Interpretational Abstraction*

Gustaf Neumann
Vienna University of Economics and Business Administration
Institute of Information Processing
Augasse 2–6
A–1090 Wien

neumann@awiwuw11.bitnet

Zusammenfassung

Interpretational abstraction is proposed as a means to overcome deficiencies in cases where procedural or data abstraction are not able to express underlying concepts explicitly. Interpretational abstraction enables the software developer to modify and extend the computational meaning of a program. Following this idea substantial benefits for maintainability and flexibility of software systems can be envisioned. Application examples for Prolog show the practicability of this approach. The instrument of program transformation is used to obtain efficient programs from abstract ones.

1 Introduction

During the last 20 years the interest of software engineers has shifted from procedural aspects towards the data objects of a system. This shift includes a move from questions of how to organize instructions in algorithmic programs for a von Neumann machine to design considerations concerned in modeling the behavior of the data and its dependencies. Abstract data types [LG86] and object oriented programming [Weg89] as it is seen today are typical instances of this data oriented view.

*Submitted to the Journal *"Computers and Mathematics with Applications"*

This shift can be observed in the various levels and forms of abstraction that are used to deal with complexity issues. Whereas instruments as Nassi–Shneiderman–Diagrams [NS73] were developed to overcome control flow problems of early days programming (control flow abstraction), modularization became a means to reduce the inherent dependencies in large monolithic programs (procedural abstraction, lambda abstraction).[1] Object oriented programming goes a step further by focusing the attention on the data objects. By defining the meaning of data objects through their behavior, locality and modifiability of the programs is improved. Object oriented programming emphasizes concepts like classification and inheritance (data abstraction) or message passing (which is a generalization of the procedure call). As Goguen and Meseguer point out in [GM87], the essence of object oriented programming lies in *"... the organization of memory into local objects, as opposed to having a single global store"*. Each object has its own identity, has internal state and exists in a context where time is important.

In this paper we propose that following these mainstream directions there is still an important dimension of abstraction missing, namely that of interpretational abstraction. Interpretational abstraction enables the software developer to modify and extend the computational meaning of a program. In a metaprogramming environment it is possible to give the same data (and the program as well) different meanings in different contexts, further to reinterpret the same situation under various aspects. As a consequence the need of reprogramming is reduced. In the next sections it will be shown that interpretational abstraction can lead to clear and concise programs in cases where procedural and data abstraction fail. Furthermore we will show that this idea is already applied in many existing programs.

2 Dimensions of abstraction

One of the main issues of programming is the question how to deal with complexity, how to create suitable building blocks that can be used to represent complex subjects as transparent as possible. When a problem reaches a certain size it is necessary to split it up into smaller units that are easier to solve. These smaller units have to be separable subproblems that should finally fit together to solve the initial problem.

[1] It is interesting to note that the first language definitions of FORTRAN did not contain user definable functions.

A very productive way to decompose complex systems is to use abstraction. Given similar problems abstraction allows for a certain task to ignore the aspects where these problems differ and to solve the common aspects once. This main idea goes back to Aristotle who tried to capture semantics in terms of genus and differentia. Applied to software design abstractions are used to deal with various aspects of programming languages. We can distinguish between

- *Control flow abstraction:* From the manipulation of a program counter to problem specific control units;
  - Nassi–Shneiderman Diagrams
  - Iteration Abstraction
- *Procedural abstraction:* From a fixed command set to the extension of the base language with new operations;
  - Abstraction by parameterization (common pieces of code using different data)
  - Abstraction by specification (describe necessary properties of the input and output data, but ignore how the results are achieved)
- *Data abstraction:* From a machine storage model to the modeling of the behavior and properties of abstract data;
  - data structures
  - classification hierarchies, inheritance
  - abstract data types
- *Interpretational abstraction:* From a fixed assignment of meaning to an evaluation relative to an environment or problem;
  - interpretational abstraction of data
  - interpretational abstraction of instructions

Each of these abstractions are means to extend the expressiveness of the virtual machine that a programmer is offered to make it a more powerful instrument for problem solving.

In this paper we will focus on the last point (a good source for the other three abstractions types is [LG86]). Interpretational abstraction is not an abstraction dimension on its own, but a generalization of the other abstractions. Throughout this paper we have symbolic languages in mind that do not stress the differences between data and programs. This generalization allows us to neglect the distinction in *"interpretational abstraction of data"* and *"interpretational abstraction of instructions"*.

We will propose interpretational abstraction as an instrument for *"programming in the middle"*, an area where the main problem lies neither in complex algorithms nor in huge projects that need large programming teams, but more in the operation of persistent systems in a changing environment.

