On GCLA, Gisela, and MedView
Studies in Declarative Programming with Application to Clinical Medicine

Olof Torgersson

Department of Computing Science
Chalmers University of Technology and Göteborg University
Göteborg, 2000

Thesis for the degree of Doctor of Philosophy

On GCLA, Gisela, and MedView
Studies in Declarative Programming with Application to Clinical Medicine

Olof Torgersson

Department of Computing Science
Chalmers University of Technology and Göteborg University
Göteborg, Sweden, 2000

On GCLA, Gisela, and MedView
Studies in Declarative Programming with Application to Clinical Medicine
Olof Torgersson
ISBN 91-628-4354-0

Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden

Printed at Chalmers, Göteborg, 2000

On GCLA, Gisela, and MedView
Studies in Declarative Programming with Application to Clinical Medicine
Olof Torgersson
Department of Computing Science
Chalmers University of Technology and Göteborg University

Abstract

Using declarative programming a programmer should be able to concentrate on *what* a program should do without worrying to much about *how* it is done. To be able to advance declarative programming methodologies, real-world examples are needed that push the limits of the proposed programming techniques. In this thesis we focus on some aspects of declarative programming, mostly from an applied angle. Specifically, various issues in the area of *definitional programming*, which has its roots in the theory of partial inductive definitions, are investigated.

The thesis consists of four separate but related parts. In the first part it is shown how functional logic programming can be achieved in the definitional programming language GCLA. Although all examples are given in GCLA the general ideas could just as well be applied to build a specialized functional logic programming language based on definitional ideas.

The second part presents the Gisela framework for definitional programming. Gisela is intended to provide a general extensible framework for definitional programming based on a new definitional computation model. The framework is also intended to facilitate building state-of-the art GUI-based applications with embedded definitional reasoning components. The computation model and various programming techniques are described.

The third part gives an overview of the MedView project. MedView is a joint project with participants from computer science and clinical medicine. The project is centered around the question: how can computing technology be used to handle clinical information in everyday work such that clinicians more systematically can learn from their gathered clinical data? All knowledge representation in MedView is based on a definitional model.

The fourth part concerns how Gisela can be used to facilitate knowledge representation and application development in the MedView project. Expressed differently, how MedView is used as a real-world example for declarative programming techniques. It is shown how the MedView knowledge base can be represented using Gisela and how to use Gisela for finding information in the knowledge base. A number of example applications are described as well as the general methodology used to incorporate definitional reasoning machinery based on Gisela into object-oriented applications with GUIs.

Keywords: declarative programming, functional logic programming, definitional programming, knowledge based systems, medical informatics.

This thesis consists of the following papers:

- Olof Torgersson. *Functional Logic Programming in GCLA.* Different parts of this paper are published as:

  - O. Torgersson. Functional logic programming in GCLA. In U. Engberg, K. Larsen, and P. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory.* Aarhus, 1994.
  - O. Torgersson. A definitional approach to functional logic programming. In *Extensions of Logic Programming, ELP'96*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1996.

  The version included here is a revised version of a paper with the same name in

  - O. Torgersson. *Definitional programming in GCLA: Techniques, functions, and predicates.* Licentiate thesis, Chalmers University of Technology and Göteborg University, 1996.

- Olof Torgersson. *Gisela—A Framework for Definitional Programming.*

- Youssef Ali, Göran Falkman, Lars Hallnäs, Mats Jontell, Ulf Mattson, Nader Nazari, and Olof Torgersson. *An Overview of MedView.*
  This paper is based on material from

  - Y. Ali, G. Falkman, L. Hallnäs, M. Jontell, N. Nazari, and O. Torgersson. Medview: Design and adoption of an interactive system for oral medicine. In *Proceedings of Medical Informatics Europe (MIE'00), Hannover, Germany, August 2000*, 2000,
  - M. Jontell, L. Hallnäs, and U. Mattson. Oralmedicinsk informationsteknologi i klinisk praxis. In *Odontologi 2000*, pages 217–230. Munksgaard, København, 2000,
  - G. Falkman, M. Jontell, and N. Nazari. Information visualisation in clinical medicine using 3D parallel diagrams: a case history. In *Proceedings of Medical Informatics Europe (MIE'00), Hannover, Germany, August 2000*, August 2000,
  - SimVis: an interaction model for exploring clinical data. In G. Szwillus and T. Turner, editors, *CHI 2000 Extended Abstracts. Conference on Human Factors in Computing Systems, 1–6 April 2000, The Hague, The Netherlands*, pages 319–320. ACM Press, New York, 2000,

  revised for the present paper and extended with some new material.

- Olof Torgersson. *MedView and Gisela.*

Acknowledgements

I would like to thank Lars Hallnäs for constant support and inspiration, and for forcing me to rewrite parts of this thesis several times when I thought it was all written, Göran Falkman for all the ideas and opinions, and for reading my papers meticulously, Peter Hammond for valuable comments and corrections, Jan Smith and Björn von Sydow for providing input on the initial version of this thesis, Mats Jontell, Nader Nazari, and Youssef Ali for collaboration in the MedView project, Bengt Nordström for help with my final course-plan, Christer Carlsson for support and for giving me the opportunity to work as a teacher during the last year, all students who have contributed to making teaching an inspiring part of my work, everyone at the computing science department for keeping up the good work, and finally: My family for being there.

On GCLA, Gisela, and MedView Studies in Declarative Programming with Application to Clinical Medicine

Olof Torgersson

1 Introduction

Since the computer was first invented, more than 50 years ago, a very large number of programming languages and programming techniques for these have been developed. From the early days of programming using machine instructions, via assembler, high-level languages, structured imperative programming, to today's sophisticated object-oriented application development environments and very high-level declarative programming languages.

The development of more and more sophisticated programming tools has been paralleled by the development of more and more powerful hardware. Whenever more powerful machines have been developed, software developers have tended to build applications demanding more computing power. Whenever a more powerful programming paradigm has evolved, it has been used to its limit, calling for even more powerful development tools. Today, most owners of an ordinary PC for use at home, have more computing power available than what was present in the machines used to travel to the moon thirty years ago.

Developing large software systems is a non-trivial task. Software is often plagued by errors, and late delivery is more of a rule than an exception when it comes to releases of major systems. Software engineering techniques have been developed to aid in the process. Much research effort has been, and is being, put into ways of proving programs correct, or automatically creating programs known to satisfy given specifications for a problem. So far, software engineering techniques are used for programming-in-the-large, while actually proving program correctness can only be done for smaller programs.

The choice of programming language is important since different languages are more or less well suited for different tasks. At a very high level the programming languages existing today can be divided into *imperative* and *declarative* languages. Imperative programming languages have been around since the beginning of computer science and are still used in an overwhelming majority of all software systems being developed. Imperative programming languages are based on the Von Neumann sequential model of computation and the programmer must specify with great detail *how* computations should be performed. Since imperative programming languages in some sense are directly based on the actual machine used, compilers can be written which generate very efficient code. On the other hand, the programmer must explicitly deal with low-level detail like allocating and de-allocating memory. This means that the gap between the cognitive model, natural for the human mind, and the encoding, as a program, might be very wide which makes coding an unnecessarily complicated and error-prone task.

Declarative programming languages, on the other hand, start from a human point of view. The foundation for declarative programming languages is typically computation-models aimed at being conceptually appealing for the human mind. Using a declarative programming language, the programmer should be able to focus more on *what* the program should do than on the details for how it should be done. Not surprisingly, this approach leads to programs that are less efficient, even though modern compilers for declarative languages come close enough to imperative ones in many cases.

In this thesis we focus on some aspects of declarative programming, mostly from an applied angle. Specifically, various issues in the area of *definitional programming* are investigated. Both programming techniques and Gisela, a framework for definitional programming based on a new computation model, are presented. The system for definitional programming developed was designed with the specific goal of making interaction between a declarative system and surrounding imperative programs natural, thus enabling us to use each paradigm for the task where it is best suited. Furthermore, we give an overview of MedView, a larger research project based on definitional knowledge representation and programming methodologies, which has served as a driving force for a substantial part of the work presented here.

2 Declarative Programming

There are several different branches of declarative programming languages, the main ones being functional and logic programming. In this section, we discuss some declarative programming paradigms, including definitional programming. In particular we will discuss how control is handled in some example languages. One reason for focusing on control issues is that management of control is a major difference between definitional programming and other declarative approaches. Another is that the higher level of abstraction used in declarative languages might make it harder for the programmer to understand the actual computational content of a program. The effect of this might be that it becomes difficult to find performance bottle-necks and programming errors.

2.1 Declarative Programming and Control

Most declarative programming languages stem from work in artificial intelligence and automated theorem proving, areas where the need for a higher level of abstraction and a clear semantic model of programs is obvious.

The basic property of a declarative programming language is that a program is a theory in some suitable logic. This property immediately gives a precise meaning to programs written in the language. From a programmer's point of view the basic property is that programming is raised to a higher level of abstraction. As mentioned above, this higher level of abstraction enables the programmer to concentrate on stating *what* is to be computed, not necessarily *how* it is to be computed. In Kowalski's terms where $algorithm = logic + control$ [28], the programmer supplies the logic but not necessarily the control.

According to [34], declarative programming can be understood in a *weak* and in a *strong* sense. Declarative programming in the strong sense then means that the programmer only has to supply the logic of an algorithm and that all control information is supplied automatically by the system. Declarative programming in the weak sense means that the programmer, apart from the logic of a program, also must give control information to yield an efficient program.

2.1.1 Functional Programming

In functional programming languages programs consist of a number of *function definitions*. To give meaning to programs these are typically mapped to some version of the λ-calculus, which is then given a denotational semantics. There are both impure (strict) functional languages like Standard ML [36], allowing things like assignment, and pure (lazy) functional languages like Haskell [27].

On the one hand, modern functional languages, such as Haskell, come rather close to achieving declarative programming in the strong sense. On the other hand, the execution order of lazy evaluation is not very easy to understand and the real computational content of a program is hidden under layers and layers of syntactic sugar and program transformations. Consequently, the programmer loses control of what is really going on and may need special tools like heap-profilers [40] to find out why a program consumes memory in an unexpected way. To fix the behavior of programs the programmer may be forced to rewrite the declarative description of the problem in some way better suited to the particular underlying implementation. Thus, an important feature of declarative programming may be lost. The programmer does not only have to be concerned with how a program is executed but has to cope with a model that is hard to understand and *very* different from the *intuitive understanding* of the program.

However, functional programming languages provide many elegant and powerful features like higher order functions, strong type systems, list comprehensions, and monads [46]. Several of these notions are being adopted in other declarative programming languages (see below).

Functional languages also support code reuse, and come with a number of useful predefined functions. While reuse of general higher-order library functions results in shorter programs, it is not obvious that it always makes programs easier to understand and maintain. An example of a general list reducing function is `foldr`, which can be used to define almost any function taking a list and reducing it somehow:

```
fun foldr _ u nil = u
|   foldr f u (x::xs) = f x (foldr f u xs)
```

Functional programming methodology makes heavy use of `foldr` and similar functions to define other functions based on recursion over lists. For instance the function `len`, computing the length of a list, can be defined as

```
val len = foldr (add o K 1) 0
```

where `add` computes the sum of two integers and `K` is the K combinator. We leave it up to the reader to understand how this definition works.

As mentioned, general higher-order functions, function composition, list comprehensions and so on, makes it possible to write very concise programs. However, it also tends to turn functional programming into an activity for experts only since it becomes difficult for non-experts to understand the resulting programs.

2.1.2 Logic Programming

For practical applications there is one logic programming language in widespread use: Prolog [12]. Prolog is used for a variety of applications in artificial intelligence, knowledge-based systems, and natural language processing. A (pure) Prolog program is understood as a set of Horn clauses, a subset of first order predicate logic, which can be given a model-theoretic semantics, see [32]. Programs are evaluated by proving queries with respect to the given program.

From a programming point of view, Prolog provides two features not present in functional languages: built-in search and the ability to compute with partial information. This is in contrast to functional languages where computations always are directed, require all arguments to be known, and give exactly one answer. A simple example of how a predicate can be used to compute in several modes and with partial information is the following program for for appending two lists:

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Not only can `append` be used to append two lists but also to take a list apart or to append some as yet unknown list to another one.

The built-in search makes it easier to formulate many problems. For instance, if we define what we mean by a subset of a list

```
subset([], []).
subset([X|Xs], [X|Ys]) :- subset(Xs, Ys).
subset([_|Xs], Ys) :- subset(Xs, Ys).
```

we can easily compute *all* subsets by asking the system to find them for us:

```
| ?- findall(Sub, subset([1,2,3], Sub), SubSets).

SubSets = [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

While the built-in search gives powerful problem solving capacities to the language it also makes the evaluation principle more complicated. Accordingly, Prolog and most other logic programming languages only provide declarative programming in the weak sense where the programmer needs to provide some control information to get an efficient program from a declarative problem description. The method used by many Prolog programmers is rather ad hoc: First, the problem is expressed in a declarative fashion as a number of predicate definitions. Then the program is tested and found slow. To remedy the problem the original program is rewritten by adding control information (cuts) as annotations in the original code and doing some other changes like changing the order of arguments to predicates. The resulting program is likely to be quite different from the original one—and less declarative.

As an example, assume that we wish to define a predicate `lookup` that finds a certain element in a list of pairs of keys and values. If the element is missing, the result should be the atom `notfound`. A first definition could be:

```
lookup(Key, [], notfound).
lookup(Key, [(Key,Value)|_], Value).
lookup(Key, [_|Xs], Value) :- lookup(Key, Xs, Value).
```

If we compute `lookup(1, [(1,a),(1,a)], V)` we will get the answer `V = a` twice and the answer `V = notfound` once. Of course, given the definition above all these answers are implied by the program. What we had in mind was probably to get one single answer `V = a`. By adding some control information, the program can be rewritten to one that gives only one answer:

```
lookup(Key, [], notfound).
lookup(Key, [(Key,Value)|_], Value) :- !.
lookup(Key, [_|Xs], Value) :- lookup(Key, Xs, Value).
```

The cut (`!`) means that the third clause is never tried if the second can be applied. Since the cut has no logical meaning, the two versions of `lookup` are identical from a logical point of view. However, the second version does not compute all answers implied by the program. Improper use of the cut, like in the present example, complicates reasoning about logic programs and is the source of many errors. A more efficient version can be defined by swapping the order of the two first arguments since it will allow the compiler to generate better code. Prolog also includes loads of impure features like arithmetic, meta predicates, operational I/O, and so on. All in all, there exist very few large-scale Prolog programs which are truly declarative.

It is possible to incorporate notions like monads and list comprehensions into logic programming. For instance, [11] shows how it can be done in the higher-order logic programming language λ-Prolog [38].

There are also languages aimed at overcoming the deficiencies of Prolog by providing a cleaner mechanism for control and meta-programming. One example is the language Gödel [26], a typed language based on a many-sorted logic. Another is Mercury [45], which claims to be a purely declarative logic programming language.

2.1.3 Functional Logic Programming

A functional logic programming language tries to amalgamate functional and logic programming into one language that is more expressive. In [34], it is argued that such a language should serve to bring together the researchers in the respective areas. This should speed up progress towards a truly declarative language and lessen the risk for duplicating research efforts. Unifying functional and logic programming should also make it easier for students to master *declarative* programming since they would only need to learn one language.

The efforts to combine functional and logic programming largely comes from the logic programming community in trying to create more powerful and elegant languages by including functional evaluation and programming techniques in logic programming.

The two most commonly used methods for functional evaluation in languages combining functional and logic programming are narrowing (see below) and residuation [1]. Languages using residuation suspend evaluation of functional expressions until they are sufficiently instantiated to be reduced deterministically. Thus, any mechanism for evaluating functions will do.

Narrowing. The theoretical foundation of languages using narrowing is Horn-clause logic with equality [39], where functions are defined by introducing new clauses for the equality predicate. Narrowing, a combination of unification and rewriting that originally arose in the context of automatic theorem proving [42], is used to solve equations. In a functional language setting this amounts to evaluating functions, possibly instantiating unknown arguments to functions. Narrowing is a sound and complete operational semantics for functional logical languages. Unrestricted narrowing is very expensive however, so much effort has gone into finding efficient versions of narrowing for useful classes of functional logic programs. A survey is given in [23].

On an abstract level, programs in all narrowing languages consist of a number of equational clauses defining functions:

$$LHS = RHS :- C_1, \ldots, C_n \quad n \geq 0$$

where a number of left-hand sides (LHS) with the same principal functor define a function. The C_i's are conditions that must be satisfied for the equality between the LHS and the right-hand side (RHS) to hold. Narrowing can then be used to solve equations by repeatedly *unifying* some subterm in the equation to be solved with a LHS in the program, and then replacing the subterm by the instantiated RHS of the rule. Examples of early languages using narrowing are ALF [20, 21], using a narrowing strategy corresponding to eager evaluation, Babel [37], and K-LEAF [15] using lazy evaluation (narrowing).

As an example of a functional logic program and a narrowing derivation, consider the definition

```
0 + N = N.
s(M) + N = s(M+N).
```

and the equation `X+s(0)=s(s(0))` to be solved. A solution is given by first doing a narrowing step with the second rule replacing `X+s(0)` by `s(Y+s(0))` binding `X` to `s(Y)`. This gives the new equation `s(Y+s(0))=s(s(0))`. A narrowing step with the first rule can then be used to replace the subterm `Y+s(0)` by `s(0)`, thus binding `Y` to `0` and therefore `X` to `s(0)`. Since the equation to solve now is `s(s(0))=s(s(0))` we have found the solution `X=s(0)`.

Curry. The language Curry [24] is a recent proposal for a standardized language in the area of functional logic programming. The intention is to combine the research efforts of researchers in the area of functional logic programming and also, hopefully, researchers in the areas of functional and logic programming respectively.

Curry is primarily an effort to design a language realizing research into efficient evaluation principles (narrowing strategies) for functional logic languages. However, in [24] it is stated that Curry should *combine the best ideas* of existing declarative languages including the functional languages Haskell and Standard ML, the logic languages Gödel and λ-Prolog, and the functional logic languages ALF, Babel, and Escher [33].

The default evaluation principle in Curry is a sophisticated lazy narrowing strategy [5, 22, 35]. The goal of this strategy is to be as deterministic as possible and to perform as few computation steps as possible. The strategy is complete in that it computes all solutions to a goal for a certain set of programs.

A simple example of a Curry program is the following:

```
function append: [A] -> [A] -> [A]
append [] Ys = Ys
append [X|Xs] Ys = [X|append Xs Ys]
```

This looks just like an ordinary definition in a functional language but can also be used to solve equations. For instance, Curry computes the answer `L = [1,2]` to the equation

```
append L [3,4] == [1,2,3,4]
```

Similarly, the ordinary map function can be defined:

```
function map: (A -> B) -> [A] -> [B]
map F [] = []
map F [X|Xs] = [(F X)|map F Xs]
```

Again, `map` can be used in the same way as in a functional programming language. However, if the function `succ`, giving the next natural number, is defined in the program, then the equation

```
map F [1,2] == [3,4]
```

will give the solution `F = succ`. For details of how Curry handles higher order function variables see [31].

Predicates are represented as boolean functions using conditional equations. A simple example of this is following:

```
perm([],[]) = true.
perm([X|Xs],[Y|Ys]) = true <= select(Y,[X|Xs],Z),
                              perm(Z,Ys).
```

The second clause means that the result is true if the condition to the right of the arrow holds (evaluates to true). Some syntactical sugaring is also provided. An alternative definition for `perm` where `select` is defined locally is:

```
perm([],[]).
perm([E|L],[F|M]) <= select(F,[E|L],N),perm(N,M)
   where select(E,[E|L],L)
         select(E,[F|L],[F|M]) <= select(E,L,M).
```

The `where` constructs are explained using implications. An implication in turn is understood by adding clauses to the program in much the same way as in λ-Prolog [38].

The default evaluation principle in Curry is an attempt at declarative programming in the strong sense. However, it is realized that it is not obvious that the lazy narrowing used as default is ideal for all applications. It is therefore possible to give *control information* in the form of *evaluation restrictions*. An evaluation restriction makes it possible to specify all sorts of evaluation strategies between lazy narrowing and residuation. If evaluation restrictions are used it is up to the user to ensure that the program computes all desired answers.

To specify that the function `append` can only be reduced if its first argument is instantiated[1] the restriction

```
eval append 1:rigid
```

is added to the program. If `append` is called with a variable as first argument the call waits, *residuates*, until the variable is bound. As another example, consider the function `leq` on natural numbers:

```
function leq: nat -> nat -> bool
eval leq 1:(s => 2)
leq 0 _ = true
leq (s M) 0 = false
leq (s M) (s N) = leq M N
```

The evaluation restriction means that the first argument should always be evaluated (to head-normal form), but that the second argument should only be evaluated if the first argument has the functor `s` at the top. Note that this might reduce an infinite computation to a finite one if the second argument can be reduced to infinitely many values. An alternative evaluation strategy, describing residuation, is

```
eval leq 1:rigid(s => 2:rigid)
```

It is also possible to specify that a function should be strict by giving the evaluation restriction `nf`(normal form) for the arguments.

Curry also allows the user to give control information by use of *encapsulated search* [25]. This means that the search required due to logical variables and back-tracking, non-deterministic predicates or functions can be encapsulated into a certain part of the program. A library of typical search-operators, including depth-first and breadth-first search, is provided.

2.1.4 Constraint Logic Programming

A constraint logic programming language performs deterministic computations over non-deterministic ones. We will not go into any details here but simply give some examples of programs in the language Life [3]. Life is not a very typical constraint programming language but conveys some interesting ideas which has influenced other languages[2]. An influential language using a concurrent constraint programming model related to constraint logic programming and aiming at several goals similar to those of Life (including object-orientation and functions evaluated using residuation) is Oz [43, 44].

[1]In [24] it is demanded that the argument must not be headed by a defined function symbol, not be a logical variable nor an application of an unknown function to some arguments.

[2]Actually, Life is no longer being developed.

Life Life is formally a constraint programming language using a constraint system based on order-sorted feature structures [2]. One of the aims of Life is to provide a synthesis of three different programming paradigms: logic programming, functional programming and object-oriented programming. Life has many similarities with Prolog but adds functions, approximation structures and inheritance.

Pure logic programming in Life is very similar to programming in Prolog. Predicates are defined using Prolog-compatible syntax—even the cut is included to prune the search space. Thus, the same problems with control mixed with declarative statements as in Prolog are present. However, the presence of functions with deterministic evaluation makes it possible to write cleaner programs.

An interesting feature of Life is that it replaces the (first-order) terms of Prolog with ψ-terms. The ψ-terms are used to represent all data structures, including lists, clauses, functions and sorts. A ψ-term has a basic sort. The partially ordered set of all sorts may be viewed as an *inheritance hierarchy.* A sort may have subsorts and the subsorts of a sort inherit all properties of the parent sort. At the the top of the sort hierarchy we find $\top$ written `@` in Life. At the base of the hierarchy we find $\bot$ written`{}`. There is no conceptual difference between values and sorts. For instance `1` is equal to the set `{1}` which is a subsort of `int`. The sort `int` is a subsort of `real`, written `int <| real`.

The user is free to specify new sorts. The declaration `truck <| vehicle` means that all trucks are vehicles and also that trucks inherit all properties of vehicles. This corresponds in some sense to subclassing in object-oriented languages. Now, if we define

```
mobile(vehicle)
```

```
useful(truck)
```

and ask the query

```
mobile(X),useful(X)?
```

we will get the answer `X = truck`. A ψ-term represents a set of objects. Each object can have attributes. Each attribute consists of a label (feature name) and a ψ-term:

```
car(nbr_of_wheels => 4,
    manufacturer => string,
    max_speed => real).
```

Note that ψ-terms do not have fixed arities. It may even be possible to unify two ψ-terms of the same principal sort but with different arities.

As mentioned, Life supports the use of functions. A function residuates until its arguments are sufficiently instantiated to *match* a clause. An example is:

```
fact(0) -> 1.
fact(N:int) -> N*fact(N-1)
```

Now, if we ask the query

```
A = fact(B)?
```

the system will respond with `A = @, B = @~`. The tilde means that `B` is a residuation variable, that is, further instantiation of `B` might lead to further evaluation of some expression. If we state that `B = 5` the call to `fact` can be computed and we get the answer `A = 120`. To use and define higher-order functions poses no problem since functions are evaluated using matching, that is, no higher order unification is involved.

2.1.5 Definitional Programming

In *definitional programming* a program is simply regarded as a *definition.* This is a more basic and low-level notion than the notion of a function or the notion of a predicate as can be seen from the fact that we talk of function and predicate *definitions* respectively.

In the definitional programming language GCLA [7, 8, 29], programs are regarded as belonging to a special class of definitions, the *partial inductive definitions* (PID) [17, 30]. Apart from being a definitional language there is one feature that sets GCLA apart from the rest of the declarative languages discussed here, namely its approach to control. In GCLA the control (or procedural) part of a program is completely separated from the declarative description. A program consists of two definitions called the (object) definition and the rule (meta) definition, where the rule definition holds the control information. Both the definition and the object definition consist of a number of definitional clauses

$$a \Leftarrow A.$$

where an atom a is defined in terms of a condition A. The most important operation on definitions is the definiens operation, $D(a)$, giving all conditions defining a in D.

The separation of the declarative description and the control information means that there is no need to destroy the declarative part with control information. It also means that one definition can be used together with several rule definitions giving different procedural behavior. Furthermore, typical control information definitions can be reused together with many different object definitions. However, for most programs the programmer *has* to be aware of control issues.

It is important to note that the rule definition giving the control information has a declarative reading as a definition giving a *declarative approach* to control. For more information on the control part see [7, 13, 30].

From a programming point of view, GCLA is essentially a logic programming language sharing backtracking and logical variables with Prolog. It was developed with the aim to find a suitable modelling tool for knowledge based systems and has been tested in several applications [8, 14, 16, 41]. One way that GCLA extends Prolog is that it allows hypothetical reasoning in a natural way. We show an example where we only give the definition. To get an executable program a suitable rule definition has to be supplied as well.

Assume we know that an object can fly if it is a bird and if it is not a penguin. We also know that Tweety and Polly are birds as are all penguins, and that Pengo is a penguin. A definition expressing this information is the following:

```
flies(X) <= bird(X),(penguin(X) -> false).

bird(tweety).
bird(polly).
bird(X) <= penguin(X).

penguin(pengo).
```

To find out which objects *cannot* fly we can pose the query

```
(flies(X) \- false).
```

and the system will respond with `X = pengo`. Note how this binds variables in a negated query.

2.2 Discussion

The implicit vision underlying declarative programming is that programmers should be able to concentrate completely on what a program should do and not need to worry about the low-level details of how to do it. As we have seen, not many declarative systems fulfill this goal. Another vision is that the semantics of programs should be sufficiently clear to enable formal reasoning about programs. A problem in many cases then is that while reasoning about the pure parts of languages may be feasible it is not practical for real applications to be restricted to such a subset.

The aim at declarative programming in the strong sense reflects an *extensional* view of programs; focus is on the functions or predicates defined not on their definitions. Definitional programming takes a quite different approach in that it studies the definitions that constitute programs. Programs are regarded as definitions and focus is on the definitions themselves as the primary objects—a focus that reflects an *intensional* view of programs. In this more low-level approach there is no obvious fixed uniform computational or procedural meaning of all definitions. Instead, different kinds of definitions require different kinds of evaluation procedures. For instance, the *definition* of a predicate is not used in the same way as the *definition* of a function. To describe the intended computational content, or procedural information, another *definition* with a fixed interpretation is used. Thus, control becomes an integrated part of the program with the same status and with a declarative reading just like the definition of the problem to be solved.

Although declarative programming languages have been around for some thirty years by now they have still not gained any widespread use for real-world applications. At the same time object-oriented programming has conquered the world and is the methodology used for most programs developed today. This is a fact even though object-oriented languages like `C++` lack most features considered important in declarative programming, such as clear and simple semantics, automatic memory management, and so on. It should be obvious that the object-oriented approach has something that is intuitively appealing to software developers. Accordingly, many attempts have been made to include object-oriented features into declarative programming and several languages like Prolog and Haskell have object-oriented extensions. Of course, another reason for the popularity of the object-oriented approach is the maturity of the existing systems. To our knowledge there is no declarative programming environment that is even close in terms of development tools and size of libraries of reusable software components. For declarative programming to gain popularity we believe that it is time to start focusing more on building real applications and making it easy to interact with legacy software. This requires clean foreign-language interfaces and better programming environments and software libraries. Unless these are developed there is a definite risk that declarative programming remains an activity for enthusiasts only for the foreseeable future.

3 An Introduction to GCLA

The first part of this dissertation concerns programming in GCLA. To make it comprehensible we give a brief overview of GCLA here.

A program in GCLA consists of two partial inductive definitions which we refer to as the *definition*, or the *object-level definition*, and the *rule definition* or the *meta-level definition* respectively. We will refer to the definition as **D** and to the rule definition as **R**.

Definitional programming as it is realized in GCLA shares several features with most logic programming languages, like logical variables allowing computations with partial information. Computations are performed by posing queries to the system and the variable bindings produced are regarded as the answer to the query. The search for proofs is performed depth-first with backtracking, and clauses of programs are tried by their textual order. We assume some familiarity with basic logic programming concepts, such as the notion of a most general unifier (*mgu*) [32], and also rudimentary knowledge of sequent calculus.

We will not go into any theoretical details of GCLA but concentrate on the aspects relevant for programming. The conceptual basis for GCLA, the theory of partial inductive definitions, is described in [17]. A finitary version is presented, and relations to Horn clause logic programming investigated in [18, 19]. Most of the details of the language as we present it were given in [29]. Several papers describing the implementation and use of GCLA can be found in [8]. Among them, [7] gives a comprehensive introduction to GCLA and describes programming methodology. More on programming methodology can be found in [13]. Finally, [30] contains a wealth of material on different finitary calculi of partial inductive definitions including details of the theoretical basis of GCLA.

3.1 Basic Notions

3.1.1 Atoms, Terms, Constants, and Variables

We start with an infinite signature, Σ, of *term constructors* and a denumerable set, $\mathcal{V}$, of *variables*. We write variables starting with a capital letter. Each term constructor has a specific arity, and there may be two different term constructors with the same name but different arities. The term constructor t of arity n is written t/n. We will leave out the arity when there is no risk of ambiguity. A *constant* is a term constructor of arity 0. *Terms* are built up using variables and constants according to the following:

1. all variables are terms,
2. all constants are terms,
3. if f is a term constructor of arity n and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

An *atom* is a term which is not a variable.

3.1.2 Conditions

Conditions are built from terms and *condition constructors*. The set $\mathcal{CC}$ of condition constructors always includes $true$ and $false$. Conditions can then be defined:

1. $true$ and $false$ are conditions,
2. all terms are conditions,
3. if $p \in \mathcal{CC}$ is a condition constructor of arity n, and $C_1, \ldots, C_n$ are conditions, then $p(C_1, \ldots, C_n)$ is a condition. Condition constructors can be declared to appear in infix position like in $C_1 \to C_2$.

In **R** the set of condition constructors is predefined, while in **D** any symbol can be declared to become a condition constructor.

3.1.3 Clauses

If a is an atom and C is a condition then

$$a \Leftarrow C.$$

is a *definitional clause*, or simply a clause for short. We refer to a as the *head* and to C as the *body* of the clause. We write

$$a.$$

as short for the clause $a \Leftarrow true$. The clause

$$a \Leftarrow false.$$

is equivalent to not defining a at all.

A *guarded definitional clause* has the form

$$a\#\{G_1, \ldots, G_n\} \Leftarrow C.$$

where a is an atom, C a condition, and each G_i is a *guard*. If t_1 and t_2 are terms then $t_1 \neq t_2$ and $t_1 = t_2$ are guards. Guards are used to restrict variables occurring in the heads of guarded definitional clauses.

3.1.4 Definitions

A definition is a finite sequence of (guarded) definitional clauses:

$$\begin{array}{l} a_1 \Leftarrow C_1. \\ \vdots \\ a_n \Leftarrow C_n. \end{array}$$

Note that both **D** and **R** are definitions in the sense described here.

3.1.5 Operations on Definitions

The *domain*, $Dom(D)$, of a definition D, is the set of all instances of heads of clauses in D, that is, $Dom(D) = \{a\sigma \mid \exists A(a \Leftarrow A \in D)\}$.

The *definiens*, $D(a)$, of an atom a is the set of all bodies of clauses in D whose heads matches a, that is $\{A\sigma \mid (b \Leftarrow A) \in D \wedge b\sigma = a\}$. If there are several bodies defining a then they are separated by ';'. A closely related notion is that of *a-sufficiency*. Given an atom a, a substitution σ is called *a-sufficient* if $D(a\sigma)$ is closed under further substitution, that is, for all substitutions τ, $D(a\sigma\tau) = (D(a\sigma))\tau$. Given an a-sufficient substitution the definiens of an atom a is completely determined. There can be more than one definiens of a however since there may be several a-sufficient substitutions. If $a \notin Dom(D)$, then $D(a) = false$. For formal definitions of the definiens operation and a-sufficient substitutions see [19, 29, 30].

3.1.6 Sequents and Queries

A *sequent* is as usual $\Gamma \vdash C$, where, in GCLA, Γ is a (possibly empty) list of *assumptions*, and C is the *conclusion* of the sequent. A *query* has the form

$$S \Vdash (\Gamma \vdash C). \tag{1}$$

where S is a *proof term*, that is, some more or less instantiated condition in **R**. The intended interpretation of (1) is that we ask for an object-level substitution σ such that

$$S\phi \Vdash (\Gamma\sigma \vdash C\sigma).$$

holds for some meta-level substitution ϕ. We will sometimes refer to a query as a *meta sequent.*

3.2 The Definition

In the definition the programmer should state the declarative content of a problem without worrying to much about its procedural behavior. Indeed, a definition **D**, has no procedural interpretation without its associated procedural part **R**. **R** supplies the necessary information to get a program fulfilling the intent behind **D**. The programmer can choose to use the predefined set of condition constructors, or replace or mix them with new ones. This gives a large degree of freedom in the declarative part of programs making it easy to express different kinds of ideas.

The default set of condition constructors include '`,`', '`;`', and '$\to$', which are understood by a calculus given by the standard rule definition. This rule definition implements the calculus $\mathcal{OLD}$ from [29], which in turn is a variant of the calculus $\mathcal{LD}$ given in [19]. We demonstrate the definition with a small example that will be used also in Section 3.5.

Let the size of a list be the number of distinct elements in the list. We can state this fact in the following definition:

```
size([]) <= 0.
size([X|Xs]) <= if(member(X,Xs),
                    size(Xs),
                   (size(Xs) -> Y) -> s(Y)).
```

with the *intended* reading: "the size of the empty list is 0, and the size of `[X|Xs]` is `size(Xs)` *if* `X` is a member of `Xs`, else evaluate `size(Xs)` to `Y` and take as result of the computation the successor of `Y`". Here `if/3` is a condition constructor, we do not give its meaning in the definition but instead interpret it through a special inference rule. To complete the program we also define `member`:

```
member(X,[X|_]).
member(X,[Y|Xs])#{X \= Y} <= member(X,Xs).
```

A few observations: Prolog notation is used for lists, `size` is a function and `member` a predicate, the guard in `member` restricts `X` to be different from `Y` even if both are variables.

3.3 The Rule Definition

In the rule definition we state *inference rules*, *search strategies*, and *provisos* which together give a procedural interpretation of the definition. The rule definition can be seen as forming a sequent calculus giving meaning to the condition constructors in **D**. The condition constructors available in **R** are fixed to ',', ';', $\rightarrow$, $true$, and $false$. Instead of interpreting these by giving yet another definition, the condition constructors in **R** are given meaning in a fixed calculus $\mathcal{DOLD}$ see Section 3.4.

Also available in the rule definition are a number of primitives to handle the communication between **R** and **D**. Some of these are described in Section 3.3.3.

3.3.1 Inference Rules

The interpretation of conditions in **D** is given by inference rules in **R**. Inference rules (or rules for short) are coded as functions from the premises of a rule to its conclusion. The inference rule

$$\frac{P_1, \ldots, P_n}{C}\ rule \quad Proviso$$

is coded by the function

$$rule(P_1, \ldots, P_n) \Leftarrow (Proviso, P_1, \ldots, P_n) \rightarrow C. \tag{2}$$

where P_i and C are object level sequents. We can read the arrow in (2) as "if ...then ...". In actual proof search, derivations are constructed bottom-up so the functions representing rules are evaluated backwards, that is, we look for instantiations of arguments giving a certain result. For more details see [7, 29].

Generally the form of an inference rule is

$$\begin{array}{rcl} rule(A_1, \ldots, A_m, PT_1, \ldots, PT_n) & \Leftarrow & P_1, \ldots, P_k, \\ & & (PT_1 \rightarrow Seq_1), \\ & & \ldots, \\ & & (PT_n \rightarrow Seq_n), \\ & & \rightarrow Seq. \end{array}$$

where

- A_i are arbitrary arguments. One way to use these is demonstrated in Section 3.5.3.
- $PT_1, \ldots, PT_n$ are proof terms, that is, more or less instantiated functional expressions representing the *proofs* of the premises, Seq_i.

- $P_1, \ldots, P_k$, for $k \geq 0$, are provisos, that is, side conditions on the applicability of the rule.
- *Seq* and Seq_i are sequents, $\Gamma \vdash C$, where Γ is a list of (object level) conditions and C is a condition.

One possible reading of *rule* is: "if P_1 to P_k hold and each PT_i proves Seq_i then $rule(A_1, \ldots, A_m, PT_1, \ldots, PT_n)$ proves *Seq*".

3.3.2 Search Strategies

Search strategies are used to combine rules together, guiding search among the rules. The basic building blocks of strategies are rules and provisos and by combining these together, with each other and with other search strategies, we can build more and more complex structures. The general form of a strategy is

$$\begin{array}{lll} strat & \Leftarrow & P_1 \rightarrow Seq_1, \\ & & \ldots, \qquad\qquad n \geq 0 \\ & & P_n \rightarrow Seq_n. \\ strat & \Leftarrow & PT_1, \ldots, PT_m. \end{array}$$

where again P_i are provisos, PT_i are proof terms, and Seq_i are sequents. We can read this as: "if P_i holds, $i \leq n$, and some PT_j, $1 \leq j \leq m$, proves Seq_i then *strat* proves Seq_i". In its simplest form $n = 0$ and the strategy becomes

$$strat \quad \Leftarrow \quad PT_1, \ldots, PT_m.$$

which is best understood as a nondeterministic choice between $PT_1, \ldots, PT_m$.

3.3.3 Provisos

A *proviso* is a side condition on the applicability of a rule or strategy. There are two kinds of provisos—predefined and user defined. We do not go into the user defined provisos here but refer to [7, 10]. Among the predefined provisos there are really three provisos handling the communication between **R** and **D** and various provisos implementing different kinds of simple tests like `var`, `atom`, `number`, etc.

The provisos handling the communication between **R** and **D** are:

- `definiens`(a, Dp, n) which holds if $\mathbf{D}(a\sigma) = Dp$, where σ is an a-sufficient substitution and n is the number of clauses defining a. If $n > 1$ then the different bodies defining a are separated by ';'. If $n = 0$ then $Dp = false$.
- `clause`(b, B) which holds if $(c \Leftarrow C) \in \mathbf{D}$, $\sigma = mgu(b, c)$, and $B = C\sigma$. If b is not defined by any clause then B is bound to $false$.
- `unify`(t, c) which unifies the two object level terms t and c.

3.4 Operational Semantics

To answer a query $S \Vdash (\Gamma \vdash C)$ can be seen as solving an equation. The right-hand side is a partially instantiated object-level sequent representing the answer, and the left-hand side (the proof term) is a partially instantiated functional expression. To answer a query it is necessary to find an instance of the proof term that evaluates to an object-level sequent which is an instance of the right-hand side. The actual search is conducted backwards starting with the query.

The operational semantics for GCLA is given by the calculus $\mathcal{DOLD}$ presented below. A more detailed description can be found in [29]. A goal in the calculus is a triple $\langle Seq_0 \cdots Seq_n, \theta, \xi\rangle$, where the first element is a sequence of queries (meta sequents), the second element is a substitution of meta variables (i.e., variables occurring in the rule definition), and the third element is a substitution of object variables (i.e., variables occurring in the object definition).

Note that the sets of meta variables and object variables are disjoint.

3.4.1 Initial Rules

1. Initial context

$$\frac{}{\langle \emptyset, \quad \{\}, \{\}\rangle}\ IC.$$

   A goal with an empty set of queries is solved.

2. Initial sequent

$$\frac{\langle \Sigma\sigma, \quad \theta, \xi\rangle}{\langle ((A \vdash B) \Vdash (C \vdash D)) \cdot \Sigma, \quad \theta\sigma, \xi\rangle}\ IS$$

   if $(A \vdash B)\sigma = (C \vdash D)$. Note that only meta variables are bound by this rule.

3.4.2 Rules for Constructed Conditions

1. Truth and Falsity

$$\frac{\langle \Sigma, \quad \theta, \xi\rangle}{\langle (\Vdash true) \cdot \Sigma, \quad \theta, \xi\rangle} \Vdash true \qquad \frac{\langle \Sigma, \quad \theta, \xi\rangle}{\langle (false \Vdash false) \cdot \Sigma, \quad \theta, \xi\rangle}\ false \Vdash$$

2. Arrow rules

$$\frac{\langle (A \Vdash C) \cdot (\Vdash B) \cdot \Sigma, \quad \theta, \xi\rangle}{\langle ((A \to B) \Vdash C) \cdot \Sigma, \quad \theta, \xi\rangle} \to\Vdash \qquad \frac{\langle (A \Vdash B) \cdot \Sigma, \quad \theta, \xi\rangle}{\langle (\Vdash (A \to B)) \cdot \Sigma, \quad \theta, \xi\rangle} \Vdash\to$$

   Note the ordering of the goals in the premise of the rule $\to\Vdash$.

3. Sum left

$$\frac{\langle (A \Vdash C) \cdot (B \Vdash C) \cdot \Sigma, \quad \theta, \xi\rangle}{\langle ((A; B) \Vdash C) \cdot \Sigma, \quad \theta, \xi\rangle}\ ;\Vdash$$

4. Sum right

$$\frac{\langle(\Vdash A)\cdot\Sigma,\quad\theta,\xi\rangle}{\langle(\Vdash(A;B))\cdot\Sigma,\quad\theta,\xi\rangle}\Vdash; \qquad \frac{\langle(\Vdash B)\cdot\Sigma,\quad\theta,\xi\rangle}{\langle(\Vdash(A;B))\cdot\Sigma,\quad\theta,\xi\rangle}\Vdash;$$

   The rules are tried from left to right by backtracking.

5. Product left

$$\frac{\langle(A\Vdash C)\cdot\Sigma,\quad\theta,\xi\rangle}{\langle((A,B)\Vdash C)\cdot\Sigma,\quad\theta,\xi\rangle}\,,\Vdash \qquad \frac{\langle(B\Vdash C)\cdot\Sigma,\quad\theta,\xi\rangle}{\langle((A,B)\Vdash C)\cdot\Sigma,\quad\theta,\xi\rangle}\,,\Vdash$$

   The rules are tried from left to right by backtracking.

6. Product right

$$\frac{\langle(\Vdash A)\cdot(\Vdash B)\cdot\Sigma,\quad\theta,\xi\rangle}{\langle(\Vdash(A,B)\cdot\Sigma,\quad\theta,\xi\rangle}\Vdash,$$

   Product right is the dual of Sum left. Note the left to right ordering of the goals in the premise.

3.4.3 Definition Rules

1. Definition Left

$$\frac{\langle(\mathbf{R}(a\sigma)\Vdash C\sigma)\cdot\Sigma\sigma,\quad\theta,\xi\rangle}{\langle(a\Vdash C)\cdot\Sigma,\quad\theta\sigma,\xi\rangle}\,D\Vdash$$

   if σ is an a-sufficient substitution with respect to $\mathbf{R}$. There is one instance of this rule for each a-sufficient substitution σ_i. All instances are tried by backtracking. Note that only meta variables are bound by this rule.

2. Definition Right

$$\frac{\langle(\Vdash C_i\sigma)\cdot\Sigma\sigma,\quad\theta,\xi\rangle}{\langle(\Vdash c)\cdot\Sigma,\quad\theta\sigma,\xi\rangle}\Vdash D$$

   if $(c_i \Leftarrow C_i)\in\mathbf{R}$ and $\sigma = mgu(c,c_i)$. All instances of this rule will be tried by backtracking over the clauses $(c_i \Leftarrow C_i)$ in the order they appear in $\mathbf{R}$. Note that only meta variables are bound by this rule.

3.4.4 Proviso Rules

The following rules are for the predefined provisos used in the communication between **R** and **D**.

1. Unify

$$\frac{\langle\Sigma\rho,\quad\theta,\xi\rangle}{\langle(\Vdash \texttt{unify}(t_1,t_2)\cdot\Sigma,\quad\theta,\xi\rho\rangle}\Vdash Unif$$

   if $\rho = mgu(t_1,t_2)$. This and the following two rules are the only rules where object variables are bound.

2. Clause
$$\frac{\langle \Sigma\rho, \quad \theta, \xi \rangle}{\langle (\Vdash \texttt{clause}(b, B_i)) \cdot \Sigma, \quad \theta, \xi\rho \rangle} \Vdash \mathit{Clause}$$
if $(b_i \Leftarrow B_i) \in \mathbf{D}$ and $\rho = mgu(b, b_i)$. All instances of this rule will be tried by backtracking over the clauses $(b_i \Leftarrow B_i)$ in the order they appear in $\mathbf{D}$.

3. Definiens
$$\frac{\langle \Sigma\rho, \quad \theta, \xi \rangle}{\langle (\Vdash \texttt{definiens}(a, Dp, n)) \cdot \Sigma, \quad \theta, \xi\rho \rangle} \Vdash \mathit{Def}$$
if ρ is an a-sufficient substitutions with respect to $\mathbf{D}$ and $\mathbf{D}(a\rho) = Dp$. There is one instance of this rule for each a-sufficient substition ρ_i. All instances are tried by backtracking.

3.5 Examples

To further illustrate GCLA we give a number of examples. The examples are mostly aimed at showing how to program the rule definition. We show most of the rules of the standard rule definition, both as sequent calculus rules and as rules coded in GCLA.

The rule that gives most of the additional power in GCLA, as compared to Horn clause languages, is the rule of *definitional reflection*, also called *D-left*. Pictured as a sequent calculus rule:

$$\frac{\Gamma\sigma, D(a\sigma) \vdash C\sigma}{\Gamma, a \vdash C \quad \sigma} \textit{ D-left} \quad \sigma \text{ is an } a\text{-sufficient substitution} \quad .$$

That is, if C follows from *everything* that define a then C follows from a. This rule comes as a standard rule in GCLA coded as