3 The concept of interpretational abstraction

Meaning of any kind of language is defined in two terms: representation and interpretation[2]. The word representation already stands for re-presentation, which can be rephrased as a means to recall objects to your mind. Symbols are used as references to real world objects. The interpretation describes how the real world objects correlate to each other by manipulating their symbols. Interpretation is used to define the meaning of a single object in a complex context, it is an instrument to reflect about a system.

Representation and interpretation are tied together loosely. Of course it is possible to give the same representation different interpretations. The human interpretation changes by learning new concepts. These ideas are not revolutionary at all when applied to human and social sciences, but appear to be mostly neglected in today's computer programming. Let us look at classical software design, where a program has to be formed out of conceptually fairly independent components such as user interface, database interaction, mathematical libraries etc., and the domain knowledge. During software development these components have to be intertwined and to be fitted into a single, linear structure. In this process the most complex system component wins in determining the structure of the program.[3] The less complex components are added accordingly. This approach is misleading when the degree of complexity was underestimated at the first place or the system is extended in a not predicted way.

Very often programmable and extensible subsystems (that are specialized for one system component) are used as a shell to host domain knowledge as an add-on. Examples can be seen for database, data entry and calculation applications. called integrated software packages. These (very productive) systems tend to fail, however, once a certain degree of complexity is reached.

[2]The terms syntax and semantics are avoided.

[3]In my view this is also true for object oriented programming where the quality of the program depends on the suitability of a class structure for a given problem. Changing the class structure entails huge efforts in rewriting the code.

As soon as an growing application needs functionality that is not supported by the shell, the strong specialization of these shells might turn out as a handicap. Similar problems occur when such an application must be integrated ex post into a larger system,

It seems to be necessary to search for abstraction methods that offer a higher flexibility than the current practice (based on procedural and data abstraction) offers. This level of abstraction should allow maintainability and replaceability of the logically orthogonal (sub-)systems based on different interpretations for different purposes.

Once a complex system is decomposed into comprehensible subsystems, the question arises, how to build a program out of this components. For this highly abstract process of design and development of a program out of fairly independent components the following approaches can be used:

- The first approach is to start with the central component and extend existing code with additional statements on all appropriate locations. Figure 1 illustrates the case where a 'small' system component has to be merged with a more complex one. In alternative A the small circles indicate the modifications in the the larger component; the programmer has to find the locations, where updates are necessary and has to modify the code accordingly. This strategy has the disadvantage that inherent dependencies are introduced through out the code. As a consequence even formerly well structured programs tend to become increasingly opaque and less maintainable by every further unpredicted modification (see eg. [Win79]). Later modifications of the system (such as replacing one user interface by another) are often prevented for reasons of practability.
- *Procedural and data abstraction* achieve large improvements towards a code that is easier to maintain. In certain cases it is possible to change local pieces of code without facing the need to adapt instructions that are scattered all over the system. This approach (although highly recommended for todays software developers) only succeeds where later changes of the system are anticipated in the design process [Lee83]. In cases where the structurization, that was determined at an early design phase, turns out to be suboptimal, later structural modifications or changes that affect many pieces of code are still very hard and tend to decrease the quality of the code.
- *Interpretational abstraction* allows to reinterpret existing subsystems

Abbildung 1: Integrating a subsystem (A) using brute force modification or (B) by encapsulation in a metasystem

Abbildung 2: Integrating a subsystem into different host systems

in a new context (see figure 2, alternative B). It avoids the manual rewriting of the code by reinterpreting the representation in a problem specific environment. Various different interpretations can be given over the same representation domain that are used for several related problems. During such an enhanced interpretation the computations can be performed that would have been programmed into the representation, when interpretational abstraction would not have been used.[4] Interpretational abstraction is also a very powerful tool for enhancements of existing programs or for creating highly problem specific languages.
For reasons of efficiency it is in certain cases possible to remove a layer of interpretation and to compile the system to a code that is equivalent to the result of the brute force programming approach. Later we will use the vehicle of metaprogramming to add completely independent subsystems to a body of domain knowledge which is formalized in Horn clauses.

4 Interpretational abstraction and other programming paradigms

From a programming language point of view we have to compare two extremes: *extensible representation and fixed interpretation* as opposed to *extensible representation and extensible interpretation*. These representation issues can be applied on data and programs.

In conventional programming the interpretation of the program is fixed. This paradigm has proven very useful for algorithmic programming. The programmer is offered a single representation layer where he models the system. This paradigm shows deficiencies in an dynamic environment with shifting objectives. The single presentation layer forces a programmer to model the system on a surface level, where each later modification has to be incorporated. Part A of Figure 3 shows how the only accessible representation layer has to be extended when a system grows; the stars, triangles and diamonds denote later added enhancements of the system. The represen-

[4]Notice that in cases where one subsystem has to be integrated in several other host systems (or changed at a later time) the advantages of interpretational abstraction become even more striking (see figure 2), because it would be necessary to perform the updates in the affected systems by hand.