```
d_left(A,I,PT) <=
        atom(A),
        definiens(A,Dp,N),
       (PT -> (I@[Dp|R] \- C))
        -> (I@[A|R] \- C).
```

where `@` is an infix append operator. Another standard rule is *a-right*, interpreting '$\rightarrow$' to the right:
$$\frac{\Gamma, A \vdash C}{\Gamma \vdash A \rightarrow C} \textit{ a-right}$$

In GCLA:

```
a_right((A -> C),PT) <=
         (PT -> ([A|G] \- C))
         -> (G \- (A -> C)).
```

We also give a possible rule for the constructor `if` used in Section 3.2. In the given program `if` will be used to the left, so we call the corresponding inference rule *if-left*:

$$\frac{\vdash Pred \quad Then \vdash C}{if(Pred, Then, Else) \vdash C}\ \textit{if-left} \qquad \frac{Pred \vdash false \quad Else \vdash C}{if(Pred, Then, Else) \vdash C}\ \textit{if-left}$$

Note that a predicate is false if it can be used to derive falsity [7, 19, 29]. Coded in GCLA *if-left* becomes:

```
if_left(PT1,PT2,PT3,PT4) <=
        ((PT1 -> ([] \- P)),
         (PT2 -> ([T] \- C))
        ;
         (PT3 -> ([P] \- false)),
         (PT4 -> ([E] \- C)))
        -> ([if(P,T,E)] \- C).
```

3.5.1 Pure Prolog

Pure Prolog programs, that is, Horn clause programs, are a subset of GCLA. All that is needed is to use some of the standard rules acting on the consequent, namely *D-right*, *v-right*, *o-right*, and *true-right*. In sequent calculus style notation these rules are written:

$$\frac{\Gamma\sigma \vdash B\sigma}{\Gamma \vdash c \quad \sigma}\ \textit{D-right} \quad (b \Leftarrow B) \in D \wedge \sigma = mgu(b, c) \qquad \frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash C_1, C_2}\ \textit{v-right}$$

$$\frac{\Gamma \vdash C_i}{\Gamma \vdash C_1; C_2}\ \textit{o-right} \quad i \in \{1, 2\} \qquad \frac{}{\Gamma \vdash true}\ \textit{true-right}$$

The coding in GCLA is straightforward. Note that the conclusion of the rule in each case is the last line in the coded version and also note the *PT*s that are used to give search strategies to guide search for proofs of the premises:

```
d_right(C,PT) <=
    atom(C),
    clause(C,B),
    (PT -> (A \- B))
    -> (A \- C).

v_right((C1,C2),PT1,PT2) <=
    (PT1 -> (A \- C1)),
    (PT2 -> (A \- C2)),
    -> (A \- (C1,C2)).
```

```
o_right((C1;C2),PT1,PT2) <=
    ((PT1 -> (A \- C1));
    (PT2 -> (A \- C2)))
    -> (A \- (C1;C2)).

true_right <= (A \- true).
```

To connect these rules together we can write the strategy:

```
prolog <= d_right(_,prolog),
          v_right(_,prolog,prolog),
          o_right(_,prolog,prolog),
          true_right.
```

A Prolog-style program to compute all permutations of a list is the following:

```
perm([], []).
perm(Xs, [Y|Ys])#{Xs \= []} <=
     delete(Y, Xs, Zs),
     perm(Zs, Ys).

delete(X, [X|Ys], Ys).
delete(X, [Y|Ys], [Y|Zs]) <= delete(X,Ys,Zs).
```

If we use the strategy `prolog` this program will give the same answers as the corresponding Prolog program. To prove the trivial fact that a singleton list is a permutation of itself the following proof is constructed:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \texttt{true}}\,\textit{true-right}}{\vdash \texttt{delete}(\mathsf{a}, [\mathsf{a}], [])}\,\textit{D-right} \qquad \dfrac{\dfrac{}{\vdash \texttt{true}}\,\textit{true-right}}{\vdash \texttt{perm}([], [])}\,\textit{D-right}}{\vdash \texttt{delete}(\mathsf{a}, [\mathsf{a}], []), \texttt{perm}([], [])}\,\textit{v-right}}{\vdash \texttt{perm}([\mathsf{a}], [\mathsf{a}])}\,\textit{D-right}$$

3.5.2 More Standard Rules

The standard rule definition consists of the rules given so far (except *if-left*) plus the six rules:

$$\dfrac{}{\Gamma, false \vdash C}\,\textit{false-left} \qquad \dfrac{\Gamma, A \vdash C}{\Gamma, pi\ X\backslash A \vdash C}\,\textit{pi-left}$$

$$\dfrac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \rightarrow B \vdash C}\,\textit{a-left} \qquad \dfrac{\Gamma, C_1, C_2 \vdash C}{\Gamma, (C_1, C_2) \vdash C}\,\textit{v-left}$$

$$\dfrac{\Gamma, C_1 \vdash C \quad \Gamma, C_2 \vdash C}{\Gamma, (C_1; C_2) \vdash C}\,\textit{o-left} \qquad \dfrac{}{\Gamma, a \vdash c \ \ \sigma}\,\textit{axiom} \quad \sigma = mgu(a, c)$$

To guide search among the standard rules a number of predefined strategies are given. One such strategy, testing the rules in the order axiom, all rules operating on the consequent, all rules operating on the antecedent, is `arl` (for axiom, right, left):

```
arl <= axiom(_,_,_),right(arl),left(arl).

right(PT) <= v_right(_,PT,PT),
             a_right(_,PT),
             o_right(_,PT,PT),
             true_right,
             d_right(_,PT).

left(PT) <= false_left(_),
            v_left(_,_,PT),
            a_left(_,_,PT,PT),
            o_left(_,_,PT,PT),
            d_left(_,_,PT),
            pi_left(_,_,PT).
```

One way to program **R** to a given **D** is to start with the standard rules and some predefined search strategy and then make refinements to these to get exactly the desired procedural behavior. Another possibility is to write a whole new set of rules and strategies.

3.5.3 Yet Another Procedural Part

We conclude by giving a possible rule definition giving the intended procedural behavior for the function `size` defined in Section 3.2. What we wish to do is to evaluate `size` as a function. For instance, the query

```
sizeS \\- size([a,b,c]) \- C.
```

should bind `C` to `s(s(s(0)))` and give no more answers. We use the standard rules `d_left`, `d_right`, `a_left`, `a_right`, `true_right`, and `false_left` plus the new rule `if_left` given in Section 3.5.

We start by writing a strategy, `sizeS`, for the query above assuming that we have a strategy that handles `member` correctly. Since `size` is used to the left, the first step is to apply the rule `d_left`. After that either the axiom rule or `if_left` should be used. This gives us the skeleton strategy:

```
sizeS <= d_left(size(_),_,sizeS),
         axiom(0,_,_),
         if_left(PT1,PT2,PT3,PT4).
```

We have instantiated the first argument in `d_left` and `axiom` to restrict their applicability properly. All that is left is to instantiate the arguments to `if_left`. Looking back at its definition we see that the first and third arguments should be able to prove and disprove `member` respectively. We assume that the strategy `memberS` does this. The second argument should simply be `sizeS` while in the fourth we give a specialized rule sequence to handle the condition correctly:

```
sizeS <= d_left(size(_),_,sizeS),
         axiom(0,_,_),
         if_left(memberS,sizeS,
            memberS,a_left(_,_,a_right(_,sizeS),axiom(s(_),_,_))).
```

The strategy `memberS` simply consists of the rules `d_right`, `d_left`, `true_right`, and `false_left`:

```
memberS <= d_right(member(_,_),memberS),
           true_right,
           d_left(member(_,_),_,memberS),
           false_left(_).
```

Now we can handle the query above as well as the more interesting

```
sizeS \\- (size([a,X,b]) \- L).
```

which first gives `L = s(s(0))`, `X = a`, then `L = s(s(0))`, `X = b`, and finally `L = s(s(s(0)))`, `a \= X`, `X \= b`. In particular note the last answer telling us that the size is `s(s(s(0)))` if `X` is anything else than `a` or `b`.

We also show the derivation built to compute the size of a list of one element:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{false} \vdash \mathtt{false}}\,\textit{false-left}}{\mathtt{member}(0,[]) \vdash \mathtt{false}}\,\textit{D-left} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\{\mathtt{Y} = 0\}}{0 \vdash \mathtt{Y}}\,\textit{axiom}}{\mathtt{size}([]) \vdash \mathtt{Y}}\,\textit{D-left}}{\vdash \mathtt{size}([]) \to \mathtt{Y}}\,\textit{a-right} \qquad \dfrac{\{\mathtt{C} = \mathtt{s}(0)\}}{\mathtt{s}(0) \vdash \mathtt{C}}\,\textit{axiom}}{(\mathtt{size}([]) \to \mathtt{Y}) \to \mathtt{s}(\mathtt{Y}) \vdash \mathtt{C}}\,\textit{a-left}}{\mathtt{if}(\mathtt{member}(0,[]), \mathtt{size}([]), (\mathtt{size}([]) \to \mathtt{Y}) \to \mathtt{s}(\mathtt{Y})) \vdash \mathtt{C}}\,\textit{if-left}}{\mathtt{size}([0]) \vdash \mathtt{C}}\,\textit{D-left}$$

The purpose of the strategy `sizeS` is to ensure that this is the only possible derivation of the goal sequent.

4 Application to Clinical Medicine

To be able to advance declarative programming methodologies, real-world examples are needed that push the limits of the proposed languages and programming techniques. Of course, this applies to definitional programming as well. An interesting review of a number of cases where functional programming has been used "in anger" for real-world tasks can be found in [47].

GCLA was used in a number of applications, the largest probably being a system for planning and management at construction sites [9]. This system was tested on a 40 000 square meter office building in 8 floors described by a database consisting of about 2000 facts.

When new application areas were needed to further develop definitional knowledge representation and programming techniques, attention was drawn to clinical medicine. To try to find ways of describing clinical procedures and model clinical experience appeared to be an interesting project.

Clinical experience is the foundation of all health care systems. However, clinical experience takes a long time to develop. In addition, it is necessary to continously update the gained clinical experience with information from new clinical cases and external sources, e.g., literature and seminars. Furthermore, the complexity of data, including both text and images, challenge the capacity of the human mind to organize information. If the total amount of data cannot be easily overviewed, there is a risk that vital information, which could be used to improve diagnosis and treatment strategies, is lost.

Computerized systems have been introduced to improve access to clinical data. However, the current systems do not generally provide advanced support for qualitative analysis of the collected data. Consequently, many open problems remain to be solved in this area, ranging from clinical procedures via knowledge representation to human-computer interaction and information visualization.

4.1 The MedView Project

In 1995, the Medview project [4], based on a co-operation between computing science and oral medicine, was initiated. The overall goal of the project is to develop models, methods, and tools to support clinicians in their diagnostic work. The project is centered around the question: how can computing technology be used to handle clinical information in everyday work such that clinicians more systematically can learn from their gathered clinical data? That is, how can the chain "formalize-collect-view-analyse-learn" be understood and implemented in the area of clinical medicine.

Essentially, the MedView project can be divided into two sub-problems: knowledge representation and development of applications for gathering and exploring clinical data. The model of clinical information used in MedView is a definitional one. Basic clinical data consists of disease history and status information from examinations. The process of assembling this information is modeled as acts of *defining* a series of descriptive parameters such as disease history (anamnesis), status, diagnosis etc. Thus, the result of gathering information is a knowledge base consisting of a large number of definitions, each representing a particular examination. The knowledge base can also contain additional definitional structures describing general knowledge not related to any particular examination.

An important goal of the project is to support clinicians in their daily work. Already from the start there has been an existing userbase of clinicians working with software developed within the project. It follows that it has been necessary to develop reliable and user-friendly applications that run on ordinary personal computers available in a clinical setting.

4.2 Declarative Programming and MedView

Symbolic computations, knowledge representation, and reasoning are typically areas where declarative programming tools are a natural choice. In fact, GCLA was specifically developed with knowledge-based systems in mind. Thus, using the declarative programming tool GCLA to implement the definitional model of information and reasoning used in the MedView project would appear to be obvious.

On the other hand, it has been equally important to build applications with graphical user interfaces for use in the real world. GCLA is not an appropriate tool for this task. It is also desirable to use tools with support for networking, distributed computing, image processing, database connectivity etc. The natural choice for the mentioned tasks is to use some industrial-strength object-oriented application development framework and environment. Accordingly, one of the components of the MedView project has been to come up with a solution that allows us to naturally have access to the benefits of both worlds. That is, to be able to both express definitional structures and computations in a clean way and to have full access to all the years of effort put into a full-fledged application development environment. The solution adopted in MedView is to use declarative programming for knowledge representation and reasoning and object-oriented programming for application development. To be able to smoothly combine the two we have developed new tools for definitional programming suitable for integration into an object-oriented environment.

5 Overview

This thesis consists of four papers illuminating different aspects of applied declarative programming. The first paper deals with how definitional programming in GCLA can be used for functional logic programming. The second paper describes a framework for definitional programming called Gisela, which among other things can be seen as a successor to GCLA. The third gives an overview of the MedView project where definitional models are used for knowledge representation. The fourth paper finally, reports on how we intend to use the Gisela framework within MedView and some of the work done so far in this direction.

Functional Logic Programming in GCLA

Many researchers have sought to combine the best of functional and logic programming into a combined language giving the benefits of both. That functions and predicates can be integrated in GCLA is nothing new. Most papers on GCLA mention in one way or the other that functions can be defined and executed in a natural way. The most in-depth treatment so far is given in [6]. Here we delve deeper into the subject and try to give a detailed description of what is needed to combine functions and predicates in a definitional setting. The functional logic GCLA programs shown build on programming techniques developed in [13].

All examples are given in GCLA but the general ideas could just as well be applied to build a specialized functional logic programming language. We also compare the definitional approach with other proposals noting that the closest relationship is with languages based on narrowing [23].

Gisela—A Framework for Definitional Programming

While GCLA is a nice realization of definitional programming it has several drawbacks which became significant when new programming techniques were developed and attempts were made to use GCLA for knowledge representation and reasoning in the MedView project.

It was therefore decided that a new definitional programming system should be developed. When we analyzed our needs we realized that what we wanted was not yet another declarative programming language based on a definitional model. Rather, we wanted to create a general framework for definitional programming which would allow us to implement definitional knowledge structures in a flexible way. Also, an important requirement was that the framework should allow us to easily build state-of-the-art desktop and web applications with embedded definitional reasoning components.

The result is an extensive object-oriented application programming framework (an API) for definitional programming. The framework, called Gisela, implements a new model for definitional programming. Since it is an object-oriented framework where certain classes may be subclassed or replaced in a well-defined manner it is open for experimentation with definitional computations. One important design-choice in the computation model is that definitions are described in an abstract manner only. Thus, any object which implements this abstract model (interface) can be used by the definitional computing machinery provided by the framework. The realization of the system as an object-oriented framework also makes it very simple to include it as a component in GUI-based applications developed using industrial-strength software tools.

It is often easier and clearer to describe definitional programs using a syntactic representation than by creating a number of objects. Therefore, the framework also provides parsers for certain classes of definitions to allow for what we might call “traditional” declarative programming.

The definitional computational model underlying Gisela is presented together with the basics of the implementation. It is also shown how to program using both syntactic representations and by using the object-oriented API directly.

An Overview of MedView

Since the MedView project was first initiated data from more than 1500 examinations has been gathered and stored into a knowledge base. A number of applications have been developed both for gathering data at examinations and for viewing and analyzing the contents of the knowledge base in different ways.

The theoretic model used for knowledge representation is definitional, which relates MedView closely to both GCLA and Gisela. Another key property of the project is that it is truly applied—applications developed within the project are used by clinicians in their daily work.

Here we give an overview of the work done in the project so far and discuss future directions.

MedView and Gisela

One of the specific goals during the development of Gisela has been to build a framework suitable for knowledge representation and reasoning in the MedView project. So far, the actual MedView system in clinical use only uses a small subset of Gisela. We are working on using Gisela as the reasoning machinery within MedView in a more uniform manner. Here we describe the general approach we intend to use to build a modified MedView system based on Gisela. Specifically, it is shown how the MedView knowledge base can be represented using Gisela and how to use Gisela for basic searches in the knowledge base. Also, a number of smaller applications developed to test Gisela in the MedView setting are described. Finally as a larger example, a functional-logic text-generator, implemented in Gisela, that can be used to create comprehensible summaries from clinical examination records in the knowledge base, is presented.

References

[1] H. Aït-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[2] H. Aït-Kaci and A. Podelski. Logic programming over order-sorted feature terms. In *Extensions of logic programming, third international workshop, ELP93*, number 660 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1993.

[3] H. Aït-Kaci and A. Podelski. Towards a meaning of Life. *Journal of Logic Programming*, 16:195–234, 1993.

[4] Y. Ali, G. Falkman, L. Hallnäs, M. Jontell, N. Nazari, and O. Torgersson. Medview: Design and adoption of an interactive system for oral medicine. In *Proceedings of Medical Informatics Europe (MIE'00), Hannover, Germany, August 2000*, 2000. To appear.

[5] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, 1994.

[6] M. Aronsson. A definitional approach to the combination of functional and relational programming. Research Report SICS T91:10, Swedish Institute of Computer Science, 1991.

[7] M. Aronsson. Methodology and programming techniques in GCLA II. In *Extensions of logic programming, second international workshop, ELP'91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[8] M. Aronsson. *GCLA, The Design, Use, and Implementation of a Program Development System.* PhD thesis, Stockholm University, Stockholm, Sweden, 1993.

[9] M. Aronsson. Planning the construction of a building. Research Report SICS R93:03, Swedish Institute of Computer Science, 1993.

[10] M. Aronsson. GCLA user's manual. Technical Report SICS T91:21A, Swedish Institute of Computer Science, 1994.

[11] Y. Bekkers and P. Tarau. Logic Programming with Monads and Comprehensions. In *Proceedings of JFPL'95*, Dijon, May 1995.

[12] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, 1996.

[13] G. Falkman and O. Torgersson. Programming methodologies in GCLA. In R. Dyckhoff, editor, *Extensions of logic programming, ELP'93*, number 798 in Lecture Notes in Artificial Intelligence, pages 120–151. Springer-Verlag, 1994.

[14] G. Falkman and J. Warnby. Technical diagnoses of telecommunication equipment: An implementation of a task specific problem solving method (TDFL) using GCLA II. Research Report SICS R93:01, Swedish Institute of Computer Science, 1993.

[15] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42:139–185, 1991.

[16] M. Gustafsson. Texttolkning med abduktion i GCLA. Master's thesis, Department of Linguistics, Göteborg University, 1994.

[17] L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.

[18] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(2):261–283, 1990. Part 1: Clauses as Rules.

[19] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(5):635–660, 1991. Part 2: Programs as Definitions.

[20] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop om Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 387–401. Springer-Verlag, 1990.

[21] M. Hanus. Improving control of logic programs by using functional languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 1–23. Springer-Verlag, 1992.

[22] M. Hanus. Combining lazy narrowing and simplification. In *Proc. 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

[23] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19/20:593–628, 1994.

[24] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[25] M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.

[26] P. Hill and J. Lloyd. *The Gödel Programming Language*. Logic Programming Series. MIT Press, 1994.

[27] P. Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[28] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, July 1979.

[29] P. Kreuger. GCLA II: A definitional approach to control. In *Extensions of logic programming, second international workshop, ELP91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[30] P. Kreuger. *Computational Issues in Calculi of Partial Inductive Definitions.* PhD thesis, Department of Computing Science, University of Göteborg, Göteborg, Sweden, 1995.

[31] H. Kuchen and J. Anastasiadis. Higher order Babel: Language and implementation. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Extensions of Logic Programming 5th International Workshop, ELP'96*, number 1050 in Lecture Notes in Artificial Intelligence, pages 193–208. Springer, 1996.

[32] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second extended edition, 1987.

[33] J. W. Lloyd. Combining functional and logic programming languages. In M. Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium.* MIT Press, 1994.

[34] J. W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 94.

[35] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming,PLIP'93*, number 714 in Lecture Notes in Computer Science, pages 184–200. Springer-Verlag, 1993.

[36] R. Milner. Standard ML core language. Internal report CSR-168-84, University of Edinburgh, 1984.

[37] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[38] G. Nadathur and D. Miller. An overview of λProlog. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827. MIT Press, 1988.

[39] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1988.

[40] C. Runciman and D. Wakeling. Heap profiling of lazy functional programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.

[41] H. Siverbo and O. Torgersson. Perfect harmony—ett musikaliskt expertsystem. Master's thesis, Department of Computing Science, Göteborg University, January 1993. In Swedish.

[42] J. J. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[43] G. Smolka. The definition of kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1994.

[44] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Current Trends in Computer Science*, number 1000 in Lecture Notes in Computer Science, pages 441–454. Springer-Verlag, 1995.

[45] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[46] P. Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM Symposium on Lisp and Functional Programming*, Nice, France, 1990.

[47] P. Wadler. Functional programming: An angry half dozen. *SIGPLAN Notices*, 33(2):25–30, February 1998.

Functional Logic Programming in GCLA

Olof Torgersson*
Department of Computing Science
Chalmers University of Technology and Göteborg University
S-412 96 Göteborg,Sweden
`oloft@cs.chalmers.se`

Abstract

We describe a definitional approach to functional logic programming, based on the theory of Partial Inductive Definitions and the programming language GCLA. It is shown how functional and logic programming are easily integrated in GCLA using the features of the language, that is, combining functions and predicates in programs becomes a matter of programming methodology. We also give a description of a way to automatically generate efficient procedural parts to the described definitions.

1 Introduction

Through the years there have been numerous attempts to combine the two main declarative programming paradigms functional and logic programming into one framework providing the benefits of both. The proposed methods varies from different kinds of translations, embedding one of the methods into the other, [39, 51], to more integrated approaches such as narrowing languages [17, 24, 37, 45] based on Horn clause logic with equality [43], and constraint logic programming as in the language Life [2].

A notion shared between functional and logic programming is that of a *definition*, we say that we *define* functions and predicates. The programming language can then be seen as a formalism especially designed to provide the programmer with as clean and elegant a way as possible to define functions and predicates respectively. Of course these formalisms are not created out of thin air but are explained by an appropriate theory.

*This work was carried out as part of the work in the ESPRIT working group GENTZEN and was funded by The Swedish National Board for Industrial and Technical Development (NUTEK).

In GCLA [7, 27] we take a somewhat different approach, we do talk about definitions but these definitions are not given meaning by mapping them onto some theory about something else, but are instead understood through a theory of *definitions* and their properties, the theory of *Partial Inductive Definitions* (PID) [19]. This theory is designed to express and study properties of definitions, so we look at the problem from a different angle and try to answer the questions: what are the specific properties of function and predicate definitions and how can they be combined and interpreted to give an integrated functional logic computational framework based on PID.

A GCLA program consists of two communicating partial inductive definitions which we call the (*object level*) *definition* and the *rule definition* respectively. The rule definition is used to give meaning to the conditions in the definition and it is also through this the user queries the definition. One could also say that the rule definition gives the procedural reading of the declarative definition.

What we present in this paper is a series of rule definitions to a class of functional logic program definitions. These rule definitions implicitly determine the structure of function, predicate, and integrated functional logic program definitions. We also try to give a description of how to write functional logic GCLA programs.

The work presented in this paper has several different motivations. One is to further develop earlier work on functional logic programming in GCLA [6, 27] to the point where it can be seen that it actually works by giving a more precise formulation of the programming methodology involved and a multitude of examples. Another motivation is to use and develop ideas from [16] on how to construct the procedural part of GCLA programs. The rule definitions given in Sections 3, 5, 6, and 7 can be seen as applications of the methods proposed in [16]. The rule definitions presented in Sections 3 and 5 confirm the claim that it is possible to use the method *Splitting the Condition Universe* to develop the procedural part to certain classes of definitions, in this case functional logic program definitions. The rule definitions developed in Sections 6 and 7 on the other hand show that the claim that it is possible to automatically generate rule definitions according to the method *Local Strategies*, holds for the definitions defining functional logic programs. Yet another motivation is the need to develop a library of rule definitions that can be used in program development.

The rest of this paper is organized as follows. In Section 2 we give some introductory examples and intuitive ideas. In Section 3 we present a minimal rule definition to functional logic program definitions. Section 4 gives an informal description of functional logic program definitions. In Section 5 the calculus of Section 3 is augmented with some inference rules needed in practical programming, and a number of example programs are given. Section 6 presents a method to automatically generate efficient rule definitions. Section 7 shows how the possibility to generate efficient rules opens up a way to write more concise and elegant definitions and Section 8, finally, gives an overview of related work.

2 Introductory Example

The key to using GCLA as a (kind of) first-order functional programming language is the inference rule *D-left*, *definition left*, also called the rule of *definitional reflection* [19, 27], which gives us the opportunity to reason on the basis of assumptions with respect to a given definition. The rule of definitional reflection tells us that if we have an atom a as an assumption and C follows from everything that defines a then C follows from a

$$\frac{\Gamma, A \vdash C \quad (A \in D(a))}{\Gamma, a \vdash C}\, D\vdash,$$

where $D(a) = \{A \mid (a \Leftarrow A) \in D\}$. With this rule at hand functional evaluation becomes a special case of hypothetical reasoning. Asking for the value of the function `f` in `x` is done with the query

```
f(x) \- C.
```

to be read as : “What value `C` can be derived assuming `f(x)`” or perhaps rather as “Evaluate `f(x)` to `C`”.

Functions are defined in a natural way by a number of equations. As a trivial example, we define the identity function with the single equation[1]

```
id(X) <= X.
```

which tells us that the value of `id(X)` is defined to be `X` (or the value of `X`). By using the standard GCLA rules *D-left* and *axiom* we can build the derivation:

$$\cfrac{\cfrac{\cfrac{\{\mathtt{a} = \mathtt{C}\}}{\mathtt{a} \vdash \mathtt{C}}\,\textit{axiom}}{\mathtt{id(a)} \vdash \mathtt{C}}\,\textit{D-left}}{\mathtt{id(id(a))} \vdash \mathtt{C}}\,\textit{D-left}$$

Note that the evaluation of a function is performed by (repeatedly) replacing a functional expression with its definiens until a canonical value is reached, in which case the axiom rule is used to communicate the result. If we use the standard GCLA inference rules the derivation above is not the only possible derivation however. Another possibility is to try the axiom rule first and bind `C` to `id(id(a))`. We see that writing a definition that intuitively defines a function is not enough, we must also have a rule definition that interprets our function definitions according to our intentions.

[1]In [19] = is used instead of the backward arrow to form clauses making the word equation more natural. In this paper however we use the notation of GCLA.

2.1 Defining Addition

In this section we give definitions of addition, using successor arithmetic, in GCLA, ML [36] and Prolog, and also make an informal comparison of the resulting programs and of how they are used to perform computations.

Functional programming languages can be regarded as syntactical sugar for typed lambda-calculus, or as languages especially constructed to define functions. In ML the definition of addition is written

```
datatype nat = zero | s of nat

fun plus(zero,N) = N
  | plus(s(M),N) = s(plus(M,N))
```

to be read as "the value of adding `zero` to `N` is `N`, and the value of adding `s(M)` and `N` is the value of `s(plus(M,N))`".

In logic programming languages like Prolog, functions are defined by introducing an extra result argument which is instantiated to the value of the function. The Prolog version becomes:

```
plus(zero,N,N).
plus(s(M),N,s(K)) :- plus(M,N,K).
```

Since pure Prolog is a subset of GCLA we can do the same thing in GCLA and define addition:

```
plus(zero,N,N).
plus(s(M),N,s(K)) <=  plus(M,N,K).
```

This definition is used by posing queries like

```
\- plus(zero,s(zero),K).
```

that is "what value should `K` have to make `plus(zero,s(zero),K)` hold according to the definition".

The naive way to define addition as a function in GCLA would then be to write a definition that is so to speak a combination of the ML program and the Prolog program:

```
plus(zero,N) <= N.
plus(s(M),N) <= s(plus(M,N)).
```

We could read this as "the value of `plus(zero,N)` is defined to be `N`, and the value of `plus(s(M),N)` is defined to be `s(plus(M,N))`". Unfortunately, this will not give us a function that computes anything useful since we have not defined what the value of `s(plus(M,N))` should be. What we forgot was that in the (strict) functional language ML the type definition and the computation rule together give a way to compute a value from `s(plus(M,N))`. The type definition `s of nat` says that `s` is a constructor function and since the language is strict the argument of `s` is evaluated before a data object is constructed. One way to achieve the same thing in GCLA is to introduce an *object constructing function* `succ` that evaluates its argument to some number `Y` and then constructs the canonical object `s(Y)`. We then get the definition:

```
succ(X) <= (X -> Y) -> s(Y).

plus(zero,N) <= N.
plus(s(M),N) <= succ(plus(M,N)).
```

The first clause could be read as "the value of `succ(X)` is defined to be `s(Y)` if the value of `X` is `Y`". The function `succ` plays much the same role as the constructor function `s` in the functional program, it is used to build data objects. We will sometimes call such functions that are used to build data objects *object functions*. We are now in a position where we can perform addition, as can be seen in the derivation below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{\mathtt{Y} = \mathtt{zero}\}}{\mathtt{zero} \vdash \mathtt{Y}}\,\textit{axiom}}{\mathtt{plus(zero,zero)} \vdash \mathtt{Y}}\,\textit{D-left}}{\vdash \mathtt{plus(zero,zero)} \rightarrow \mathtt{Y}}\,\textit{a-right} \qquad \dfrac{\{\mathtt{X} = \mathtt{s(zero)}\}}{\mathtt{s(Y)} \vdash \mathtt{X}}\,\textit{axiom}}{(\mathtt{plus(zero,zero)} \rightarrow \mathtt{Y}) \rightarrow \mathtt{s(Y)} \vdash \mathtt{X}}\,\textit{a-left}}{\mathtt{succ(plus(zero,zero))} \vdash \mathtt{X}}\,\textit{D-left}}{\mathtt{plus(s(zero),zero)} \vdash \mathtt{X}}\,\textit{D-left}$$

One of the things that normally distinguishes a function definition in a functional programming language from a relational function definition, like the Prolog program above, is that the functional language allows arbitrary expressions as arguments, that is, expressions like `plus(plus(zero,zero),zero)` can be evaluated. In Prolog on the other hand a goal like `plus(plus(zero,zero),zero,K)` will fail. The functional GCLA definition we have given so far comes somewhere in between; it allows arbitrary expressions in the second argument but not in the first. An expression like `plus(plus(zero,zero),zero)` is simply undefined. In a strict functional language like ML this problem is solved by the computation rule, which says that all arguments to functions should be evaluated before the function is called. We see that again we have to have as part of the definition something that in a conventional functional programming language is part of the external computation rule. What we do is to add an extra equation to the definition of addition which is used to force evaluation of the first argument when necessary:

```
plus(zero,N) <= N.
plus(s(M),N) <= succ(plus(M,N)).
plus(Exp,N)#{Exp \= zero, Exp \= s(_)} <= (Exp -> M) -> plus(M,N).
```

Now, the last equation matches arbitrary expressions except `zero` and `s(_)`. It could be read as "the value of `plus(Exp,N)` is defined to be `plus(M,N)` if `Exp` evaluates to `M`". The guard is needed to make the clauses defining addition mutually exclusive.

At last we have a useful definition of addition. If we use the standard rules we are unfortunately still able to derive some undesired results, as can be seen below:

$$\frac{\{\mathtt{X} = \mathtt{plus(s(zero),zero)}\}}{\mathtt{plus(s(zero),zero)} \vdash \mathtt{X}}\ \textit{axiom}$$

$$\cfrac{\cfrac{\cfrac{}{\mathtt{false} \vdash \mathtt{X}}\ \textit{false-left}}{\mathtt{s(zero)} \vdash \mathtt{X}}\ \textit{D-left}}{\mathtt{plus(zero,s(zero))} \vdash \mathtt{X}}\ \textit{D-left}$$

The last derivation succeeds without binding `X`. The problem with the first derivation is that the axiom rule can be applied to an expression we really want to evaluate by applying the rule *D-left*. In the second derivation the problem is the opposite, the rule *D-left* can be applied to a canonical value that cannot be evaluated any further. To prevent situations like these the canonical values must be distinguished as a special class of atoms. We achieve this by giving them circular definitions:

```
zero <= zero.
s(X) <= s(X).
```

Now the canonical values are defined so they cannot be absurd, but on the other hand the definition is empty in content so there is nothing to be gained from using *D-left*. We also restrict our inference rules so that the *axiom* rule only is applicable to such circularly defined atoms, and symmetrically restrict the rule *D-left* to be applicable only to atoms not circularly defined. With these restrictions in the inference rules and with the definition below $plus(m, n) \vdash k$ holds iff $m + n = k$, see [19]. We call atoms with circular definitions *canonical objects*. Note that in the final definition below the first three clauses correspond to the *type definition* in the ML program, while the last three really constitute the *definition* of the addition function. We also use `0` instead of `zero` which we only used since ML does not allow `0` as a constructor.

```
0 <= 0.
s(X) <= s(X).

succ(X) <= (X -> Y) -> s(Y).

plus(0,N) <= N.
plus(s(M),N) <= succ(plus(M,N)).
plus(Exp,N)#{Exp \= 0, Exp \= s(_)} <= (Exp -> M) -> plus(M,N).
```

2.2 Discussion

From the above example one might wonder if it would not be easier to abandon functional programming in GCLA and stick to relational programming in the Prolog vein. Still, we hope to be able to show that it is possible to write reasonably elegant functional logic programs and perhaps most of all that we get a very natural integration of functional and logic programming in the same computational framework. We will also in Section 7 show how the information of the third clause of the definition of `plus` can be moved to the rule definition thus making the definition shorter and more elegant.

3 A Calculus for Functional Logic Programming

In this section we describe a basic rule definition to functional logic program definitions. We present the inference rules as a sequent calculus to enhance readability. In appendix B the same rules are presented using GCLA code.

We call the rule definition consisting of the rules in Section 3.2 *FL* (for functional logic). This rule definition shows implicitly the choices we have made as for what we regard as a valid functional logic program. Whenever we write a query

```
a \- c.
```

without specifying the search strategy we assume that *FL* is used.

3.1 Design Goals

One can imagine more than one way to integrate functions and predicates in GCLA, both concerning syntax and perhaps most of all concerning the expressive power and generality of function and predicate definitions respectively.

We have chosen to work with the common GCLA condition constructors '`,`', '`;`', '`pi`', '`^`', and '`->`', which all have more or less their usual meaning. However, the inference rules are restricted to specialize them to functional logic programs. We have also added a special condition constructor, '`not`', for negation. In delimiting the class of functional logic programs we have tried to keep as much expressive power as possible while still allowing a *useful simple deterministic* rule definition *FL*.

3.1.1 Making it Useful

One of our goals has been to create a rule definition towards a well-defined class of definitions in such a way that it can be part of a library and used without modification when needed. Practically all the standard search strategies of the GCLA system are far too general in the sense that they will give too many derivations for most definitions and queries. *FL*, on the other hand, is not general but does not give nearly the same amount of redundant answers. To make *FL* useful we must then delimit the notion of a functional logic program definition so that the library rule definition can be used directly without modification.

The most typical problem when function evaluation is involved is to decide what is to be regarded as "data" in a derivation. If we wish to solve this in a way that allows one rule definition for all functional logic programs, the information of what atoms are to be treated as canonical values *must be part of the definition.* This has led us to a solution where the canonical values are defined in circular definitions, that is, they are defined to be themselves. We have then altered the definitions of the rules *axiom* and *D-left*, so that they are mutually exclusive; the former so that it can only be applied to atoms with circular definitions and the latter so that it can only be applied to atoms not circularly defined. The same solution is advocated in [28] in the context of interpreting the rule definition of GCLA programs.

3.1.2 Keeping it Simple

Functional logic programming is actually inherent in GCLA, the only problem is to describe it in some manner which makes it easy to understand and learn to program. Indeed, most examples in Sections 2 and 4 can be run with the standard rules at the cost of a number of redundant answers.

In order to keep the rule definition simple we have decided not to try to make it handle general equation solving. How such a rule definition should be designed and what additional restrictions on the definitions that would be needed to make equation solving work generally remain open questions even though some results are given in [50].

Instead, *FL* guarantees that if $F(t_1, \ldots, t_n)$ is ground, F is a deterministic function definition, and F is defined in $(t_1, \ldots, t_n)$ then the query

$$F(t_1, \ldots, t_n) \vdash C.$$

will give exactly one answer, the value of $F(t_1, \ldots, t_n)$. We feel that in most cases it is much more important that a function has a deterministic behavior than that it can be used backwards, since then it could have been defined as a relation in the first place.

As it happens, functions can often be used backwards anyway, as mentioned in [6]. For instance, the query

```
plus(X,s(0)) \- s(s(0)).
```

gives `X` = `s(0)` as the first answer but loops forever if we try to find more answers.

3.1.3 Achieving Determinism

Generally, an important feature of GCLA is the possibility to reason from assumptions, to ask queries like “if we assume `a`, `b` and `q(X)`, does `p(X)` hold”, that is,

```
[a,b,q(X)] \- p(X).
```

This gives very powerful reasoning capabilities but if we want to set up a general, useful and simple calculus to handle functional logic program definitions we get into trouble. To begin with, what should we try to prove first in a query like the one above, that `p(X)` holds, or that

```
q(X) \- p(X).
```

holds, or perhaps that

```
[(d;e),b,q(X)] \- p(X).
```

holds, where `d;e` is the definiens of `a`? There are simply too many choices involved here if we want to describe a general way to write functional logic program definitions that can use a simple library rule definition.

In *FL* we have decided to get rid of this entire problem in the easiest way possible—we do not allow any other kind of hypothetical reasoning than functional evaluation and negation. This makes the design of the rule definition much more simple and also allows us to make *FL* deterministic in the sense that at most one inference rule can be applied to each object level sequent. Of course, we lose a lot of expressive power, but what we have left is not so bad either. Also, if parts of a program need to reason with a number of assumptions we can always use another rule definition for those parts and only use *FL* for the functional logic part of the program.

Determinism is achieved more or less in the same manner as in the calculus $\mathcal{DOLD}$, used to interpret the rule definition of GCLA programs [27]. We make the following restrictions:

- at most one condition is allowed in the antecedent,
- rules that operate on the consequent can only be applied if the antecedent is empty,
- the axiom rule, *D-ax*, can only be applied to atoms with circular definitions,
- if the condition in the antecedent is (C_1, C_2) then C_1 and C_2 are tried from left to right by backtracking,
- if the condition in the consequent is $(C_1; C_2)$ then C_1 and C_2 are tried from left to right by backtracking.

Note that since *FL* is deterministic it does not matter in which order the inference rules are tried. The search strategy could therefore be "try all rules in any order". Other search strategies allowing for different kinds of extensions are discussed in Sections 5, 6 and 7.

3.2 Rules of Inference

The inference rules of *FL* can be naturally divided into two groups, rules relating atoms to a definition and rules for constructed conditions.

3.2.1 Rules Relating Atoms to a Definition

$$\frac{\vdash C\sigma}{\vdash c \ \ \sigma} \textit{ D-right} \quad (b \Leftarrow C) \in D, \sigma = mgu(b,c), C\sigma \neq c\sigma$$

$$\frac{D(a\sigma) \vdash C\sigma}{a \vdash C \ \ \sigma} \textit{ D-left} \quad \sigma \text{ is an } a\text{-sufficient substitution}, a\sigma \neq D(a\sigma)$$

$$\frac{}{a \vdash c \ \ \sigma\tau} \textit{ D-ax} \quad \sigma \text{ is an } a\text{-sufficient substitution}, a\sigma = D(a\sigma), \tau = mgu(a\sigma, c\sigma)$$

The restrictions we put on these definitional rules are such that they are mutually exclusive, a very important feature to minimize the number of possible answers. For a more in-depth description and motivation of these rules, in particular the rule *D-ax*, see [28, 29].

3.2.2 Rules for Constructed Conditions

The rules for constructed conditions are essentially the standard GCLA and PID rules [19, 27] restricted to allow at most one element in the antecedent:

$$\frac{}{\vdash true} \textit{ truth} \qquad \frac{}{false \vdash false} \textit{ falsity}$$

$$\frac{A \vdash B}{\vdash A \to B} \textit{ a-right} \qquad \frac{\vdash A \quad B \vdash C}{A \to B \vdash C} \textit{ a-left}$$

$$\frac{\vdash C_1 \quad \vdash C_2}{\vdash (C_1, C_2)} \textit{ v-right} \qquad \frac{C_i \vdash C}{(C_1, C_2) \vdash C} \textit{ v-left} \quad i \in \{1,2\}$$

$$\frac{\vdash C_i}{\vdash (C_1; C_2)} \textit{ o-right} \quad i \in \{1,2\} \qquad \frac{C_1 \vdash C \quad C_2 \vdash C}{(C_1; C_2) \vdash C} \textit{ o-left}$$

$$\frac{C \vdash false}{\vdash not(C)} \textit{ not-right} \qquad \frac{\vdash A}{not(A) \vdash false} \textit{ not-left}$$

$$\frac{A \vdash C}{(pi\ X \backslash A) \vdash C} \textit{ pi-left} \qquad \frac{\vdash C}{\vdash X\hat{}C} \textit{ sigma-right}$$

3.3 Queries

It should be noted that the restriction to at most one element in the antecedent does not rule out the possibility to ask more complex queries than simply asking for the value of a function or whether a predicate holds or not. As can be seen from the inference-rules, both the left and right-hand side of sequents may be arbitrarily complex as long as we remember that whenever there is something in the left-hand side the right-hand side *must* be a *variable* or a (partially instantiated) *canonical object*.

Assuming that `f`, `g` and `h` are functions, `p` and `q` predicates and `a`, `b` and `c` canonical objects some possible examples are:

```
f(a) \- C.
```

"What is the value of `f` in `a`."