Abbildung 3: Fixed interpretation and extensible representation versus extensible interpretation and extensible representation. The stars, triangles and diamonds stand for later added enhancements of the system.

tation becomes incrementally complex and scattered, implicit dependencies are introduced.

The proposed alternative is to keep the representation as simple and fixed as possible and to enhance the interpretation whenever additional functionalities are to be incorporated. Various interpreters can be written for various problems concerning (parts of) the same representation. By variable interpretation the programmer gains control over representation as well as interpretation. Common aspects can be addressed at a higher abstraction level (see figure 3, side B). The main difference to fixed interpretation is that the meaning of the representation is now relative to a given environment that is kept in an interpreter. An array of problem specific interpreters can be used for different application specific purposes.

In the later section of this paper we will use interpretational abstraction for logic programs. This abstraction method does not only apply for logic programs. There are many similarities (both in the aims and the methods) to established approaches.

- The idea of interpretational abstraction fits well into the *"knowledge* *representation paradigm"*. It states that the task of representing knowledge should be done mostly independent from question of applications of this knowledge.

This paradigm is not only a matter of AI: Files or databases can be seen as fixed and persistent representations which are interpreted differently by various applications. When the number of applications accessing this representation grows in becomes necessary to use generalized representations with clear meaning and well defined access methods (eg. relational databases and relational algebra).

When this notion is applied to Logic Programming as it is seen today Horn clauses are used as a general representation language. The Kowalski doctrine [Kow79] *"Algorithm = Logic + Control"* emphasizes the separability of the declarative meaning (*"Logic"*) of a program and of the operational aspects of executing it (*"Control"*) such as problem solving strategies using the domain knowledge expressed in a declarative way. The control can be changed without changing the meaning of the program.

- Both object oriented programming and interpretational abstraction try to improve the reusability and flexibility of programs. Object oriented programming concentrates on making objects and classes reusable through encapsulation and inheritance of attributes and methods. The inheritance is performed in accordance with an object classification. We show in the next sections that interpretational abstraction can also be used to inherit data or methods to other programs but now according to the syntactical constructs of the programs (predicates and terms) and their properties.

  Interpretational abstraction and object oriented programming have, however, different aims: Whereas the power of object oriented programming lays in modeling the behavior of objects with identity, interpretational abstraction can be used efficiently to reason about the meaning of a specific constellation of various entities for a certain aspect in a certain context.

- An abstract data type is defined by the tuple $\langle objects, operations \rangle$ ([LG86]). In order to access an *object*, the user is required to call *operations* manipulating these objects instead of accessing their representation directly. Thus, in order to implement an abstract data type, it is essential to implement the relevant operations. Outside of the implementation of the operations the representation is invisible.

Implementing this abstraction type in a logic program is a matter of programming style. For example in order to define the type "`list`" with its operations "`is_empty`", "`first_element`" and "`rest_list`", very simple predicates can be used. This way, the internal representation of the datatype will not be directly accessible by a program using this type. The internal representation of the abstract datatype (in our example the list constructor, or more generally *"rep type"* in [LG86]) can easily be changed.

The principle of *"information hiding"* is somehow violated in logic programs, because all data objects have to be passed around in the arguments of predicates. Very often it is necessary to introduce extra arguments only to pass some global data objects from a predicate to its subgoals. As a consequence the argument lists of the predicates tend to become large, the comprehensibility and maintainability of the program decreases. A clean way to circumvent this problem is to keep the information about such objects in the interpreter, keeping the program layer as simple as possible. Various interpreters can be used for different implementations of the same abstract datatype.[5]

- Logic Programs arc well-suited for program transformation. New programs can be efficiently derived from existing ones using fold-unfold techniques (as eg. the clausal join in [LS88a]) or compilation techniques (see for example [FF88a, Neu86, Neu88, TF86, SS86]).

- Interpretational abstraction can be used similarly to a macro processor. Where a standard macro processor (eg. the C-preprocessor) works on the tokens of a given language, interpretational abstraction is applied to the parse tree of the program in order to perform computations according to certain syntactical or semantical properties of the program. An instance of this semantic macro processing can be found in [Che83] where transformation rules of the form

$$\textit{pattern} \quad \textbf{Where} \quad \textit{predicate} \quad \leftrightarrow \quad \textit{replacement}$$

[5]In the book [LG86] the term *"different interpretation for same representation"* is used in context with data abstraction (page 72). The questions addressed there concern however the many-to-one relationship between internal representation types and abstract data types.

are used. *pattern* denotes a syntactical construct of the source language, *predicate* stands for semantical properties, and *replacement* replaces *pattern* in the resulting program.

- Interpretational abstraction improves the locality of problem specific information. It reduces the need to inspect various pieces of code in order to examine a certain property of the system. This is also true for situations where the interesting information is not explicitly manifested in the structure of the representation. Independent subcomponents can be modeled separately and be easily exchanged. The maintainability and elegance of the program is increased. Elegance, which can be formalized as

$$\frac{\textit{number of connections}}{\textit{number of items}}$$

  is normally not one of the most important issues in software engineering, but there is a certain hope that reducing the number of symbols leads to a code that is easier to understand and to maintain. As we show in later examples, interpretational abstraction using metainterpreters is a very elegant approach.

5 Interpretational abstraction and metainterpreter

After we have introduced the abstract concept of interpretational abstraction, we address the question how it can be used or implemented in existing environments. The answer is, that it interpretational abstraction is already used widely but its full power has not been exploited yet.

For the implementation of interpretational abstraction, a metaprogram is needed. A metaprogram is a program that reasons about other program, that performs computations according to some object programs. Thus, a metaprogram defines the meaning of an object program. A metaprogram can be either a compiler (source and target language are different), a program transformator (source and target language are identical) or an interpreter (a program that implements a virtual machine). A program transformator is a special case of a compiler.

It has been shown that a wide class of interpreters can be transformed into a compiler by a meaning preserving operation, where the interpreted program has the same meaning as the compiled result. This transformation has been shown for procedural languages (see [Ers78, Pag88]), functional languages such as a subset of pure Scheme (see [Jon85]) and logic based languages (see [FF88a, LS88b, Neu86, Neu88, TF86, SS86]). The reasons for transforming interpreters to compilers are mainly speed issues that will not be addressed further.

Throughout the rest of the paper we will use the instrument of interpreters to implement interpretational abstraction, simply because interpreters are easier to understand and to use than compilers.[6] symbolical languages such as Lisp or Prolog program statements can be processed with the same ease as data structures. In the Prolog context interpreters for Prolog-like languages are commonly called metainterpreters. Interpreters written for a language that is identical to the implementation language of the interpreter are called metacircular evaluators [ASS85] or autoprojectors [SS86].

Of course, metainterpreters can be written for many different purposes. Sometimes applications of interpretational abstraction can also be found in programs that were not be written in the intention to write a metaprogram. We will show in section 6.2 such a program that shares many properties of a metainterpreter. The following classification applies for all types of metainterpreters:

- *The program to be interpreted by the metainterpreter can be executed in a meaningful way also without the interpreter.* The metainterpreter changes the operational meaning of a given program. The metalevel can be used to add enhancements to an existing program. A commonly used introductory example for metainterpreters belonging to this class is the explanation generating interpreter, which computes in addition to values for a certain query an explanation how these values are derived (this metainterpreter is discussed in section 6.1.2).
- *The program to be interpreted by the interpreter cannot be executed in a meaningful way without the interpreter.* The program to be interpreted relies on certain properties of the interpreter and cannot be executed in the underlying environment. These properties can be

[6] Note that interpreters are only one way to implement the ideas of interpretational abstraction.

either of syntactical (for instance Prolog's definite clause grammars [PW80]) or of semantical (eg. parallel language constructs that are implemented by the interpreter) nature.

When a program depends on certain syntactical or semantical properties, also the according interpreter must deal with this properties (second type of interpreter). The reusability of such a specialized interpreter is somewhat more limited compared to an interpreter that works on standard program clauses that can be potentially used for each Prolog program.

Interpreters that enhance or modify the meaning of given programs, can be classified as follows:

- *Interpreters that implement conservative modifications of a program:* For a logic programming language several interpreters can be given that compute the same results according to the programs declarative meaning ([Kow79]). While the variety of semantic equivalent interpreters is rather high for pure logic programs, it is much more limited for programs issuing side-effects. As the computational meaning of pure programs is defined in terms of success/failure and variable substitutions of the initial goal, the meaning of impure programs has to comprise runtime errors, side-effects and the order in which the side-effects are issued ([Neu86, Neu88]).

  An interpreter that does not alter for a given program the declarative meaning of the goals issued is called an *enhancement* or an enhanced metainterpreter. An enhanced metainterpreter might compute in further arguments of the interpreter additional values, or it might issue additional side-effects during the execution of the goal (for a detailed discussion of modulants, enhancements and mutants see [SL88]).