```
\- p(X).
```

"Does `p` hold for some `X`."

```
p(b) -> g(a,b) \- C
```

"What is the value of `g` in `(a,b)` provided that `p(b)` holds."

```
p(X) -> (f(X),h(X)) \- b.
```

"Is there a value of `X` such that `f` or `h` has the value `b` if `p(X)` holds."

```
(p(X);q(X)) -> (f(X);h(X)) \- a.
```

"Find a value of `X` such that `p(X)` *or* `q(X)` holds and both `f` *and* `h` has the value `a` in `X`."

More precisely there are two forms of possible queries, functional queries and predicate or logic queries. The functional query has the form

$$FunExp \vdash C.$$

where *FunExp* is a functional expression as described in Section 6.2 and C is a variable or a (partially instantiated) canonical object. The predicate (logic) query has the form

$$\vdash PredExp.$$

where *PredExp* is a predicate expression as described in Section 6.2.

3.4 Discussion

FL gives the interpretation of definitions we have in mind when we in the next section describe how to write functional logic programs. At the same time it determines the class of definitions and queries that can be regarded as functional logic programs. *FL* can be classified as a rule definition constructed according to the method *Splitting the Condition Universe* described in [16], where we have split the universe of all atoms into atoms with circular definitions and atoms with non-circular definitions. The code given in appendix B is a rule definition to a well-defined class of definitions and queries and does therefore confirm the claim that the this method can be useful for certain classes of programs.

In practical programming it is convenient to augment the rule definition given here with a number of additional rules, for example to do arithmetic efficiently. These additional rules do no affect the basic interpretation given here nor the methodology described in Section 4.

The rule *falsity* deserves some extra comments; it is restricted so that it can only be used in the context of negation, that is, to prove sequents where the consequent is *false*. One motivation for this is that we do not want this rule to interfere with functional evaluation; if a function `f` is undefined for some argument `x` we do not want to be able to construct the proof

$$\dfrac{\dfrac{}{\texttt{false} \vdash \texttt{Y}}\ \textit{false-left}}{\texttt{f(x)} \vdash \texttt{Y}}\ \textit{D-left}$$

but instead want the derivation to fail. On the other hand we do not want to rule out the possibility of negation so we use the presented rule.

FL is very similar to the calculus $\mathcal{DOLD}$ [27] used to interpret the meta-level of a GCLA program (that is, $\mathcal{DOLD}$ is used to interpret the GCLA code of the rules described here). This should not be surprising since the meta-level of a GCLA program is nothing but a non-deterministic functional logic program run backwards. One important similarity with the calculus $\mathcal{DOLD}$ is that our rule definition is deterministic in the sense that there is at most one inference rule that can be applied to each given object level sequent. The most important difference, apart from that we use our rule definition to another kind of programs, is that we use the definition to guide the applicability of the rules *D-left*, *D-ax* and *D-right*, an approach we find very natural in the context of functional logic programming.

3.4.1 A Problem with D-ax

We have chosen to formulate the rule *D-ax* in the same way as in [28]. However, there is one possible case not covered in [28] which must be dealt with to use this rule in programming, namely what we should do if both the condition in the antecedent and the consequent are variables.

One solution is to make the rule only applicable to atoms but this would rule out too many interesting programs and queries. Another solution would be to unify the two variables and instantiate the resulting variable to some atom with a circular definition, but this would really require that we type our programs. The solution we take in the GCLA-formulation given in appendix B uses the fact that a variable is a place-holder for some as yet unknown atom. When both the antecedent and the consequent are variables *D-ax* succeeds unifying the two variables, but with the constraint that when the resulting variable is instantiated it must be instantiated to an atom with a circular definition.

3.4.2 Alternative Approaches

Controlling when the inference rules *D-left* and *axiom* may be applied is definitely the key to successful functional logic programming in GCLA. Consequently, several solutions have been proposed for this problem through the years.

In the first formulation of GCLA [8], there was no rule definition but programs consisted only of the definition which could be annotated with control information. To prevent application of the rules *D-right* and *D-left* to an atom it was possible to declare the atom "total". It was also possible to annotate directly in a definition that only the axiom rule could be applied to certain atoms.

When the rule definition was introduced in GCLA [27], new methods were needed. Basically, it is possible to distinguish two methods to handle control in functional logic programs.

In [6], *axiom* and *D-left* are made mutually exclusive by introducing a special proviso in the *rule* definition which defines what is to be regarded as data in a program. In this approach there is no information in the definition of what is to be regarded as data and we have to write a new rule definition for each program. An interactive system that among other things can be used to semi-automatically create this kind of rule definitions for functional logic programs is described in [42].

Another approach is taken in [16, 27], where the rule sequence needed to evaluate a function or prove a predicate is described by strategies in such detail that there is no need for a general way to choose between different inference rules. Of course, with this approach we have to write new rule definitions for each program. Circular definitions of canonical objects are included in [27] but these are never used in any way. In Section 6 we will show that this kind of highly specialized rule definition may be automatically generated providing an efficient alternative to a general rule definition like *FL*.

Finally, [28] presents more or less exactly the same definition of addition as we did in Section 2.1 as an example of the properties of the rule *D-ax*.

4 Functional Logic Program Definitions

Now that we have given a calculus for functional logic program definitions, it is in order that we also describe the structure of the definitions it can be used to interpret and how to write programs. Note that all readings we give of definitions, conditions and queries are with respect to the given rule definition, *FL*, and that there is nothing intrinsic in the definitions *themselves* that forces this particular interpretation. For example, if we allow contraction, and several elements in the antecedent, function definitions as we describe them cease to make sense since it is then possible to prove things like

```
plus(s(0),s(0)) \- s(s(s(0))).
```

using the definition of addition given in Section 2.1.

As mentioned in the previous section, conditions are built up using the condition constructors '`,`', '`;`', '`->`', '`true`', '`false`', '`not`', '`pi`' and '`^`'. Both functions and predicates are defined using the same condition constructors and there is no easily recognizable syntactic difference between functions and predicates. The difference in interpretation of functions and predicates instead comes from whether they are intended to be used to the left or to the right of the turnstile, '`\-`', in queries.

When we read and write functional logic program definitions the condition constructors have different interpretations depending on whether they occur to the left or to the right, for instance '`;`' is read as *or* to the right and as *and* to the left. Thus, when we write a function or a predicate we must always keep in mind whether the condition we are currently writing will be used to the left or to the right to understand its meaning. The basic principle is that predicates are used to the right (negation excepted), while functions are used to the left.

In order to show that it is not always obvious to see what constitutes function and predicate definitions respectively, we look at a simple example:

```
q(X) <= p(X) -> r(X).
```

If we read this as defining a predicate, we get "`q(X)` holds if the value of `p(X)` is `r(X)`". On the other hand, read as a (conditional) function it becomes "the value of `q(X)` is `r(X)` provided that `p(X)` holds". Of course, if we look at the definition of `p`, we might be able to determine if `q` is intended to be a function or a predicate, since if `p` is a function then `q` is a predicate and vice versa. In the following sections we look more closely at definitions of canonical values, predicates and functions respectively.

4.1 Defining Canonical Objects and Canonical Values

Each atom that should be regarded as a canonical object in a definition, in the sense that it could possibly be the result of some function call, must be given a circular definition. From an operational point of view this is essential to prevent further application of the rule *D-left* and allow application of the rule *D-axiom*. These circular definitions also set canonical objects apart as belonging to a special class of terms since they cannot be proven true or false in *FL*.

Compared to functional languages, these circular or total objects correspond to what are usually called constructors, but with one important difference, constructor functions may be strict and take an active part in the computation, whereas our canonical objects are passive and simply define objects.

Generally, the definition of the canonical object S of arity n is

$$S(X_1, \ldots, X_n) \Leftarrow S(X_1, \ldots, X_n).$$

where each X_i is a variable.

Note that we make a distinction between a *canonical object* and a *canonical value*. Any atom with a circular definition is a canonical object, while a canonical value is a canonical object where each subpart is a canonical value (a canonical object of arity zero is also a canonical value).

We have already in Section 2 seen definitions of the canonical objects `0` and `s`, some more examples are given below:

```
[] <= [].
[X|Xs] <= [X|Xs].

'True' <= 'True'.
'False' <= 'False'.

empty_tree <= empty_tree.
node(Element,Left,Right) <= node(Element,Left,Right).
```

Note that a definition like

```
[_|_] <= [_|_].
```

is not circular since the variables in definiens and definiendum are different.

4.2 Defining Predicates

Since pure Prolog is a subset of GCLA, defining predicates is very much the same thing in this context as in Prolog even though the theoretical foundation is different. Given a pure Prolog program, the corresponding GCLA definition is obtained by substituting '`<=`' for '`:-`' in clauses. SLD-resolution, [33], corresponds to that we only use the rules *D-right*, *truth*, *v-right*, and *o-right* to construct proofs of queries. Of course, in such queries the antecedent is always empty. The relationship between SLD-resolution and the more proof-theoretical approach taken in GCLA is thoroughly investigated in [20].

However, the predicate definitions allowed by *FL* also provide two extensions of pure Prolog in predicates that go beyond the power of SLD-resolution: use of functions in conditions defining predicates and constructive negation.

4.2.1 Calling Functions In Predicates

As an example of how functions can be used in the definitions of predicates we define a predicate `add` such that $\vdash add(m, n, k)$ whenever $m + n = k$. Using the function `plus` already defined in Section 2.1 the definition becomes one single clause:

```
add(N,M,K) <= plus(N,M) -> K.
```

we read this as "`add(N,M,K)` is defined to be equal to the value of `plus(N,M)`". We show an example derivation below. Note that the predicate `add` is used to the right, that is, "does `add(0,s(0),K)` hold", while `plus` is evaluated to the left, that is, "what is the value of `plus(0,s(0))`" or "what follows from `plus(0,s(0))`".

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathtt{K} = \mathtt{s(0)}\}}{\mathtt{s(0)} \vdash \mathtt{K}}\;D\text{-}ax}{\mathtt{plus(0,s(0))} \vdash \mathtt{K}}\;D\text{-}left}{\vdash \mathtt{plus(0,s(0))} \to \mathtt{K}}\;a\text{-}right}{\vdash \mathtt{add(0,s(0),K)}}\;D\text{-}right$$

4.2.2 Negation

The usual way to achieve negation in logic programming is negation as failure [33]. In GCLA however, we have the possibility to derive falsity from a condition which gives us a natural notion of negation. This kind of negation and its relation to negation as failure is described in [21], here we will simply give a very informal description and discuss its usefulness and limitations in practical programming.

We say that a condition C is true with respect to the definition at hand if $\vdash C$, and false if $C \vdash false$, that is if C can be used to derive falsity. It is now possible to achieve negation by posing the query

$$\vdash C \to false.$$

or equivalently

$$\vdash not(C).$$

The constructor `not`, and the inference rules *not-right* and *not-left* are really superfluous; we have included them as syntactic sugar and to reserve the arrow constructor for usage in functional expressions only.

In order to understand how and why we are able to derive falsity, there are two consequences of the interpretation of a GCLA definition as a PID that must be kept in mind, namely:

1. if we take a definition like

   ```
   nat(0).
   nat(s(X)) <= nat(X).
   ```

   it should properly be read as "`0` is a natural number, `s(X)` is natural number if `X` is and *nothing else is a natural number*",

2. as a consequence of the above property, we have that for any atom a that is not defined in a definition D, $D(a) = false$.

Remembering this we can derive that `s(?)` is not a natural number as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\texttt{false} \vdash \texttt{false}}\,\textit{falsity}}{\texttt{nat}(?) \vdash \texttt{false}}\,\textit{D-left}}{\texttt{nat(s}(?)) \vdash \texttt{false}}\,\textit{D-left}}{\vdash \texttt{not(nat(s}(?)))}\,\textit{not-right}$$

Unfortunately it is not possible to use this kind of constructive negation for all the predicates we are able to define. Apart from the fact that the search space sometimes becomes impracticably large, there are two restrictions we must adhere to if we want to be able to negate a defined atom correctly. First of all, since we use the rule *D-left*, all clauses $a \Leftarrow C$ defining a must fulfill the no-extra-variable condition, that is, all variables occurring in C must also occur in a. If this condition does not hold the generated substitutions in the definiens operation will not be a-sufficient [21, 27, 29] and we may be able to derive things which do not make sense logically. Secondly, it is not possible to use functions in conjunction with negation, that is, no clause $a \Leftarrow C$ defining a may contain a function call of the form $C_1 \rightarrow C_2$.

To see why functions in predicates and negation cannot be used together consider the definition of `add` and the failed derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\text{this goal fails,}}{\vdash \texttt{plus}(0,0)} \quad \cfrac{\text{as does this}}{\texttt{s}(0) \vdash \texttt{false}}}{\texttt{plus}(0,0) \rightarrow \texttt{s}(0) \vdash \texttt{false}}\,\textit{a-left}}{\texttt{add}(0,0,\texttt{s}(0)) \vdash \texttt{false}}\,\textit{D-left}}{\vdash \texttt{not}(\texttt{add}(0,0,\texttt{s}(0)))}\,\textit{not-right}$$

We see that we end up trying to prove that `plus(0,0)` holds which of course is nonsense. It is easy to see that using *FL* the query $\vdash F$ fails for every function F.

As described in [7] we are also able to instantiate variables, both positively and negatively in negative queries. These possibilities are best illustrated with a small example:

```
p(1).
p(2) <= q(2).

:- complete(append/3).

append([],Xs,Xs).
append([X|Xs],Ys,[X|Zs]) <= append(Xs,Ys,Zs).
```

In this definition we have declared `append` to be complete [7], which means that the definition of `append` is completed to make it possible to find bindings of variables such that `append(X,Y,Z)` is not defined. We may now ask for some value of `X` such that `p(X)` does not hold:

```
\- not(p(X)).

X = 2 ?;
```

Another possible query is

```
\- not(append([1,2],L,[1,2,3,4])).

append([],L,[3,4]) \= append([],_A,_A) ? ;
```

which tells us that `L` must not be the list `[3,4]`.

4.3 Defining functions

A function definition, defining the function F, consists of a number of equational clauses

$$\begin{array}{ll} F(t_1,\ldots,t_n) \Leftarrow C_1. & \\ \vdots & n \geq 0, m > 0 \\ F(t_1,\ldots,t_n) \Leftarrow C_m. & \end{array}$$

where each right-hand side condition, C_i, is on one of the following forms

- universally quantified, $(pi\ X \backslash C)$,
- atomic, that is C_i is a variable, a canonical value or a function call,
- conditional, $(C_1 \rightarrow C_2)$,
- choice, (C_1, C_2),
- split, $(C_1; C_2)$

each of which are described below.

If the heads of two or more clauses defining a function are overlapping all the corresponding bodies must evaluate to the same value, since the definiens operation used in *D-left* collects all clauses defining an atom. For example consider the function definition:

```
f(0) <= 0.
f(N) <= s(0).
```

Even though `f(0)` is defined to be `0` the derivation of the query

```
f(0) \- C.
```

will fail, as seen below, since `f(0)` is also defined to be `s(0)`. The definition of `f` is *ambiguous.*

$$\dfrac{\dfrac{\dfrac{\{\texttt{C} = \texttt{0}\}}{\texttt{0} \vdash \texttt{C}}\,\textit{D-ax} \qquad \dfrac{\text{fails since } \texttt{C} = \texttt{0}}{\texttt{s(0)} \vdash \texttt{C}}}{\texttt{0}; \texttt{s(0)} \vdash \texttt{C}}\,\textit{o-left}}{\texttt{f(0)} \vdash \texttt{C}}\,\textit{D-left}$$

We will mainly describe how to write function definitions with non-overlapping heads. The methodology involved if we use overlapping heads is only slightly touched upon in Section 4.3.5.

4.3.1 Universally Quantified Condition

In predicates, variables not occurring in the head of a clause are thought of as being existentially quantified. In function definitions, however, they should normally be seen as being universally quantified. Universal quantification is expressed with the construct $pi\ X\backslash C$. As an example the function `succ` of Section 2.1 should be written as:

```
succ(X) <= pi Y \ ((X -> Y) -> s(Y)).
```

Since evaluation of universal quantification on the left-hand side of sequents is carried out by simply removing the quantifier, we will often omit it in our function definitions to avoid clutter. All variables in an equation not occurring in the head should then be understood as if they were universally quantified.

4.3.2 Atomic Condition

With an atomic condition we simply define the value of a function to be the value of the defining atomic condition. Using the definition of addition given in Section 2.1 a function definition consisting of a single atomic equation is:

```
double(X) <= plus(X,X).
```

Note that atomic conditions with circular definitions are the only possible end-points in functional evaluation since the derivation of a functional query, $F \vdash C$, always starts (ends) with an application of the inference rule *D-axiom*.

A useful special kind of function definition, defined with a single atomic equation, is constant functions or defined constants:

```
max <= s(s(0)).

double_max <= plus(max,max).

add_max(X) <= plus(X,max).
```

A drawback of constants like `double_max` is that they, contrary to in functional languages, will be evaluated every time their value is needed in a derivation.

4.3.3 Conditional Condition

A conditional clause:

$$F \Leftarrow C_1 \rightarrow C_2.$$

states that the value of F is C_2 provided that the condition C_1 holds, otherwise the value of F is *undefined*. The schematic derivation below shows how this kind of equation may be used to incorporate predicates into functions; C_1 is proved to the right of the turnstile, that is, in exactly the same manner as if we posed the query $\vdash C_1$, "does C_1 hold". Of course, C_1 and C_2 may be arbitrarily complex conditions in which case the meaning of different parts depend on whether they end up on the left or right hand side of sequents.

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash C_1} \quad \dfrac{\vdots}{C_2 \vdash C}}{C_1 \rightarrow C_2 \vdash C}\,\textit{a-left}}{F \vdash C}\,\textit{D-left}$$

We have already seen some examples of conditional equations, which have all been of the form

$$F \Leftarrow (C_1 \rightarrow C_2) \rightarrow C_3.$$

(the definition of `succ` for instance). A clause like this could be read as "the value of F is C_3 provided that C_1 evaluates to C_2". Note that the two arrows have different meanings since the first will be proved to the right and the second used to the left of the turnstile '$\vdash$' .

As another example we write a function `odd_double`. This function returns the double value of its argument and is defined on odd numbers only. When we define it we use the definition of `double` from Section 4.3.2. We define `odd` as a predicate using the auxiliary predicate `even`:

```
odd_double(X) <= odd(X) -> double(X).

odd(s(X)) <= even(X).

even(0).
even(s(X)) <= odd(X).
```

Since `s(s(0))` is an even number the query

```
odd_double(s(s(0))) \- C.
```

fails, while

```
odd_double(s(0)) \- C.
```

binds `C` to `s(s(0))`.

If we try to compute the value of `odd_double(plus(s(0),0)))`, the derivation will fail since `odd` is only defined for canonical values. To avoid this we can write a slightly more complicated conditional equation in the definition of `odd_double`:

```
odd_double(X) <=
    (X -> X1),
    odd(X1)
    -> double(X1).
```

The condition `(X -> X1)` corresponds closely to a let expression in a functional language, it gives a name to an expression and ensures (in the strict case, see below) that it will only be evaluated once.

4.3.4 Choice Condition

Remembering the rule *v-left*, we see that to derive C from (C_1, C_2) it is enough to derive C from either C_1 or C_2. Accordingly, an equation like

$$F \Leftarrow C_1, C_2.$$

means that the value of F is that of C_1 *or* C_2. The alternatives are tried from left to right and if backtracking occurs C_2 will be tried even if C can be derived from C_1. This gives us a possibility to write non-deterministic functions. For instance, we can define a function `nats` to enumerate all natural numbers on backtracking with the equations:

```
0 <= 0.
s(X) <= s(X).

nats <= nats_from(0).

nats_from(X) <= X,nats_from(s(X)).
```

Of course, if C_1 and C_2 are mutually exclusive the choice condition will be deterministic.

If we combine a conditional condition with a choice condition we get a kind of multiple choice mechanism, something like a case expression in a functional language. The typical structure of a functional equation containing such a condition is:

$$F \Leftarrow (C \rightarrow Val) \rightarrow ((P_1 \rightarrow E_1), \ldots, (P_n \rightarrow E_n)).$$

The meaning of this is that the value of F is E_i if C evaluates to Val and P_i holds. Note that P_i will be proved on the right-hand side of '$\vdash$', and that if P_i holds for more than one i this kind of condition will also be non-deterministic. As an example we write the boolean function `and`. Since the alternatives in the choice condition are mutually exclusive this function is deterministic.

```
'True' <= 'True'.
'False' <= 'False'.

X=X.

and(X,Y) <=
   (X -> Z) ->
   ((Z = 'True' -> Y),
    (not(Z = 'True') -> 'False')).
```

We also give a derivation of the query

```
and('False','True') \- C.
```

since it shows most of the rules involved in functional evaluation plus negation in action.

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathtt{Z} = \mathtt{False}\}}{\mathtt{False} \vdash \mathtt{Z}}\;\textit{D-ax}}{\vdash \mathtt{False} \rightarrow \mathtt{Z}}\;\textit{a-right} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{false} \vdash \mathtt{false}}\;\textit{falsity}}{\mathtt{False} = \mathtt{True} \vdash \mathtt{false}}\;\textit{D-left}}{\vdash \mathtt{not(True = False)}}\;\textit{not-right} \quad \dfrac{\{\mathtt{C} = \mathtt{False}\}}{\mathtt{False} \vdash \mathtt{C}}\;\textit{D-ax}}{\mathtt{not(True = False)} \rightarrow \mathtt{False} \vdash \mathtt{C}}\;\textit{a-left}}{\mathtt{(Z = True \rightarrow True), (not(Z = True) \rightarrow False)} \vdash \mathtt{C}}\;\textit{v-left}}{\mathtt{(False \rightarrow Z) \rightarrow ((Z = True \rightarrow True), (not(Z = True) \rightarrow False))} \vdash \mathtt{C}}\;\textit{a-left}}{\mathtt{and(False, True)} \vdash \mathtt{C}}\;\textit{D-left}$$

4.3.5 Split Condition

The main subject of this paper is to describe how GCLA allows what we might call traditional functional logic programming. When we include the condition constructor '`;`' to the left and also allow function definitions with overlapping heads it is possible to use the power of the definiens operation to write another kind of function definition. Since this is not really the subject of this paper we just give a couple of examples as an aside, more material may be found in [14, 15].

Recall from the rule *o-left*, that to show that C follows from $(A_1; A_2)$ we must show that C follows from both A_1 and A_2. An alternative definition of `odd_double` using a split condition is:

```
odd_double(X) <= (odd(X) -> Y);double(X).
```

Using the power of the definiens operation we can write the equivalent definition:

```
odd_double(X) <= odd(X) -> Y.
odd_double(X) <= double(X).
```

Since the two heads of this definition are unifiable the definiens of `odd_double(X)` consists of both clauses separated (in GCLA) by the constructor '`;`'.

As a last example consider:

```
odd_double(X) <= odd_double(X,_).

odd_double(X,Xval) <= (X -> Xval) -> Y.
odd_double(_,Xval) <= odd(Xval) -> Y.
odd_double(_,Xval) <= double(Xval).
```

Here we use an auxiliary function `odd_double/2` which might need some explanation. The first clause evaluates `X` to a canonical value `Xval`, and since the three heads are unified by the definiens operation this value is communicated to the second and third clauses, thus avoiding repeated evaluations. The second clause checks the side condition that the argument is an odd number, and finally the third computes the result.

4.4 Laziness and Strictness

We will now proceed to describe how by writing different kinds of function definitions we are able to define both lazy and strict functions.

It should be noted that our notions of strictness and laziness differ both from their usual meaning in the functional programming community and from their meanings in earlier work on functional GCLA programming [5, 6, 27].

In functional programming terminology a function is said to be strict (or eager) if its arguments are evaluated before the function is called. With the logical interface *FL* however, arguments to functions are never evaluated unless the function definition contains some clause that explicitly forces evaluation. A property of the strict function definitions we present, shared with strict functional languages, is that whenever an expression is evaluated it will be reduced to a canonical value with no remaining inner expressions. That a functional language is lazy on the other hand means that arguments to functions are evaluated only if needed and then *only once*. Also evaluation stops when the value of an expression is a data object (canonical object) even if the parts of the object are not evaluated. While the lazy function definitions we present share the property that evaluation stops whenever a canonical object is reached, the rule definition we use cannot avoid repeated evaluation of expressions.

In earlier work on functional programming in GCLA it is not the definition but the rule definition that determines if lazy or eager evaluation is used. Lazy evaluation then means that an expression is evaluated as little a possible before an expression is returned and backtracking is used as the engine to evaluate expressions further.

Given the definition of addition in Section 2.1, and with lazy evaluation, the query

```
plus(s(0),0)\- C.
```

would then produce the answers:

```
C = plus(s(0),0);
C = succ(plus(0,0));
C = s(plus(0,0));
C = s(0);
no.
```

The eager evaluation strategy gives the same answers in the opposite order.

Although this might be interesting we do not feel that it is very useful for a deterministic function like `plus` since all the answers represent exactly the same value.

The definition of addition in Section 2.1 is an example of a strict function definition. Using the calculus *FL* the query

```
plus(s(0),0) \- C.
```

will give the single answer `C = s(0)`. It is actually a property of *FL* together with the function definitions we present that, in both the lazy and strict case, we will get only one answer unless the function itself is non-deterministic. In short, given a definition of a deterministic function F, there is only one possible derivation of $F \vdash C$.

4.4.1 Defining Strict Functions

By a strict function definition we mean a definition which ensures that the value returned from the function is a fully evaluated canonical value, that is, a canonical object with no inner expressions remaining to be evaluated. We must therefore find a systematic way to write function definitions that guarantees that only fully evaluated expressions are returned from functions.

The solution to this problem is to be found in the method we use to build data objects in definitions. Recall the ML type definition

```
datatype nat = zero | s of nat
```

of Section 2.1, and that we discussed that it gives information on *what* objects are constructed from and (together with the computation rule) on *how* objects are constructed. We then made the same information explicit in our definition by putting the corresponding information in three clauses, two defining of *what* natural numbers are constructed and one defining *how* a natural number is built from an expression representing a natural number. These clauses provide us with the necessary type definition needed to work with successor arithmetic in strict function definitions.

```
0 <= 0.
s(X) <= s(X).

succ(X) <= (X -> Y) -> s(Y).
```

Now, the idea is that whenever we need to build a natural number in a function definition, we always use `succ` to ensure that anything built up using `s` is truly a canonical value (although not necessarily a natural number).

A general methodology is given by generalization: to each canonical object S defined by a clause:

$$S(X_1, \ldots, X_n) \Leftarrow S(X_1, \ldots, X_n).$$

we create an object function F to be used whenever we want to build an object of type S in a definition. F has the definition:

$$F(X_1, \ldots, X_n) \Leftarrow ((X_1 \to Y_1), \ldots, (X_n \to Y_n)) \to S(Y_1, \ldots, Y_n).$$

In function definitions S is used when we define functions by pattern-matching while F is used to build data objects. We call the definition of the canonical objects of a data type together with their object functions an *implicit type definition.*

Lists is one of the most common data types. In GCLA, Prolog syntax is used to represent lists. If we call the object function associated with lists `cons`, the implicit type definition needed to use lists in strict function definitions is:

```
[] <= [].
[X|Xs] <= [X|Xs].

cons(X,Xs) <= (X -> Y),(Xs -> Ys) -> [Y|Ys].
```

The elements of lists must also be defined as canonical objects somewhere, but it does not matter what they are.

We can now write a strict functional definition of `append`:

```
append([],Ys) <= Ys.
append([X|Xs],Ys) <= cons(X,append(Xs,Ys)).
append(E,Ys)#{E \= [], E \= [_|_]} <= (E -> Xs) -> append(Xs,Ys).
```

This append function can of course be used with any kind of list elements. If we use append together with our definition of addition we can ask queries like

```
append(append(cons(plus(s(0),0),[]),[]),[s(s(0))]) \- C.
```

which gives `C = [s(0),s(s(0))]` as the only answer.

As another example we define the function `take` which returns the first n elements of a list. If there are fewer than n elements in the list the value of `take` is undefined. In the second clause below we take the argument `L` apart by evaluating it to `[X|Xs]`, note how we use this form to inspect the head and tail of a list and that we use `cons` when we put elements together to form a list.

```
take(0,_) <= [].
take(s(N),L) <=
    (L -> [X|Xs]) ->  cons(X,take(N,Xs)).
take(E,L)#{E \= 0, E \= s(_)} <= (E -> N) -> take(N,Xs).
```

4.4.2 Defining Lazy Functions

By a lazy function definition, we mean a definition where arguments to functions are only evaluated if necessary and where evaluation stops whenever we reach a canonical object, regardless of whether all its subparts are evaluated or not. Lazy function definitions make it possible to compute with infinite data structures like streams of natural numbers in a natural way.

In strict function definitions, object functions were used to ensure that all parts of canonical objects were fully evaluated. It is obvious that if the object functions do not force evaluation of subparts of canonical objects we will get lazy evaluation in the above sense.

If we simply replace the definition of `cons` by the clause:

```
cons(X,Xs) <= [X|Xs].
```

neither `X` nor `Xs` will be evaluated and the unique answer to the query

```
append([0],[0]) \- C.
```

will be `C = [0|append([],[0])]`.

Of course, if the object function serves no other purpose than to replace one structure by another one, identical up to the name of the principal functor, it can just as well be omitted which is what we will do in our lazy function definitions. To illustrate the idea we show the lazy versions of `plus` and `append`. The definitions are identical to the strict ones except for that we do not use `cons` and `succ` when we build data objects.

```
0 <= 0.
s(X) <= s(X).

plus(0,N) <= N.
plus(s(M),N) <= s(plus(M,N)).
plus(E,N)#{E \= 0,E \= s(_)} <= (E -> M) -> plus(M,N).

[] <= [].
[X|Xs] <= [X|Xs].

append([],Ys) <= Ys.
append([X|Xs],Ys) <= [X|append(Xs,Ys)].
append(E,Ys)#{E \= [], E \= [_|_]} <= (E -> Xs) -> append(Xs,Ys).
```

The only problem with these definitions is that the results we get are generally not fully evaluated values but we need something more to force evaluation at times. Before we address that problem however, we will give one example of how infinite objects are handled.

Assume that we wish to compute the first element of the list consisting of the five first elements of the infinite list `[a,b,a,b,a,b,a,...]`. How should this be done as a functional program in GCLA? We start by defining the canonical objects of our application which are the constants `a` and `b`, the natural numbers and lists. This is done as usual with circular definitions:

```
a <= a.
b <= b.

0 <= 0.
s(X) <= s(X).

[] <= [].
[X|Xs] <= [X|Xs].
```

The next thing to do is to define the infinite list. We call this list `ab`, it is defined as the list starting with an `a` followed by the list `ba`, which in turn is defined as the list starting with a `b` followed by the list `ab`.

```
ab <= [a|ba].

ba <= [b|ab].
```

To conclude the program, we need the function `hd`, which returns the first element of a list and the function `take`, which returns the first n elements. Note that it is important that we use a lazy version of `take` here and not the one in Section 4.4.1.

```
hd([X|_]) <= X.
hd(E)#{E \= [], E \= [_|_]} <= (E -> Xs) -> hd(Xs).

take(0,_) <= [].
take(s(N),L) <= (L -> [X|Xs]) -> [X|take(N,Xs)].
take(E,L)#{E \= 0, E \= s(_)} <= (E -> N) -> take(N,L).
```

Now we can pose the query

```
hd(take(s(s(s(s(s(0))))),ab)) \-C.
```

which solves our problem binding `C` to `a`. The unique derivation, in *FL*, producing this value is shown below:

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{
            \dfrac{\{\mathtt{Y}=\mathtt{a},\mathtt{Ys}=\mathtt{ba}\}}{[\mathtt{a}|\mathtt{ba}]\vdash[\mathtt{Y}|\mathtt{Ys}]}\;\textit{D-ax}
          }{\mathtt{ab}\vdash[\mathtt{Y}|\mathtt{Ys}]}\;\textit{D-left}
        }{\vdash\mathtt{ab}\rightarrow[\mathtt{Y}|\mathtt{Ys}]}\;\textit{a-right}
        \qquad
        \dfrac{\{\mathtt{Xs}=[\mathtt{a}|\mathtt{take}(4,\mathtt{ba})]\}}{[\mathtt{Y}|\mathtt{take}(4,\mathtt{Ys})]\ \vdash\mathtt{Xs}}\;\textit{D-ax}
      }{(\mathtt{ab}\rightarrow[\mathtt{Y}|\mathtt{Ys}])\rightarrow[\mathtt{Y}|\mathtt{take}(4,\mathtt{Ys})]\vdash\mathtt{Xs}}\;\textit{a-left}
    }{\mathtt{take}(5,\mathtt{ab})\vdash\mathtt{Xs}}\;\textit{D-left}
  }{\vdash\mathtt{take}(5,\mathtt{ab})\rightarrow\mathtt{Xs}}\;\textit{a-right}
  \qquad
  \dfrac{\dfrac{\{\mathtt{C}=\mathtt{a}\}}{\mathtt{a}\vdash\mathtt{C}}\;\textit{D-ax}}{\mathtt{hd}(\mathtt{Xs})\vdash\mathtt{C}}\;\textit{D-left}
}{
  \dfrac{(\mathtt{take}(5,\mathtt{ab})\rightarrow\mathtt{Xs})\rightarrow\mathtt{hd}(\mathtt{Xs})\vdash\mathtt{C}}{\mathtt{hd}(\mathtt{take}(5,\mathtt{ab}))\vdash\mathtt{C}}\;\textit{D-left}
}\;\textit{a-left}
$$

Forcing Evaluation. The technique we use to force evaluation of expressions is similar to the approach taken in lazy functional languages where a printing mechanism is used as the engine to evaluate expressions [44]. One important difference should be noted, if the result of a computation is infinite we will not get any answer at all while in a functional language the result is printed while it is computed which may give some information. We have chosen a method which does not present the answer until it is fully computed since backtracking generally makes it impossible to do otherwise—we cannot know that it will not change before the computation is completed. In each particular case it may of course be possible to write a definition that presents results as they are computed.

To force evaluation we write an auxiliary function `show` whose only purpose is to make sure that the answer presented to the user is fully evaluated. To get a fully evaluated answer to the query

$$F \vdash C.$$

we instead pose the query

$$\mathit{show}(F) \vdash C.$$

thus forcing evaluation of F.

The show function for a definition involving natural numbers and lists is

```
show(0) <= 0.
show(s(X)) <= (show(X) -> Y) -> s(Y).
show([]) <= [].
show([X|Xs]) <=
   (show(X) -> Y),
   (show(Xs) -> Ys)
   -> ([Y|Ys]).
show(Exp)#{Exp\=0,Exp\=s(_),Exp\=[],Exp\=[_|_]}<=
    (Exp -> Val) -> show(Val).
```

In general, for a definition defining the canonical objects $S_1, \ldots, S_n$ the definition of the show function consists of:

- for each canonical object S of arity 0 a clause

$$\mathit{show}(S) \Leftarrow S.$$

- for each canonical object $S(X_1, \ldots, X_n)$ of arity $n > 0$, a clause

$$\begin{aligned}\mathit{show}(S(X_1, \ldots, X_n)) \Leftarrow\ & ((\mathit{show}(X_1) \rightarrow Y_1),\\ & \ldots,\\ & (\mathit{show}(X_n) \rightarrow Y_n))\\ & \rightarrow S(Y_1, \ldots, Y_n).\end{aligned}$$

- a clause to evaluate anything that is not a canonical object. This clause becomes

$$\mathit{show}(E)\#\{E \neq S_1, \ldots, E \neq Sn\} \Leftarrow (E \rightarrow V) \rightarrow \mathit{show}(V).$$

Writing the show function could be done automatically given a definition. It is also possible, as discussed in Section 7, to lift it to become part of the rule definition instead. To move the information of the show function to the rule definition is well-motivated if we regard it as containing purely procedural information. The reason why we keep it as part of the definition is that we wish to describe how one rule definition may be used to any functional logic program definition adhering to certain conventions.

Pattern matching problems. We have already seen that it is more complicated to define functions by pattern matching in GCLA than in a functional language. The basic reason for this is that in GCLA we have much less hidden information used to explain a function definition. A positive effect of this is that the function definition in GCLA mirrors in greater detail the computational content of a function, that is, what computations are necessary to produce a value.

When we write lazy function definitions, of the kind we have described here, pattern matching becomes even more restricted. Not only must the clauses defining a function be unique and arguments evaluated "manually" when necessary, but each individual pattern is severely restricted. In a lazy function definition the only possible patterns, except for variables, are canonical objects where each argument is a variable, that is, if $S(t_1, \ldots, t_n)$ is a pattern used in the head of a function definition, then $S(t_1, \ldots, t_n)$ must have a circular definition and each t_i must be a variable. In [31] these are called shallow patterns. Finally, patterns must be linear, that is no variable may occur more than once in the head of a clause.

The reason for these restrictions is obvious, unless we use a show function nothing is evaluated inside a canonical object. Of course, it is possible to write more general patterns like

```
rev(rev(L)) <= L.

f([[],[a,b]]) <= a.
```

but they will probably not work as we expect in a program.

One simple example will suffice to demonstrate the problem, a definition of the function `last`. A definition of `last` inspired by functional programming is:

```
last([X]) <= X.
last([X|Xs])#{Xs \= []} <= last(Xs).
last(Exp)#{Exp \= [], Exp \= [_|_]}<= (Exp -> List) -> last(List).
```

The problem with this definition is that we cannot tell whether a list contains one or more than one element. The query

```
last([a|append([],[])]) \- C.
```

will fail since the second clause of `last` will be matched even though the expression `append([],[])` evaluates to `[]`. Instead we must write something like:

```
last([X|Xs]) <= (Xs -> Ys) -> ((null(Ys) -> X),
                               (not(null(Ys)) -> last(Ys)).
last(E)#{E \= [],E \= [_|_]} <= (E -> L) -> last(L).

null([]).
```

It should be noted that in a lazy functional language a definition like

```
last([X]) = X.
last([X,Y|Ys]) = last([Y|Ys])
```

is really taken as syntactical sugar for another definition consisting of a number of case expressions which in turn are transformed a number of times yielding something like the following, where we can see the computations needed better:

```
last(L) = case L of
            [] => error
            [X|Xs] => case Xs of
                        [] => X
                        [Y|Ys] => last(Xs).
```

4.4.3 Examples

We conclude our discussion of how to write programs with two more examples, one that uses guarded clauses and negation and one that combines lazy evaluation with non-determinism and backtracking giving an elegant way to code generate and test algorithms.

Let the size of a list be the number of distinct elements in the list. One definition, adopted from [4, 31], that states this fact is:

```
size([]) <= 0.
size([X|Xs]) <= (member(X,Xs) -> size(Xs)),
                (not(member(X,Xs)) -> succ(size(Xs))).
size(E)#{E \= [], E \= [_|_]} <= (E -> Xs) -> size(Xs).

member(X,[X|_]).
member(X,[Y|Ys])#{X \= Y} <= member(X,Ys).
```

In this example we assume that strict evaluation is used. To use the definition given so far we also need some implicit type definitions, we only show the one for pairs since the ones for lists and numbers were given in section 4.4.1:

```
pair(X,Y) <= pair(X,Y).
mk_pair(X,Y) <= (X -> X1),(Y -> Y1) -> pair(X1,Y1).
```

A simple query using this definition is

```
size(cons(mk_pair(0,succ(size([]))),cons(pair(0,s(0)),[]))) \- S.
```

which gives the single answer `S = s(0)`. More interesting is to leave some variables in the list uninstantiated and see if different instantiations of these give different sizes:

```
size([pair(X,0),pair(Y,Z)]) \- S.
```

Here we first get the answer `S = s(0)`, `Y = X`, `Z = 0` and then on backtracking `S = s(s(0))`, `pair(X,0) \= pair(Y,Z)`, that is, the list has size 2 if the variables in the antecedent are instantiated so that the two pairs are not equal.

Finally, in an example adopted from [41], we consider the following problem: Given a number k and a set S of positive numbers, generate all subsets of S such that the sum of the elements is equal to k. It is obvious that if a subset Q of S has been generated whose sum is greater than k, then all sets that can be created by adding elements to Q can be discarded. This idea can be coded in the following definition where `subset` is a lazy function which ensures that we only create as much as is needed to determine if a subset is a feasible candidate:

```
sum_of_subsets(S,K) <= sum_eq(subset(S),0,K).

sum_eq([],Acc,K) <= (Acc = K) -> [].
sum_eq([X|Xs],Acc,K) <= (X+Acc -> N),
                        (N =< K)
                         -> [X|sum_eq(Xs,N,K)].
sum_eq(E,Acc,K)#{E \= [], E \= [_|_]} <=
   (E -> Xs) -> sum_eq(Xs,Acc,K).