- *Interpreters that implement nonconservative modification of a program:* If an interpreter alters the computational meaning of a program it is called a *mutant* of the base interpreter.

  A simple example of a mutant is a depth bounded interpreter that behaves like Prolog, except that it limits the deduction depth to a given value. This method can be used for debugging ([Sha83b]) of programs that do not terminate.

6 Interpretational abstraction in Prolog

Standard Prolog interpreters and compilers execute Prolog rules using top down, depth first, left to right search. For this reason, it is sometimes thought that a program in Prolog can only be executed in this way.

However, there are at least two commonly used programming methods that effectively tailor Prolog's built-in inference engine to particular applications. The first method is to simply reprogram, – for example it is possible to rewrite a Prolog program in a way that it produces explanations of its own run. The second method, which forms the basis of many expert system shells, is to write a metainterpreter consisting of Prolog rules that define how the object rules will be processed.

In Prolog it is especially simple to write a metainterpreter which is able to run pure Prolog programs. A very common approach is to start off with this interpreter and enhance it with the needed functionality. The first example given in the following was formed this way. It is not necessary for programs running on a metainterpreter to be executed after the Prolog execution scheme. [Neu88] contains a collection of 12 different application examples that range from expert systems applications such as inexact reasoning [Sha83a] and forward chaining to simple computer graphics, parallel Prolog variants for intelligent simulation applications (a T-Prolog dialect [FS83], see also [AN85]) and to the integration of user-interfaces via metaprogramming.

For some applications it is easy to distinguish between the metainterpreter and the program that is to be interpreted. All mentioned examples are typical instances of this situation. Sometimes the interpreters were written to attach additional functionalities to a given program (explanation generation, integration of a user interface), in other cases the programs were written in a way such they could be interpreted by a given metainterpreter (eg. the T-Prolog dialect). Section 6.2, however, shows that the same techniques can also be applied in cases where it was not intended to write metainterpreters. Many well written programs already follow this principle although they do not have a program part that will normally be called interpreter. When data and program are treated uniformally it just depends on the point of view whether a program is called metaprogram or not.

6.1 Forming a system out of independent subcomponents

In the following example it is shown how a given program P_1 (in our case a simple part hierarchy program) can be extended with an additional, fairly orthogonal functionality: The task is to modify the program P_1 in a way such it becomes able to explain the results of its own run. The explanation will consist of the derivation tree of the stated goal. Two independent subsystems (here "part hierarchy" and "explanation generation") will be merged to a common system.

6.1.1 The part hierarchy example

Program P_1 (see figure 4) defines the relationship between parts and subparts of a mechanical assembly. The predicate "`parts`" specifies how an abstract "`Part`" is made up of "`SubParts`" and "`SubSubParts`" respectively. The predicate "`immediateParts`" contains information on how many parts and subparts are needed for a certain assembly, in our case a unicycle.[7]

```
parts(Part,SubSubPart,N) :-
    immediateParts(Part,SubPart,N0),
    parts(SubPart,SubSubPart,N1),
    N is N0 * N1.

parts(Part,SubPart,N) :-
    immediateParts(Part,SubPart,N).

immediateParts(unicycle,wheel,1).
immediateParts(unicycle,pedal,2).
immediateParts(wheel,spoke,28).
immediateParts(spoke,spokenipple,1).
```

Abbildung 4: P_1, a part hierarchy in Prolog

A sample goal for P_1 is "`parts(unicycle,spokenipple,X)`". This goal computes the number of "`spokenipples`" used in a "`unicycle`". As result the variable "`X`" is bound with the value "`28`".

[7] We use the term *"spokenipple"* to denote the device connecting a spoke with the rim.

6.1.2 Explanation generation

In a next step our aim is to enhance the functionalities of the program P_1 in such a way that it computes not only the results as above, but also an explanation of how these results are derived. The explanation, we want to obtain, consists of the proof tree of the initial goal.

```
parts(unicycle,spokenipple,28)
   immediateParts(unicycle,wheel,1)
   parts(wheel,spokenipple,28)
      immediateParts(wheel,spoke,28)
      parts(spoke,spokenipple,1)
         immediateParts(spoke,spokenipple,1)
      28 is 28 * 1
   28 is 1 * 28
```

Abbildung 5: An explanation that should be generated from a run of P_1

Figure 5 contains the derivation tree for the goal "`parts(unicycle, spokenipple, X)`". Each indentation layer expresses conjunctive goals. It can be read as follows: "`parts(unicycle, spokenipple, 28)`" is true because the subgoals "`immediateParts(unicycle, wheel, 1)`" and "`parts( wheel, spokenipple, 28)`" and "`28 is 1 * 28`" can be shown. On the next layer the explanation for "`parts(wheel, spokenipple, 28)`" is to be read likewise. The built-in predicate "`is`" is not explained further.