subset([]) <= [].
subset([X|Xs]) <= [X|subset(Xs)],subset(Xs).
subset(E)#{E \= [], E \= [_|_]} <= (E -> Xs) -> subset(Xs).
```

We do not show the implicit type definitions and show function needed to compute a result. Note that if the first argument to `sum_eq` is a list but `Acc` is greater than `K` then `sum_eq` fails thus forcing evaluation of a new subset.

4.5 Discussion

As we have seen, the condition constructors '`->`' and '`,`' may be used in ways that produce expressions similar to functional let and case expressions. A natural question then is why we have not integrated them as condition constructors in our function definitions, that is, why not write

```
f(X) <= let(Y = g(X) in h(Y,Y)).
```

instead of:

```
f(X) <= (g(X) -> Y) -> h(Y,Y).
```

The answer to this question is that we have chosen to stick as close as possible to the general GCLA rules and the underlying theory of PID. Working with the ordinary condition constructors both shows what can be expressed using only the basic constructors and what restrictions are necessary in the general inference rules to achieve a working logical interface towards functional logic program definitions.

A problem with the function definitions we have presented is that they may loop forever if they are used with incorrect arguments. For example if `gcla` is defined as a canonical object we can go on forever trying to evaluate `plus(gcla,0)`.

5 Extending FL

While *FL* describes the basic properties of our functional logic program definitions well, it is not quite as well-suited for practical programming. There are several reasons for this, for example successor arithmetic is not particularly efficient, there is no way to write something to a file etc. In this section we discuss *FLplus* which is *FL* augmented with a number of inference rules.

5.1 Arithmetics

In *FLplus* numbers are represented with the construct `n(Number)`, where `n/1` has a circular definition:

```
n(X) <= n(X).
```

To see why we need to have the extra functor around numbers consider the following definition of the factorial function:

```
fac(n(0)) <= n(1).
fac(N)#{N = n(_),N \= n(0)} <= N*fac(N-n(1)).
fac(Exp)#{Exp \= n(_)} <= (Exp -> Num) -> fac(Num).
```

Without the functor `n`, denoting numbers, we cannot distinguish between a number and an expression that needs to be evaluated. In Section 7 we will discuss how we can generate a rule definition which allows us to drop evaluation clauses like the last clause of `fac`. We will then not need the extra functor around numbers.

5.1.1 Evaluation Rules

The rules for evaluating arithmetical expressions are almost identical to the rules used in [7], the only difference is that we, as usual, only allow one element in the antecedent of object level sequents.

We allow the usual operations on numbers. The rule for the operator *Op* is

$$\frac{X \vdash n(X_1) \quad Y \vdash n(Y_1) \quad n(Z) \vdash C}{X \; Op \; Y \vdash C} \quad Z = X_1 \; Op \; Y_1$$

The operation X_1 *Op* Y_1 is a proviso, a side condition which must hold for the rule to hold. The GCLA code of a rule for addition is

```
add_left((X+Y),PT1,PT2,PT3) <=
   (PT1 -> ([X] \- n(X1))),
   (PT2 -> ([Y] \- n(Y1))),
   add(X1,Y1,Z),
   (PT3 -> ([n(Z)] \- C))
   -> ([X+Y] \- C).
```

How the proviso `add` is defined and executed is not really important as long as `Z` is the sum of `X1` and `Y1`.

5.1.2 Comparison Rules

We also have a number of rules to compare numbers. These operate on conditions occurring to the right since we regard them as efficient versions of a number of predicates comparing numbers. We use the same operators as in Prolog, thus `=:=` is used to test equality of two numerical expressions. The rule for the operator *Op* is:

$$\frac{X \vdash n(X_1) \quad Y \vdash n(Y_1)}{\vdash X \mathit{Op}\, Y} \quad X_1 \mathit{Op}\, Y_1$$

The corresponding GCLA code for the operation less than is:

```
lt_right((X < Y),PT1,PT2) <=
   (PT1 -> ([X] \- n(X1))),
   (PT2 -> ([Y] \- n(Y1))),
   less_than(X1,Y1)
   -> ([] \- (X < Y)).
```

The comparison rules can only be used to test if a relation between two expressions holds, not to prove that it does not hold, we cannot prove a query like:

```
\- not(n(4) < n(3)).
```

Also, both the evaluation rules and the comparison rules will work correctly only if the arithmetical operators are applied to ground expressions.

5.2 Rules Using the Backward Arrow

Sometimes we wish to cut possible branches of the derivation tree depending on whether some condition is fulfilled or not. To do this we use the backward arrow '`<-`' described in [7]. The backward arrow is a kind of meta-level if-then-else. It has the operational semantics:

$$\frac{\mathit{If} \vdash \mathit{Seq} \quad \vdash \mathit{Then}}{((\mathit{If} \leftarrow \mathit{Then}), \mathit{Else}) \vdash \mathit{Seq}} \qquad \frac{\mathit{Else} \vdash \mathit{Seq}}{((\mathit{If} \leftarrow \mathit{Then}), \mathit{Else}) \vdash \mathit{Seq}} \quad \mathit{If} \nvdash \mathit{Seq}$$

Using this backward arrow we can introduce two very useful (but not so pure) conditions; *if-expressions* in functions and *negation as failure* in predicates.

5.2.1 If Conditions

If-expressions are commonly used in functional programming. A kind of if-expression can be defined and executed using *FL* as well:

```
if(Pred,Then,Else) <= (Pred -> Then),(not(Pred) -> Else).
```

If if/3 is used to the left we can read it as "if Pred holds then the value of if(Pred,Then,Else) is Then, else if not(Pred) holds then the value is Else". This is very nice and logical but not so efficient. In practice we are often satisfied with concluding that the value of if(Pred,Then,Else) is Else if we cannot prove that Pred holds, without trying to prove the falsity of Pred. To implement this behavior we introduce if/3 as a condition constructor and handle it with a special inference rule, *if-left*:

$$\frac{\vdash P \quad T \vdash C}{if(P,T,E) \vdash C} \qquad \frac{E \vdash C}{if(P,T,E) \vdash C} \ \not\vdash P$$

Note that $\vdash P$ may succeed any number of times. We leave out the somewhat complicated encoding of this rule, it can be found in Appendix C.

5.2.2 Negation as Failure

We have already seen that we can derive falsity in GCLA. However, just as we in if conditions do not care to prove falsity, in many applications we do not care to prove a condition false in predicates either. We therefore introduce the condition constructor '\+'. The condition \+ c is true if we cannot prove that c holds. The inference rule handling this is called *naf-right* (negation as failure right) and is coded using the backward arrow:

```
naf_right((\+ C),PT) <=
   (((PT -> ([] \- C)) -> ([] \- (\+ C)) <- false),
   ([] \- (\+ C)).
```

This rule is of course not sound. It is very important that a possible proof of C is included in the set of proofs represented by PT or we can prove practically anything.

5.3 A Rule for Everything Else

Sometimes it is necessary to communicate with the underlying computer (operating) system. We take a very rudimentary and pragmatically oriented approach to doing this. We introduce a new condition constructor system/1 and a corresponding inference rule *system-right*. Communication with the underlying computer system is then handled by giving the command to be executed as an argument to system/1, which so to speak lifts it up to the rule level. Since in the current implementation the underlying system is Prolog, only valid Prolog commands are allowed.

The definition of the rule *system-right* simply states that $\vdash system(C)$ holds if the proviso C can be executed successfully:

```
system_right(system(C)) <=
    C -> ([] \- system(C)).
```

It is now very easy to define primitives for I/O for instance. The definition below allows us to open and close files and to do formatted output.

```
open(File,Mode,Stream) <= system(open(File,Mode,Stream)).
close(Stream) <= system(close(Stream)).

format(T,Args) <= system(format(T,Args)).
format(File,T,Args) <= system(format(File,T,Args)).
```

5.4 Yet Another Example

A classical algorithm to generate all prime numbers is the sieve of Eratosthenes. The algorithm is: start with the list of all natural numbers from 2, then cross out all numbers divisible by 2, since they can not be primes, take the next number still in the list (3) and cross out all numbers divisible by this number since they cannot be primes either, take the next number still in the list (5) and remove all numbers divisible by this, and so on.

To implement this algorithm we use a combination of lazy functions, predicates, and some of the additional condition constructors of *FLplus*. First of all we define what the canonical objects of the application are, in this case numbers and lists:

```
n(X) <= n(X).

[] <= [].
[X|Xs] <= [X|Xs].
```

Next we define the prime numbers, `primes` is defined as the result of applying the function `sieve` to the list of all numbers from 2:

```
primes <= sieve(from(n(2))).
```

The function `sieve` is the heart of the algorithm. Given a list `[P|Ps]`, where we know that `P` is prime, the result of `sieve` is the list beginning with this prime followed by the list where we have applied `sieve` to the result of removing all multiples of `P` from `Ps`.

```
sieve([P|Ps]) <= [P|sieve(filter(P,Ps))].
sieve(E)#{E \= [], E \= [_|_]} <= (E -> Xs) -> sieve(Xs).
```

All that remains is to define the functions `from` and `filter`. Note that we evaluate the argument of `from`, this is done to avoid repeated evaluations which would otherwise be the case:

```
filter(_,[]) <= [].
filter(N,[X|Xs]) <= if(divides(N,X),
                       filter(N,Xs),
                       [X|filter(N,Xs)]).
filter(N,E)#{E \= [],E \= [_|_]} <= (E -> Xs) -> filter(N,Xs).

divides(A,B) <= (B mod A) =:= n(0).

from(M) <= (M -> N) -> [N|from(N+n(1))].
```

Of course, one problem remains. Since we use lazy evaluation of functions the answer to the query

```
primes \-C.
```

is `[n(2)|sieve(filter(n(2),from(n(2)+n(1))))]`. We cannot use a show function since the result is infinite. The solution is to define a predicate `print_list` used to print the elements of the infinite list of primes:

```
print_list(E) <= (E -> [n(X)|Xs]),
                 print(X),
                 print_list(Xs).

print(X) <= system((format('~ w',[X]),ttyflush)).
```

Now the query

```
\- print_list(primes).
```

will print `2 3 5 7 11` and so on until the system runs out of memory.

5.5 Discussion

We have presented some useful extensions of *FL*. Other extensions are possible as well, for instance we could introduce an apply function with a corresponding inference rule enabling us to write functions like:

```
map(_,[]) <= [].
map(F,[X|Xs]) <= [apply(F,X)|map(F,Xs)].
map(F,E)#{E \= [], E \= [_|_]} <= (E -> Xs) -> map(F,Xs).
```

The function `sieve` given in Section 5.4 is not particularly efficient and consumes a considerable amount of memory. In fact, if run long enough, execution will be aborted due to lack of space.

The reason behind the inefficiency of `sieve` (and other programs we present) is that while the current GCLA system is designed to be as general as possible, allowing for many different styles of programming, it cannot detect the limited kind of control needed in functional logic programming and optimize the code accordingly. This does not mean that it is impossible in principle to efficiently execute the kind of programs we describe in this paper. We believe that with a specialized compilation scheme for functional logic program definitions the programs we describe could be executed just as fast as programs written in existing functional logic programming languages, that is, at a speed approaching that of functional or logic programming languages [24].

6 Generating Specialized Rule Definitions

In Sections 3 and 5 we noticed that both *FL* and *FLplus* were deterministic thus making search strategies more or less superfluous; the answers will be the same no matter in which order the inference rules are tried.

However, there are several reasons to use search strategies anyway. The most important concerns integration of functional logic programs into large systems using other programming paradigms as well; without a proper search strategy all the rules and strategies concerning the rest of the system will also be tested. Another reason is to enhance efficiency, the strategies we present in this section will almost always try the correct rule immediately.

In [16], the method *Local Strategies* was suggested as a way to write efficient rule definitions to a given definition. It was also suggested that it should be possible to more or less automatically generate rule definitions according to this method for some programs. In this section we will show how such rule definitions may be generated for functional logic program definitions.

Note that the rule generation process take it for granted that the clauses in function definitions are mutually exclusive. We put this demand on function definitions in Section 4, but with the difference that the rule definitions *FL* and *FLplus* could deal with overlapping function definitions. Also, we do not attempt to generate specialized strategies to handle negation through the constructor `not/1` since we feel that this is a typical case which requires ingenuity, not mechanization. Instead, the general rule definition *FLplus* is used.

6.1 A First Example

The method *Local Strategies* described in [16] states that each defined atom should have a corresponding procedural part, specialized to handle the particular atom in the desired way. In a functional logic program definition this means that each function, predicate, and canonical object have a specialized procedural interpretation given by the rule definition.

We illustrate with a small example, consisting of two predicate definitions, defining naive reverse:

```
rev([],[]).
rev([X|Xs],Zs) <= append(Ys,[X],Zs),rev(Xs,Ys).

append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) <= append(Xs,Ys,Zs).
```

To this definition we need two search strategies, one for each defined predicate, which we call `rev_strat` and `append_strat`. Since `rev` is a predicate it will always be used to the right (remember that we do not consider negation). The first (and only) rule to try to prove a query

```
\- rev(X,Y).
```

is *D-right*, therefore the definition of `rev_strat` becomes:

```
rev_strat <= d_right(rev(_,_),rev_cont(_)).
```

`rev_cont` (`cont` for continuation) is the strategy to be used to construct the rest of the proof.

If we look at the definition of `rev`, we see that when we have applied `d_right` there are two possible cases. Either the body is `true` and we are finished, or it is `append(Zs,[X],Ys),rev(Xs,Ys)` and we have to continue building a proof. It would be nice if we could check this and choose the correct continuation depending on what we have got to prove. To achieve this we define `rev_cont`:

```
rev_cont(C) <= ([] \- C).
rev_cont(C) <= rev_next(C).
```

This kind of strategy is standard GCLA programming methodology and is described in [6]. The effect of `rev_cont` is that we get the argument of `rev_next` bound to the consequent we are trying to prove thus making it possible to use pattern-matching to choose the correct rule to continue with:

```
rev_next(true) <= truth.
rev_next((append(_,_,_),rev(_,_)) <=
    v_right(_,append_strat,rev_strat).
```

When we define `rev_next` we use the knowledge that `append` is a predicate and thus has a corresponding strategy `append_strat`.

When we wrote the strategy `rev_strat` we did not depend on any sophisticated knowledge that could not be put in a program. Really, the only knowledge used was:

- `append` and `rev` are predicates,
- predicates are used to the right,

- each predicate has a corresponding strategy,
- *FL* is deterministic and therefore for each occurrence of a condition constructor there is only one possible rule to apply.

In programs combining functions and predicates we also need, among other things, to distinguish between predicates and functions.

If we analyze the code of `rev_strat` we see that if we unfold the call to `d_right` we can write more efficient and compact code. Also, we do not need to check at run time that we do not use `d_right` on canonical objects. Furthermore, the strategy `rev_cont` is not needed. With these changes the rule definition in our example becomes:

```
rev_d_right <=
  functor(C,rev,2),   % just to make sure...
  clause(C,B),
  (rev_next(B) -> ([] \- B))
  -> ([] \- C).

rev_next(true) <= truth.
rev_next((append(_,_,_),rev(_,_)) <=
  v_right(_,append_d_right,rev_d_right).

append_d_right <=
  functor(C,append,3),
  clause(C,B),
  (append_next(B) -> ([] \- B))
  -> ([] \- C).

append_next(true) <= truth.
append_next(append(_,_,_)) <= append_d_right.
```

We have changed the name from `rev_strat` to `rev_d_right` to signify that what we get is really a specialized version of the inference rule *D-right*, which may only be used to prove the predicate `rev`.

6.2 Algorithm

The rule generation process consists of three phases which we will describe one by one.

In the first phase the functional logic program definition is analyzed to separate the defined atoms into three classes: canonical objects, defined functions, and defined predicates. The second phase creates an abstract representation of a specialized rule definition for each function and predicate. The third phase finally takes this abstract representation and creates a rule file. We could of course merge phase two and three into one and write rules and strategies to a file as soon as they have been created. We have chosen to separate them to make it easier to experiment with different kinds of output from the rule generator.

Our prototype implementation is implemented as a special kind of rule definition. This rule definition is specialized to handle the single query

```
rulemaster \\- (\- makerules(DefinitionFile,RuleFile)).
```

where `DefinitionFile` is the name of the definition we wish to generate rules for and `RuleFile` is the name of the generated file. The interesting thing is that it is a rule definition to reason *about* definitions, not to interpret definitions to perform computations.

6.2.1 Splitting the Definition

To find the canonical objects of a definition is not difficult, we simply collect all clauses with identical head and body. Distinguishing function definitions from predicate definitions is not quite as easy, in fact it is so hard that in the rule definitions of GCLA programs (which as pointed out in Section 3 are a kind of functional logic programs) predicates are syntactically distinguished by using '`:-`' instead of '`<=`' . This is done to ensure that it is *always* possible to tell functions (rules and strategies) and predicates (provisos) apart.

When we automatically generate rule definitions, for functional logic programs, we are satisfied if it is almost always possible to decide what constitutes a predicate definition and what constitutes a function definition. In case the rule-generator can not make up its mind we let an oracle (the user) decide.

Before we start separating functions from predicates we collect the names and arities of all functions and predicates into a list, $[N_1/A_1, \ldots, N_n/A_n]$. For example if the definition is

```
a <= a.

p(X) <= q(X,X).

q(a,a).
```

this list becomes `[p/1,q/2]`.

The goal of the separation process is to split this list into one list of functions and one list of predicates. To do this we traverse the list of names and arities and use the following heuristics to decide if the atom N of arity A is a function or a predicate; N/A is a function if:

- N/A is already known to be a function,
- the body of some clause defining N/A is an atom which is either a canonical object or a function,

- the body of some clause defining N/A is a constructed condition which is a functional expression as described below.

If we could decide that N/A is a function we store this fact. Note that as a side effect we might find other functions (and predicates) as well. If N/A could not be shown to be a function we try to show that it is a predicate. N/A is a predicate if:

- N/A is already known to be a predicate,
- the body of some clause defining N/A is the condition `true` or an atom which is a predicate,
- the body of some clause defining N/A is a constructed condition which is a predicate expression as described below.

If we could decide that N/A is a predicate we store this fact. Note that as a side effect we might find other predicates (and functions) as well. If we could not decide that N/A is a predicate we let the oracle decide.

We say that the condition C is a functional expression if:

- C is an atom which is either a canonical object or a function,
- the main condition constructor of C is `pi`, or in the case of *FLplus* one of `pi`, `if`, `+`, `-`, `*`, `/`, and `mod`,
- $C = (C_1, C_2)$ or $C = (C_1; C_2)$ and C_1 or C_2 is a functional expression,
- $C = C_1 \rightarrow C_2$ and C_2 is a functional expression but not a canonical object or C_1 is a predicate expression and C_2 is a functional expression.

We also say that the condition C is a predicate expression if:

- C is either the condition true or an atom which is a predicate,
- the main constructor of C is `^`, or in the case of *FLplus* one of , `^`, `\+`, `system`, `<` , `>`, `=<`, `>=`, `=:=`, and `=\=`,
- $C = (C_1, C_2)$ or $C = (C_1; C_2)$ and C_1 or C_2 is a predicate expression,
- $C = C_1 \rightarrow C_2$ and C_1 is a functional expression or C_2 is canonical object.

The given heuristics are correct but not complete. Typically, there are two cases not covered, namely definitions like

```
q(X) <= X.
```

and

```
q(X) <= r(X).
```

where `r/1` is not defined at all.

6.2.2 Specialized Rules

When we know which definitions constitute functions, predicates, and canonical objects, we proceed to create an efficient rule definition according to the method *Local Strategies*. We describe phase two and three together but one should keep in mind that it is possible to imagine other more or less equivalent ways to write the exact details of the created rule definitions.

We need two things, specialized rules and strategies to handle each separate function and predicate, and a top level strategy for arbitrary queries.

Specialized D-right rules for Predicates. A proof of a predicate in *FL* will always end with an application of the rule *D-right*. Since proofs are built backwards the first rule to use in a proof of a predicate is `d_right`. We generate a specialized version of this rule for each predicate. Given a predicate, say `pred` of arity 2, the following rule is generated:

```
pred2_d_right <=
  functor(C,pred,2),
  clause(C,B),
  (pred2_next(B) -> ([] \- B))
  -> ([] \- C).
```

Note that since we know that `pred` is a predicate we do not need to check at run time that `B` does not have a circular definition. The strategy `pred2_next` chooses the correct continuation for the rest of the proof depending on the structure of the chosen clause.

Specialized D-left rules for Functions. Symmetrically to predicates the evaluation of a function, in *FL*, always ends with an application of the rule *D-left*. We generate a specialized version of *D-left* for each function. Given a function `fun` of arity 3 for instance, the generated rule becomes:

```
fun3_d_left <=
  functor(T,fun,3),
  definiens(T,Dp,_),
  (fun3_next(Dp) -> ([Dp] \- C))
  -> ([T] \- C).
```

Again, it is not necessary to check at run time that `fun` does not have a circular definition. The strategy `fun3_next` chooses the correct continuation depending on the definiens of `T`.

Generalized Axiom Rule. We do not create any special rules or strategies for canonical objects. It is worth noting that since the rules are pre-compiled we can omit the check for circularity at run time, thus enhancing efficiency.

6.2.3 Creating Continuation Strategies

Each function and predicate definition consists of a number of clauses:

$$\begin{aligned}&A_1 \Leftarrow B_1.\\&\vdots\\&A_1 \Leftarrow B_1.\end{aligned}$$

When we have looked up a body with *clause* or *definiens* we would like to continue in the correct manner depending on which B_i was chosen. The naive approach to do this is to define a strategy with one clause for each B_i, thus:

$$\begin{aligned}&FP\_next(B_1) \Leftarrow S_1.\\&\vdots\\&FP\_next(B_n) \Leftarrow S_n.\end{aligned}$$

This works well enough as long as fewer than two bodies are unifiable, but is otherwise inefficient. To see why consider the definition:

```
p(1,X) <= q(X).
p(X,1) <= q(X).

q(1).
q(2).
```

The naive approach would generate:

```
p2_next(q(_)) <= q1_d_right.
p2_next(q(_)) <= q1_d_right.
```

Now if we ask the query

```
p2_d_right \\- \- p(1,1).
```

the body of `p` will be proved twice, once for each clause of `p2_next`.

To remedy this problem, we have to analyze the bodies of each function and predicate definition and merge overlapping bodies together before the *next*-strategies are created. There is one problem in the merging process though; variables. As an example consider the definition:

```
p(X) <= X,r,q.
p(X) <= r,X,q.
```

The two bodies are possibly overlapping but neither is an instance of the other, so we cannot simply take one body and throw the other away. Instead we have to create a generalized condition which can be instantiated to both, we must merge and generalize. The rule-generator will produce:

```
p1_next((X,Y,q)) <= somestrategy.
```

Generally, given the list of bodies defining a function or a predicate we do the following:

- the list of bodies is split into non-variable bodies and variable bodies. The variable bodies are immediately merged into one. The purpose of this split is that it sometimes is convenient to treat the variable bodies separately,
- the non-variable bodies are merged and generalized according to the procedure described below.

Merging non-variable bodies. We describe the algorithm used to merge and generalize bodies with a GCLA rule definition. The definition is designed to handle one single query:

```
cases \\- (\- cases(InBodies,OutBodies).
```

The code given in Figure 1 looks (and is) very much like a Prolog-program. The main reason for this is that at the rule level the only possible canonical values, that is, the only possible results from functions, are object-level sequents, thus forcing us to write everything as predicates. Some possible queries are:

```
cases \\- ( \- cases([q(_),q(_)],B)).

B = [q(_A)] ?

cases \\- ( \- cases([(r,X,q),(X,r,q)],B)).

B = [(_B,_A,q)] ?
```

Creating Strategies for each Body. When the bodies have been merged and generalized the only remaining problem is to create a suitable strategy for each. With a fixed set of deterministic inference rules this poses no great problem since for each occurrence of a constructed condition there is at most one rule to apply and also for each function and predicate there is a specialized rule to use. The only problem is that there may be variables denoting conditions which are unknown when we the strategies are created. This problem is solved by having two top level strategies, one called `eval` for functional expressions (the left hand side of sequents), and one called `prove` for predicates (the right hand side of sequents), which are used whenever a variable is found in the generation process.

We do not describe in detail how the strategies for each body are generated since it is not very interesting (an interesting question is how we best can allow for new rules and condition constructors) but merely demonstrate with a couple of examples:

```
cases <=
   cases(In,Out)
   -> ([] \- cases(In,Out)).

cases(In,Out):-
   cases(In,Mid,Flag),
   (    Flag == nochange,
        unify(Mid,Out) ->  true
   ;
        Flag == change,
        cases(Mid,Out)
   ).

cases([],[],nochange).
cases([],[],Flag):- Flag == change.
cases([X|Xs],[Y|R],Flag):-
   rem_and_gen(X,Xs,Y,Xs1,Flag),
   cases(Xs1,R,Flag).

rem_and_gen(X,[],X,[],Flag).
rem_and_gen(X,[Y|Xs],Z,R,Flag):-
   match_gen(X,Y,Z1,Flag),
   rem_and_gen(Z1,Xs,Z,R,Flag).
rem_and_gen(X,[Y|Xs],Z,[Y|R],Flag):-
   %\+match_gen(X,Y...
   rem_and_gen(X,Xs,Z,R,Flag).

match_gen(X,Y,Z,change):- var(X),nonvar(Y).
match_gen(X,Y,Z,change):- nonvar(X),var(Y).
match_gen(X,Y,Z,Flag) :-  var(X),var(Y).
match_gen(X,Y,X,Flag):-
   functor(X,N,A),
   A =:= 0,
   functor(Y,N,A).
match_gen(X,Y,Z,Flag):-
   functor(X,N,A),
   A > 0,
   functor(Y,N,A),
   X =.. [F|ArgsX],
   Y =.. [F|ArgsY],
   match_gen_args(ArgsX,ArgsY,ArgsZ,Flag),
   Z =.. [F|ArgsZ].

match_gen_args([],[],[],Flag).
match_gen_args([X|Xs],[Y|Ys],[Z|Zs],Flag):-
   match_gen(X,Y,Z,Flag),
   match_gen_args(Xs,Ys,Zs,Flag).
```

Figure 1: Code to split definitions into functions and predicates.

- the functional expression `p(X) -> succ(X)` will generate (provided that `p` is defined as a predicate and `succ` as a function):
  `a_left(_,a_right(_,p1_d_right),succ1_d_left))`,
- the functional expression `(X -> Y) -> s(Y)` will get the strategy(provided that `s` is a canonical object): `a_left(_,a_right(_,eval),axiom(_,_)))`,
- the predicate expression `q(X),r(X)` will have the corresponding strategy `v_right(_,q1_d_right,r1_d_right)`.

Whenever the merging process results in a single body the corresponding strategy is inserted directly into the function or predicates *D*-rule, thus omitting the next-strategies. For instance

```
from(N) <= [N|from(s(N))].
```

will get the rule:

```
from1_d_left <=
  functor(T,from,1),
  definiens(T,Dp,_),
  (axiom(_,_) -> ([Dp] \- C))
  -> ([T] \- C).
```

6.2.4 Top Level Strategies

All files created with the rule generator described here includes a file with all the rules of *FLplus* and three top level strategies `fl_gen`, `eval` and `prove`. These top level strategies are implemented to find the correct rule or strategy to use, including all the specialized rules for each predicate and function created by the rule generator. They are defined as follows:

```
fl_gen <= fl_gen(_).

fl_gen(A) <= (A \- _).
fl_gen([]) <= prove.
fl_gen([A]) <= eval.

eval <= left(_).

left(T) <= ([T] \- _).
left(T) <= (nonvar(T),case_l(T,PT) -> PT) <- true,
            var(T) -> d_axiom(_,_).
```

The proviso case_l is a listing of the appropriate rule to continue a proof with for each possible condition T. As a default, it lists the correct continuation for each constructed condition having an inference rule in *FLplus*. When a rule is file created case_l is augmented with clauses for each function and canonical object. The definition of prove is analogously (case_r is augmented with clauses for each predicate):

```
prove <= right(_).

right(C) <= ([] \- C).
right(C) <= nonvar(C),case_r(C,PT) -> PT.
```

6.3 Example

One of the most commonly used examples in papers on functional logic programming in GCLA is quick sort. We will use it again here to make it possible to compare the results from the rule generation process with previous hand-coded suggestions.

We use the same definition as in [6, 16], with the exception that we add circular definitions to define canonical objects. We also use the possibility of the rule-generator to create rules where numbers are regarded as canonical objects. The definition, which is a combination of the strict functions qsort and append and the predicate split, is:

```
[] <= [].
[X|Xs] <= [X|Xs].

cons(X,Xs) <= pi Y \ (pi Ys \ ((X -> Y),(Y -> Ys) -> [Y|Ys])).

qsort([]) <= [].
qsort([X|Xs]) <= pi L \ (pi G \
   (split(X,Xs,L,G)
    -> append(qsort(L),cons(X,qsort(G))))).

append([],Ys) <= Ys.
append([X|Xs],Ys) <= cons(X,append(Xs,Ys)).
append(Exp,Ys)#{Exp \= [],Exp \= [_|_]} <= pi Xs \
   ((Exp -> Xs) -> append(Xs,Ys)).

split(_,[],[],[]).
split(E,[F|R],[F|Z],X) <= E >= F,split(E,R,Z,X).
split(E,[F|R],Z,[F|X]) <= E < F,split(E,R,Z,X).
```

The fact that we use explicit quantification in this definition makes it very easy to find functions and predicates and the code given in Figure 2 is generated without any questions being asked.

6.4 Discussion

If we use explicit quantification as in the example in Section 6.3 it is usually possible to divide the defined atoms into functions and predicates automatically, for most definitions it is even enough to use explicit quantification in the object functions only. It would of course be trivial to get rid of the entire splitting problem by introducing some kind of declarations, but we wish to keep our definitions free from this kind of external information.

The rule-generation process is really very much like a kind of partial evaluation of a general rule definition like *FL* with respect to a certain definition and a given set of queries. An interesting question is if this process can be extended to more general classes of definitions as well. Another possibility to investigate is to unfold as many rule calls as possible thus minimizing the number of rule calls and increasing performance.

When we generate rules according to the method local strategies, what we get is so to speak a *basic procedural interpretation* for each function and predicate. Since we have a distinct procedural part for each function and predicate it is very easy to manually alter the procedural behavior of a particular function or predicate. For example, given the definition

```
member(X,[X|_]).
member(X,[_|Xs]) <= member(X,Xs).
```

the created rule `member2_d_right` will enumerate all instances of `X` in `Xs`. If we only want to find the first member somewhere in a program we can write another procedural part achieving this and substitute it for `member2_d_right` at the appropriate places.

7 Moving Information to the Rule Level

Almost all function definitions shown so far have contained some clause to force evaluation of arguments when necessary. If we want to have function definitions which by themselves explicitly describe the computations needed to evaluate a function this makes perfect sense and, as we have seen, it is possible to get by with very simple rule definitions.

There is one major problem though, it is sometimes very complicated to define functions by pattern matching. We have not seen many examples of this but then we have avoided the problem by only writing function definitions where at most one argument has a pattern other than a variable.

When we try to define functions using pattern matching on several arguments we immediately run into problems, as seen below.

```
:- include_rules(lib('FLRules/flnumplus.rul')).
:- include_rules(lib('FLRules/flnumplus_basic_strats.rul')).

cons2_d_left <=
   functor(A,cons,2),
   definiens(A,Dp,_),
   (pi_left(_,
      pi_left(_,
         a_left(_,v_right(_,a_right(_,eval),a_right(_,eval)),
                  axiom))) -> ([Dp] \- C))
   -> ([A] \- C).

qsort1_d_left <=
   functor(A,qsort,1),
   definiens(A,Dp,_),
   (qsort1_next(Dp) -> ([Dp] \- C))
   -> ([A] \- C).

qsort1_next(([])) <= axiom.
qsort1_next((pi A\pi B\split(C,D,A,B)-> append(qsort(A),cons(C,qsort(B))))) <=
   pi_left(_,pi_left(_,a_left(_,split4_d_right,append2_d_left))).

append2_d_left <=
   functor(A,append,2),
   definiens(A,Dp,_),
   (eval -> ([Dp] \- C))
   -> ([A] \- C).

split4_d_right <=
   functor(A,split,4),
   clause(A,B),
   (split4_next(B) -> ([] \- B))
   -> ([] \- A).

split4_next((true)) <= truth.
split4_next((A>=B,split(A,C,D,E))) <=
   v_right(_,gte_right(_,eval,eval),split4_d_right).
split4_next((A<B,split(A,C,D,E))) <=
   v_right(_,lt_right(_,eval,eval),split4_d_right).

case_l([],axiom).
case_l([A|B],axiom).

case_l(cons(A,B),cons2_d_left).
case_l(qsort(A),qsort1_d_left).
case_l(append(A,B),append2_d_left).

case_r(split(A,B,C,D),split4_d_right).
```

Figure 2: Generated rules for the quick sort example.

7.1 Why Pattern Matching Causes Problems

Let us try to define the function `min` returning the smallest value of two natural numbers. If we only allow canonical objects as arguments the natural definition is:

```
min(0,_) <= 0.
min(s(_),0) <= 0.
min(s(X),s(Y)) <= succ(min(X,Y)).
```

When we wish to allow arbitrary expressions as arguments we need at least one more clause to evaluate arguments. First we try to define a version that only evaluates the arguments which are not natural numbers, that is, we evaluate exactly the needed arguments. The difficulty in doing this is to write evaluation clauses without introducing overlapping clauses while still covering all possible cases. One solution is to add four more clauses giving a total of seven clauses:

```
min(0,_) <= 0.
min(s(_),0) <= 0.
min(s(X),s(Y)) <= succ(min(X,Y)).
min(E,s(X))#{E \= 0, E \= s(_)} <= (E -> V) -> min(V,s(X)).
min(E,0)#{E \= s(_),E \= 0} <= 0.
min(s(X),E)#{E \= 0,E \= s(_)} <= (E -> V) ->  min(s(X),V).
min(E1,E2)#{E1 \= 0,E1 \= s(_), E2 \= 0, E2 \= s(_)} <=
  (E1 -> V1),(E2 -> V2) -> min(V1,V2).
```

This is rather terrible and can not be considered as a serious alternative. We can do slightly better if we evaluate both arguments when none of the original clauses match, that is, we add a fourth clause:

```
min(E1,E2)# Guard <= (E1 -> V1),(E2 -> V2) -> min(V1,V2).
```

When `E1` or `E2` is already a canonical object this clause will perform redundant computations when one of the arguments is evaluated to itself, but that cost is negligible compared to the gain in readability. What we need is a guard that excludes the three first cases but catches all cases where one of the arguments is something other than `0` or `s(_)`. The guards are built-up of conjunctions of inequalities. One guard that does not work is the one in the last clause above since it also excludes all cases where one argument is a canonical object. Instead we have to write the fourth clause:

```
min(E1,E2)#{min(E1,E2) \= min(0,_),
            min(E1,E2) \= min(s(_),0),
            min(E1,E2) \= min(s(_),s(_))} <=
            (E1 -> V1),(E2 -> V2) -> min(V1,V2).
```

This version is obviously better than the previous one. It should be mentioned that since it is a common problem in GCLA to define functions like `min`, where it is not trivial to write a correct guard to exclude all other cases, there is a special construct in the language for this. If we write $a\#else \Leftarrow C$ a guard which excludes all other clauses defining a is generated, thus making the following definition possible:

```
min(0,_) <= 0.
min(s(_),0) <= 0.
min(s(X),s(Y)) <= succ(min(X,Y)).
min(E1,E2)#else <= (E1 -> V1),(E2 -> V2) -> min(V1,V2).
```

However, the specialized *D-left* rules we create when we generate rule definitions in Section 6 opens up the way for another approach where the fourth clause is not needed at all. What we do is to create a specialized *D-left* rule which ensures that the arguments to *min* are evaluated before we use the definiens operation to substitute definiens for definiendum. The rule connected with *min* becomes:

$$\frac{X \vdash X_1 \quad Y \vdash Y_1 \quad M \vdash C}{min(X,Y) \vdash C} \quad M = D(min(X_1,Y_1))$$

Naturally, rules like this could be coded manually but it gets rather tiresome to write specialized rules for each function and predicate.

7.2 Another Way to Define Functions and Predicates

When we remove the evaluation clauses from function definitions it reflects a shift of our view of the relation between the definition and the rule definition. The function definitions of previous sections are in some sense complete, we stated all information needed to perform computations explicitly, the role of the rule definition was passive, it merely stated ways to combine atoms (interpret condition constructors) and replace them with their definiens.

By removing the evaluation clauses it is the definition which so to speak becomes the passive part, its only role is to statically define substitutions between atoms. Instead, the rule definition becomes the vehicle that forces evaluation and determines the meaning of expressions not defined in the definition. The resulting programs express the fact that the definition and the rule definition are two equivalent parts in GCLA.

There are several different possible choices concerning what arguments to functions and predicates that should be evaluated by the rules. We suggest some conventions below. Other possible schemes are discussed in Section 7.4.

7.2.1 Strict Function Definitions and their D-left Rules

The usual meaning of a strict function definition is that its arguments are evaluated before the function is called. It is therefore reasonable to associate with each strict function a rule that evaluates each of the arguments and then looks up the definiens of the resulting atom. If F is a function of arity n the rule becomes:

$$\frac{X_1 \vdash Y_1 \ldots X_n \vdash Y_n \quad Dp \vdash C}{F(X_1, \ldots, X_n) \vdash C} \qquad Dp = D(F(Y_1, \ldots, Y_n))$$

Since arguments to functions are evaluated before the function is called the only meaningful patterns are canonical objects and variables. To see why, consider the definition:

```
rev(rev(L)) <= L.
rev([]) <= [].
rev([X|Xs]) <= append(rev(Xs),[X]).
```

The first clause is intuitively correct but it can never be applied since the argument is evaluated to a canonical object before `rev` is called.

Besides the removed evaluation clauses there is one more difference in strict function definitions—the implicit type definitions. Recall that in Section 4 we used object functions which evaluated their arguments to build canonical objects. A typical example is the function `succ`:

```
succ(X) <= (X -> Y) -> s(Y).
```

When we evaluate the argument of `succ` before it is called the condition `(X -> Y)` becomes redundant. The entire implicit type definition of natural numbers thus becomes:

```
0 <= 0.
s(X) <= s(X).

succ(X) <= s(X).
```

The new version of `succ` may look a bit strange but it has the same purpose as the old one; to evaluate `X` before the number `s(X)` is built. Generally using this kind of strict evaluation the object function connected to the canonical object with definition

$$S(X_1, \ldots, X_n) \Leftarrow S(X_1, \ldots, X_n)$$

becomes:

$$F(X_1, \ldots, X_n) \Leftarrow S(X_1, \ldots, X_n)$$

We may now reformulate our first example of Section 2.1. We assume that the implicit type definition above is used:

```
plus(0,N) <= N.
plus(s(M),N) <= succ(plus(M,N)).
```

7.2.2 Lazy Functions and their D-left Rules

The ideal in lazy function definitions is to only evaluate arguments if it is absolutely necessary. A reasonable and easy to implement compromise is to evaluate arguments which have another pattern than a variable in some defining clause. To ensure avoiding evaluating unnecessary arguments one should then only write uniform function definitions.

As an example we define append once more. The implicit type definitions remain identical compared with previous lazy functions:

```
append([],Ys) <= Ys.
append([X|Xs],Ys) <= [X|append(Xs,Ys)].
```

Since the second argument is not needed to match any clause we do not evaluate it before `append` is applied. Thus, the *D-left* rule connected to `append` becomes:

$$\frac{X \vdash X_1 \quad A \vdash C}{\mathit{append}(X,Y) \vdash C} \quad A = D(\mathit{append}(X_1,Y))$$

It should be noted that since arguments with variable patterns in all clauses are not evaluated before we apply definiens we can not allow repeated variables in the heads of lazy function definitions.

We also remove the show functions from lazy function definitions and instead introduce a strategy `show` that is used to fully evaluate expressions. This strategy can be automatically generated based on what the canonical values of the definition are.

The top-level strategy `show/0` is simply defined in terms of a rule `show/1` which does all the work. The definitions of `show/0` and `show/1` are as follows:

```
show <= show(eval).

show(PT) <=
  eval_these(T,PT,Exp,C1),
  Exp,
  unify(C,C1)
  -> ([T] \- C).
```

In the rule `show/1`, `T` is the functional expression to be evaluated and `PT` is some kind of general strategy for functional evaluation such as `eval` described in Section 6. The purpose of the proviso `eval_these` is to define which parts of `T` that need to be further evaluated and what the resulting value is. The third argument of `eval_these` provides a (meta-level) condition specifying the necessary computations and the fourth the result which is unified with `C`, the conclusion of the rule. The definition of `eval_these` is in terms of a proviso `show_case`, which varies depending on the canonical objects of the application:

```
eval_these(T,PT,Exp,C1) :- nonvar(T),show_case(T,PT,Exp,C1).
eval_these(T,_,true,T) :- var(T),circular(T).
```

Recall that what the show functions presented earlier in Section 4 did was to evaluate the subparts of canonical objects. There were three different kinds of cases in the definition of a show function: the expression to be showed could be either a canonical object of arity zero, a canonical object of arity greater than zero or a functional expression other than a canonical object. The corresponding definitional clauses of `show_case` are:

- for each canonical object S of arity zero a clause

  $$show\_case(S, \_, true, S).$$

- for each canonical object S of arity n a clause

  $$\begin{aligned} show\_case(S(X_1,\dots,X_n), PT, \; & ((show(PT) \to ([X_1] \vdash Y_1)),\dots, \\ & (show(PT) \to ([X_n] \vdash Y_n))), \\ & S(Y_1,\dots,Y_n)). \end{aligned}$$

- and finally a clause to handle expression that are not canonical objects, it becomes

  $$\begin{aligned} show\_case(E, PT, \; & ((PT \to ([E] \vdash CanObj)), \\ & (show(PT) \to ([CanObj] \vdash CanVal))), \\ & CanVal)\#\{Guard\}. \end{aligned}$$

  where $Guard$ contains the inequality $E \neq S_i$ for each canonical object S_i.

As a simple example, assume that we have a definition where lists and numbers are the only canonical objects. Then the definition of `show_case` becomes:

```
show_case(0,_,true,0).
show_case(s(X),PT,(show(PT) -> ([X] \- Y)),s(Y)).
show_case([],_,true,[]).
show_case([X|Xs],PT,((show(PT) -> ([X] \- Y)),
                     (show(PT) -> ([Xs]\-Ys))),[Y|Ys]).
show_case(E,PT,((PT -> ([E] \- CanObj)),
                (show(PT) -> ([CanObj] \- CanVal))),Canval)
                #{E \= 0, E \= s(_), E \= [], E \= [_|_]}.
```

Now if we ask the query

```
show \\- append(append([],[0]),[s(0)]) \- C.
```

the only answer will be `C = [0,s(0)]`.

7.2.3 Predicate Definitions and their D-right Rules

We also take the approach that arguments to predicates (symmetrically) may be any functional expressions. A consequence of this is that the only allowed patterns in predicate definitions, as in function definitions, are canonical objects and variables.

When we create specialized versions of *D-right* to predicates we let them evaluate all arguments before we try to find a unifiable clause. The reason for this is of course the two-way nature of predicates. Generally, if P is a predicate of arity n its corresponding *D-right* rule becomes:

$$\frac{X_1 \vdash Y_1 \ldots X_n \vdash Y_n \quad \vdash B}{\vdash P(X_1, \ldots, X_n)} \quad B \in D(P(Y_1, \ldots, Y_n)).$$

If we use strict functions in the arguments of predicates this approach works well enough, but if we combine lazy functions and predicates the situation becomes, as usual, more complicated.

For instance, consider the usual member definition

```
member(X,[X|_]).
member(X,[_|Ys]) <= member(X,Ys).
```

If `append` is a lazy function and we ask a query like

```
\- member(3,append([2+1],[])).
```

it will of course fail since the functional expression `append([2+1],[])` will be evaluated to `[2+1|append([],[])]`.

There are two simple solutions to this problem, the first is to write the definitions so that problems of this kind does not occur. The membership predicate may instead be written

```
member(X,[Y|_]) <= X=Y.
member(X,[_|Ys]) <= member(X,Ys).
```

provided a proper definition of '`=`', see Section 8.2.2. A first step to write predicates which work correctly with lazy functions as arguments is to adhere to all restrictions we have mentioned concerning pattern matching in functions. The second way to avoid the problem is that when the *D-right* rules are generated use the strategy `show` instead of `eval` to evaluate arguments thus forcing evaluation of arguments to predicates.

Of course these solutions are far from perfect, leaving us with some problems remaining to be solved concerning integration of functions and predicates in GCLA.

7.3 Examples

In order to show the differences and similarities of the function and predicate definitions described in this section and the previous sections we present a couple of examples here. More examples may be found in appendix A.

Our first example is the type definition for lists, the definition of the canonical objects remains the same, only the definition of cons is changed:

```
[] <= [].
[X|Xs] <= [X|Xs].

cons(X,Xs) <= [X|Xs].
```

In Section 4 we defined the function take returning the n first elements of a list using pattern matching on the first argument only. We can now write the more compact definition

```
take(0,_) <= [].
take(s(N),[X|Xs]) <= cons(X,take(N,Xs)).
```

with the corresponding generated *D-left* rule:

```
take_d_left <=
  (eval -> ([N] \- N1)),
  (eval -> ([L] \- L1)),
  definiens(take(N1,L1),Dp,1),
  (take_next(Dp) -> ([Dp] \- C))
   -> ([take(N,L)] \- C).
```

A lazy version of take is:

```
take(0,_) <= [].
take(s(N),[X|Xs]) <= [X|take(N,Xs)].
```

Note that if we use the conventions of Section 7.2 both definitions of take will act lazily if rules are generated according to the lazy scheme. The reason for this is that the function cons will not evaluate its arguments under the lazy scheme. This means that we are back in a situation where one and the same definition may be used both for lazy and eager evaluation depending on the rule definition used as in [5, 6]. The notions lazy and strict (eager) evaluation are quite different though as discussed in Section 4.4.

Sections 7.1 and 7.2 also give definitions of plus, append and member. Using these definitions and take we can pose queries like (strict evaluation is assumed):

```
fl_gen \\- take(min(s(0),s(s(0)),append([0],[s(0)])) \- C.

C = [0];
```

```
no

fl_gen \\- \- member(X,append([0,s(0)],[s(s(0))])).

X = 0;
X = s(0);
X = s(s(0));
no

fl_gen \\- \- member(plus(s(0),s(0)),cons(0,cons(s(s(0)),[]))).

true ?;
no
```

The rule generator also allows us to stipulate that numbers should be regarded as if they were canonical objects, that is as if we had the clauses

```
0 <= 0.
1 <= 1.
```

and so on. We may then restate the factorial function from Section 5:

```
fac(0) <= 1.
fac(N)#{N \= 0} <= N > 0 -> N*fac(N-1).
```

7.4 Discussion

The functional logic program definitions and rule definitions we have presented in this section are not equivalent to the ones in previous sections. A typical example is the difference in behavior if we call a function with an incorrect argument like:

```
plus([],0) \- C.
```

If the empty list is defined as a canonical object, the definition of Section 2.1 will loop forever trying to evaluate it to `0` or `s(X)`. The definition of Section 7.2 together with its generated rule definition on the other hand will fail.

The computational behavior of the functional logic programs described in this section is easily mapped into definitions executable with *FLplus* though, by writing each function in two steps—the first step evaluates each argument needed and the second step is identical to the function definitions described in this section. A strict definition of addition according to this two-step scheme is:

```
plus(X,Y) <= (X -> X1),(Y -> Y1) -> plus1(X1,Y1).

plus1(0,N) <= N.
plus1(s(M),N) <= succ(plus(M,N)).
```

It should also be noted that the restriction of patterns in clause heads to canonical objects is really very much the same as the restriction to constructors in so called constructor-based languages [24], although differently motivated.

A central idea in GCLA-programming is that it is possible to write a procedural part which gives exactly the desired procedural behavior for each specific definition and query. Since there is nothing absolute in the conventions for specialized D-rules described in Section 7.2 the rule-generator allows the programmer to customize the produced rule definitions. In addition to the D-rules of Section 7.2 it is possible to work in a manual mode where for each function and predicate definition it is possible to stipulate exactly which arguments should be evaluated.

None of our schemes is really satisfactory for lazy functional logic programs, the main reasons being the severely restricted pattern matching and the fact that we will often evaluate too many arguments. The usual approach to solve this in other languages is to use different kinds of program transformation and analysis techniques [18, 35, 37, 44]. We could of course use similar methods in GCLA. For instance, [49] describes an automatic transformation of programs from a lazy functional language into GCLA which uses such techniques to simplify the program before it is mapped into a GCLA definition.

A last question is if it is necessary to have such highly specialized D-rules, could we not just as well have general D-rules producing the same behavior? To show how this can be done we give the code for a general *D-right* rule which evaluates each argument (*D-left* could be defined analogously):

```
d_right(C,PT) <=
   atom(C),
   not(circ(C)),
   all_args_canonical(C,PT,C1),
   clause(C1,B),
   (PT -> ([] \- B))
   -> ([] \- C).

all_args_canonical(C,PT,C1) :-
   functor_args(C,Functor,Args),
   eval_args(Args,PT,Args1),
   functor_args(C1,Functor,Args).

eval_args([],_,[]).
eval_args([X|Xs],PT,[Y|Ys]) :-
   (PT -> ([X] \- Y)),
   eval_args(Xs,PT,Ys).
```

The proof term `PT` must be a strategy not containing any variables (since it is used several times). The proviso `functor_args/3` is defined using the prolog primitive '`=..`':

```
functor_args(C,F,A):- C =..[F|A].
```

and is thus not "pure" GCLA code.

The main disadvantages of this approach are that it is less efficient and also that the possibility to describe the desired behavior for each function and predicate separately is lost.

8 Related Work

Through the years much has been written about different approaches combining functional and logic programming, for surveys see [9, 12, 24]. An interesting, albeit somewhat dated, overview classifying different approaches together as *embedding*, *syntactic*, *algebraic*, and *higher-order logic* respectively is included in [1]. Today, most research seems to focus on functional logic languages using narrowing as their operational semantics, these correspond roughly to the algebraic approach in [1].

8.1 Syntactic Approaches

The syntactic approach to the combination of functional and logic programming is based on the idea that a functional (equational) program may be transformed into a logic (Prolog) program that may then be executed using ordinary SLD-resolution [33]. This is a well-known idea going back at least to [51]. Some more examples of this approach may be found in [3, 39, 40, 48]. By regarding function definitions as syntactical sugar that is transformed away the problem of giving a computational model suitable for both functions and predicates is avoided.

We illustrate with some examples. In [40] a method is described that makes it possible to transform function definitions into Prolog programs in such a way that lazy evaluation is achieved. This is done by defining a relation `reduce/2` based on the function definitions at hand. For instance the definition

```
append(X,Y) = if null(X)
              then Y
              else [hd(X)|append(tl(X),Y)]
```

is transformed into:

```
reduce(append(X,Y),Z) :- reduce(X,[]),reduce(Y,Z).
reduce(append(X,Y),[FX|append(RX,Y)]) :- reduce(X,[FX|RX]).
```

To perform computations it is also necessary to define values of lists:

```
reduce([],[]).
reduce([U|V],[U|V]).
```

Higher-order functions can be handled by another trick reducing higher-order variables to first-order. We illustrate with an example adopted from [1] showing how higher-order and curryed functions are handled in [51]. A possible equational syntax for defining the higher-order function `map` is:

```
map(F,[]) = [].
map(F,[X|Xs]) = [F(X)|map(F,Xs)].
```

In a functional language this kind of definition is regarded as a sugaring of the λ-calculus, but it could also be interpreted as rewriting rules or, as we will see, as a sugaring of a set of Horn clauses. To begin with, each n-ary function is seen as an $(n+1)$-ary predicate where the last argument gives the value of the function. Higher-order function variables are then reduced to first-order by expressing everything at a meta-level with a binary function `apply` denoting (curryed) function application, thus `F(X)` becomes `apply(F,X)`. Representing the function `apply/2` with the predicate `apply/3`, [51] desugars the definition of `map` into:

```
apply(map,F,map(F)).
apply(map(F),[],[]).
apply(map(F),[X|Xs],[FX|FXs]) :-
   apply(F,X,FX),
   apply(map(F),X,FXs).
```

A variant of the approach to handle higher-order functions has been implemented in a transformation from a lazy functional language into GCLA [49] and could of course, as mentioned in Section 5.5, be added to our programs as well.

8.2 Narrowing

The notion of a functional logic programming language goes back to [45] that suggests using narrowing as the operational semantics for a functional language, thus defining a functional logic language as a programming language with functional syntax that is evaluated using narrowing. The name may also be used in a broader sense, like in this paper, denoting any language combining functional and logic programming.

The theoretical foundation of languages using narrowing is Horn-clause logic with equality [43], where functions are defined by introducing new clauses for the equality predicate. Narrowing, a combination of unification and rewriting that originally arose in the context of automatic theorem proving [46], is used to solve equations, which in a functional language setting amounts to evaluate functions, possibly instantiating unknown functional arguments.

Several languages based on Horn-clause logic with equality and narrowing have been proposed, among them are ALF [22, 23], BABEL [37], and SLOG [17]. The language K-LEAF [18] is based on Horn-clause logic with equality but uses a resolution-based operational semantics that is proved to be equivalent to conditional narrowing.

8.2.1 Narrowing Strategies

Narrowing is a sound and complete operational semantics for functional logical languages (Horn-clause Logic with Equality) if a fair computation rule is used[2]. Unrestricted narrowing is very expensive however, so a lot of work has gone into finding efficient versions of narrowing for useful classes of functional logic programs. A detailed discussion of most narrowing strategies is given in [24], here we will simply try to give the basic ideas of narrowing and mention something about different strategies used.

On an abstract level programs in all narrowing languages consist of a number of equational clauses defining functions:

$$LHS = RHS :- C_1, \ldots, C_n \quad n \geq 0$$

where a number of left-hand sides (LHS) with the same principal functor define a function. The C_i's are conditions that must be satisfied for the equality between the LHS and the right-hand side (RHS) to hold. Narrowing can then be used to solve equations by repeatedly *unifying* some subterm in the equation to be solved with a LHS in the program, and then replacing the subterm by the instantiated RHS of the rule.

In order to be able to use efficient but complete forms of narrowing, and to ensure certain properties of programs, there are usually a number of additional restrictions on equational clauses. The exact formulations of these varies between languages but most of the following are usually included:

- The set of function symbols is partitioned into a set of *constructors*, corresponding to our canonical objects, and a set of *defined functions*. The LHS of equations are then restricted so that no defined functions are allowed in patterns.
- The set of variables in the RHS should be included in the set of variables in the LHS. Sometimes extra variables are allowed in the conditional part.
- No two left-hand sides should be unifiable, or if they are then the right-hand sides must have the same value, or alternatively the conditional parts of the equations must not both be satisfiable.
- The rewrite system formed by the equational clauses most fulfill certain properties, for instance that it is confluent and terminating.

The restricted forms of narrowing can be given efficient implementations using specialized abstract machines, see [24] for more details and references. Indeed, [23] argues that functional logic programs are at least as, and often more, efficient than pure logic programs. The possibility to get more efficient programs is due

[2]Just as in Prolog most actual implementations use depth-first search with backtracking, so answers may be missed due to infinite loops

to improved control and to the possibility to perform functional evaluation as a deterministic rewriting process. In purely functional languages like BABEL or K-LEAF predicates are simulated by boolean functions with some syntactic sugaring to make them similar to Prolog predicates.

As an example of a functional logic program and a narrowing derivation consider the definition

```
0 + N = N.
s(M) + N = s(M+N).
```

and the equation `X+s(0)=s(s(0))` to be solved. A solution is given by first doing a narrowing step with the second rule replacing `X+s(0)` by `s(Y+s(0))` binding `X` to `s(Y)`. This gives the new equation `s(Y+s(0))=s(s(0))`. A narrowing step with the first rule can then be used to replace the subterm `Y+s(0)` by `s(0)`, thus binding `Y` to `0` and therefore `X` to `s(0)`. Since the equation to solve now is `s(s(0))=s(s(0))` we have found the solution `X=s(0)`.

Basic Innermost Narrowing. Innermost narrowing is performed inside out and therefore corresponds to eager evaluation of functions. That a narrowing strategy is basic means that narrowing cannot be applied at subterms introduced by substitutions but only at subterms present in the original program or goal. This means that the possible narrowing positions can be determined at compile time which of course is much more efficient than looking through the entire term to be evaluated and trying all positions.

Normalizing Narrowing. A normalizing narrowing strategy prefers deterministic computations. Therefore the equation to be solved is reduced to normal form by rewriting before each narrowing step. Normalizing narrowing may reduce an infinite search space to a finite one since a derivation can be safely terminated if the sides of an equation are rewritten to normal forms that can never yield a solution, see Section 8.2.2 below.

Lazy Narrowing. Lazy narrowing strategies correspond to lazy evaluation of functions. To give a good lazy narrowing strategy is much more difficult than to evaluate a lazy functional language due to the complications introduced by non-determinism and backtracking. Outermost narrowing only allows narrowing at outermost positions but is generally too weak [24]. Therefore, variants like lazy narrowing have been proposed. Lazy narrowing allows narrowing at inner positions if it is necessary to enable some outer narrowing. Another problem is that different rules may require evaluation of different subterms to be applicable. As a solution, the implementation of the language BABEL suggested in [30] transforms programs into a flat uniform (c. f. Section 7.2.2) form. Consider the following equational program [24]:

```
f(0,0) = 0.
f(s(X),0) = 1.
f(X,s(Y)) = 2.
```

Here the second, but not the first, argument must always be evaluated to find a suitable rule. The transformation into flat uniform programs makes this explicit by giving the new program:

```
f(X,0) = g(X).
f(X,s(Y)) = 2.

g(0) = 0.
g(s(X)) = 1.
```

Other recent proposals for efficient lazy evaluation of functional languages include demandedness analysis and needed narrowing, see [24] for more details.

8.2.2 Examples and Comparison with GCLA

To give some kind of intuitive feeling of the behavior of different narrowing strategies and their relationship to the definitional approach taken in this paper we give some simple examples.

Addition. In Section 3 we mentioned that the query

```
X+s(0) \- s(s(0)).
```

using a strict function definition like the one in Section 2.1 will loop forever after finding the first answer. This corresponds to the behavior of basic innermost narrowing for the definition in Section 8.2.1. An alternative solution is to use the following lazy definition

```
0 <= 0.
s(X) <= s(X).

0 + N <= N.
s(M) + N <= s(M+N).
```

together with a specialized generated rule file. We also need an appropriate definition of equality:

```
0 = 0.
s(X) = s(Y) <= X = Y.
```

Now there is one unique proof to show that `X+s(0) = s(s(0))` given below. In the derivation `=/2` has a corresponding *D-right* rule that evaluates the first argument.

$$\dfrac{\dfrac{\dfrac{\{\mathtt{Y} = \mathtt{s(M+s(0))}\}}{\mathtt{s(M+s(0))} \vdash \mathtt{Y}}\;\textit{D-ax}}{\mathtt{X+s(0)} \vdash \mathtt{Y}}\;\textit{add-Dl} \qquad \dfrac{\dfrac{\dfrac{\{\mathtt{Z} = \mathtt{s(0)}\}}{\mathtt{s(0)} \vdash \mathtt{Z}}\;\textit{D-ax}}{\mathtt{M+s(0)} \vdash \mathtt{Z}}\;\textit{add-Dl} \qquad \dfrac{\dfrac{\{\mathtt{W} = \mathtt{0}\}}{\mathtt{0} \vdash \mathtt{W}}\;\textit{D-ax} \qquad \dfrac{}{\vdash \mathtt{true}}\;\textit{truth}}{\vdash \mathtt{0} = \mathtt{0}}\;\textit{eq-Dr}}{\vdash \mathtt{M+s(0)} = \mathtt{s(0)}}\;\textit{eq-Dr}}{\vdash \mathtt{X+s(0)} = \mathtt{s(s(0))}}\;\textit{eq-Dr}$$

Rejection. Innermost normalizing narrowing is more powerful than any method to achieve eager evaluation presented in this paper. To see why consider the rules

```
append([],L) = L.
append([X|Xs],L) = [X|append(Xs,L)].
```

and the equation `append(append([0|V],W),Y) = [1|Z]`. This equation can be reduced by deterministic rewriting to `[0|append(append(V,W),Y)] = [1|Z]` and can therefore be rejected since `0` and `1` are different constructors. A corresponding strict definition of `append` according to the methods we have presented will fail to terminate both for the query

```
append(append([0|V],W),Y) \-  [1|Z].
```

and for:

```
\- append(append([0|V],W),Y) = [1|Z].
```

8.3 Residuation

Both the programs we have presented and languages based on narrowing allow unknown arguments to functions. Although this may be advantageous in equation solving it destroys the deterministic nature of functional evaluation when values for functions are guessed in a non-deterministic way. Several researchers have therefore suggested that functional expressions should only be reduced if arguments are ground (or sufficiently instantiated), and that all non-determinism should be represented by predicates. Predicates may then be proved using SLD-resolution extended so that a function call in a term is evaluated before the term is unified with another term. The exact computational model of functions is not important as long as it ensures that each expression has a unique value. The language Le Fun [1] uses λ-calculus to define functions, Life [2] rewrite rules, and in [32] Standard ML is used to compute functions.

There is one problem with this method however, how should functional expressions containing unknown values be handled? The usual approach is what is called *residuation* in Le Fun, similar methods are used in [2, 32, 39, 47]. We illustrate residuation with an example adopted from [24].

Assume that we have the following definition relating a number to its square:

```
square(X,X*X).
```

Since relations in logic programming usually can be used in both directions we would expect to be able to prove `square(3,9)` as well as find instantiations of variables occurring in `square`. To find a solution to a literal like `square(3,Z)`, `X` is first unified with `3` and then *the value* of `X*X` is computed before it is unified with `Z`, thus binding `Z` to `9`. But if we try the query

```
?- square(V,9), V=3.
```

it leads to failure even though the solution is obviously implied by the program. The reason for this failure is that `9` and the unevaluable function call `V*V` cannot be unified. To avoid failures like this residuation is used; instead of failing the evaluation of the functional expression `X*X` is postponed until the variable X becomes bound and unification of `square(V,9)` and `square(X, X*X)` succeeds with the *residuation* that `9 = V*V`. When `V` later becomes bound to `3` the residuation can be proved and the entire goal is proved to be true. Residuation is satisfactory for many programs, but it may also happen that solutions are not found since variables never become instantiated, see [24] for more details and references.

8.4 Other methods

There have of course been many more proposals to combine functional and logic programming than those we have discussed here. Also, some languages mentioned, e.g. the language Life, do not only combine functional and logic programming but also attempt to include object-oriented and constraint logic programming into one computational framework.

A recent ambitious proposal for a language combining functional and logic programming is the language Escher [34]. Escher attempts to combine the best parts of the logic programming languages Gödel [25] and λ-Prolog [38] and the functional language Haskell [26], with the aim to make learning of declarative programming easier since students will only have to learn one language, and also to bring the functional and logic programming communities closer together, thus avoiding some duplication of research. Escher has its theoretical foundations in Church's simple theory of types and an operational semantics without the usual logic programming operations unification and backtracking. Instead, in Escher, a goal term is rewritten to normal form using function calls and more than 100 rewrite rules.

There are also extensions to functional programming languages to give them some logical features of logical languages. One approach is to extend function definitions with logical guards that have to be proved to make a clause applicable

[13], another used in the language LML (Logical Meta Language) [11], is to have a special (built-in) data type for logical theories and then use the functional language as a kind of glue to combine different theories together.

8.5 Discussion

We have a presented a definitional approach to functional logic programming and given a brief overview of some prominent proposals by others. We believe that the definitional approach has many advantages including:

- Programs are understood through a simple and elegant theory where both predicates and functions are easily defined.
- Compared to narrowing languages an important conceptual difference is that we differ between functions and predicates—predicates are something more than syntactical sugar for boolean functions.
- The two-layered nature of GCLA gives the programmer very explicit control of control and at the same time gives a clean separation between the declarative and the procedural content of a program.
- It is up to the programmer to choose lazy or strict evaluation or even combine them in the same program.
- The rule-generator presented in Sections 6 and 7 provides efficient rule definitions for free, also the specialized definitional rules presented in Section 7 gives a very natural way to handle nested terms in both functions and predicates.

Our approach is far from perfect however, some disadvantages and areas for future work are:

- Programs cannot be run very efficiently as discussed in Section 5.5. To solve this we could either develop a specialized definitional functional language or try to build a better GCLA-compiler.
- More work needs to be done on lazy evaluation strategies and/or program transformations to be able to have less restrictions on patterns in lazy programs. Presumably ideas from the area of lazy narrowing can be used also in the definitional setting.
- More work needs to be done on the theoretical side for instance to investigate the relation between narrowing (Horn Clause Logic with equality) and the definitional approach.

- The current rule-generator gives no support for modular program development and is less efficient than it could be. An easy way to increase performance would be to optimize the rules generated by unfolding as many rule calls as possible.

References

[1] H. Aït-Kaci and R. Nasr. Integrating logic and functional programming. *Lisp and Symbolic Computation*, 2:51–89, 1989.

[2] H. Aït-Kaci and A. Podelski. Towards a meaning of Life. *Journal of Logic Programming*, 16:195–234, 1993.

[3] S. Antoy. Lazy evaluation in logic. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 371–382. Springer-Verlag, 1991.

[4] P. Arenas-Sánchez, A. Gil-Luezas, and F. López-Fraguas. Combining lazy narrowing with disequality constraints. In *Proc. of the 6th International Symposium on Programming Language Implementation and Logic Programming,PLIP'94*, number 844 in Lecture Notes in Computer Science, pages 385–399. Springer-Verlag, 1994.

[5] M. Aronsson. A definitional approach to the combination of functional and relational programming. Research Report SICS T91:10, Swedish Institute of Computer Science, 1991.

[6] M. Aronsson. Methodology and programming techniques in GCLA II. In *Extensions of logic programming, second international workshop, ELP'91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[7] M. Aronsson. *GCLA, The Design, Use, and Implementation of a Program Development System.* PhD thesis, Stockholm University, Stockholm, Sweden, 1993.

[8] M. Aronsson, L.-H. Eriksson, A. Gäredal, L. Hallnäs, and P. Olin. The programming language GCLA: A definitional approach to logic programming. *New Generation Computing*, 7(4):381–404, 1990.

[9] M. Bellia and G. Levi. The relation between logic and functional languages: A survey. *Journal of Logic Programming*, 3:217–236, 1986.

[10] H. Boley. Extended logic-plus-functional programming. In *Extensions of logic programming, second international workshop, ELP'91*, number 596 in Lecture Notes in Artificial Intelligence, pages 45–72. Springer-Verlag, 1992.

[11] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Logic programming within a functional framework. In *Proc. of the 2nd Int. Workshop om Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 372–386. Springer-Verlag, 1990.

[12] D. DeGroot and G. Lindstrom, editors. *Logic Programming, Functions, Relations and Equations.* Prentice Hall, New York, 1986.

[13] R. Dietrich and H. Lock. Exploiting non-determinism through laziness in a guarded functional language. In *TAPSOFT'91, Proc. of the Int. Joint Conference on Theory and Practice of Software Development*, number 494 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[14] G. Falkman. Program separation as a basis for definitional higher order programming. In U. Engberg, K. Larsen, and P. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory.* Aarhus, 1994.

[15] G. Falkman. Definitional program separation. Licentiate thesis, Chalmers University of Technology, 1996.

[16] G. Falkman and O. Torgersson. Programming methodologies in GCLA. In R. Dyckhoff, editor, *Extensions of logic programming, ELP'93*, number 798 in Lecture Notes in Artificial Intelligence, pages 120–151. Springer-Verlag, 1994.

[17] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 172–184. IEEE Computer Soc. Press, 1985.

[18] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42:139–185, 1991.

[19] L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.

[20] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(2):261–283, 1990. Part 1: Clauses as Rules.

[21] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(5):635–660, 1991. Part 2: Programs as Definitions.

[22] M. Hanus. Compiling logic programs with equality. In *Proc. of the 2nd Int. Workshop om Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 387–401. Springer-Verlag, 1990.

[23] M. Hanus. Improving control of logic programs by using functional languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 1–23. Springer-Verlag, 1992.

[24] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19/20:593–628, 1994.

[25] P. Hill and J. Lloyd. *The Gödel Programming Language.* Logic Programming Series. MIT Press, 1994.

[26] P. Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[27] P. Kreuger. GCLA II: A definitional approach to control. In *Extensions of logic programming, second international workshop, ELP91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[28] P. Kreuger. Axioms in definitional calculi. In R. Dyckhoff, editor, *Extensions of logic programming, ELP93*, number 798 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994.

[29] P. Kreuger. *Computational Issues in Calculi of Partial Inductive Definitions.* PhD thesis, Department of Computing Science, University of Göteborg, Göteborg, Sweden, 1995.

[30] H. Kuchen, F. J. López-Fraguas, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a lazy functional logic language with disequality constraints. In *Proc. of the 1992 Joint International Conference and Symposium on Logic Programming.* MIT Press, 1992.

[31] H. Kuchen, R. Loogen, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proceedings of the Second International Conference on Algebraic and Logic Programming*, number 463 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[32] G. Lindstrom, J. Maluszyński, and T. Ogi. Our lips are sealed: Interfacing functional and logic programming systems. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 24–38. Springer-Verlag, 1992.

[33] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second extended edition, 1987.

[34] J. W. Lloyd. Combining functional and logic programming languages. In M. Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium.* MIT Press, 1994.

[35] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming,PLIP'93*, number 714 in Lecture Notes in Computer Science, pages 184–200. Springer-Verlag, 1993.

[36] R. Milner. Standard ML core language. Internal report CSR-168-84, University of Edinburgh, 1984.

[37] J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.

[38] G. Nadathur and D. Miller. An overview of λProlog. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 810–827. MIT Press, 1988.

[39] L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 15–26. Springer-Verlag, 1991.

[40] S. Narain. A technique for doing lazy evaluation in logic. *Journal of Logic Programming*, 3:259–276, 1986.

[41] S. Narain. Lazy evaluation in logic programming. In *Proceedings of the 1990 International Conference on Computer Languages*, pages 218–227, 1990.

[42] N. Nazari. A rulemaker for GCLA. Master's thesis, Department of Computing Science, Göteborg University, 1994.

[43] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1988.

[44] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[45] U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 138–151. IEEE Computer Soc. Press, 1985.

[46] J. J. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[47] G. Smolka. The definition of kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1994.

[48] A. Togashi and S. Noguchi. A program transformation from equational programs into logic programs. *Journal of Logic Programming*, 4:85–103, 1987.

[49] O. Torgersson. Translating functional programs to GCLA. In *Proceedings of La Wintermöte 94*. Department of Computing Science, Chalmers University of Technology, 1994.

[50] O. Torgersson. A definitional approach to functional logic programming. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Extensions of Logic Programming 5th International Workshop, ELP'96*, number 1050 in Lecture Notes in Artificial Intelligence, pages 273–287. Springer-Verlag, 1996.

[51] D. H. D. Warren. Higher-order extensions to Prolog—are they needed? In D. Mitchie, editor, *Machine Intelligence 10*, pages 441–454. Edinburgh University Press, 1982.

[52] D. H. D. Warren, L. M. Pereira, and F. Pereira. Prolog—the language and its implementation compared with Lisp. *SIGPLAN Notices*, 12(8):109–115, 1977.

A Examples

In order to make it easier to compare the style of our programs with the style of some other proposals integrating functional logic programming, we provide some examples adopted from different sources. The following implicit type definitions are assumed to be included in programs using natural numbers and lists:

```
0 <= 0.
s(X) <= s(X).
succ(X) <= s(X).

[]<= [].
[X|Xs] <= [X|Xs].
cons(X,Xs) <= [X|Xs].
```

A.1 Quick Sort

The code for quick sort given in Section 6.3 is cluttered with universally quantified conditions and evaluation clauses. Here we have a nicer syntax for quantifiers and let the rule level evaluate arguments to functions. We also use the operator `@` to denote append.

```
qsort([]) <= [].
qsort([X|Xs]) <= pi [L,G] \
     split(X,Xs,L,G) -> (qsort(L) @ cons(X,qsort(G))).

split(_,[],[],[]).
split(E,[F|R],[F|Z],X) <= E >= F,split(E,R,Z,X).
split(E,[F|R],Z,[F|X]) <= E < F,split(E,R,Z,X).

[] @ Ys <= Ys.
[X|Xs] @ Ys <= cons(X,Xs@Ys).
```

A.2 Sieve Revisited

The code below for `sieve` is basically the same as in Section 5.4, but without evaluation clauses for arguments.

```
primes <= sieve(from(2)).

sieve([P|Ps]) <= [P|sieve(filter(P,Ps))].

filter(N,[X|Xs]) <= if(X mod N =:= 0,
                       filter(N,Xs),
                       [X|filter(N,Xs)]).

from(M) <= [M|from(M+1)].

print_list([X|Xs]) <= system((format('~w ',[X]),ttyflush)),
                      print_list(Xs).
```

The generated rule definition is shown below, note that we have instructed the rule-generator to create a rule that evaluates the argument to `from` (c. f. Section 5.4):

```
primes0_d_left <=
   functor(A,primes,0),
   definiens(A,Dp,_),
   (sieve1_d_left -> ([Dp] \- C))
   -> ([A] \- C).
```

```
sieve1_d_left <=
   (eval -> ([X1] \- Y1)),
   definiens(sieve(Y1),Dp,_),
   (axiom -> ([Dp] \- C))
   -> ([sieve(X1)] \- C).

filter2_d_left <=
   (eval -> ([X2] \- Y2)),
   definiens(filter(X1,Y2),Dp,_),
   (if_left(_,eq_right(_,eval,eval),filter2_d_left,
                         axiom) -> ([Dp] \- C))
   -> ([filter(X1,X2)] \- C).

from1_d_left <=
   (eval -> ([X1] \- Y1)),
   definiens(from(Y1),Dp,_),
   (axiom -> ([Dp] \- C))
   -> ([from(X1)] \- C).

print_list1_d_right <=
   (eval -> ([X1] \- Y1)),
   clause(print_list(Y1),B),
   (v_right(_,system_right(_),print_list1_d_right) -> ([] \- B))
   -> ([] \- print_list(X1)).

case_l([],axiom).
case_l([A|B],axiom).

case_l(primes,primes0_d_left).
case_l(sieve(A),sieve1_d_left).
case_l(filter(A,B),filter2_d_left).
case_l(from(A),from1_d_left).

case_r(print_list(A),print_list1_d_right).
```

A.3 Serialise

Our next example is adopted from [10, 52]. It defines a function `serialise` which transforms a string (list of characters) into a list of their alphabetic serial numbers, for instance `serialise("prolog")` should give the result `[4,5,3,2,3,1]`. The definition using lazy evaluation becomes:

```
nil <= nil.
```

```
node(E,L,R) <= node(E,L,R).

p(X,Y) <= p(X,Y).

serialise(L) <= (numbered(arrange(zip(L,R)),1) -> _) -> R.

zip([],[]) <= [].
zip([X|L],[Y|R]) <= [p(X,Y)|zip(L,R)].

arrange([]) <= nil.
arrange([X|L]) <= partition(L,X,L1,L2)
                  -> node(X,arrange(L1),arrange(L2)).

partition([],_,[],[]).
partition([X|L],X,L1,L2) <= partition(L,X,L1,L2).
partition([X|L],Y,[X|L1],L2) <= before(X,Y),partition(L,Y,L1,L2).
partition([X|L],Y,L1,[X|L2]) <= before(Y,X),partition(L,Y,L1,L2).

before(p(X1,_),p(X2,_)) <= X1 < X2.

numbered(nil,N) <= N.
numbered(node(p(X,N1),T1,T2),N0) <=
        numbered(T2,((numbered(T1,N0) -> N1) -> N1 + 1)).
```

A short explanation is appropriate; `zip` combines the input list of characters with a list `R` of unbound logical variables into a list of pairs, the list of pairs is then sorted and put into a binary tree. Finally, `numbered` assigns a number to each logical variable variable in the tree, simultaneously binding the variables in `R`.

A.4 N-Queens

We also show a definition (inspired by [40]) combining lazy functions predicates and non-determinism into a generate and test program for the N-Queens problem. Note how `fromto` is made strict by using `cons` and also note the non-deterministic function `insert`.

```
queens(N) <= safe(perm(fromto(1,N))).

safe([]) <= [].
safe([Q|Qs]) <= [Q|safe(nodiag(Q,Qs,1))].

nodiag(_,[],_) <= [].
nodiag(Q,[X|Xs],N) <= noattack(Q,X,N) -> [X|nodiag(Q,Xs,N+1)].
```

```
noattack(Q1,Q2,N) <= Q1 > Q2,N \= Q1-Q2.
noattack(Q1,Q2,N) <= Q1 < Q2,N \= Q2-Q1.

perm([])<= [].
perm([X|Xs]) <= insert(X,perm(Xs)).

insert(X,[]) <= [X].
insert(X,[Y|Ys]) <= [X,Y|Ys],[Y|insert(X,Ys)].

fromto(N,M) <= if(N=:=M,
                  [N],
                  cons(N,fromto(N+1,M))).
```

A.5 Imitating Higher Order Functions

This program uses an extra function `apply` to imitate higher order programming. The function `gen_bin` computes all binary numbers of length n. The operator '`@`' is as defined in the quick sort example.

```
1 <= 1.
cons(X) <= cons(X).

map(_,[]) <= [].
map(F,[X|Xs]) <= cons(apply(F,X),map(F,Xs)).

gen_bin(0) <= [[]].
gen_bin(s(X)) <=
    (gen_bin(X) -> Nums)
     -> map(cons(0),Nums) @ map(cons(1),Nums).

apply(cons(X),Y) <= cons(X,Y).
```

A.6 Hamming Numbers

Finally, a program computing hamming numbers. In this program we combine both lazy (`ham`, `mlist`, `merge`) and strict (addition and multiplication) functions with predicates (`nth_hamming`, `nth_mem`, '<').

```
nth_hamming(N,M) <= nth_mem(N,ham,M).

nth_mem(0,[X|Xs],X).
nth_mem(s(N),[X|Xs],Y) <= nth_mem(N,Xs,Y).

ham <= [s(0)|merge(mlist(s(s(0)),ham),
               merge(mlist(s(s(s(0))),ham),
                 mlist(s(s(s(s(s(0))))),ham)))].

mlist(N,[X|Xs])<= (N*X -> M) -> [M|mlist(N,Xs)].


merge([X|Xs],[Y|Ys]) <= if(X < Y,
                           [X|merge(Xs,[Y|Ys])],
                           if(Y < X,
                              [Y|merge([X|Xs],Ys)],
                              [X|merge(Xs,Ys)])).

0 + N <= N.
s(M) + N <= succ(M + N).

0 * N <= 0.
s(M) * N <= (M * N) + N.

0 < s(_).
s(M) < s(N) <= M < N.
```

The predicate `nth_hamming` can be used both to compute the nth hamming number, to give the number of a certain hamming number, and to enumerate all hamming numbers on backtracking.

The rule definition shown below was generated by manually telling the rule generator what arguments to evaluate for each function and predicate, thus making it possible to freely mix functions, predicates, strict, and lazy evaluation in one program. We only show the definitional rules for each function and predicate since that is enough to see how arguments are evaluated.

```
% specialized d_left-rules for each function
nth_hamming2_d_right <=
   functor(A,nth_hamming,2),
   clause(A,B),
   (nth_mem3_d_right -> ([] \- B))
   -> ([] \- A).

nth_mem3_d_right <=
   (eval -> ([X1] \- Y1)),
   (eval -> ([X2] \- Y2)),
   clause(nth_mem(Y1,Y2,X3),B),
   (nth_mem3_next(B) -> ([] \- B))
   -> ([] \- nth_mem(X1,X2,X3)).
```

```
ham0_d_left <=
   functor(A,ham,0),
   definiens(A,Dp,_),
   (axiom -> ([Dp] \- C))
   -> ([A] \- C).

mlist2_d_left <=
   (eval -> ([X2] \- Y2)),
   definiens(mlist(X1,Y2),Dp,_),
   (a_left(_,a_right(_,'*2_d_left'),axiom) -> ([Dp] \- C))
   -> ([mlist(X1,X2)] \- C).

merge2_d_left <=
   (eval -> ([X1] \- Y1)),
   (eval -> ([X2] \- Y2)),
   definiens(merge(Y1,Y2),Dp,_),
   (if_left(_,'<2_d_right',axiom,
       if_left(_,'<2_d_right',axiom,axiom))
        -> ([Dp] \- C))
   -> ([merge(X1,X2)] \- C).

'+2_d_left' <=
   (eval -> ([X1] \- Y1)),
   definiens(+(Y1,X2),Dp,_),
   (eval -> ([Dp] \- C))
   -> ([+(X1,X2)] \- C).

'*2_d_left' <=
   (eval -> ([X1] \- Y1)),
   definiens(*(Y1,X2),Dp,_),
   ('*2_next'(Dp) -> ([Dp] \- C))
   -> ([*(X1,X2)] \- C).

'<2_d_right' <=
   (eval -> ([X1] \- Y1)),
   (eval -> ([X2] \- Y2)),
   clause(<(Y1,Y2),B),
   ('<2_next'(B) -> ([] \- B))
   -> ([] \- <(X1,X2)).

succ1_d_left <=
   (eval -> ([X1] \- Y1)),
   definiens(succ(Y1),Dp,_),
   (axiom -> ([Dp] \- C))
   -> ([succ(X1)] \- C).
```

B FL in GCLA

This appendix shows how the calculus *FL* presented in Section 3 is coded as a rule definition in GCLA. The code contains no search strategies since the deterministic nature of *FL* makes them superfluous.

```
:- multifile(constructor/2).

%%% declarations of the condition constructors used in FL.
constructor(true,0).
constructor(false,0).
constructor(',',2).
constructor(';',2).
constructor((->),2).
constructor(pi,1).
constructor(^,2).
constructor(not,1).

%%% Rules Relating Atoms to a Definition
d_right(C,PT) <=
  atom(C),
  clause(C,B),
  C \== B,
  (PT -> ([] \- B))
  -> ([] \- C).

d_left(T,PT) <=
  atom(T),
  definiens(T,Dp,N),
  T \== Dp,
  (PT -> ([Dp] \- C))
  -> ([T] \- C).

d_axiom(T,C) <=
  term(T),
  term(C),
  unify(T,C),
  circular(T)
  -> ([T] \- C).
```

```
%%% Rules for Constructed Conditions
truth <= ([] \- true).

falsity <= functor(C,false,0) -> ([C] \- false).

a_right((A -> B),PT) <=
  (PT -> ([A] \- B))
  -> ([] \- (A -> B)).

a_left((A -> B),PT1,PT2) <=
  (PT1 -> ([]\- A)),
  (PT2 -> ([B] \- C))
  -> ([(A -> B)] \- C).

v_right((C1,C2),PT1,PT2) <=
  (PT1 -> ([] \- C1)),
  (PT2 -> ([] \- C2))
  -> ([] \- (C1,C2)).

v_left((C1,C2),PT1,PT2) <=
  ((PT1 -> ([C1] \- C)) -> ([(C1,C2)] \- C)),
  ((PT2 -> ([C2] \- C)) -> ([(C1,C2)] \- C)).

o_right((C1 ; C2),PT1,PT2) <=
  ((PT1 -> ([] \- C1)) -> ([] \- (C1 ; C2))),
  ((PT2 -> ([] \- C2)) -> ([] \- (C1 ; C2))).

o_left((A1 ; A2),PT1,PT2) <=
  (PT1 -> ([A1] \- C)),
  (PT2 -> ([A2] \- C))
  -> ([(A1 ; A2)] \- C).

pi_left((pi X \ A),PT) <=
  inst(X,A,A1),
  (PT -> ([A1] \- C))
  -> ([(pi X \ A)] \- C).

sigma_right((X^C),PT) <=
  inst(X,C,C1),
  (PT -> ([] \- C1))
  -> ([] \- (X^C)).

not_right(not(C),PT) <=
  (PT -> ([C] \- false))
  -> ([] \- not(C)).

not_left(not(A),PT) <=
  (PT -> ([] \- A))
  -> ([not(A)] \- false).

%%% Definition of the proviso circular/1
circular(T) :- when_nonvar(T,canonical_object(T)).

canonical_object(T) :- definiens(T,Dp,1),T == Dp.

when_nonvar(A,B) :- user:freeze(A,B).
```

C Flplus

FLplus is made up of all the rules of *FL* plus the rules listed here. Note that *FLplus* is deterministic so we do not show any search-strategies. We have also included rules for dynamically changing the definition. These are really the same as the standard ones with restricted antecedents and are discussed in [6].

```
constructor(if,3).
constructor(\+ ,1).
constructor(system,1).
constructor(add_def,2).
constructor(rem_def,2).

if_left(if(B, T, E), P1, P2, P3) <=
   (((P1->([] \- B)) -> ([if(B,T,E)] \- C)) <- (P2->([T] \- C))),
   ((P3 -> ([E] \- C)) -> ([if(B,T,E)] \- C)).

naf_right((\+ C), PT) <=
     (((PT -> ([] \- C)) -> ([] \- (\+ C) )) <- false),
     ([] \- (\+ C)).

system_right(system(C)) <=
     C  -> ([] \- system(C)).

add_left(add_def(X,Y),PT) <=
     add(X),
     (PT -> ([Y] \- C))
     -> ([add_def(X,Y)] \- C).
```

```
rem_left(rem_def(X,Y),PT) <=
     rem(X),
     (PT -> ([Y] \- C))
     -> ([rem_def(X,Y)] \- C).

add_right(add_def(X,Y),PT) <=
     add(X),
     (PT -> ([] \- Y))
     -> ([] \- add_def(X,Y)).

rem_right(rem_def(X,Y),PT) <=
     rem(X),
     (PT -> ([] \- Y))
     -> ([] \- rem_def(X,Y)).
```

We do not actually list all the rules to handle arithmetics since they are really all the same, only the arithmetical operation differ. Instead we list the constructor declarations and four example rules.