The task of generating an explanation is not related with the task of computing parts and subparts of an assembly. Using standard programming techniques, however, it is necessary to intertwine these two subcomponents manually to a new system. For this task procedural abstraction cannot help, because the change affects every computation step of the program.

6.1.3 Obtaining a common system using either brute force programming or metainterpreters

In the first step the program P_1 will be 'manually' rewritten to a program $P_1{}'$ which in addition to the previous results will generate explanation trees that show how the results are computed.

```
parts(V0,V1,V2,(parts(V0,V1,V2) :-
          (immediateParts(V0,V3,V4) :- V5),
          (parts(V3,V1,V6) :- V7),  s(V2 is V4 * V6))) :-
    immediateParts(V0,V3,V4,immediateParts(V0,V3,V4) :- V5),
    parts(V3,V1,V6,parts(V3,V1,V6) :- V7),
    V2 is V4 * V6.

parts(V0,V1,V2,parts(V0,V1,V2) :- (immediateParts(V0,V1,V2) :- V3)) :-
    immediateParts(V0,V1,V2,immediateParts(V0,V1,V2) :- V3).

immediateParts(unicycle,wheel,1,
    immediateParts(unicycle,wheel,1) :- s(true)).
immediateParts(unicycle,pedal,2,
    immediateParts(unicycle,pedal,2) :- s(true)).
immediateParts(wheel,spoke,28,
    immediateParts(wheel,spoke,28) :- s(true)).
immediateParts(spoke,spokenipple,1,
    immediateParts(spoke,spokenipple,1) :- s(true)).
```

Abbildung 6: Obtaining explanations from P_1 using the brute force programming approach

Figure 6 shows the modified program. Each user defined predicate of P_1 appears with one additional argument in $P_1{}'$ that will keep the explanation of that goal. The term "`s(X)`" is used to denote the presence of built-in predicates which cannot be explained further. The goal "`parts(unicycle, spokenipple, X, Expl)`" binds "`X`" with "`48`" and the variable "`Expl`" with the derivation tree of this goal, which is can be prettyprinted as the content of figure 6.1.2 by omitting the operators, empty leaf nodes and the annotation for system predicates.

Using the brute force programming approach it is a tedious task to make Prolog programs self explanatory. A much more elegant approach is to use a metainterpreter where the derivation tree is built in an additional argument during interpretation.

```
ie((V0,V1), (V2,V3)) :- !,
    ie(V0,V2),
    ie(V1,V3).

ie(V0, s(V0)) :-
    sys(V0),
    !,
    call(V0).

ie(V0, (V0 :- V1)) :-
    clause((V0 :- V2)),
    ie(V2,V1) .
```

Abbildung 7: I_e, an interpreter for generating positive explanations

The explanation generating interpreter in figure 7 consists of three clauses: The first clause is applied for conjunctive goals that are decomposed recursively. The second clause is used for built-in predicates, the last clause deals with Prolog defined predicates which are unfolded and recursively processed. The first argument of this interpreter contains the goal that is to be processed, the second argument keeps the explanation. The call of the interpreter "`ie(parts(unicycle, spokenipple, X), Expl)`" binds "`X`" with "`48`" and the variable "`Expl`" with the derivation tree of this goal, which is identical with the result of the brute force programming approach.

This example shows how the same results can be obtained using the brute force programming approach leading to the program $P_1{}'$ or following the metaprogramming idea using the interpreter I_e and the original program P_1. It has been shown in [Neu86, Neu88] how to derive the program $P_1{}'$ automatically from the interpreter I_e and the program P_1. This transformation is done in two steps: first a compiler generator converts the interpreter I_e into a compiler, in a second step the compiler is applied on program P_1. The resulting residual program is $P_1{}'$.

The program P_1 and the interpreter I_e are a very simple example but we hope that it shows the ideas behind interpretational abstraction clearly. As mentioned above [Neu88] contains a collection of 12 different application interpreters that demonstrate the flexibility of the approach. In our next example we will show that the application program P_1 can also be treated as an interpreter.

6.2 The part hierarchy example as an instance of interpretational abstraction

Program P_1 (figure 4) consists of two different predicates: The predicate "`parts`" defines how parts and subparts can be associated and which arithmetic operations are needed to compute the actual figures. "`parts`" is independent of an actual assembly. The predicate "`immediateParts`" supplies information about a specific assembly.

Paraphrasing the last paragraph in another way we could say that the "`parts`" predicate defines how the contents of the "`immediateParts`" clauses correlate to each other, or that the "`parts`" predicate interprets the data in the "`immediateParts`" facts.