```
constructor(int,1).
constructor(=:=,2).
constructor(=\=,2).
constructor(<,2).
constructor(>=,2).
constructor(>,2).
constructor(=<,2).
constructor('*',2).
constructor('/',2).
constructor('//',2).
constructor('+',2).
constructor('-',2).

integer_left(int(X),PT,PT1) <=
     (PT -> ([X] \- n(X1))),
     Y is integer(X1),
     (PT1 -> ([n(Y)] \- C))
     -> ([int(X)] \- C).

mul_left(*(A,B),PT1,PT2,PT3) <=
     (PT1 -> ([A] \- n(A1))),
     (PT2 -> ([B] \- n(B1))),
     X is A1 * B1,
     (PT3 -> ([n(X)] \- C))
     -> ([(A * B)] \- C).
```

```
gt_right(>(X,Y),PT1,PT2) <=
     (PT1 -> ([X] \- n(NX))),
     (PT2 -> ([Y] \- n(NY))),
     NX > NY
     -> ([] \- X > Y).

eq_right(=:=(X,Y),PT1,PT2) <=
     (PT1 ->  ([X] \- n(N))),
     (PT2 ->  ([Y] \- n(M))),
     N =:= M
     -> ([] \- (X=:=Y)).
```

D Building Blocks for Generated Rules

All rules created by the rule-generator include some common building blocks and top-level strategies as described in Section 6.2.4. If the basic rules are pure *FL* these strategies are as shown below. If *FLplus* is used instead some clauses are added to `case_l` and `case_r`. Apart from generating specialized procedural parts to each function and predicate the rule-generator adds a number of clauses to the provisos `case_l`, and `case_r` and if lazy evaluation is suspected creates the proviso `show_cases`.

```
% The file "fl.rul"  must be loaded.
% :- include_rules(lib('FLRules/fl.rul')).

% Clauses may be added to case_l/2 and case_l/2 from other files
:- multifile(case_l/2).
:- multifile(case_r/2).

% Additional simple axiom rule, only to be used in generated rules
% at places where we know that axiom should be applied.
axiom <=
     unify(T,C)
     -> ([T] \- C).

% Top-level strategies.
fl_gen <= fl_gen(_).

fl_gen(A) <= (A \- _).
fl_gen([]) <= prove.
fl_gen([A]) <= eval.
```

```
eval <= left(_).

left(T) <= ([T] \- _).
left(T) <=
     (left1(T) <- true),
     (var(T) -> d_axiom(_,_)).
left1(T) <= nonvar(T),case_l(T,PT) -> PT.

prove <= right(_).

right(C) <= ([] \- C).
right(C) <= nonvar(C),case_r(C,PT) -> PT.

% The basic definitions of case_l and case_r state which rule
% to use for each predefined condition constructor.
case_l(false,falsity).
case_l((_ -> _),a_left(_,right(_),left(_))).
case_l((_,_),v_left(_,left(_),left(_))).
case_l((_;_),o_left(_,left(_),left(_))).
case_l((pi_ \ _),pi_left(_,left(_))).
case_l(not(_),not_left(_,fl)).

case_r(true,truth).
case_r((_,_),v_right(_,right(_),right(_))).
case_r((_;_),o_right(_,right(_),right(_))).
case_r((_ -> _),a_right(_,left(_))).
case_r(not(_),not_right(_,fl)).
case_r((_^_),sigma_right(_,right(_))).

% Show is a top level strategy used to force evaluation,
% the definition of show_case/4 is added by the rule-generator.
show <= show(eval).

show(PT) <=
     eval_these(T,PT,Exp,C1),
     Exp,
     unify(C,C1)
     -> ([T] \- C).

eval_these(T,PT,Exp,C) :- nonvar(T),show_case(T,PT,Exp,C).
eval_these(T,_,true,T) :- var(T),circular(T).
```

Gisela—A Framework for Definitional Programming

Olof Torgersson
Department of Computing Science
Chalmers University of Technology and Göteborg University
S-412 96 Göteborg,Sweden
oloft@cs.chalmers.se

Abstract

We describe Gisela, a framework for developing systems based on definitional models. The framework can be seen as a successor to the definitional programming tools GCLA and GCLAII. Compared to these, Gisela was designed to provide for a cleaner definitional programming methodology and to allow for new ideas on programming with definitions not covered by previous systems. Another important goal has been to create a system suitable for use as an embedded deductive database engine in object-oriented applications with GUIs. The computational model and implementation are described, and a number of example programs are given to illustrate how the framework can be used.

1 Introduction

Declarative programming comes in many flavors. There are functional languages, lazy functional languages, logic languages, constraint logic languages, functional logic languages and so on. Common to most of these is the concept of a definition. Function definitions are given, predicates are defined etc. Yet another approach to declarative programming is what we call *definitional* programming. In a definitional program, the definition is the basic notion, not functions, predicates, or constraints. Since both functions and predicates conceptually are given using definitions, taking the definition as the basic notion puts definitional programming at a lower level. This said, we also believe that definitional programming does, and should, provide for a large degree of freedom. Using the tools presented here many different kinds of programs and evaluation strategies can be expressed, although it might take some more work than using a higher-level declarative system. The situation can be compared to imperative languages. A language like

Ada or Java puts programming on a higher level than C. On the other hand, no other widely used imperative language gives the freedom provided by C.

With *Gisela*[1], we have tried to keep the flexibility of the definitional programming tool GCLAII and move on to another level. Gisela should *not* be seen as a programming language with a fixed syntax and semantics, but as a framework for definitional programming. The intention is to provide a set of tools that are useful for realizing definitional (knowledge) models into executable programs. The framework gives an abstract description of definitions in terms of sets and operations. It also provides a general machinery for computing with definitions, which does not depend on the details of how definitions are realized. Depending on the application at hand, the tools may be used to implement programs using a variety of different kinds of definitions and evaluation orders. Furthermore, the framework also contains all the building blocks we need to construct a complete definitional program, if we have no requirements beyond those provided by the Gisela framework.

Theoretically, the basis for all work on definitional programming so far is, in some sense, the theory of *Partial Inductive Definitions* (PID) [25]. Although the definitions and proof-systems presented in this paper differ in many ways from the original PIDs the heritage should be obvious. The model presented here builds on, and borrows from, earlier definitional programming systems, reformulated, augmented and constrained to fit the needs set up as goals for Gisela.

The definitional programming tools GCLA [9, 10] and GCLAII [6, 11, 36], were based on finitary versions [29, 30, 38] of the infinitary PID theory. The basic motivation for the development of these tools was to find a suitable modeling tool for knowledge-based systems. GCLAII was successfully used in a number of applications, including construction planning [7], music theory [47], and reasoning about circuits [24]. It was also used for knowledge representation in the initial phases of the MedView project [1, 28].

As definitional programming evolved, and new demands were set by the MedView project, it became obvious that a replacement for GCLAII was needed. Some of the problems with GCLAII and ideas for a new definitional tool are discussed in [58]. Gisela is the result of our efforts at building this replacement. During development, several of the initial requirements have changed, both with respect to the theoretic model used and the realization as a framework for definitional programming. However, several central ideas remain the same.

Among the goals we have had in mind while developing Gisela are:

- To design a framework for definitional programming rather than a definitional programming language.

[1]In Swedish the preceding definitional language GCLAII was pronounced "Gisela 2". Since GCLA was an acronym for Generalized Horn Clause Language, which did not feel appropriate in the current setting, we kept the way GCLA was pronounced but changed the spelling.

- To describe definitional computations in a sufficiently abstract manner to make the above possible.
- To provide a framework suitable for use for knowledge representation and reasoning in the MedView project.
- To build a framework which can easily be integrated into a modern object-oriented application programming environment.
- To provide a machinery that can be used as a definitional programming language based on a particular concrete syntax.
- To give a description of definitional programming that breaks the links to Prolog present in GCLA.
- To provide a framework that is complete enough to be used as is, but which can also easily be extended. Accordingly, the behavior of Gisela can be modified and extended by providing specialized observers (see Section 4.3) or new definition object classes.
- To keep the distinction between declarative and procedural parts of programs used in GCLAII, thus separating declarative descriptions and control information.
- To allow a more fine-grained definiens operation. The definiens operation as described in [7, 30, 38] is very costly. By implementing several different versions for different tasks, efficiency can be gained in many cases.
- To allow any number of distinct definitions in programs. The GCLAII system used two: the declarative (object) definition and the procedural (rule) definition.
- To allow for new definitional programming ideas [18, 19, 21, 22, 58] while keeping many techniques developed for GCLAII.
- To create a portable implementation.

To meet the goal of smooth integration into a modern object-oriented application programming environment, Gisela is realized as an object-oriented framework for definitional computing. This framework provides a complete object-oriented application programming interface (API) for building definitional components for use in applications, see Section 5.6. The realization as an object-oriented framework also solves the issue of flexibility since it is possible to introduce new classes, or subclass existing ones, to customize the behavior of the system. A second API is provided in terms of equational syntactic representations, which enables the use of Gisela as a "traditional" declarative programming language, see Section 5.1. In addition, the two APIs may be mixed freely.

Gisela is still very much of an ongoing project. However, we believe that the basic design will remain the same and we are developing various applications to test and investigate programming using the Gisela framework.

The general organization of the rest of this paper is that we make a number of iterations through Gisela where each iteration provides more detail than the previous ones. Thus, in Section 2 we give a few examples of programs built using Gisela to give a flavor of the general ideas. In Section 3 we introduce, in a general way, the definitional computation model of Gisela. It is followed by a description of Gisela and its operational semantics in Section 4. Section 5 gives a variety of examples showing how Gisela can be used in various ways. Section 6 refines the computational model into a more fine-grained operational semantics that is more suited as a basis for implementation. In Section 7 an overview of the current implementation is given. Section 8 finally, sums it all up with a discussion of Gisela, its relation to other declarative programming systems, future directions etc.

2 Samples

Programs in Gisela consist of an arbitrary number of *data definitions* and *method definitions*. The data definitions describe the declarative content of the program, and the method definitions give the algorithms, or search strategies, used to compute solutions. A query is built from a method definition and an initial *state definition*. The method definition tells the system how to compute an answer from the initial state definition. The result of a computation is another state definition, referred to as the *result definition*, and an answer substitution for variables in the initial state definition. By default, the result definition is rather complex and contains information, not only about the answer as such, but information about how it was computed as well. Depending on the application, the result definition may be simplified in different ways, since the full definition may not be interesting and building it requires a lot of resources.

All examples in this section are based on the syntactic representations of definitions described in Section 5.1. The example queries have been run using the interactive system discussed in Section 5.8.

2.1 A Toy Expert System

As a first example we consider a toy expert system adopted from [6]. The knowledge base of the system is the following data definition:

```
definition diseases.

symptom(high_temp) = disease(pneumonia).
symptom(high_temp) = disease(plague).
symptom(cough) = disease(pneumonia).
symptom(cough) = disease(cold).
```

The data definition, named `diseases`, contains the connections between symptoms and diseases, but no facts. To ask the system what a possible disease might be, based on observed facts, e.g. symptoms, we form a query using a method definition and an initial state definition. For instance, assume that the patient has the symptom `high_temp`, from which diseases does this follow?

```
G3> lra(diseases){disease(X) = symptom(high_temp)}.
```

The meaning of the query is "use the method definition `lra` instantiated with the domain knowledge in the data definition `diseases` to compute a result definition from the initial state definition `{disease(X) = symptom(high_temp)}`". Generally, the form of a query is $m\{e_1, \ldots, e_n\}$ where m is the method definition to use to compute a result from the initial state definition $\{e_1, \ldots, e_n\}$.

Now, let us run the above query and have a look at the result:

```
G3> lra(diseases){disease(X) = symptom(high_temp)}.

X = pneumonia,

{
[lra,r:D] = {
  [D] = {
    disease(pneumonia) = disease(pneumonia)
    }
  }
}
?
```

The question mark at the end of the system's response indicates that there may be more answers to the query. Typing a semi-colon will cause the system to attempt to compute the next answer. What has been computed here is the full result definition for the query. The result definition can be viewed as a partial trace of the computation. More details are given in Section 3. As with most examples adopted from GCLA, the result definition is of no particular interest in this case. Therefore, we ask the system to display only the computed answer substitution and re-run the query:

```
G3> restype(vars_only).
G3> lra(diseases){disease(X) = symptom(high_temp)}.
X = pneumonia
? ;
X = plague
? ;
no
```

The answer tells us that `high_temp` could be caused by `pneumonia` or `plague`.

The method definition `lra` is actually a reusable *method-scheme* that can be instantiated with different data definitions:

```
method lra:[D].

lra = [lra, l:D] # some l:in_dom(D).
lra = [lra, r:D] # some r:in_dom(D) & all not(l:in_dom(D)).
lra = [D] # all not(l:in_dom(D); r:in_dom(D)).
```

The first line states that `lra` is a method definition that takes one parameter, a data definition `D`. The remaining lines are three equations that describe the behavior implemented by `lra`. The general form of equations in method definitions is $M = Condition\#Guard$, where *Guard* decides if the equation can be applied and *Condition* describes possible sequences of operations to perform.

The method `lra` attempts to replace left and right-hand sides of equations in the current state definition using the data definition `D`. If this is not possible, the third equation of `lra` will try to unify the left and right-hand side of some equation in the state definition. If no equation of `lra` can be applied, the answer to the query is `no`.

One way to view `lra` is as a method definition implementing a subset of the general inference machinery of GCLA. More on how Gisela can be used for GCLA-style programming can be found in Section 5.2.

2.2 Logic Programming

Pure Prolog [51] is a subset of Gisela, just as it is a subset of GCLA [29]. Pure Prolog programs can be transformed into valid Gisela definitions simply by substituting '`=`' for '`:-`' throughout.[2] Some examples of Prolog-style predicates are given in the following data definition:

```
definition lpsample.

permutation([], []).
permutation([X|Xs], [Y|Ys]) =
    select(Y, [X|Xs], Zs),
    permutation(Zs, Ys).

select(X, [X|Xs], Xs).
select(Y, [X|Xs], [X|Ys]) = select(Y, Xs, Ys).

hanoi(s(0), A, B, C, [mv(A,B)]).
hanoi(s(N), A, B, C, Moves) =
```

[2]Actually, Prolog programs can be handled directly, as is, by a special definition class.

```
    hanoi(N, A, C, B, Ms1),
    hanoi(N, C, B, A, Ms2),
    append(Ms1, [mv(A,B)|Ms2], Moves).

append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) = append(Xs, Ys, Zs).
```

The predicate `append/3` is the canonical logic programming example of how predicates can be used in several modes. We use a method definition called `prolog`, shown below, to try out logic programming in Gisela. Again, we have set the system to show only the substitution part of the answer:

```
G3> prolog(lpsample){true = append([a,b],[c],Xs)}.
Xs = [a,b,c]
? ;
no

G3> prolog(lpsample){true = append(Xs, Ys, [a,b])}.
Xs = [],
Ys = [a,b]
? ;
Xs = [a],
Ys = [b]
? ;
Xs = [a,b],
Ys = []
? ;
no
```

Another query using the definition `lpsample` and the method `prolog` is

```
G3> prolog(lpsample){true = permutation([a,b,c],L)}.
```

The intended reading of this query is "is there an `L` such that `L` is a permutation of `[a,b,c]`". The computed answer substitutions are the six possible permutations of `[a,b,c]`. The result definition, again, is of no particular interest.

Finally, the predicate `hanoi/5` solves the well-known towers of Hanoi problem. The problem is to move a tower of n disks from one peg to another with the help of an auxiliary peg. Only one disk can be moved at a time and a larger disk can never be placed on top of a smaller disk. The first argument of `hanoi` is the number of disks to move. The result is a list of moves where `mv(A,B)` means "move the top disk from A to B".

To solve the problem for 3 disks we run the query

```
G3> prolog(lpsample){true = hanoi(s(s(s(0))), a, b, c, Moves)}.
```

```
Moves =
[mv(a,b),mv(a,c),mv(b,c),mv(a,b),mv(c,a),mv(c,b),mv(a,b)]
? ;

no
```

which computes a single answer just as we would expect it to do.

The method definition `prolog` is very simple:

```
method prolog:[P].

prolog = [] # some r:matches(true).
prolog = [prolog, r:P] # all not(r:matches(true)).
```

Note that it is assumed that the initial state definition used contains only one equation. Otherwise, the query does not correspond to a Prolog query. What `prolog`, parameterized with the program definition `P`, does is simply to apply the correct computation rule (see Sections 3.5.1 and 4.2) to the right-hand side of the the single equation of the state definition as long as it does not equal `true`. When the right-hand side equals `true` the query is proved and evaluation stops.

2.3 Functional Evaluation

As in GCLA, a kind of (first-order) functional programming is possible in Gisela. If we have the data definitions

```
definition nats:matching.

zero = zero.
s(X) = s(X).

definition plus:matching.

plus(zero, N) = N.
plus(s(M), N) =
    (plus(M, N) -> K)
    -> s(K).
```

and a method definition `fun`, which takes two parameters, a definition defining data objects and a definition defining functions, the query

```
G3> fun(nats, plus){plus(s(zero),s(zero)) = X}.
```

will compute the expected answer substitution `{X = s(s(zero))}`. The slightly complex method definition `fun` is not shown here. If we combine logic programming with functional evaluation we get *functional logic* programming. General programming methodology and method definitions for functional logic programming using the Gisela framework are discussed in Section 5.5.

2.4 Hamming Distance

All the examples above are adopted from GCLA programs. Since GCLA is essentially an extension to logic programming, the interesting part of the answer is the computed answer substitution for variables in the initial state definition. One of the objectives of Gisela is to allow for other ways of computing with definitions, where the computed result definition is the interesting part of the answer. Studying properties of definitions, such as similarity, is an example of this.

Hamming distance is a notion generally used to measure difference with respect to information content. The Hamming distance between two code-words, for example `001001110110` and `101100101100`, is the number of positions where the words differ, in this case six. If the Hamming distance between two words is d it takes d simple bit-transformations to transform one word into the other.

In this example we let each code-word be represented by a data definition. Thus, the word `1101` is represented by

```
definition w1.

w(0) = 1.
w(1) = 1.
w(2) = 0.
w(3) = 1.
```

and the word `0110` by:

```
definition w2.

w(0) = 0.
w(1) = 1.
w(2) = 1.
w(3) = 0.
```

To compute the Hamming distance between `w1` and `w2` we ask the query

```
G3> lr(w1,w2){w(0)=w(0), w(1)=w(1), w(2)=w(2), w(3)=w(3)}
```

which computes the result definition `{1=0, 1=1, 0=1, 1=0}`. What we have computed is, so to speak, how *similar* `w1` and `w2` are. From this similarity measure, it is easy to see that the Hamming distance is 3. Definitional similarity measures are described in [20], another example using Gisela can be found in Section 5.4.

The method `lr` expands the initial state definition as far as possible by replacing atoms according to the actual data definitions used. When a state where no equation can be changed is reached the computation stops:

```
method lr:[L,R].

lr = [lr,l:L] # some l:in_dom(L).
lr = [lr,r:R] # (all(not(l:in_dom(L))) & some(r:in_dom(R))).
lr = [] # all((not(l:in_dom(L)) , not(r:in_dom(R)))).
```

The first equation of `lr` can be read as follows: "If the left-hand side of some equation in the current state definition is in the domain of the definition `L`, then use `L` to replace the left-hand side of some equation by its definiens and continue the computation using the method definition `lr`".

2.5 Database Search

Imagine a database built as a large number of data definitions:

```
definition record1.
id = t(14).
status = active.

definition record2.
id = t(23).
status = passive.

definition record3.
id = t(11).
status = active.
...
```

A suitable method definition here will be one that replaces right-hand sides in the state definition until both sides are equal in some equation:

```
method sri:[Record].

sri = [sri, r:Record] # some r:in_dom(Record) & all not(identity).
sri = [] # some identity.
```

The guard of the first equation of `sri` holds if the right-hand side of some equation in the state definition is in the domain of `Record` and no equation is an identity. The guard of the second equation holds if some equation in the state definition is an identity.

Now, if we wish to find all records in the database which have the value `active` for the attribute `status` we can instantiate a method-scheme with a *parameter set* instead of, as in earlier examples, with a single definition. This will have the effect that all the definitions in the parameter set are used to create instances of the method `sri`. A query could be:

```
sri(R <- records){active = status}.
```

The query has the answers:

```
{active = active}, R = record1,

{active = active}, R = record3
```

which tells us that `record1` and `record3` are the ones we are looking for.

We conclude this section by showing how the above query can be setup and run in Objective-C using the Gisela framework. Assuming that `sri` is an object representing the method-scheme `sri`, `recordDB` contains all the records in the database, and that `state` is the initial state definition, the following will collect all matching records (definitions) in the array `matches`:

```
// (1) Some declarations
DFDMachine *dMachine;
DFQuery *query;
DFAnswer *answer = nil;
NSMutableArray *matches = [NSMutableArray array];

// (2) Create Gisela machine.
dMachine = [[DFDMachine alloc] init];
// Create query.
query = [[DFQuery alloc] initWithMethodScheme:sri
                                 stateDefinition:state
                                 andParameterSets:recordDB];
// (3) Set query and run while there are answers.
[dMachine setQuery:query];
while (answer = [dMachine nextAnswer]) {
   [matches addObject:[[answer parameterValues] objectAtIndex:0]];
}
```

All entities used in definitional computations can be represented using objects of various classes provided by the Gisela framework. A small subset of these classes are used in the present example. For example, the code following `(2)` creates a machine for definitional computations and a query object set up to represent the database search query shown above. The code following `(3)` tells the definitional machine to use the created query and run it as long as more answers can be computed.

3 Computing with Definitions

In this section we present the basic notions of definitions and computations using definitions which form the basis of Gisela. Many notions are shared with previous work on PID and definitional programming. However, the differences compared to earlier work with respect to both terminology, ideas, and presentation are significant enough to motivate a separate description. The presentation used is one without variables and is in many ways similar to those in [20, 22, 27, 28], with some significant extensions. In Section 4 variables are introduced and the system refined to provide an operational semantics for Gisela.

In computations we will consider three different kinds of definitions: *data definitions*, *state definitions*, and *method definitions*. We first describe what a definition is, generally, and then proceed to describe the different definition types and their respective roles in computations.

3.1 Definitions

A definition D is given by

1. two sets: the *domain* of D, written $\mathit{dom}(D)$, and the *co-domain* of D, written $\mathit{com}(D)$, where $\mathit{dom}(D) \subseteq \mathit{com}(D)$,

2. and a *definiens* operation: $D : \mathit{com}(D) \to \mathcal{P}(\mathit{com}(D))$.

Objects in $\mathit{dom}(D)$ are referred to as *atoms* and objects in $\mathit{com}(D)$ are referred to as *conditions*.

A natural presentation of a definition is that of a (possibly infinite) system of equations

$$D \begin{cases} a_0 & = & A_0 \\ a_1 & = & A_1 \\ & \vdots & \\ a_n & = & A_n \\ & \vdots & \end{cases} \qquad n \geq 0,$$

where atoms, $a_0, \ldots, a_n, \ldots \in \mathit{dom}(D)$, are defined in terms of a number of conditions, $A_0, \ldots, A_n, \ldots \in \mathit{com}(D)$, i.e., all pairs (a_i, A_i) such that $A_i \in D(a_i)$, and $a_i \in \mathit{dom}(D)$. Note that an equation $a = A$ is just a notation for A being a member of $D(a)$. Expressed differently, the left-hand sides in an equational presentation of a definition D are the atoms for which $D(a_i)$ is not empty.

Given a definition D the presentation as a system of equations is unique modulo the order of equations. However, given an equational presentation of a definition it is not generally possible to determine which definition the equations represent. The reason for this is that it is not possible to decide the domain and co-domain of a definition from its equational presentation. When an equational presentation of a definition D is given without further specifying $dom(D)$ and $com(D)$, it is assumed that the definition is uniquely determined by its presentation.

Intuitively, the definiens operation gives further information about its argument. For an atom a, $D(a)$ gives the conditions defining a, that is, $D(a) = \{A \mid (a = A) \in D\}$. For a condition $A \in com(D) \setminus dom(D)$, $D(A)$ gives the constituents of the condition. For example, $D((A \rightarrow B)) = \{A, B\}$.

It should be kept in mind that although we frequently use equational presentations of definitions, a definition is any object which adheres to 1 and 2 above.

3.1.1 Operations on Definitions

We will use some primitive operations on definitions:

- $(A/B)D$, is the definition given by replacing all left-hand sides of D identical to B with A.
- $D(A/B)$, is the definition given by replacing all right-hand sides of D identical to B with A.
- $D \downarrow A$. If A is a condition and D a definition, then if $D \neq \emptyset$ then $D \downarrow A$ is the definition given by:

  1. $dom(D \downarrow A) = dom(D) \cup \{\top\}$, $com(D \downarrow A) = com(D) \cup \{A\}$,
  2. $D \downarrow A(a) = \{A\}$ for all $a \in dom(D)$ such that $D(a) \neq \emptyset$,

  else $D \downarrow A$ is $\{\top = A\}$.
- $A \ominus D$. If A is a condition and D a definition then $A \ominus D$ is the definition given by:

  1. $dom(A \ominus D) = dom(D)$, $com(A \ominus D) = com(D)$,
  2. $A \ominus D(A) = \emptyset$, $A \ominus D(C) = D(C)$ for $C \neq A$.
- $A \oplus D$. If A is a condition and D is a definition then $A \oplus D = \top \ominus (A \oplus' D)$ where $A \oplus' D$ is the definition given by

  1. $dom(A \oplus' D) = dom(D) \cup \{A\}$, $com(A \oplus' D) = com(D) \cup \{A\}$,
  2. $A \oplus' D(A) = \bigcup_{b \in dom(D)} D(b)$, $A \oplus' D(B) = D(B)$ for $B \neq A$.
- $D_1 + D_2$. If D_1 and D_2 are definitions then $D_1 + D_2$ is the definition given by:

  1. $dom(D_1 + D_2) = dom(D_1) \cup dom(D_2)$, $com(D_1 + D_2) = com(D_1) \cup com(D_2)$,
  2. $D_1 + D_2(A) = D_1(A) \cup D_2(A)$.

3.2 Data Definitions

Data definitions are used to model declarative knowledge. These definitions are the building blocks which computations operate on.

In principle there could be a great number of different kinds of conditions. In the present work we will use the following to define the set $\mathcal{C}$ of all conditions:

1. all atoms are conditions,
2. $\top$ and $\bot$ are conditions,
3. if A and B are conditions then (A, B) and $(A \rightarrow B)$ are conditions.

For any data definition D, the definiens of $A \in (com(D) \setminus dom(D))$ is defined as follows:

1. $D(\top) = \emptyset$,
2. $D(\bot) = \emptyset$,
3. $D((A, B)) = \{A, B\}$,
4. $D((A \rightarrow B)) = \{A, B\}$.

3.3 State Definitions

A computation is a transformation of an initial state definition into a final state definition. State definitions will always be considered with respect to given data definitions. We make a distinction between ordinary state definitions and result definitions. State definitions are the initial state definition and all following definitions representing the state of a computation. Result definitions are used for answers.

3.3.1 State Definition Details

The domain and co-domain of state definitions is the union of all the co-domains of all the data definitions used in a computation. Expressed in another way, all conditions as described in Section 3.2. We will generally denote state definitions $S, S_1, S_2 \ldots$ and write them as a sequence of equations:

$$\{e_1, e_2, \ldots, e_n\}.$$

For example

$$\{a = b, (c, d) = b, e \rightarrow b = b, f = b\}.$$

3.3.2 Result Definitions

All result definitions are uniquely determined by their equational presentations. In result definitions the right-hand side of an equation can be another result definition. Thus, a result definition can contain other result definitions nested within itself. We will use $X, X_1, X_2 \ldots$ to denote result definitions.

In the general case, the result of a computation is rather complex. Not only does it contain a number of equations that may be viewed as being the answer to a query, but also information on *how* the result was computed.

We illustrate with an example, a result definition nested several levels:

$$X \left\{ cm\overline{R_1} \quad = \left\{ \begin{array}{lcl} white & = & \left\{ cm\underline{R_2} \;=\; \left\{ \epsilon \;=\; \{white = white \right.\right. \\ brown & = & \left\{ cm\underline{R_2} \;=\; \left\{ \epsilon \;=\; \{brown = white \right.\right. \end{array} \right.\right. \tag{1}$$

The right-most equations in (1) are the end-points, or final equations, of the computation. The rest essentially contains information about where the computation was split into different branches. The details for how result definitions are constructed is defined by the computation rules in Section 3.5.1.

In most cases, we are only interested in the final equations, not the complete structure of the result definition. Thus, result definitions can be transformed to give the representation most suited for a particular application. Examples of transformations are $flatten(X)$, which gives a definition containing the leaf equations of a nested definition only, and $null(X)$ which gives the empty definition $\{\}$. Applying $flatten$ to (1) gives:

$$X \left\{ \begin{array}{lcl} white & = & white \\ brown & = & white \end{array} \right.$$

A flattened result definition is a valid state definition and can therefore be used as the initial state definition for a new computation. Also, flattened result definitions where all left-hand sides are atoms are valid data definitions for use in computations.

3.3.3 Definitions as a Generalization of Sequents

PID and GCLA use sequent calculus notation. In Gisela we try to use definitions as the only structure wherever possible. Thus, sequents have been replaced by state definitions. Each sequent of PID or GCLA can be represented as a state definition. Also, compared to GCLA, state definitions generalize sequents to include what would be sequents with an arbitrary number of consequents.

Instead of describing how sequents correspond to definitions, we give some examples from which the general idea should be obvious. The sequent

$$a \vdash b$$

corresponds to the state definition

$$\{a = b\}$$

and the sequent

$$a, b, c \vdash d$$

to the state definition

$$\{a = d, b = d, c = d\}.$$

Along the same lines, a sequent calculus rule such as

$$\frac{a, C \vdash b}{C \vdash a \to b}$$

can be represented as

$$\frac{\{a = b, C = b\}}{\{C = a \to b\}}\ .$$

We use $\top$ to write sequents with an empty set of conditions in the antecedent. Thus

$$\frac{A \vdash B}{\vdash A \to B}$$

yields

$$\frac{\{A = B\}}{\{\top = A \to B\}}.$$

Since $\top$ simply is the representation corresponding to an empty antecedent it is not part of the premise of the rule.

3.4 Method Definitions

A method definition describes the sequence of steps to be performed in a computation. The description can be more or less precise. We may have method definitions that set up general search strategies, or method definitions that in great detail describe what to do next, given the current state definition. We say that a method definition defines a *computation method.*

3.4.1 Method Definition Details

Let $\mathcal{V}$ be a set of atoms (computation method names). Given a set of data definitions $\mathcal{D}$, let $\mathcal{O}$ be a set of formal notations $\overline{D}, D, \underline{D}$ for all definitions D in $\mathcal{D}$. Let $\mathcal{W}$ be the set of all computation words over $\mathcal{V}$ and $\mathcal{O}$: $\mathcal{W} = (\mathcal{V} \cup \mathcal{O})^*$. The empty word is denoted by ϵ.

The set of computation conditions, $\mathcal{WC}$, for use in method definitions is defined as follows:

1. All words in $\mathcal{W}$ are computation conditions.

2. If W_1 and W_2 are computation conditions then (W_1, W_2) is a computation condition.

3. If W_1 and W_2 are computation conditions then W_1W_2 is a computation condition.

A method definition is a definition with $\mathcal{V}$ as its domain and $\mathcal{WC}$ as its co-domain.

A method definition m can be presented as a system of (guarded) equations:

$$m \begin{cases} m &= W_1 \;\#\; C_1 \\ m &= W_2 \;\#\; C_2 \\ &\vdots \\ m &= W_n \;\#\; C_n \end{cases}$$

where each condition $W_i \in \mathcal{WC}$, and each guard C_i is a boolean function that is used to decide whether the equation can be applied or not.

Given a data definition, D, we refer to the word constituents D, $\overline{D}$, and $\underline{D}$ as *operations* on D. For the sake of simplicity we assume that each computation method is defined in a method definition with the same name as the method. That is, the only atom defined in a method named m is m. The meaning of the operations is given by the calculus in Section 3.5.1.

A method acts on the present state definition. We could think of it as that there is a hidden argument present in method definitions:

$$m \begin{cases} m(S) &= W_1 \;\#\; C_1(S) \\ m(S) &= W_2 \;\#\; C_2(S) \\ &\vdots \\ m(S) &= W_n \;\#\; C_n(S) \end{cases}$$

3.5 Computations

We now give a presentation of what it means to compute a result definition X given an initial state definition S and a computation condition W. We write $W\colon S \Rightarrow X$, meaning "$W\colon S$ can be computed to X". We call $W\colon S \Rightarrow X$ a *goal*. Depending on the application at hand, we will interpret X and $W\colon S$ in different ways. For instance, X can be taken as a measure of the *distance* between definitions with respect to S and computation methods used, or we can view $W\colon S$ as a logic programming goal to be proved, in which case only result definitions where some right-hand side is $\top$ will be accepted. In any case, "$W\colon S$ can be computed to X" means that we try to move from the initial state definition S to a result definition X using W. This may fail, which means that W could not be used to move from S to X.

The possible computation steps are given using a number of inference rules. The presentation is aimed mainly at making the intuition of definitional computing in Gisela clear. A similar, but fully detailed, calculus for Gisela is given in

Section 4.2. An even more fine-grained version, presented as a number of rewrite rules more suitable as a basis for implementation, is given in Section 6.1.

3.5.1 Computation Rules

In all rules D denotes *any* data definition and M *any* computation method.

(1) Termination

$$\frac{}{\epsilon{:}\, S \Rightarrow S}\; T \quad .$$

(2) Method

$$\frac{WW_1{:}\, S \Rightarrow X_1, \ldots, WW_n{:}\, S \Rightarrow X_n}{WM{:}\, S \Rightarrow X}\; M$$

where $\mathcal{M}(M) = \{W_1, \ldots, W_n\}, n \geq 1$. $\mathcal{M}(M)$ is the definiens of M in the method $\mathcal{M}$, that is

$$\{W_i \mid M = W_i \# C_i \in \mathcal{M} \wedge C_i(S)\}$$

and X is the result definition

$$X \begin{cases} W_1 & = & X_1 \\ & \vdots & \\ W_n & = & X_n \end{cases} .$$

(3) Choice

$$\frac{WW_i{:}\, S \Rightarrow X}{W(W_1, W_2){:}\, S \Rightarrow X}\; C$$

where $W_i \in \{W_1, W_2\}$.

(4) Definition Left

Let $e \in S$. Then, depending on the left-hand side of e we have:

(4.1) If $e = (a = C)$

$$\frac{W{:}\, (A_1/a)S \Rightarrow X_1, \ldots, W{:}\, (A_n/a)S \Rightarrow X_n}{W\overline{D}{:}\, S \Rightarrow X}\; \overline{D}_D$$

where $D(a) = \{A_1, \ldots, A_n\}$ and X is the result definition

$$X \begin{cases} A_1 & = & X_1 \\ & \vdots & \\ A_n & = & X_n \end{cases} .$$

(4.2) If $e = ((A, B) = C)$

$$\frac{W\colon (C'/(A,B))S \Rightarrow X}{W\overline{D}\colon S \Rightarrow X}\ \overline{D}_V$$

where $C' \in D((A, B))$.

(4.3) If $e = ((A \to B) = C)$

$$\frac{W\colon S_1 \Rightarrow X_1 \quad W\colon S_2 \Rightarrow X_2}{W\overline{D}\colon S \Rightarrow X}\ \overline{D}_A$$

where S_1 and S_2 are given by

- $S_1 = ((A \to B) \ominus S) \downarrow A)$,
- $S_2 = (B/(A \to B))S$,

and X is the result definition

$$X \left\{ \begin{array}{lcl} A & = & X_1 \\ B & = & X_2 \end{array} \right. .$$

(5) Definition Right

Let $e \in S$. Then, depending on the right-hand side of e we have:

(5.1) If $e = (A = a)$

$$\frac{W\colon S(B/a) \Rightarrow X}{W\underline{D}\colon S \Rightarrow X}\ \underline{D}_D$$

where $B \in D(a)$.

(5.2) If $e = (A = (B, C))$

$$\frac{W\colon S(A/(B,C)) \Rightarrow X_1 \quad W\colon S(B/(B,C)) \Rightarrow X_2}{W\underline{D}\colon S \Rightarrow X}\ \underline{D}_V$$

where X is the result definition

$$X \left\{ \begin{array}{lcl} A & = & X_1 \\ B & = & X_2 \end{array} \right. .$$

(5.3) If $e = (A = (B \to C))$

$$\frac{W\colon S' \Rightarrow X}{W\underline{D}\colon S \Rightarrow X}\ \underline{D}_A$$

where $S' = B \oplus (S(C/(B \to C)))$.

(6) Identity

Let $e \in S$. If $e = (a = a)$ for some a then:

$$\frac{W{:}\, S \Rightarrow X}{WD{:}\, S \Rightarrow X}\ I\ .$$

3.5.2 Comments

Note that the rules (1) through (6) only describe how state definitions and computation conditions connect to each other and that the empty word means that a computation terminates. In particular, there are no rules for the conditions $\bot$ and $\top$. If we wish to interpret these in a special way, for instance as *false* and *true*, the interpretation has to be given in a method definition. Another way to explain computations is that the given rules describe what state definitions may be generated given an initial state definition S and a computation condition W. The rules (4.1) and (5.1) connect computation methods to the data definitions used in method definitions. The set of definitions that can be generated from a state definition is thus given by the above *inference rules*, the *form* of the method definitions involved, and the *contents* of the particular data definitions used in methods.

The computation system described shares many properties with PID and the definitional programming system GCLA, most notably the duality between the left and right-hand sides of equations. Also, the rules are very similar if we look at which sequents the different state definitions represent. However, state definitions are more general in nature than sequents, and, as mentioned, a method definition is necessary to further describe the permitted computation steps.

3.6 An Example

Consider the two data definitions R_1 and R_2

$$R_1 \left\{ \begin{array}{l} status = direct \\ direct = mucos \\ direct = palpation \\ mucos = mucos\_site \\ mucos = mucos\_col \\ mucos\_site = 112 \\ mucos\_col = white \\ mucos\_col = brown \\ palpation = palp\_site \\ palp\_site = 112 \end{array} \right. \qquad R_2 \left\{ \begin{array}{l} status = direct \\ direct = mucos \\ direct = palpation \\ mucos = mucos\_site \\ mucos = mucos\_col \\ mucos\_site = 232 \\ mucos\_col = white \\ palpation = palp\_site \\ palp\_site = 242 \end{array} \right. ,$$

which are adoptions of examination records from the MedView project. We will investigate the similarity of R_1 and R_2 with respect to to the attribute *mucos_col*. To this end we need a computation method. A typical method definition for this kind of computation using the definitions R_1 and R_2 is:

$$cm \left\{ \begin{array}{lcll} cm & = & cm\overline{R_1} & \#\ \ 'eq \in dom(R_1) \\ cm & = & cm\underline{R_2} & \#\ \ eq' \in dom(R_2) \wedge \neg 'eq \in dom(R_1) \\ cm & = & \epsilon & \#\ \ otherwise \end{array} \right. . \tag{2}$$

If S is the state definition to which cm is applied, we may interpret (2) as follows: If the left-hand side of some equation in S ($'eq$) is in the domain of R_1, then replace it with its definiens and continue computing using cm. Otherwise, if the right-hand side of some equation in S is in the domain of R_2, then replace it with a condition from its definiens and continue computing using cm. Otherwise, end the computation. Thus, what cm does is to replace atoms according to the definitions R_1 and R_2 until the state definition can no longer be changed.

If we apply cm to the state definition $\{mucos\_col = mucos\_col\}$, that is, we compute the goal $cm : \{mucos\_col = mucos\_col\} \Rightarrow X$, the answer X is the following result definition:

$$X \left\{ \begin{array}{lcl} cm\overline{R_1} & = & \left\{ \begin{array}{lcl} white & = & \left\{ \begin{array}{lcl} cm\underline{R_2} & = & \left\{ \begin{array}{lcl} \epsilon & = & \{white = white \end{array} \right. \end{array} \right. \\ brown & = & \left\{ \begin{array}{lcl} cm\underline{R_2} & = & \left\{ \begin{array}{lcl} \epsilon & = & \{brown = white \end{array} \right. \end{array} \right. \end{array} \right. \end{array} \right. .$$

We also show a derivation. All result definitions are abbreviated with some X_i:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon\colon \{brown = white\} \Rightarrow X_6}\,T}{cm\colon \{brown = white\} \Rightarrow X_4}\,M}{cm\underline{R_2}\colon \{brown = mucs\_col\} \Rightarrow X_4}\,D_D}{cm\colon \{brown = mucs\_col\} \Rightarrow X_2}\,M \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon\colon \{white = white\} \Rightarrow X_7}\,T}{cm\colon \{white = white\} \Rightarrow X_5}\,M}{cm\underline{R_2}\colon \{white = mucs\_col\} \Rightarrow X_5}\,D_D}{cm\colon \{white = mucs\_col\} \Rightarrow X_3}\,M}{\dfrac{cm\overline{R_1}\colon \{mucs\_col = mucs\_col\} \Rightarrow X_1}{cm\colon \{mucs\_col = mucs\_col\} \Rightarrow X}\,M}\,D_D$$

Note that we also allow *method-schemes* which are parameterized method definitions. A method-scheme is one that covers all methods differing only with respect to the data definitions used. The parameterized version of the method cm with parameters L and R is written

$$cm_{L,R} \left\{ \begin{array}{lcll} cm & = & cm\overline{L} & \#\ \ 'eq \in dom(L) \\ cm & = & cm\underline{R} & \#\ \ eq' \in dom(R) \wedge \neg 'eq \in dom(L) \\ cm & = & \epsilon & \#\ \ otherwise \end{array} \right. .$$

At the time of computation, this scheme must be instantiated with particular data definitions for the parameters. Thus, the instance cm_{R_1,R_2} of $cm_{L,R}$ is identical to the method cm.

4 Gisela—Programs and Computations

Section 3 showed the principles of computations in the Gisela framework. However, several things were left out, e.g., the treatment of variables and how choices are made. A more complete description is given here.

The basic computation model provided by the Gisela framework is a very general one, allowing for several different approaches for how to program using definitions. In part, this generality is achieved by leaving certain choices in the description open to be handled by an *observer.* The observer is an abstract concept. In any particular case it might be the user running a program, an intelligent software agent or, as in most applications developed so far, a simple object returning default choices. The other important thing is that definitions are described in an abstract way only. Thus, any object which fits into the abstract description is a valid definition to use in a program.

The main components involved in the description of computations are:

- Data Definitions. Compared to the description above, data definitions in Gisela allow logical variables. A data definition may be created in several ways, one of them being the syntactic representations given in Section 5.1.
- State Definitions. As before, but can contain variables.
- Method Definitions. The description of method definitions provided in Section 3.4 is sufficient in this section also. Details of how to create method definitions in the Gisela framework are given in Sections 5.1.2 and 5.6.2.
- Queries.
- An observer. The observer handles choices as mentioned above.
- A D-Machine. Computations are performed by a *D-Machine.* The behavior of this machine is given by the operational semantics in Section 4.2. This operational semantics involves an observer.

Compared to the presentation in Section 3, what is added in this section is variables in data definitions and state definitions, the notion of an observer, and information on the order in which things are computed. The rules of the operational semantics in Section 4.2 together with an observer define how computations are performed. The default observer is described in Section 4.3.

4.1 Gisela Programs

A program in the Gisela framework consists of a number of data and method definitions. The data definitions are used to describe the declarative content of an application and the method definitions define how solutions should be computed. Expressed differently, the data definitions give connections between atoms and conditions and method definitions describe the possible sequences of operations, or applications of the built-in computation rules, a program can perform.

To run a program we pose a query $M{:}S \Rightarrow X$. The meaning of this is "can S be computed to *some* result system X using the method M". If the computation is successful, we take X and any bindings for variables in S to be the answer to the query. Otherwise, the answer is no. Of course, computations may not terminate. A computation requires an observer to handle choices left open in the basic computation rules. The same query run with different observers can give different sets of answers. The power of the observer is restricted to making choices. Thus, a complete search through all possible alternatives will include all answer sets given by different observers. If no particular observer is provided choices are handled left to right and from top to bottom with backtracking as discussed in Section 4.3. Since search is performed depth-first with backtracking the actual computing machinery may fail to find existing solutions.

4.1.1 Data Definitions

We have chosen to define the computation model of Gisela using an abstract description only of what a data definition is. This is because we want to provide a framework where users are free to create data definitions with as few restrictions as possible. Also, a more detailed description is not really needed. Of course, this means that we cannot here give any details for how the definiens operation is computed. More details on this for certain definition classes are provided in Section 5.1 below.

Atoms, Terms, Constants, and Variables We start with an infinite signature, Σ, of *term constructors* and a denumerable set, $\mathcal{V}$, of *variables.* We write variables starting with a capital letter. Each term constructor has a specific arity, and there may be two different term constructors with the same name but different arities. The term constructor t of arity n is written t/n. The arity will be omitted when there is no risk of ambiguity. A *constant* is a term constructor of arity 0. The set $\mathcal{T}$ of all *terms* is built up using variables and constants as follows:

1. all variables are terms,
2. all constants are terms,
3. if f is a term constructor of arity n and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

An *atom* is a term which is not a variable.

Conditions The set $\mathcal{C}$ of all conditions is given by:

1. $\top$ and $\bot$ are conditions,
2. all terms are conditions,
3. if A and B are conditions then (A, B) and $(A \rightarrow B)$ are conditions.

Substitutions A *substitution* is a (possibly empty) finite set of equalities

$$\{(x_1 = t_1), (x_2 = t_2), \ldots, (x_n = t_n)\}$$

where each $x_i \in \mathcal{V}$, $t_i \in \mathcal{T}$, $\forall i(x_i \neq t_i)$, and $\forall i, j(x_i = x_j \rightarrow i = j)$. We use σ, τ, ϕ, σ_1,... to denote substitutions.

Definitions To describe data definitions in the presence of variables we make some minor modifications to the definition given in Section 3. Thus, a definition D is given by

1. two sets: the *domain* of D, written $dom(D)$, and the *co-domain* of D, written $com(D)$, where $dom(D) \subseteq com(D)$, also $dom(D) \subseteq \mathcal{T}$ and $com(D) \subseteq \mathcal{C}$,
2. and a *definiens* operation: $D : com(D) \rightarrow \mathcal{P}(com(D))$.

Let $\mathcal{VD}$ be the set of all variables in $com(D)$. We assume that for all data definitions D_i and D_j, $i \neq j$, $(\mathcal{VD}_i \cap \mathcal{VD}_j) = \emptyset$. Further, we assume that the variables occurring in state definitions are not part of $\mathcal{VD}$ for any definition D and that variables can be renamed to make sure that these conditions hold.

Given a term a, a substitution σ is called *a-sufficient* if $D(a\sigma)$ is closed under further substitution, that is, for all substitutions τ, $D(a\sigma\tau) = (D(a\sigma))\tau$.

For any data definition D we assume that the following can be computed:

1. $D_{suff}(a)$, which is a sequence of the a-sufficient substitutions for a with respect to D.
2. $D_{mgu}(a)$, which is a sequence of the most general unifiers (mgus) [41] between a and $b \in dom(D)$ such that $D(b) \neq \emptyset$.

On a-sufficient substitutions Given an a-sufficient substitution the definiens of a is completely determined. There can be more than one definiens of a however, since there may be several a-sufficient substitutions.

With the completely abstract and variable-free system used in Section 3 it was easy to state what $D(a)$ should be. When variables are introduced the situation becomes more complex. The situation has an exact parallel in GCLA where the infinitary PID calculus is replaced by a system with variables. The problem was first investigated in [30] where the notion a-sufficiency was introduced. Algorithms for computing a-sufficient substitutions for definitions based on equational presentations can be found in [8, 30, 38].

4.1.2 Method Definitions

Conceptually, method definitions correspond to the rule definition of GCLAII. However, they are expressed and operate in a completely different manner. Also, Gisela works with a fixed set of inference rules given below in Section 4.2. Thus, what can be expressed in method definitions is which rule or computation method to use given the current state definition. The description of method definitions in Section 3.4 is sufficient to describe the operational behavior of Gisela.

Note that there are no variables in method definitions. Instead, in a method-scheme like

$$m_D \left\{ \begin{array}{lllll} m & = & m\overline{D} & \# & {}'eq \in dom(D) \\ m & = & m\underline{D} & \# & eq' \in dom(D) \wedge \neg{}'eq \in dom(D) \\ m & = & D & \# & otherwise \end{array} \right.$$

we have a *parameter* D, see also Section 3.6. Parameters are a notational convenience only. Before a computation starts the parameters must be replaced by the actual data definitions to use in the computation.

4.1.3 State Definitions

State Definitions and result definitions are as in Section 3.3.1, with the addition of variables. The scope of a variable is the entire goal in which it occurs.

4.1.4 Queries

A query is simply a goal $W: S \Rightarrow X$. The answer to the query is the result definition X and a substitution σ with bindings for variables in the initial state definition S. The purpose of a query is to compute a *result* X *from* $W{:}S$.

4.2 Operational Semantics

We give an operational semantics to describe how computations are performed. The operational semantics is expressed as a number of inference rules operating on computation states. The following notations are used:

- A computation state is a tuple $< \Gamma, \quad \theta >$ where Γ is a list of goals and θ a substitution.
- A goal is of the form $W{:}S \Rightarrow X$ where W is a computation condition, S is a state definition, and X is a result definition.
- $X \cdot Xs$ is the list with head X and tail Xs.
- D denotes *any* data definition and M *any* method.

- $\mathcal{O}_{seq}(S)$ is an operation where an observer selects a sequence of elements from a set S.

- $\mathcal{O}_{trans}(X)$ is an operation performed by an observer which transforms the result definition X.

By using a list of qoals it is possible to write the rules with only one premise, making them correspond to state transitions.

Note that in rules (5) through (7) it is an observer who selects which equations of the current state definition S that may be used. Also note that an equation is selected before it is decided which rule to apply.

(1) Termination

$$\frac{}{<\{\},\quad \emptyset>}\,T \quad .$$

The inference rules are applied backwards and the computation stops when the list of goals is empty, thus the name termination.

(2) Empty

$$\frac{<\Sigma,\quad \sigma>}{<(\epsilon{:}S \Rightarrow S)\cdot\Sigma,\quad \sigma>}\,E \quad .$$

A goal is fully evaluated, or proved, when the computation condition is empty.

(3) Method

$$\frac{<(WW_1{:}S \Rightarrow X_1)\cdot \ldots \cdot (WW_n{:}S \Rightarrow X_n)\cdot\Sigma,\quad \sigma>}{<(WM{:}S \Rightarrow X)\cdot\Sigma,\quad \sigma>}\,M$$

where $M(M) = \{W_1, \ldots, W_n\}, n \geq 1$. $M(M)$ is the definiens of the method name M in the method definition M, that is,

$$\{W_i \mid M = W_i\#C_i \in M \wedge C_i(S)\},$$

and $X = \mathcal{O}_{trans}(X')$ where X' is the result definition

$$X' \begin{cases} W_1 & = & X_1 \\ & \vdots & \\ W_n & = & X_n \end{cases} .$$

Whenever a compound result definition is built, an observer gets a chance to transform it.

(4) Choice

$$\frac{<(WW_i{:}\, S \Rightarrow X)\cdot\Sigma,\quad \sigma>}{<(W(W_1,W_2){:}\, S \Rightarrow X)\cdot\Sigma,\ \ \sigma>}\ C$$

where W_i is an element of $ws = \mathcal{O}_{seq}(\{W_1, W_2\})$. The elements of ws are tried from left to right by backtracking. The selection ws must not be empty. This construction lets an observer decide in which order W_1 and W_2 are tried and to decide to only use one of them.

(5) Definition Left

Let $es = \mathcal{O}_{seq}(S)$ be the sequence of equations of S considered for rule-application. All elements of es are tried from left to right by backtracking. Let e be the currently selected equation. Then, depending on the left-hand side of e we have:

(5.1) If $e = (a = C)$ then

$$\frac{<((W{:}\,(A_1/a\sigma)S\sigma \Rightarrow X_1,\ldots,W{:}\,(A_n/a\sigma)S\sigma \Rightarrow X_n)\cdot\Sigma)\sigma,\quad \theta>}{<(W\overline{D}{:}\, S \Rightarrow X)\cdot\Sigma,\quad \theta\sigma>}\ \overline{D}_D$$

where $\sigma \in D_{suff}(a)$, $D(a\sigma) = \{A_1,\ldots,A_n\}$, and X is the result definition

$$X\begin{cases} A_1 &= X_1\\ &\vdots\\ A_n &= X_n\end{cases}.$$

Note that we have one instance of this rule for each a-sufficient substitution in $D_{suff}(a)$. All instances are tried by backtracking over these a-sufficient substitutions.

(5.2) If $e = ((A,B) = C)$ then

$$\frac{<(W{:}\,(C'/(A,B))S \Rightarrow X)\cdot\Sigma,\quad \sigma>}{<(W\overline{D}{:}\, S \Rightarrow X)\cdot\Sigma,\quad \sigma>}\ \overline{D}_V$$

where C' is an element of $cs = \mathcal{O}_{seq}(D((A,B)))$. The elements of cs are tried from left to right by backtracking. The selection cs must not be empty. This construction lets an observer decide in which order A and B are tried and to decide to only use one of them.

(5.3) If $e = ((A \rightarrow B) = C)$ then

$$\frac{<(W{:}\, S_1 \Rightarrow X_1)\cdot(W{:}\, S_2 \Rightarrow X_2)\cdot\Sigma,\quad \sigma>}{<(W\overline{D}{:}\, S \Rightarrow X)\cdot\Sigma,\quad \sigma>}\ \overline{D}_A$$

where S_1 and S_2 are given by

- $S_1 = ((A \to B) \ominus S) \downarrow A)$,
- $S_2 = (B/(A \to B))S$,

and $X = \mathcal{O}_{trans}(X')$ where X' is the result definition

$$X' \left\{ \begin{array}{lcl} A & = & X_1 \\ B & = & X_2 \end{array} \right. .$$

(6) Definition Right

Let $es = \mathcal{O}_{seq}(S)$ be the sequence of equations of S considered for rule-application. All elements of es are tried from left to right by backtracking. Let e be the currently selected equation. Then, depending on the right-hand side of e we have:

(6.1) If $e = (A = a)$ then

$$\frac{< ((W{:}S\sigma(B/a\sigma) \Rightarrow X) \cdot \Sigma)\sigma, \quad \theta >}{< (W\underline{D}{:}S \Rightarrow X) \cdot \Sigma, \quad \theta\sigma >}\ \underline{D}_D$$

where $\sigma \in D_{mgu}(a)$ and $B \in D(a\sigma)$. All elements of $D(a\sigma)$ are tried by backtracking.

Note that we have one instance of this rule for each element in $D_{mgu}(a)$. All instances are tried by backtracking.

(6.2) If $e = (A = (B, C))$ then

$$\frac{< (W{:}S(B/(B,C)) \Rightarrow X_1) \cdot (W{:}S(C/(B,C)) \Rightarrow X_2) \cdot \Sigma, \quad \theta >}{< (W\underline{D}{:}S \Rightarrow X) \cdot \Sigma, \quad \theta >}\ \underline{D}_V$$

where $X = \mathcal{O}_{trans}(X')$ and X' is the result definition

$$X' \left\{ \begin{array}{lcl} B & = & X_1 \\ C & = & X_2 \end{array} \right. .$$

(6.3) If $e = (A = (B \to C))$ then

$$\frac{< (W{:}S' \Rightarrow X) \cdot \Sigma, \quad \theta >}{< (W\underline{D}{:}S \Rightarrow X) \cdot \Sigma, \quad \theta >}\ \underline{D}_A$$

where $S' = B \oplus (S(C/(B \to C)))$.

(7) Identity

Let $es = \mathcal{O}_{seq}(S)$ be the sequence of equations of S considered for rule-application. All elements of es are tried from left to right by backtracking. Let e be the currently selected equation. Then

$$\frac{< ((W{:}S \Rightarrow X) \cdot \Sigma)\sigma, \quad \theta >}{< (WD{:}S \Rightarrow X) \cdot \Sigma, \quad \theta\sigma >} \; I$$

provided that $e = (a = b)$, and and $\sigma = mgu(a, b)$.

4.3 The Observer

There are two main motivations for introducing an observer. First, to make it possible to describe computations in a general manner without making all choices with respect to execution order explicit. Second, to set up hooks where the user, or some other process, may interact with computations. The Gisela framework provides a default observer, which is used if nothing else is stated explicitly. The default observer implements the following behavior:

- In rule 4, $\mathcal{O}_{seq}(\{w_1, w_2\})$ returns $[w_1, w_2]$.
- In rule 5.2, $\mathcal{O}_{seq}(\{A, B\})$ returns $[A, B]$.
- The selection of a sequence of equations from the state definition in rules 5, 6, and 7 is handled in the same way. When the guard of an equation in a method definition is evaluated and holds, it is reasonable to assume that an equation in the state definition that contributes to making the guard hold should be considered. The default observer therefore uses the heuristic to select all equations which make the guard hold. The selected equations are tried from left to right. If no equation in the state definition can be detected to make the quard hold all equations are selected.
- The result of $\mathcal{O}_{trans}(X)$ depends on the result type currently set in the observer. The default observer allows three result types: *full*, which means that no transformation is performed, *flat*, which means that the result definition is flattened to contain only the leaves of the full definition, and *empty*, which returns the empty result definition. The same transformation is performed throughout the entire computation.

5 Programming in the Gisela Framework

In this section we explain how to use Gisela for different kinds of programming. So far, only a limited set of programs have been developed using the Gisela framework. Apart from the examples shown in this paper, and various other minor programs, we have also built some tools for use in the MedView project.

There are two main approaches to programming in Gisela: to use syntactic representations (Section 5.1) or to use object representations (Section 5.6). Using syntactic representations is the easier way and yields readable programs. Using object representations is appropriate when we need some special kind of definition or observer. When we use object representations we have full access to programs written using syntactic representations, thus the two can be mixed freely.

Another view is that when we use syntactic representations only, in particular in conjunction with the interactive system discussed in Section 5.8, we do in effect work with a programming language. This programming language is what we get from specializing the model from Section 4.2 to use only the definition classes and methods for which we give syntactic representations, plus the default observer. Using object, or mixed, representations we work with a framework which provides a customizable model for definitional programming.

5.1 Syntactic Representations

When we use syntactic representations we create data definitions and method definitions using an equational presentation. The syntax used in Gisela is closely related to the syntax of GCLA and Prolog.

5.1.1 Terms and Data Definitions

The syntax used for data definitions is as follows:

1. Variables: A *variable* is a string beginning with an uppercase letter or the character '_', for example `X`, `LongVariableName`, `_Foo`.

2. Functors and constants:

   - A *functor* is a string beginning with a lowercase letter, or an arbitrary quoted string, which can be applied to some number of arguments. Some examples are `p/1`, `member/2`, `'Any name whatever'/0`.
   - A *constant* is a functor with no arguments.
   - Gisela also allows numbers and strings as special constants. Some examples are `4`, `"abc"`, and `3.76`.

3. Terms:

   - Each variable and constant is a *term*.
   - If $t_1, \ldots, t_n$ are terms and f is a functor of arity n then $f(t_1, \ldots, t_n)$ is a term.

- Gisela allows the same shorthand notation as Prolog for lists. Thus, `[]` denotes the empty list, and the lists `[X|Xs]` and `[a,b,c]`, the lists `X.Xs` and `a.(b.(c.nil))` respectively.
- Gisela allows infix notation for the ordinary arithmetic operators, `+`, `-`, `*`, and `/`. Thus, `4*5` is shorthand for the term `'*'(4,5)`.

4. Conditions:
   - Each term is a *condition*.
   - `true` and `false` are conditions.
   - If C_1 and C_2 are conditions then `(`C_1`,`C_2`)` and `(`C_1`->`C_2`)` are conditions. The parentheses may be omitted when there is no risk for ambiguity.
5. Equations. If a is a term and C is a condition then a `=` C. is an *equation*. The equation a. is shorthand for a `= true.` .
6. Guards. If t_1 and t_2 are terms then t_1 `\=` t_2 is a *guard*.
7. Guarded Equations. If $G_1, \ldots, G_n$ are guards then

$$a\#\{G_1, \ldots, G_n\} = C.$$

   is a *guarded equation*. Currently, guards are only allowed in matching definitions or equations restricted as matching (see below).
8. Directives. The following are *directives*:
   - `definition` *Name*`.`, where *Name* is a constant denoting the name of the definition.
   - `definition` *Name*`:`*Type*`.`, where *Name* is as above and *Type* is a constant giving the type of the definition. Currently, possible types are `constant`, `matching`, `unifying`, `fl`, and `gcla`. If no value is given the type of a data definition defaults to `unifying`.
   - `restrict` *N*`/`*A*`:`*Val*`.`, where *Val* is one of `right` and `matching`.
9. Data Definitions. A *data definition* is a finite sequence of (guarded) equations and directives starting with a directive giving the name of the definition.

The scope of a variable is the equation where it occurs. Comments are allowed as usual, that is, `%` or `//` means that the rest of the line is a comment, arbitrary comments are enclosed in between `/*` and `*/`.

Note that each data definition starts with a directive giving its name and type. With a `restrict` directive the programmer informs the system that it can use a simpler algorithm to compute the definiens operation. A `right` restriction means that the term will only be used in the right-hand side of equations in computations. A `matching` restriction tells the system that the definiens operation will only be applied to fully instantiated terms.

The meaning of the different definition types is as follows:

- A `constant` definition allows only constants as left-hand sides in equations. The domain consists of all the constants in the left-hand side of the equations of the definition.
- A `matching` definition uses matching only to find the definiens of a term. Thus, $D(a)$ is only valid for fully instantiated terms. The domain consists of all terms with the same principal functor as some term occurring as a left-hand side in an equation.
- A `unifying` definition is as a matching definition but uses full unification.
- An `fl` definition uses unification and has as its domain all terms with the same principal functor as some left-hand side in the definition. The difference compared to a unifying definition is that for terms in the domain, but not defined, an `fl` definition returns `{false}`, whereas a unifying definition returns `{}`.
- A `gcla` definition has as its domain the set of all terms, uses unification and returns `{false}` for terms not defined in the definition.

When a definition is presented as a number of equations using the syntax described above, the type of the definition together with the given equations fully determines which definition the description represents.

5.1.2 Method Definitions

In the description of the syntax used for method definitions we start by describing the building blocks and then show how they are combined into complete methods:

1. Parameters. A *Parameter* is a string beginning with an uppercase letter which denotes any data definition given as parameter to a method.
2. Constants. A *constant* is a string beginning with a lowercase letter. Depending on the context a constant denotes a computation method or a data definition with the same name.
3. Guard Constraints. A *guard constraint* is a boolean function which operates on a single condition C selected from the current state definition. The provided guard constraints are:
   - `in_dom(` D `)`, which holds if C is in the domain of D.

- `def_in_dom(` D `)`, which holds if some element of $D(C)$ is in the domain of D.
- `in_com(` D `)`, which holds if C is in the co-domain of D.
- `def_in_com(` D `)`, which holds if some element of $D(C)$ is in the co-domain of D.
- `matches(` T `)`, which holds if C matches T. No variables are bound.
- `var`, which holds if C is a variable.
- `nonvar`, which holds if C is not a variable.

In all cases D may be a parameter or a constant denoting a data definition.

4. Guard Primitives. A *guard primitive* is a boolean function which operates on a single equation e selected from the current state definition. The provided guard primitives are:

   - `false`, which never holds.
   - `true`, which always holds.
   - `identity`, which holds if the left and right-hand sides of e are identical.
   - `l:`GC, which holds if the guard constraint GC holds for the left-hand side of e.
   - `r:`GC, which holds if the guard constraint GC holds for the right-hand side of e.
   - `not(` GP `)`, the negation of the guard primitive GP.
   - `(` GP_1 `,` GP_2 `)`, which holds if both the guard primitives GP_1 and GP_2 hold for e.
   - `(` GP_1 `;` GP_2 `)`, which holds if any of the guard primitives GP_1 or GP_2 hold for e.

5. Guards. A *guard* is a boolean function which operates on the current state definition S. The following forms are provided:

   - `some(` GP `)` which holds if the guard primitive GP holds for some equation of S.
   - `all(` GP `)` which holds if the guard primitive GP holds for all equations of S.
   - `(` G_1 `&` G_2 `)`, which holds if both the guards GP_1 and GP_2 hold for S.
   - `(` G_1 `|` G_2 `)`, which holds if any of the guards G_1 or G_2 hold for S.

6. Equations. A method definition consists of a number of equations which have the general form
$$m = W\#Guard.$$
where m is a constant which is the same as the name of the method, W a computation condition, described below, and *Guard* a guard as described above.

7. Word Constituents. Computation words are built from *word constituents*. A word constituent is one of the following:
   - M, where M denotes any method or method instance in the current scope. Scoping rules are given below.
   - D, where D is any parameter or definition constant.
   - `l:` D, where D is any parameter or definition constant. This is the concrete syntax for $\overline{D}$.
   - `r:` D, where D is any parameter or definition constant. This is the concrete syntax for $\underline{D}$.

8. Computation words. A *computation word* is a (possibly empty) sequence of word constituents. A computation word is one of the following:
   - `[]`, the empty word.
   - `[`W_1`,...,`W_n`]`, where the W_i are word constituents.

9. Computation Conditions. A *computation condition* is one of the following:
   - All computation words are computation conditions.
   - W_1 `;` W_2, where the W_i are computation conditions. This is the concrete syntax for (W_1, W_2).
   - `[`W_1`,...,`W_n`]`, where the W_i are computation conditions.

10. Imports. The following are used to import method and data definitions:
    - `import_definition(` *Name* `).`, where *Name* is the name of the file where the data definition is stored, or in case of built-in definitions, simply the name of the definition.
    - `import_methods(` *Name* `).`, where *Name* is the name of the file where the method definitions are stored.

    After a method or a data definition has been imported its name can be used in subsequent method definitions.

11. Instantiations. An *instantiation* of a method scheme is an equation

$$Iname = \texttt{instance}(Mname, [D_1, \ldots, D_n]).$$

where $Iname$ is the name introduced to be used to denote an instance of the method-scheme $Mname$ created by instantiating it with the data definitions $D_1, \ldots, D_n$.

12. Method Definitions. A *method definition* has the following general form:

$$\begin{array}{ll} Imports & \\ \texttt{method} & m(D_1, \ldots, D_n). \quad n \geq 0 \\ Instantiations & \end{array}$$

$$\begin{array}{rcccl} m & = & W_1 & \# & G_1. \\ & & \vdots & & \\ m & = & W_m & \# & G_m. \end{array}$$

Method-schemes are method definitions where $n > 0$.

13. Scoping rules:

    - The scope of a parameter is the method-scheme for which it is a parameter.
    - Defined methods are visible throughout the file where they are defined.
    - Imported method and data definitions are visible throughout the file into which they are imported.
    - A method created with an instantiation is visible within the method definition where it is created.

The syntax for comments is the same as in data definitions.

5.1.3 Queries

We describe the syntax of queries for the interactive system in Section 5.8:

- State Definitions. A *state definition* is written `{`$e_1, \ldots, e_n$`}` where each e_i is an equation. The scope of a variable is the entire state definition. Each equation is of the form C_1 `=` C_2 where the both C_1 and C_2 are conditions.

- Queries. A *query* is written m`(`$D_1, \ldots, D_n$`)`S, $n \geq 0$, where m is a method, $D_1, \ldots, D_n$ are parameters used to create on instance of m, and S is the initial state definition.

The answer to a query is the computed result definition and any bindings to variables occurring in the initial state definition. If no result can be computed the answer is `no`.

5.1.4 Computing Definiens and Clause

For a definition represented as a sequence of equations, the definiens, $D(a)$, of an atom a is the set of all right-hand sides of equations in D whose left-hand sides matches a, that is $\{A\sigma \mid (b \Leftarrow A) \in D, b\sigma = a\}$. All the different equational definition types in the Gisela framework order the bodies in $D(a)$ in the order in which they appear in the definition.

To perform the operation $\overline{D}$ we need to compute an a-sufficient substitution for a. In the general case this is a very costly operation which involves finding a maximal set of left-hand sides in D which can be unified with each other and with a. To perform the operation $\underline{D}$ (clause) we only need to find some left-hand side in D unifiable with a. For `constant` and `matching` definitions the computation of an a-sufficient substitution is not needed which is why they should be used whenever possible to improve performance. The `restrict` directive has the same purpose, to avoid attempts at computing a-sufficient substitutions whenever possible. Some more details on how the definiens operation is performed can be found in Section 7.

5.1.5 A Note on Variables and Completeness

Variables and calculi of PID are covered in [16, 30, 38]. In GCLA explicit quantification can be used for variables in the bodies of clauses not occurring in the head. Existential quantification can easily be handled if it occurs in the right-hand side of an equation in a state definition. Likewise, universal quantification is easily handled to the left. Gisela has no way to express explicit quantifiers. Instead it is assumed that users are aware how free variables in bodies of equations should be understood.

The algorithm used for computing definiens for data definitions which use unification is not complete since it does not compute all a-sufficient substitutions as discussed in [7, 38]. Both [7] and [38] present algorithms based on some notion of guarded variables or disequalities to solve this problem. In Gisela guards are only allowed in matching definitions. If guards are extended to be allowed in unifying definitions we will be able to implement some version of these algorithms. Doing so is not trivial though.

5.2 GCLA-style Programming

In GCLA [11] a program consisted of a single definition. Queries were proved using a fixed PID-calculus. Some control of the search for proofs of queries could be given using annotations in the definition, and by setting certain global parameters. GCLAII [6, 37] introduced a second definition, the *rule definition*, which made it possible to describe proof search strategies and inference rules in a very sophisticated declarative manner. In this section we discuss how Gisela can be used for GCLA style programming. First we give the basics, which

$$\frac{\Gamma \vdash C\sigma}{\Gamma \vdash c \;\; \sigma}\; \textit{D-right} \quad (b \Leftarrow C) \in D, \sigma = mgu(b, c)$$

$$\frac{\Gamma, A \vdash C\sigma \;\;\; A \in D(a\sigma)}{\Gamma, a \vdash C \;\; \sigma}\; \textit{D-left} \quad \sigma \text{ is an } a\text{-sufficient substitution}$$

$$\frac{}{\Gamma, a \vdash c \;\; \tau}\; \textit{axiom} \quad \tau = mgu(a, c)$$

$$\frac{}{\Gamma \vdash \texttt{true}}\; \textit{true-right} \qquad \frac{}{\Gamma, \texttt{false} \vdash C}\; \textit{false-left}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\; \textit{a-right} \qquad \frac{\Gamma \vdash A \quad \Gamma, B \vdash C}{\Gamma, A \to B \vdash C}\; \textit{a-left}$$

$$\frac{\Gamma \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \vdash (C_1, C_2)}\; \textit{v-right} \qquad \frac{\Gamma, C_i \vdash C}{\Gamma, (C_1, C_2) \vdash C}\; \textit{v-left} \quad i \in \{1, 2\}$$

$$\frac{\Gamma \vdash C_i}{\Gamma \vdash (C_1; C_2)}\; \textit{o-right} \quad i \in \{1, 2\} \qquad \frac{C_1 \vdash C \quad C_2 \vdash C}{(C_1; C_2) \vdash C}\; \textit{o-left}$$

Figure 1: GCLA Sequent Calculus Rules.

essentially correspond to GCLA, and then we discuss control issues focusing on the similarities and differences between GCLAII and Gisela.

5.2.1 Basics

Figure 1 shows a sequent calculus which is essentially the calculus used in GCLA to prove queries. In GCLA a query is a sequent ($\Gamma \vdash C$), where Γ is a list of conditions and C is a condition. The meaning of the query is: "Does C follow from Γ using the given definition". If the query can be proved the result is an answer substitution containing the variables in the query, otherwise the answer is `no`. The logic used to prove a query is local to the definition D used [25], as can be seen from the inference rules.

We need to define a computation method and describe how to write the initial state definition in such a way that running a query in Gisela corresponds to proving an equivalent query in GCLA. We will base our method on the following observations and restrictions:

- Programs consist of a single data definition just as in GCLA,
- The data definitions used will be GCLA-definitions, that is, $D(a) = false$, for all atoms a not occurring as left-hand sides in D.
- All right-hand sides in the initial state definition must be identical. This corresponds to the single condition in the consequent of sequents in GCLA.

If all right-hand sides are identical in the initial state definition they will remain so throughout the computation.

- There is nothing in Gisela corresponding the the rules *o-right* and *o-left*. Indeed, $(A; B)$ is not a condition in Gisela. We note that *or* in logic programming is mainly for convenience. If desired, an extra data definition defining *or* could be introduced.
- Gisela has no rules corresponding to the rules *false-left* and *true-right*. This is since Gisela only has a limited number of built in rules providing a number of ways to transform an initial state definition, but no particular interpretation of *true* and $false$. We will have to write method definitions giving the desired interpretation.
- The rest of the rules in Figure 1 have direct counterparts in Gisela.

An essential difference from the method definitions shown in Section 2 is that a basic search strategy for GCLA is inherently non-deterministic. Typically, for each sequent more than one sequent calculus rule apply. In Gisela non-deterministic method definitions are written by having more than one computation condition to choose from in an equation of a method definition.

The default behavior of GCLA is to use a search strategy called `arl` which first tries the *axiom* rule, then *x-right* rules, and finally *x-left* rules. This behavior is captured by the following method definitions:

```
method true_right.
true_right = [] # some r:matches(true).

method false_left.
false_left = [] # some l:matches(false).

method arl:[D].

arl = [D];[true_right];[arl,r:D];[false_left];[arl,l:D].

method gcla:[D].

arl_inst = instance(arl, [D]).

gcla = [arl_inst].
```

We have defined `gcla` to be a cover for the computation method `arl`. Most interesting is the definition of `arl` where the computation continues with any of the computation conditions separated by ‘`;`’. We assume that the default observer is used, thus all alternatives are tried from left to right.

Of course, other search orders could be used. For instance, `lra` and `lar`:

```
method lra:[D].

lra = [false_left];[lra,l:D];[true_right];[lra,r:D];[D].

method lar:[D].

lar = [false_left];[lar,l:D];[D];[true_right];[lar,r:D].
```

Typically, too many answers are computed. One of the reasons is that atoms to the left are reduced to `false` more often than desired. In GCLA atoms could be declared total to prevent these reductions. In Gisela we could introduce another data definition defining such atoms to be regarded as data. More on issues like this can be found in Section 5.5.

5.2.2 Example: Default Reasoning

Assume we know that an object can fly if it is a bird and if it is not a penguin. We also know that Tweety and Polly are birds as are all penguins, and finally we know that Pengo is a penguin. A data definition expressing this information is the following:

```
definition birds:gcla.

flies(X) =
   bird(X),
   (penguin(X) -> false).

bird(tweety).
bird(polly).
bird(X) = penguin(X).

penguin(pengo).
```

The definition is adopted from [23]. If we want to know which birds can fly, we pose the query

```
G3> gcla(birds){true = flies(X)}.
X = tweety
? ;
X = polly
? ;
no
```

which gives the expected answers. More interesting is that we can also infer negative information, i.e., which birds cannot fly:

```
G3> gcla(birds){true = flies(X) -> false}.
X = pengo
? ;
no
```

This kind of negation has been treated at length in a number of papers on GCLA for instance [6, 11, 36]. It works the same in Gisela.

5.2.3 Control

Both GCLAII and Gisela separate the declarative and the procedural part of a program. The way control issues are handled are very different though, as are parts of the general computation models.

GCLAII has a default set of inference rules similar to the calculus shown in Section 5.2.1 and a number of search-strategies built from these rules. To program the control part, the user could define new search-strategies but it was also possible to define new inference rules, discarding the default calculus completely if desired. The system was very powerful and a lot of work was put into developing suitable programming methodologies [6, 23, 57].

Compared to the rule definitions of GCLAII, method definitions in Gisela are very restricted. Although most parts of method definitions may be modified using specialized object representations, the structure, as such, remains very simple. A method definition is just a number of equations where each equation contains a computation condition. The conditions are simple flat structures describing a sequence of actions to perform. It is natural to think of a method definition as a function which takes an initial state definition and transforms it according to the actions specified by the computation conditions. On the other hand, proof-search in GCLAII is really a matter of equation-solving and rules and strategies are functions which are run “backwards”.

From a practical point of view the key differences are:

- Gisela does not permit us to write new inference rules, e.g., change the set of ways to move from one state definition to the next one. What we could do is to write a number of method definitions corresponding to the default rules in GCLA and use these as a basis for programming control. Such a set of method definitions is given in appendix A.

- In method definitions in Gisela, it is not possible to control explicitly which equation of the current state definition an operation should be applied to. In particular, it is not possible to specify that the next operation should be applied to the same equation as the current operation.

- In Gisela an arbitrary number of data definitions may be used. This opens up for new programming methodologies which could be used to regain some power lost in other respects, i.e., splitting a program into several data definitions and using method definitions to select between these in different ways. This approach has been explored to some extent in the setting of definitional program separation [18, 19].

- Gisela can be programmed using object representations through which method definitions can be modified in a multitude of ways.

- In Gisela, the observer concept for tuning computational behavior is present. However, so far this concept is rather unexplored.

5.3 Separated Definitional Programming

Separated definitional programming or *definitional program separation* has been discussed in [17, 18, 19, 21, 26]. Gisela is in several ways better suited for this technique than GCLAII. We give a brief description of the technique and demonstrate with an example.

5.3.1 Background

The central idea in definitional program separation is a program separation scheme based on the notions of *form* and *content* of an algorithm. Since many different algorithms can be expressed using the same form and varying the content, definitional program separation has also been proposed as a candidate for higher order definitional programming.

Definitional program separation relies heavily on the use of multiple data definitions. Since GCLAII only supports a single data definition it was not particularly well-suited for implementing separated programs. In [18, 19] an idealized definitional programming language based on GCLAII was used. Essentially, this language augmented GCLAII with the possibility of having multiple data definitions and a number of provisos like `in_dom/1`. To test programs an interpreter was written using GCLAII. The way Gisela supports program separation is closer to the original descriptions given in [26] than to the GCLA inspired notations of the idealized language in [18, 19].

When developing a separated version of an algorithm we try to split the description of the algorithm into its form and content. In other words, we try to separate the *global structure* or (recursive) form of the algorithm from the *operations* needed to compute the algorithm. One of the interesting things about this is that many algorithms share the same form, but use different operations. Thus, it becomes possible to classify algorithms in new ways.

In Gisela the form of an algorithm is expressed using a method definition and the content in a number of data definitions.

5.3.2 A Separated Algorithm

Consider the primitive recursive definition of addition:

$$D \left\{ \begin{array}{lcl} plus(0,m) & = & m. \\ plus(s(n),m) & = & s(plus(n,m)). \end{array} \right.$$

A stepwise description of the intended algorithm computing $plus(n,m)$ associated with this definition could be:

1. If $n = 0$, then the result is $D(plus(n,m))$, that is, m.
2. If $n = s(x)$, then first compute $plus(x,m)$ to y and then apply s to get the result $s(y)$.

In a separated program the *local* operations should be separated from the *global* content. The operations involved in this example are:

1. From $plus(0,m)$ move to m.
2. From $plus(s(x),m)$ move to $plus(x,m)$.
3. From a number y compute $s(y)$.

Expressed as two simple definitions:

$$P \left\{ \begin{array}{lcl} plus(0,m) & = & m. \\ plus(s(n),m) & = & plus(n,m). \end{array} \right.$$

$$N \left\{ \begin{array}{lcl} n & = & s(n). \end{array} \right.$$

Now, given these operations we need a form which will compute the algorithm implicit in D. Such a form, F, described entirely in definitional terms is:

$$F \left\{ \begin{array}{lcll} F(x) & = & P(x) & \#P(x) \notin Dom(P). \\ F(x) & = & NFP(x) & \#P(x) \in Dom(P). \end{array} \right.$$

So, F defines the form of an algorithm adding two natural numbers and P and N provide the content.

5.3.3 Separated Gisela programs

The examples given in this section are described as functions, that is, what we want to do is to evaluate a functional expression to a value. Following the approach in [18, 19, 21], where the expression to evaluate was given in the antecedent of sequents, the expression to evaluate will be the left-hand side in a state definition containing a single equation.

First, we look at the separated program discussed in the previous section. We rename the definitions P and N `plus` and `nats`, respectively:

```
definition plus:matching.

plus(zero, M) = M.
plus(s(X), M) = plus(X, M).

definition nats:matching.

zero = s(zero).
s(X) = s(s(X)).
```

Since in this case we are interested in computing answers only, not solving equations, we have declared that `plus` and `nats` are *matching* definitions, that is, the definiens operation can be applied only to fully instantiated terms.

Describing the form F in Gisela is also rather straightforward. F can be implemented using a method definition with two equations, corresponding to the two equations of F. The implementation uses the built-in guard constraint `def_in_dom`. Also, the data definitions `nats` and `plus` are imported into the method definition, which has no parameters:

```
import_definition(nats).
import_definition(plus).

method form1.

form1 = [l:plus] # all not(l:def_in_dom(plus)).
form1 = [l:nats, form1, l:plus] # some l:def_in_dom(plus).
```

What `form1` does is to reduce the expression on the left-hand side of the chosen equation to its value. For instance,

```
G3> form1{plus(s(zero),s(zero)) = value}.
```

will compute the flattened result definition `{s(s(zero)) = value}`.

Of course, things become more interesting if we parameterize the method definition `form1`, since then several algorithms sharing the same form can be computed, simply by switching data definitions. The parameterized version becomes:

```
method form1:[D1, D2].

form1 = [l:D1] # all not(l:def_in_dom(D1)).
form1 = [l:D2, form1, l:D1] # some l:def_in_dom(D1).
```

With this version of `form1` we use a slightly modified query which computes the same answer as before:

```
G3> form1(plus,nats){plus(s(zero),s(zero)) = value}.
```

Although we can tell what the sum of two numbers is from a result such as `{s(s(zero)) = value}`, it is arguable that it is not the most intuitive of answers. An alternative is to use a variable in the right-hand side of the equation and bind it to the result of the computation. This method is in accordance with the technique used in GCLA. To be able to do this we modify `form1` somewhat and add an extra step which unifies the left and right-hand sides of the equation after a result has been computed:

```
method f1:[D1, D2].

f1 = [l:D1] # all not(l:def_in_dom(D1)).
f1 = [l:D2, f1, l:D1] # some l:def_in_dom(D1).

method form1:[D1, D2].

f = instance(f1, [D1, D2]).

form1 = [D1, f].
```

In this version `form1` simply uses the method `f` which corresponds to previous versions of `form1` to compute a value and then the two sides of the resulting state definition are unified with each other. Now, if we decide to view only the answer substitution, the answer to the query

```
G3> form1(plus,nats){plus(s(zero),s(zero)) = N}.
```

is the substitution `{N = s(s(zero))}`.

We round up the example by showing, *len* and *min*, two more recursive functions having the same form as *plus* but different content. Both can be split in a manner very similar to *plus* and use `nats` to get the successor of a natural number. We simply show the data definitions providing the content and a sample query:

```
definition min:matching.

min(zero,N) = zero.
min(s(M), zero) = zero.
min(s(M), s(N)) = min(M, N).

definition len:matching.

len([]) = zero.
len([X|Xs]) = len(Xs).
```

Compute the length of `[a,b,c]`:

```
G3> form1(len,nats){len([a,b,c]) = N}.

N = s(s(s(zero)))
```

5.3.4 Discussion

Definitional program separation, and especially the way to describe methods and computations used in [26], has had a major influence on the development of Gisela. It was a programming technique which required use of several data definitions, a feature not available in GCLA.

Most of the work on definitional program separation so far is presented in [18, 19] which goes through a large number of examples and presents a number of different forms. As mentioned above, all examples are given in an idealized definitional programming language similar to GCLA.

We have not, as yet, thoroughly tested how well the developed techniques may be transferred to Gisela. Some examples use specialized provisos testing properties and performing operations on terms not present in Gisela. However, in most cases we believe that program separation is handled in a cleaner way using Gisela.

5.4 Computing Similarity Measures

Assume that we have the following two partial cases adopted from MedView:

```
definition s1:constant.              definition s2:constant.

anamnesis = common.                  anamnesis = common.
common = drug.                       common = drug.
common = allergy.                    common = allergy.
common = smoke.                      common = smoke.
drug = no.                           drug = no.
allergy = oranges.                   allergy = lemons.
smoke = '8 cigarettes/day'.          smoke = '4 cigarettes/day'.
```

Suppose we wish to compute somehow how similar the cases are to each other. One possibility is to compare all the common attributes pair-wise and run the query

```
G3> cm(s1,s2){drug=drug, allergy=allergy, smoke=smoke}.
```

where `cm` is the same as the method definition used in Section 3.6. The flattened result definition for this query is

```
{no=no, oranges=lemons, '8 cigarettes/day'='4 cigarettes/day'}
```

If the interpretation of the result is obvious we can stop here. However, if an interpretation is not abvious we can use the computed result definition as the initial state definition in a new query to get a better estimation of how similar `s1` and `s2` are. For instance, we may have additional knowledge in a data definition `groups`:

```
definition groups:constant.

oranges = citrus_fruits.
lemons = citrus_fruits.
'8 cigarettes/day' = '< 10 cigarettes/day'.
'4 cigarettes/day' = '< 10 cigarettes/day'.
```

One way to learn more about the similarity of `s1` and `s2` is to use the result definition computed above and a single-stepping method `ss` which replaces some left or right-hand side by its definition in `groups`:

```
G3> ss(groups){no = no, oranges = lemons,
              '8 cigarettes/day' = '4 cigarettes/day'}.