As before it is possible to convert the "`parts`" predicate into a compiler which generates the programs in figures 8–10 from program P_1. As before the information kept on a higher abstraction layer (now in the predicate "`parts`") is incorporated into the clauses at a lower abstraction layer. The information how to correlate unit information now appears implicitly through out the code. The programs P_2 to P_4 are not as general as the program P_1 and if they would have been written by a programmer they would be regarded as bad programming style. These programs are however more efficient than the source program.

```
parts(unicycle,V0,V1) :-
   parts(wheel,V0,V2),
   V1 is 1 * V2.
parts(unicycle,wheel,1).

parts(unicycle,pedal,2).

parts(wheel,V0,V1) :-
   parts(spoke,V0,V2),
   V1 is 28 * V2.
parts(wheel,spoke,28) .

parts(spoke,spokenipple,1) .
```

Abbildung 8: P_2, a program that computes the same results as P_1, but where the information of how to combine elementary facts appears implicitly in each clause

```
unicycle(V0,V1) :-
   wheel(V0,V2),
   V1 is 1 * V2.
unicycle(wheel,1).

unicycle(pedal,2).

wheel(V0,V1) :-
   spoke(V0,V2),
   V1 is 28 * V2.
wheel(spoke,28).

spoke(spokenipple,1).
```

Abbildung 9: P_3, a program to compute from a given part corresponding subparts

```
wheel(V0,V1) :-
   unicycle(V0,V2),
   V1 is 1 * V2.
wheel(unicycle,1).

pedal(V0,V1) :-
   wheel(V0,V2),
   V1 is 2 * V2.
pedal(unicycle,2) .

spoke(V0,V1) :-
   wheel(V0,V2),
   V1 is 28 * V2.
spoke(wheel,28).

spokenipple(V0,V1) :-
   spoke(V0,V2),
   V1 is 1 * V2.
spokenipple(spoke,1).
```

Abbildung 10: P_4, a program to compute from subparts the parts where they are used

6.3 Reusability and exchangability though metainterpreters

As discussed in section 5 metainterpreters that work on ordinary Prolog clauses can be freely used for each Prolog program. For instance the explanation generating interpreter I_e can be applied to the programs presented here, such as the original part example P_1 (figure 4) or the specialized programs P_2 to P_4 (figures 8-10). For programs containing cut symbols, it is either necessary to write interpreters that can handle cut correctly, or to transform the used interpreters into compilers, where it is easier to deal with the cut (see [Neu88]). This way it is even possible to apply I_e on itself and to generate explanations about how explanations are generated.

When the way how to generate explanations should be changed, it is just necessary to change change the interpreter. Interpretational abstraction allows here to make local changes in the interpreter that would need global changes in the program without making use of this abstracion level.

Interpretational abstraction can also be used to "apply know-how on data": For example the parts program P_1 (figure 4) knows how to combine arbitrary parts and subparts. We have shown how various differently specialized programs can be derived from a "know-how" in form of an interpreter and data (in form of facts) about some assembly. If the data is changed programs of a different application domain may be derived containing the same knowledge. Similarly [Lak89] shows how metainterpreters can also be used to incorporate programming techniques (for example the accumulator technique) into logic definitions.

7 Conclusion

In this paper we have presented the ideas of interpretational abstraction which can be elegantly applied on symbolic languages such as Prolog. Similar ideas are expressed somewhat disguised or with different intentions in various other work. We have the feeling that the current mainstream activities towards deeper understanding of applications for object oriented programming can be complemented by the ideas presented in this paper. Interpretational abstraction can lead to more flexible programming environments where an array of application specific interpreters can coexist that act on a central body of knowledge.

8 Acknowledgment

Many thanks to Ulrich Neumerkl for fruitful discussions on this topic and to Arun Lakhotia for his comments on the draft version of this paper.

Literatur

[ASS85] H. Abelson, G.J. Sussman, J. Sussman: *"Structure and Interpretation of Computer Programs"*, The MIT-Press, Cambridge 1985.

[AN85] H.H. Adelsberger, G. Neumann: *"Goal Oriented Simulation Modeling using Prolog"*, in: Lavery R.G., Modeling & Simulation on Microcomputers, Society of Computer Simulation, La Jolla 1985.

[Che83] T.E. Cheatham: *"Reusability Through Program Transformations"*, in: Readings in Artificial Intelligence and Software Engineering, edited by C. Rich and R.C. Waters, pp. 185–190, Morgan Kaufmann Publishers, Los Altos 1986.

[Ers78] A.P. Ershov: *"On the Essence of Compilation"*, in: IFIP Working Conference on Formal Descriptions of Programming Concepts, edited by E.J. Neuhold, pp. 391–420, North–Holland, New York 1978.

[FF86] K. Fuchi, K. Furukawa: *"The Role of Logic Programming in the Fifth Generation Computer Project"*, London 1986, LNCS 225, pp. 1-24.

[FF88a] H. Fujita, K.. Furukawa: *"A Self-Applicable Partial Evaluator and its use in Incremental Compilation"*, in: New Generation Computing, Vol. 6, Nos. 2 and 3, Springer Verlag 1988.

[FS83] I. Futo, P. Szeredi: *"T–Prolog User Manual"*, Version 4.2, Institute for Coordination of Computer Technology, Budapest 1983.

[GM87] J.A. Goguen, J. Meseguer: *"Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics"*, in: [SW87].

[Jon85] N.D. Jones, P. Sestoft, H. Søndergaard: *"The Generation of a Compiler Generator, 1st International Conference on Rewriting Techniques and Applications"*, Dijon 1985, LNCS 202, pp. 120-140.

[KC79] S. Kondoh, T. Chikayama: *"Macro Processing in Prolog"*, in: Proceedings of the Fifth International Conference and Symposium, MIT-Press Cambridge 1988.

[Kow79] R.A. Kowalski: *"Algorithm = Logic + Control"*, in: CACM, Vol. 22, No. 7, July 1979.

[Lee83] R.M. Lee: *"Application Software and Organizational Change: Issues in the Representation of Knowledge"*, Information Systems, Vol. 8, No. 3.

[LG86] B. Liskov, J. Guttag: *"Abstraction and Specification in Program Development"*, The MIT-Press, Cambridge 1986.

[LS88a] A. Lakhotia, L. Sterling: *"Composing Logic Programs with Clausal Join"*, in: New Generation Computing, Vol. 6, Nos. 2 and 3, Springer Verlag 1988.

[LS88b] A. Lakhotia, L. Sterling: *"How to Control Unfolding when Specializing Interpreters"*, Case Western Reserve University, Technical Report CWRU CES–88–08, May 1988.

[Lak89] A. Lakhotia: *"Incorporating 'Programming Techniques' into Logic Programs"*, in [LO89].

[LO89] E.L. Lusk, R.A. Overbeek (eds): *"Logic Programming: Proceedings of the North American Conference 1989"*, MIT Press, Cambridge 1989.

[Neu86] G. Neumann: *"Meta-Interpreter Directed Compilation of Logic Programs into Prolog"*, IBM-Research Report RC 12113 (#54357), Yorktown Heights, New York 1986.

[Neu88] G. Neumann: *"Metaprogrammierung und Prolog"*, Addison–Wesley, Bonn 1988.

[NS73] I. Nassi, B. Shneiderman: *"Flowchart Techniques for Structured Programming"*, SIGPLAN Notices Vol. 8, No. 8, August 1973.

[Pag88] F.G. Pagan: *"Converting Interpreters into Compilers"*, Software–Practice and Experience, Vol. 18, No. 6, June 1988.

[PW80] F.C.N. Pereira, D.H.D. Warren: *"Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparism with Augmented Transition Networks"*, Artificial Intelligence, Vol. 13, Nr. 3, pp. 231-278, May 1980.

[SS86] S. Safra, E. Shapiro: *"Meta-Interpreters for Real"*, in: IFIP 1986, H.-J. Kugler, ed., Elsevier Science Publishers 1986.

[Sha83a] E. Shapiro: *"Logic Programs with Uncertainties: A Tool for Implementing Rule-Based Systems"*, Proceeding of the Eighth International Joint Conference on Artificial Intelligence, 8-12 August 1983, Karlsruhe, West Germany.

[Sha83b] E. Shapiro: *"Algorithmic Program Debugging"*, MIT Press, Cambridge 1986.

[SW87] B. Shriver, P. Wegner: *"Research Directions in Object-Oriented Programming"*, The MIT-Press, Cambridge 1987.

[SL88] L. Sterling, A. Lakhotia: *"Composing Prolog Meta–Interpreters"*, in: Proceedings of the Fifth International Conference and Symposium, MIT-Press Cambridge 1988.

[TF86] A. Takeuchi, K. Furukawa: *"Partial Evaluation of Prolog Programs and its Application to Meta-Programming"*, in: IFIP 1986, H.-J. Kugler, ed., Elsevier Science Publishers 1986.

[Weg89] P. Wegner: *"Learning the Language"*, Byte, March 1989.

[Win79] T. Winograd: *"Beyond Programming Languages"*, CACM, Vol. 22, No. 7, 1979.