{no = no, citrus_fruits = lemons,
          8 cigarettes/day = 4 cigarettes/day}
```

We take the result as the initial state definition in a new computation:

```
G3> ss(groups){no = no, citrus_fruits = lemons,
              '8 cigarettes/day' = '4 cigarettes/day'}.

{no = no, citrus_fruits = lemons,
          < 10 cigarettes/day = 4 cigarettes/day}
```

Repeating this process, we will finally arrive at a result definition containing identities only. Now, the similarity can be defined as follows: The fewer steps we need to arrive at a definition which consists of identities only, the more similar `s1` is to `s2`. Alternatively, we could use some more complicated method and query and take the *size* of the full result definition as a similarity measure.

5.5 Functional Logic Programming

Functional logic programming using GCLA has been covered in depth in [55, 56, 57]. In particular [57] covers a wide range of topics from how to go about writing functional logic programs to generating specialized rule definitions for efficient evaluation. The functional logic programming methodology is based on a few crucial restrictions to the general GCLA machinery, namely:

- at most one condition is allowed in the antecedent,

- rules that operate on the consequent can only be applied if the antecedent is empty,
- the axiom rule, can only be applied to atoms with circular definitions,
- if the condition in the antecedent is (C_1, C_2) then C_1 and C_2 are tried from left to right by backtracking.

With these restrictions, evaluation of functional logic programs becomes deterministic in the sense that only one inference rule can be applied to each sequent. The functional logic programming methodology following from this is not aimed at general equation solving, but at combining functions and predicates in a natural way.

Now, the restrictions above can be directly applied to Gisela:

- each state definition contains exactly one equation,
- rules that operate on the right-hand side of equations can only be applied if the left-hand side is `true`,
- the identity rule can only be applied to atoms in the domain of a special data object definition,
- if the condition in the left-hand side is (C_1, C_2) then C_1 and C_2 are tried from left to right by backtracking.

5.5.1 A Computation Method for Functional Logic Programs

Expressing the above evaluation strategy as a method definition in Gisela is relatively straightforward. To give an illustration of how computations are performed we give a number of deduction rules in Figure 2 showing state definition transformations in functional logic computations. To distinguish data from functions and predicates we use one data definition D to define canonical data objects, and another data definition P to define functions and predicates. A method definition which implements functional logic computations along the lines of the calculus in Figure 2 is `fl`, which takes two parameters, a program definition `P`, and a data object definition `D`:

```
method fl:[P,D].
// t, done when true to the right.
fl = [] # some l:matches(true) &
          some r:matches(true).

// f, both sides false.
fl = [] # some l:matches(false) &
          some r:matches(false).
```

$$\frac{\{\texttt{true} = C\}}{\{\texttt{true} = c\}}\,dr \quad c \in \mathit{Dom}(P), C \in P(c)$$

$$\frac{\{A_1 = C\} \quad \ldots \quad \{A_n = C\}}{\{a = C\}}\,dl \quad a \in \mathit{Dom}(P), P(a) = A_1, \ldots, A_n$$

$$\frac{}{\{a = a\}}\,ax \quad a \in \mathit{Dom}(D)$$

$$\frac{}{\{\texttt{true} = \texttt{true}\}}\,t \qquad \frac{}{\{\texttt{false} = \texttt{false}\}}\,f$$

$$\frac{\{A = B\}}{\{\texttt{true} = A \rightarrow B\}}\,ar \qquad \frac{\{\texttt{true} = A\} \quad \{B = C\}}{\{A \rightarrow B = C\}}\,al$$

$$\frac{\{\texttt{true} = C_1\} \quad \{\texttt{true} = C_2\}}{\{\texttt{true} = (C_1, C_2)\}}\,vr \qquad \frac{\{C_i = C\}}{\{(C_1, C_2) = C\}}\,vl \quad i \in \{1, 2\}$$

Figure 2: Schematic state definition transformations for functional logic computations using a data object definition D and a program definition P.

```
// ax, data, unify left and right.
fl = [D] # some l:in_dom(D).

// al, vl, conditions to the left.
fl = [fl, l:P]  # some l:matches((_, _));l:matches((_->_)).

// ar, vr, conditions to the right.
fl = [fl, r:P]  # some l:matches(true) &
                  some r:matches((_, _));r:matches((_-> _)).

// dl, definiens
fl = [fl, l:P]  # some l:in_dom(P).

// dr, clause
fl = [fl, r:P]  # some l:matches(true) & some r:in_dom(P).
```

5.5.2 Writing Functional Logic Programs

As mentioned above, [57] covers functional logic programming using GCLAII in detail. Among other things, a calculus called *FL* for handling functional logic programming is given. The method `fl` makes it possible to reuse the general methodology using the Gisela framework. Since most of the basic material on writing functional logic programs carries right over to Gisela we only give a brief overview and refer to [57] for details. Certain extensions of *FL*, such as using generated specialized rule definitions, cannot be applied to Gisela. We discuss alternative approaches in Section 5.5.3 below.

Queries In the following we use the terminology from [57] and call data *canonical objects*. Assume that we have a data definition P defining a number of functions and predicates, and a data definition D defining the canonical objects of the application domain. Using the method `fl` there are two kinds of queries:

1. Functional queries:
$$\texttt{fl}(P, D)\{FunExp = C\},$$
where $FunExp$ is a condition and C a variable or a (partly instantiated) canonical object.

2. Predicate (logic) queries:
$$\texttt{fl}(P, D)\{\texttt{true} = PredExp\},$$
where $PredExp$ is a condition.

The intended meaning of the functional query is "evaluate $FunExp$ to C". The intended meaning of the predicate query is "does $PredExp$ hold?". We see that conditions to the left are understood as expressions to evaluate, and conditions to the right as predicates to be proved.

Canonical Objects The computation method `fl` is intended for use with data definitions of type `fl` (5.1.1). The canonical objects of an application are defined in a special data definition. Since the only thing this definition is used for is to test whether a term is in its domain or not, it does not really matter how the canonical objects are defined. However, following the approach of [57] we use circular definitions. For instance:

```
definition nats:fl.

zero = zero.
s(X) = s(X).
```

Defining Functions A function definition, defining the function F, consists of a number of equations

$$\begin{array}{ll} F(t_1, \ldots, t_n) = C_1. & \\ \vdots & n \geq 0, m > 0. \\ F(t_1, \ldots, t_n) = C_m. & \end{array}$$

Two observations of interest are: (i) If the heads of two or more equations are overlapping then the corresponding bodies must have the same value, (ii) If $C_i = A \rightarrow B$ then it is understood as “the value of C_i is B if A *holds*”.

Defining Predicates The method `fl` handles pure Prolog programs. Thus, defining predicates is just like writing a program in pure Prolog. The interesting thing is how to use functions in predicates. Just as in function definitions the arrow, ‘$\rightarrow$’, works as a switch between functions and predicates. For instance, if we have an equation like

$$P = F \rightarrow C.$$

in a predicate definition it should be understood as “P holds if F can be evaluated to C”. The arrow can also be used in the context of negation as in Section 5.2.2.

Examples In [57] a large number of example programs dealing with functional logic programming in GCLA are given. Most of these can be more or less directly transferred to the Gisela setting. We show such an example here.

Let the definition `nats` be as above. We will define a (partial) function `double_odd` which doubles all odd numbers but computes no value for even numbers. First, we state that if `X` is odd then the value of `double_odd(X)` is computed by the function `double/1`:

```
double_odd(X) = odd(X) -> double(X).
```

Then we define the predicate `odd` and the function `double`:

```
odd(s(X)) = even(X).

even(zero).
even(s(X)) = odd(X).

double(zero) = zero.
double(s(M)) =
    (double(M) -> K)
    -> s(K).
```

With this we are done and can proceed to ask queries:

```
G3>  fl(fldemo,nats){double_odd(s(zero)) = X}.
X = s(s(zero))
? ;
no

G3> fl(fldemo,nats){double_odd(zero) = X}.
```

```
no

G3>  fl(fldemo,nats){double_odd(N) = M}.
N = s(zero),
M = s(s(zero))
? ;
N = s(s(s(zero))),
M = s(s(s(s(s(s(zero))))))
?
yes
```

where all functions and predicates are defined in the data definition `fldemo`.

5.5.3 Discussion

The most significant restriction on queries imposed in functional logic programs is that the state definition must contain exactly one equation. Due to the properties of Gisela and the method `fl`, this means that all goals throughout the computation will contain exactly one equation. This of course eliminates the need for an observer to choose the equation, and makes evaluation very simple indeed.

We have only showed the most basic methods for using Gisela for functional logic programming here. To test Gisela we have written one major functional logic program which generates text summaries in HTML or LaTeX format from patient data gathered in the MedView project. The in-depth description of functional logic programming using GCLA in [57] covers a number of topics not mentioned here. Some of these are:

- Methodology for writing lazy and strict functions.
- Extensions to *FL* such as efficient arithmetics, if-then-else, negation as failure, and IO.
- Generation of specialized rule definitions for management of nested function calls and more efficient computations.

We discuss how these issues could be handled in Gisela.

Evaluation strategies In [57] programming methods were presented for both strict and lazy evaluation of functions. In principle, all this material can be applied to Gisela without modification. It should be noted that “lazy” in this setting does not mean that expressions are evaluated at most once (sharing), but simply that they are only evaluated when needed.

Extensions The implementations of if-then-else and negation as failure presented in [57] rely on a built-in if-then-else at the meta-level of GCLA. This built-in meta-level if-then-else works as the built-in if-then-else of Prolog, that is, the if part is only evaluated once if successful. Gisela so far has no such primitive. We would rather try to find a more declarative solution. From a practcial point of view, however, the need for an if-then-else construct is obvious. The other extensions of *FL* mentioned can be implemented through extra definitions.

Nested Function Calls If we have a data definition like

```
double(zero) = zero.
double(s(M)) =
    (double(M) -> K)
    -> s(K).
```

and try to use it to evaluate `double(double(s(zero))` it will not work since there is nothing in the definition or in the method definition `fl` which tells us how to evaluate the argument to `double`. In GCLA two approaches were used to handle this. Either adding an extra clause to the definition of `double` or using a specialized rule definition which ensured that arguments were evaluated. The second approach cannot be used in Gisela since it is not within reach of what can be expressed in method definitions. The first approach can be used, but yields rather complicated data definitions.

A better alternative might be to use the Gisela framework as a low-level engine for functional logic programming and build a programming language on top of it. In its most naive form such a language could simply add clauses for evaluation of arguments to a definition. For instance, a definition like:

```
min(zero,N) = zero.
min(s(M), zero) = zero.
min(s(M), s(N)) = succ(min(M, N)).
```

would become

```
min(M,N) =
    (M -> M1),
    (N -> N1)
    -> min1(M1, N1).

min1(zero,N) = zero.
min1(s(M), zero) = zero.
min1(s(M), s(N)) = succ(min(M, N)).
```

From a computational point of view, this corresponds directly to what the specialized rule definitions used in [57] do. Of course, this is not optimal since it will attempt to re-evaluate already evaluated arguments. However, a lot of work has been put into finding efficient solutions to this problem, both in the area of functional [12, 45, 46] and functional logic programming [2, 3, 4, 5, 31, 39, 43], which could be applied in a translation of a high-level source language into Gisela.

5.6 Object Representations

At a suitably abstract level, a program in the Gisela framework is just a collection of data and method definitions, plus a query which is evaluated according to the rules given in Section 4.2. Thus, whether the data and method definitions are created using syntactic representations or by some other means is not important. With this in mind, Gisela was from the start designed to make it simple to build programs directly as objects, from components and classes in the framework, instead of using traditional syntactic representations. All that is required to use the Gisela framework in an Objective-C program is to create a new instance of the class `DFDMachine`, some data and method definition objects and start computing.

In this section we give an overview of how Gisela can be used in this manner. A couple of applications are discussed in Sections 5.7 and 5.8. Some more details are given in Section 7. The examples use Objective-C, an object-oriented extension to C. A nice introduction to Objective-C and object-oriented programming is found in [40]. A very brief overview is given in appendix B.

5.6.1 General Idea

The general idea behind the Objective-C interface to Gisela is that each kind of entity used to build programs, variables, terms, conditions, definitions, guards etc., is represented by objects of a corresponding class. Thus, a constant is represented by an object of the class `DFConstant`, a guard primitive by an object of the class `DFGuardPrimitive` and so on. It follows that if we have a conceptually clear definitional model of a system it can be realized directly using object representations.

The aim of Gisela is to provide a general framework for implementing definitional models of various kinds of systems. As such, we want as few restrictions as possible on what the definitional model permits. To allow for flexibility, the computation model described in Section 4.2 only gives very abstract descriptions of certain parts of computations. Specifically, data definitions are described in an abstract manner, guards in method definitions only as boolean functions, and the behavior of the observer is essentially left open. The syntactic representations presented above provide specific implementations of these notions. Using object representations alone does not extend Gisela in any way, apart from providing a second API. What we can do, however, is to extend the framework by subclassing existing classes or writing new ones which adhere to the restrictions of the Gisela computation model. It is mainly through the mentioned parameters, data definitions, guards, and observers, the framework is open for modification. Given specific implementations of data definitions, guards and observers, the behavior of the system is fully defined by the model in Section 4.2.

5.6.2 Creating Data and Method Definitions

Since this paper is not a manual or reference for using the Gisela framework we will only give some brief examples. We start by showing how to build data and method definitions from objects.

Assume that we want to create a data definition defining the identity function, `id/1`, for use as part of a definitional computation in an Objective-C program. That is, an object representing the data definition having the syntactic representation:

```
definition id:matching.
id(X) = X.
```

There are two ways to create the data definition `id` of which only one will be shown here. First, we can use the classes of the Gisela framework and build up the definition from objects of these classes step by step. Second, it would be trivial to write a definition class, implementing the required methods, which for any term $id(X)$ returned $\{X\}$ as the definiens.

We illustrate the API for building the data definition `id` from objects of classes in the framework. The definition is built bottom-up starting with the variable `X`:

```
// Declarations of needed variables.
DFVariable *x;
DFCompoundTerm *idX;
NSArray *eqs;
DFDefinition *idDef;

// Create a new variable.
x = [DFVariable variable];
// Create the term id(X)
idX = [DFCompoundTerm compoundTermWithName:@"id"
               andArguments:[NSArray arrayWithObject:x]];

// Create an array containing the equation id(X) = X.
eqs = [NSArray arrayWithObject:[DFEquation equationWithLeft:idX
                                                   andRight:x]];

// Create a definition named id from the equations in eqs.
idDef = [[DFMatchingDefinition alloc] initWithName:@"id"
                                      andEquations:eqs];
```

In the current implementation, the definition `idDef` constructed above is identical to a definition resulting from parsing a string containing the syntactic representation. An alternative way to build the definition is therefore:

```
// Create a parser object.
DFDefinitionParser *parser = [[DFDefinitionParser alloc] init];
DFDefinition *idDef;

// Create the definition from its syntactic representation.
idDef = [parser parseDefinitionWithString:
                     @"definition id:matching. id(X) = X."
        ];
```

To build a method definition is no different, just slightly more cumbersome. As an example let us create the method definition that has the syntactic representation:

```
method rightAx.

rightAx = [id,r:id].
```

The following Objective-C code builds the corresponding object representation:

```
// Declarations of needed variables.
NSString *rAx = @"rightAx";
DFOperator *anOp;
DFWord *word;
DFDefinition *idDef; // created as above
DFGuardedEquation *eq;
DFMethod *raMethod;

// Create the method definition.
raMethod = [[DFMethod alloc] initWithName:rAx];

word = [[DFWord alloc] initWithCapacity:2];

// Create the operator id used in the syntactic representation.
anOp = [DFOperator operatorWithDefinition:idDef
                          andOperatorType:DFBothOperator];
[word addConstituent:anOp];

// Create the operator r:id and add it to word.
anOp = [DFOperator operatorWithDefinition:idDef
                          andOperatorType:DFRightOperator];
[word addConstituent:anOp];
```

```
// Create the single equation and add it to the method definition.
eq = [DFGuardedEquation equationWithLeft:
                            [DFMethodConstant constantWithName:rAx]
                                andRight:word];
[raMethod addEquation:eq];
```

The structural similarity between syntactic and object representations should be clear from the examples. Also, the fact that syntactic representations are generally easier to handle, which of course is the reason why we use them in the first place.

5.6.3 Using a D-Machine

The heart of the definitional machinery is the `DFDMachine` which is a class implementing the calculus in Section 4.2.

The machine may be set up in different ways depending on the context where it is to be used. It is possible to have a machine that runs in the same thread as the object creating the machine or in a separate thread, which might be more appropriate for interactive applications. It is also possible to set the machine's observer to any object implementing the appropriate methods. Some of the methods available to initialize a `DFDMachine` are:

```
// Create a machine that uses the default observer.
- (id)initWithDelegate:(id)anObject;

// Create a machine that uses the default observer
// and runs computations in a separate thread.
// Messages from the computation are handled by the delegate.
- (id)initWithInteractiveDelegate:(id)anObject;

// Create a machine that uses a custom observer
// which does not interact with other objects.
- (id)initWithDelegate:(id)anObject
          andObserver:(id<DFComputingObserver>)anObserver;

// Create a machine that uses a custom observer
// that may interact with the calling application.
// Computations are run in a separate thread.
- (id)initWithDelegate:(id)anObject
      andInteractiveObserver:(id<DFComputingObserver>)anObserver;

// Create a machine where the delegate and the observer
// are the same object.
// Computations are run in a separate thread.
```

```
- (id)initWithInteractiveObserverDelegate:
            (id<DFComputingObserver>)anObserver;
```

The delegate is an object which handles certain things for the machine and receives notifications at times. It can be the same as the observer or another object.

5.6.4 Extending the Framework

So, if using object representations is just a more cumbersome way to write programs, why bother? The answer, of course, is that by providing means to introduce new behavior we can easily extend the framework to allow more general definitional models. We give a few examples of how this can be done.

Introducing New Data Definitions String constants are allowed in Gisela. With the current representation, they are just atomic constants which cannot be modified.[3] To handle strings the Gisela framework provides a built-in data definition class called `DFStringsDefinition` which implements common string operations. Some of the operations available are:

```
// Convert a char code to a string
restrict char_string/1:matching.
char_string(97) = "a"

// Split a string into a list of characters
restrict char_string/1:matching.
explode_string("foo") = ["f","o"."o"]

// Append two strings.
restrict string_append/1:matching.
string_append("foo", "bar") = "foobar"

// Compare two strings
restrict equals_string/1:matching.
equals_string("foo", "foo") = true.
```

Note that all entities defined have a `matching` restriction. Recall that a definition D is given by the sets $dom(D)$ and $com(D)$ and the definiens operation (Section 3.1). Using informal pseudo-code we can describe the definiens operation for a definition with the four operations above:

```
def(char_string(N)) = {string_for_char(N)}.
def(explode_string(S)) = {explode(S)}.
```

[3]Taking the common approach of letting string constants be syntactic sugar for lists of characters is an alternative to consider for the future, of course.

```
def(string_append(A,B)) = {A++B}.
def(equals_string(A,B)) = if A==B
                          then {true}
                          else {}.
```

A good choice for $dom(D)$ is the set of all terms having the same principal functor as any of the given operations. As $com(D)$ we can use the set of all conditions. That the class `DFStringsDefinition` can be implemented in Objective-C should be obvious. It also fulfills the requirements the computation model sets on definitions. Thus, for all purposes, a `DFStringsDefinition` is no different from a definition created using syntactic representations or a definition built up from objects as in Section 5.6.2

Adding a Guard Primitive In the computational model for Gisela, guards in method definitions are only defined to be boolean functions. The framework provides a number of classes from which guards corresponding to the description in 5.1.2 can be built. This provides a reasonable set of building-blocks sufficient for most applications. However, it does not attempt to cover all possible guards needed. If some new guard is needed it can be programmed using object representations, preferrably using the provided classes as a basis.

The framework includes a guard primitive which tests if the two sides of an equation are identical. A more general operation, which is not included, is to test whether the left-hand side matches the right-hand side. A simple subclass of the general class `DFGuardPrimitive` can handle this. In principle we only have to override the method `holds`:

```
- (BOOL)holds:(DFEquation *)eq {
    return [[eq right] matches:[eq left]];
}
```

Another Observer The observer is responsible for selecting the order in which equations are selected for rule application. If we want to restrict rule application to the left-most equation only, we can introduce a new observer class. In this class we override the appropriate method from the default observer to ensure that only the left-most equation is selected:

```
// A LeftMostObserver inherits from DFDefaultObserver.
@interface LeftMostObserver:DFDefaultObserver
{
}
@end

@implementation LeftMostObserver
```

```
- (NSArray *)selectEquationsWithWord:(DFWord *)aWord
                     stateDefinition:(DFStateDefinition *)stateDef
                            andHints:(NSArray *)hints
{
     return [NSArray arrayWithObject:[NSNumber numberWithInt:0]];
}
@end
```

The left-most equation is the one at index `0`. The power of object-oriented programming, in general, and inheritance in particular, lets us experiment with definitional computations using the Gisela framework as a basis.

5.6.5 Other Possibilities

Sometimes it might be better to use only part of the Gisela framework and develop the rest of an application directly in the surrounding programming language. The typical scenario is that some data definition classes are used to represent domain knowledge but that the general computing machinery is replaced by hard-wired behavior.

An example of this is the application MedSummary developed in the MedView project. In MedSummary definition classes of Gisela are used to represent examination records and parts of text templates for text generation. The application also implements a number of specialized subclasses for data definitions. The definition objects are glued together by Objective-C code. The result is a system with excellent performance partly based on a framework for declarative programming.

5.7 ExaminationFinder—A Simple Application

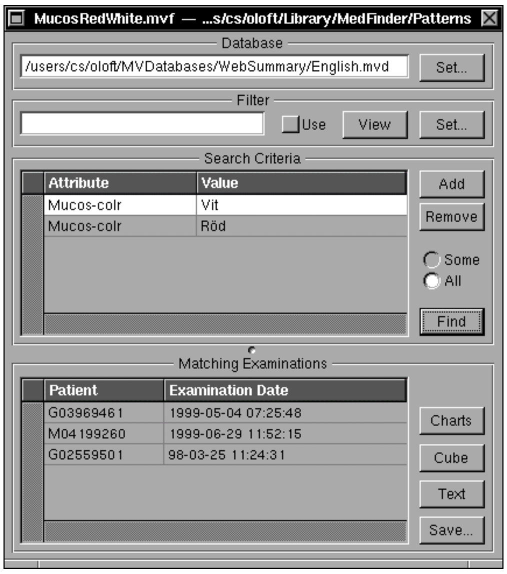
In this section we discuss a simple application with a graphical user interface which uses Gisela for definitional computations.

5.7.1 Using ExaminationFinder

ExaminationFinder, a simple prototype application, lets the user enter a pattern of attribute-value pairs, and then searches a MedView database for examination records matching the pattern. The search panel is shown in Figure 3. The selected records can be used for different tasks by viewing them in different applications.

ExaminationFinder allows two kinds of searches:

1. To look for records having the values of attributes specified in the search panel. It is possible to look for records matching *all* or *some* given criteria. For instance: “Find all records for patients born in Sweden who have the diagnosis oral lichen planus”.



Database: /users/cs/oloft/MVDatabases/WebSummary/English.mvd

Attribute	Value
Mucos-colr	Vit
Mucos-colr	Röd

Patient	Examination Date
G03969461	1999-05-04 07:25:48
M04199260	1999-06-29 11:52:15
G02559501	98-03-25 11:24:31

Figure 3: ExaminationFinder search panel.

2. To look for records in the same way but using an extra definition which collects values into different groups. For instance, if we have a definition where countries are grouped into regions we might try: “Find all records for patients born in Europe”.

If a search pattern is found useful it can be saved for future use.

5.7.2 Set Up

ExaminationFinder is written in Objective-C using OpenStep’s AppKit framework [44]. An application developed using this framework consists of an executable and a number of resources needed by the application. The resources can be pretty much anything, including text files containing Gisela definitions. Thus, data and method definitions needed for definitional computations can be put into the application’s resources folder and then be loaded by the application at run time.

The general methodolgy to use Gisela to build applications including syntactic representations is:

- Decide what data and method definitions are needed for the definitional part of the application.
- Write the syntactic representations of the Gisela program part and add the resulting files to the application’s resources.
- At run time, load the definitional resources into objects representing them and create the desired number of `DFDMachine` objects for running queries.

- Build a `DFQuery` object, representing the query, from user input somehow.
- Send a message with the `DFQuery` object to a `DFDMachine` and ask it to run the query.
- Present the result, represented by a `DFAnswer` object, to the user somehow.

ExaminationFinder uses a single definitional resource file containing method definitions for the computation methods used to search the database.

A MedView database is represented by an object of the class `MVDatabase`. This class knows how to read the database from disk and present it as a number of data definitions, each representing a single examination. ExaminationFinder uses a multi-document architecture, that is, any number of search panels, or documents, can be used at the same time. Each search panel has its own `DFDMachine` object performing definitional computations.

When a new search panel is opened, its controller object creates a new object of the `DFDMachine` class and loads the method definitions to use for computations. This is done with a few lines of code:

```
// Create a method definition parser.
DFMethodParser *mParser = [[DFMethodParser alloc] init];

methods = [mParser parseMethodsAtPath:mPath];
// Create and initialize a machine for definitional computing.
dMachine = [[DFDMachine alloc] initWithDelegate:self];
// Use flat result definitions.
[dMachine setResultSystemType:DFFlatSystemResultType];
```

where `methods` is an array which holds the loaded method definition objects and `mPath` is the path to the text file, in the application's resources folder, where the methods are defined.

5.7.3 Finding Matches

In ExaminationFinder, the user enters attribute-pairs using an ordinary table view. In definitional terms, as used in MedView, that an attribute A has a value V means that there is a connection from A to V using an examination record R. To examine if such a connection exists, we use a state definition $\{V = A\}$ and reduce the right-hand side as far as possible, or until both sides are equal. When there are several attribute-value pairs ExaminationFinder creates a state definition $\{V_1 = A_1, \ldots, V_n = A_n\}$ for *some* queries and a separate state definition for each attribute-value pair for *all* queries.

The attributes and values entered by the user are represented by strings and stored in a special object which works as a data source for the table view. Before we can send a query to the `DFDMachine` these strings must of course be turned into suitable definitional objects. Since Gisela constants are built from strings this is easy to do. The following code creates an equation from the strings `attribute` and `value`:

```
eq = [DFEquation equationWithLeft:
                     [DFConstant constantWithName:value]
                 andRight:[DFConstant constantWithName:attribute]];
```

Using other methods from the Gisela frameworks, some of which were shown in Section 2.5, an object representing the query is constructed.

ExaminationFinder uses two different method definitions, `sri` shown in Section 2.5, and the method definition `srfi` (for *s*ome *r*ight *f*ilter *i*dentity) shown below. Which method definition to use depends on whether a grouping or filtering of values is used or not. When filtering is on `srfi` is used. The meaning of the equations in `srfi` is: (i) if there is an equation with identical left and right-hand sides, the computation is finished (ii) if some attribute can be reduced using `Record`, reduce it and continue (iii) if a value can be grouped using `Filter`, do that and continue.

```
method srfi:[Record,Filter].

srfi = [] # some identity.
srfi = [srfi, r:Record] # some r:in_dom(Record) &
                          all not(identity).
srfi = [srfi, r:Filter] # some r:in_dom(Filter) &
                          all not(identity) &
                          all not(r:in_dom(Record)).
```

5.8 An Interactive System

Following the tradition of declarative programming systems, we have written a (simple) interactive system useful for developing and testing Gisela programs. Since the framework contains almost all functionality needed, the interactive system is written using a few hundred lines of code only. Most of the code is for parsing commands and queries. Parsers for data and method definitions are provided by the Gisela framework. Also, all classes for terms, conditions, equations, methods etc. have a method `stringValue` which gives the syntactical representation of the object.

The architecture of the interactive system is the same as that for ExaminationFinder, e.g. a `DFDMachine` is created to handle computations. The machine is connected to a default observer. While simple, the interactive system does its job. Adding a `DFDMachine` class suitable for debugging would of course be a valuable improvement.

5.9 Discussion

Of course, most declarative programming languages have foreign-language interfaces which allow them to call, or be called from, imperative programming languages, typically C or Java. There are also several implementations [34, 52, 13] which compile programs into an object-oriented model, again typically using Java as the target language. Some of these feature a programming model similar to the object representations discussed here. Jinni [52, 53, 54] is an interesting attempt to combine ideas from Prolog and Java into a tool for gluing together knowledge processing components and Java objects in distributed applications.

The special thing about Gisela is that we take neither representation as being *the* language. Instead, there is a framework providing a number of tools to implement definitional programs. The tools can be used to write programs using syntactic representations and running them in the interactive system. On the other hand, they can be used as an extensible API for building definitional components in Objective-C programs. How to use the tools is up to the user of the framework.

More programs must be written to evaluate the system and we might expect this to lead to some revision of Gisela. To increase the usefulness of the system we must also provide a suitable set of built-in data definitions and standard computation methods to build programs from. Generally, this is one of the areas where existing declarative programming systems are lacking in comparison to traditional imperative or object-oriented ones.

6 Towards a D-Machine

The set of inference rules given in Section 4.2 is a suitable representation to provide an understanding of how definitional computing is realized in Gisela. However, they are at a somewhat too high level to be used as a basis for an implementation. Therefore, we provide a number of state transition rules, which at a lower level, describe how an initial state definition is transformed into a final result definition. The rules describe a machine using depth-first search with backtracking and are the basis for the actual implementation of Gisela. The most notable difference compared to the rules in Section 4.2 is that a computation is described as rewriting an initial goal into a final result definition.

6.1 Rewrite Rules

The notations used are based on the ones in Sections 3 and 4. We only describe modifications and extensions:

- A *goal* is of the form $W\colon S$ where W is a computation condition and S a state definition.

- An *index-set* is a sequence $\{I_1, \ldots, I_n\}$ where the elements are conditions or computation conditions.

- A *computation element* is either a goal, an equation, or an index set.

- A *computation stack* is a list of computation elements. We use Δ to denote a computation stack. $[Y|\Delta]$ is the stack with top Y.

- A *result stack* is a list of result definitions.

- A *computation frame* is a triple $\langle \Delta, R, \theta \rangle$, consisting of a computation stack Δ, a result stack R, and a substitution θ.

- A *computation state* is a stack $F; \Phi$ of computation frames. F is the active or topmost frame, and Φ the rest of the stack. Each computation frame represents an alternative way to compute a solution. We write $\{\}$ for the empty computation state.

The final states of the transition system are $yes(X, \theta)$, where X is the computed result definition and θ a substitution, and *no* which indicates that no answer could be computed.

(1) Init

$$M{:}S \to \langle [M{:}S], [], \emptyset \rangle \ .$$

At the top level only a single method is allowed.

(2) Success

$$\langle [], [X], \theta \rangle ; \Phi \to yes(X, \theta) \ .$$

Alternative solutions are computed by restarting the machine from the state Φ.

(3) Failure

$$\{\} \to no \ .$$

(4) Goal Success

$$\langle [\epsilon{:}S|\Delta], R, \theta \rangle ; \Phi \to \langle \Delta, [S|R], \theta \rangle ; \Phi \ .$$

When a goal is fully evaluated the result S is moved to the result stack.

(5) Index

$$\langle [\{I_1, \ldots, I_n\} | \Delta], [X_1, \ldots, X_n | R], \theta \rangle; \Phi \rightarrow \langle \Delta, [X | R], \theta \rangle; \Phi ,$$

where $n \geq 0$, $X = \mathcal{O}_{trans}(\{I_1 = X_n, \ldots, I_n = X_1\})$. When an index-set is on top of the computation stack a new result definition is built from pending definitions previously pushed onto the result stack.

(6) Choice

$$\langle [W(W_1, W_2){:}\, S | \Delta], R, \theta \rangle; \Phi \rightarrow \langle [WV_1{:}\, S | \Delta], R, \theta \rangle; \ldots; \langle [WV_m{:}\, S | \Delta], R, \theta \rangle; \Phi ,$$

where $\{V_1, \ldots, V_m\} = \mathcal{O}_{seq}(\{W_1, W_2\}), m \in \{1, 2\}$.

(7) Method

$$\langle [(WM{:}\, S) | \Delta], R, \theta \rangle; \Phi \rightarrow \langle [(WW_1{:}\, S), \ldots, (WW_n{:}\, S), \{W_1, \ldots, W_n\} | \Delta], R, \theta \rangle; \Phi ,$$

where $\mathcal{M}(M) = \{W_1, \ldots, W_n\}, n \geq 1$. $\mathcal{M}(M)$ is the definiens of the method name M in the method $\mathcal{M}$, that is

$$\{W_i \mid M = W_i \# C_i \in \mathcal{M} \wedge C_i(S)\}.$$

If $\mathcal{M}(M) = \{\}$ then

$$\langle [(WM : S) | \Delta], R, \theta \rangle; \Phi \rightarrow \Phi .$$

Note that the index-set $\{W_1, \ldots, W_n\}$ is pushed onto the computation stack to make it possible to build the desired result definition once the required goals have been evaluated.

(8) Equation Left

$$\langle [W\overline{D}{:}\, S | \Delta], R, \theta \rangle; \Phi \rightarrow \langle [e_1, W\overline{D}{:}\, S | \Delta], R, \theta \rangle; \ldots; \langle [e_n, W\overline{D}{:}\, S | \Delta], R, \theta \rangle; \Phi ,$$

where $\{e_1, \ldots, e_n\} = \mathcal{O}_{seq}(S), n \geq 1$.

(9) Definition Left

$$\langle [(a = B), W\overline{D}{:}\, S | \Delta], R, \theta \rangle; \Phi \rightarrow \langle [G_{11}, \ldots, G_{1k}, I_1 | \Delta\sigma_1], R\sigma_1, \theta\sigma_1 \rangle; F_2; \ldots; F_n; \Phi ,$$

where we have

- $D_{suff}(a) = \{\sigma_1, \ldots, \sigma_n\}$, $D_i = D(a\sigma_i) = \{A_{i1}, \ldots, A_{ik}\}, n \geq 1, k \geq 0$,
- $G_{ij} = W{:}\, (A_{ij}/a\sigma_i) S\sigma_i$,
- $I_i = \{A_{i1}, \ldots, A_{ik}\}$,
- $F_i = \langle [G_{i1}, \ldots, G_{ik}, I_i | \Delta\sigma_i], R\sigma_i, \theta\sigma_i \rangle$.

Note that k can be different for each n.

(10) Vector Left

$$\langle[((A,B)=C),W\overline{D}{:}S|\Delta],R,\theta\rangle;\Phi \rightarrow \langle[W{:}S_1|\Delta],R,\theta\rangle;\ldots;\langle[W{:}S_m|\Delta],R,\theta\rangle;\Phi\ ,$$

where $\{C_1,\ldots,C_m\}=\mathcal{O}_{seq}(D((A,B))), m\in\{1,2\}$ and $S_i=C_i/(A,B)S$.

(11) Arrow Left

$$\langle[((A\rightarrow B)=C),W\overline{D}{:}S|\Delta],R,\theta\rangle;\Phi \rightarrow \langle[W{:}S_1,W{:}S_2,\{A,B\}|\Delta],R,\theta\rangle;\Phi\ ,$$

where S_1 and S_2 are given by

- $S_1=(((A\rightarrow B)\ominus S)\downarrow A)$,
- $S_2=(B/(A\rightarrow B))S$.

(12) Fail Left

$$\langle[(A=C),W\overline{D}{:}S|\Delta],R,\theta\rangle;\Phi\rightarrow\Phi\ ,$$

if A is a variable or $A=\top$ or $A=\bot$.

(13) Equation Right

$$\langle[W\underline{D}{:}S|\Delta],R,\theta\rangle;\Phi\rightarrow\langle[e_1,W\underline{D}{:}S|\Delta],R,\theta\rangle;\ldots;\langle[e_n,W\underline{D}{:}S|\Delta],R,\theta\rangle;\Phi\ ,$$

where $\{e_1,\ldots,e_n\}=\mathcal{O}_{seq}(S), n\geq 1$.

(14) Definition Right

$$\langle[(B=a),W\underline{D}{:}S|\Delta],R,\theta\rangle;\Phi\rightarrow F_{11};\ldots;F_{nk_n};\Phi\ ,$$

where we have

- $D_{mgu}(a)=\{\sigma_1,\ldots,\sigma_n\}, D(a\sigma_i)=\{A_{i1},\ldots,A_{ik}\}, n\geq 0, k\geq 0$,
- $G_{ij}=W{:}S\sigma_i(A_{ij}/a\sigma_i)$,
- $F_{ij}=\langle[G_{ij}|\Delta\sigma_i],R\sigma_i,\theta\sigma_i\rangle$.

Note that k can be different for each n. If $n=0$ or $k_1=0$ the rules becomes:

$$\langle[(B=a),W\underline{D}{:}S|\Delta],R,\theta\rangle;\Phi\rightarrow\Phi\ .$$

(15) Vector Right

$$\langle[(A=(B,C)),W\underline{D}{:}S|\Delta],R,\theta\rangle;\Phi\rightarrow\langle[W{:}S_1,W{:}S_2,\{B,C\}|\Delta],R,\theta\rangle;\Phi\ ,$$

where $S_1=S(A/(A,B))$ and $S_2=S(B/(A,B))$.

(16) Arrow Right

$$\langle[(A = (B \to C)), W\underline{D}{:}\, S|\Delta], R, \theta\rangle; \Phi \to \langle[W{:}\, S'|\Delta], R, \theta\rangle; \Phi\ ,$$

where $S' = B \oplus (S(C/(B \to C)))$.

(17) Fail Right

$$\langle[(A = C), W\underline{D}{:}\, S|\Delta], R, \theta\rangle; \Phi \to \Phi\ ,$$

if C is a variable or $C = \top$ or $C = \bot$.

(18) Identity Equation

$$\langle[WD{:}\, S|\Delta], R, \theta\rangle; \Phi \to \langle[e_1, WD{:}\, S|\Delta], R, \theta\rangle; \ldots; \langle[e_n, WD{:}\, S|\Delta], R, \theta\rangle; \Phi\ ,$$

where $\{e_1, \ldots, e_n\} = \mathcal{O}_{seq}(S), n \geq 1$.

(19) Identity

$$\langle[(a = b), WD{:}\, S|\Delta], R, \theta\rangle; \Phi \to \langle[W{:}\, S|\Delta]\sigma, R\sigma, \theta\sigma\rangle; \Phi\ ,$$

if $\sigma = mgu(a, b)$.

$$\langle[(A = B), WD{:}\, S|\Delta], R, \theta\rangle; \Phi \to \Phi\ ,$$

if A and B are terms which are not unifiable, or A or B is a condition which is not a term.

6.2 Result Definitions

A good question is whether there is ever any point in building a full result definition. The introduction of a complex result definition was motivated by a wish to study properties of computations and a need to find out from *what* a specific equation in a flattened result definition was computed. However, very little work has been done in this area so far. The developed applications and examples have been either (functional) logic programs, where the result definition is not needed at all, or programs where the flattened form of the result definition is the interesting part of the answer.

From an efficiency point of view, the problem with building full result definitions is that the size grows relative to the number of steps in the computation and thus consumes a very large amount of memory. Even when all result definitions are flattened, a large number of index sets are created and put on the computation stack, only to be discarded later on. Whether full result definitions are needed and exactly what they should contain is an area for future investigations. That they are present in Gisela is in line with the goal of providing a framework useful for several different tasks.

6.3 Discussion

We have chosen to implement Gisela as a system which uses depth-first search and backtracking to find answers to queries. This choice is debatable since, in general, the search procedure is not complete and may miss obvious answers implied by the program.

Historically, using depth-first search is the most common approach in programming languages involving search for answers, among them Prolog and Mercury [50]. Today, it is possible to discern a trend where other approaches are used, e.g. systems like Curry [33], Escher [42], and Oz [48, 49].

Breadth-first search was used in an earlier version of Gisela, see Section 8. However, it was deemed that for a system with a focus on being *practical*, such as Gisela, the efficiency gained by using depth-first search instead was more important than the loss of completeness.

7 Implementation

The Gisela framework has been implemented in Objective-C using the Foundation framework of OpenStep [44]. The Foundation framework provides a level of operating system independence, to enhance portability. Thus, Gisela runs on any platform for which the appropriate OpenStep runtime system is available. We are considering implementing a version of Gisela in Java for even greater portability. This should be trivial due to the similarity between Java and Objective-C.

The implementation of Gisela is divided into three frameworks, one for data definitions, one for method definitions and one implementing computations. A framework in this setting corresponds to a package in Java and is a collection of classes that are grouped together, since they conceptually form a unit. This unit should provide some functionality useful for building other frameworks and applications. All entities of Gisela are represented by objects of various classes. It follows that, since a definitional machine is just another object, it can be directly used in any Objective-C application.

It should be noted that the purpose of this section is not to give a detailed description of the implementation, but rather to hint at the general ideas and the design philosophy used. We discuss possible alternatives in Section 7.5 below.

7.1 Overall Structure

The main design goal behind the implementation of Gisela is to create a portable implementation that can easily be integrated into real-world applications with graphical user-interfaces. The most practical way to achieve this, in our opinion, is to make it very simple to include Gisela as a component for reasoning in applications using existing frameworks for GUI, not to provide GUI facilities in Gisela. Thus, we have implemented Gisela as a framework (package) which provides all functionality through a number of objects that can be used in Objective-C applications.

The three frameworks which together make up Gisela are:

- `DFDefinitions`, where terms, conditions and data definitions are implemented. This framework is the basis for Gisela and is needed by the other two.
- `DFMethods`, which implements all classes needed to build method definitions.
- `DFComputing`, which uses classes from both `DFDefinitions` and `DFMethods` and implements the classes which manage actual definitional computations.

The main motivation for the separation is that definition classes may be useful by their own without the rest of the definitional computing machinery. The other motivation is to have reasonably sized frameworks.

The design of the frameworks is not particularly dependent on any specific features of Objective-C, thus a port to another object-oriented language should not be to hard to do.

7.2 Implementing Data Definitions

In terms of lines of code and number of public classes, `DFDefinitions` is by far the largest of the three frameworks. In part, this is because `DFDefinitions` contains a number of classes needed to handle the files used to store examination records in MedView. From a design point of view, it can be argued that these classes should not be part of the basic framework but be defined in an extension. Nevertheless, since Gisela is intended for use in MedView we have put them into the framework.

7.2.1 Terms and Conditions

Data definitions are built using terms, conditions, and equations. The common properties of terms are implemented by the abstract class `DFTerm`, the common properties of conditions by the abstract class `DFCondition`. Both these classes implement the `DFConditionProtocol`. A protocol in Objective-C corresponds to an interface in Java. The `DFConditionProtocol` in turn inherits a number of methods from the `DFVariableCopyingProtocol` which describes different kinds of copying. Thus:

```
DFVariableCopying
   DFCondition
      DFTerm
```

As an example we show `DFTerm.h`

```
#import <Foundation/Foundation.h>
#import <DFDefinitions/DFTermProtocol.h>

@interface DFTerm : NSObject<DFTerm, NSCoding, NSCopying>
{
}
@end
```

This tells us that `DFTerm` is a subclass of the root class `NSObject` which implements the protocols `DFTerm`, `NSCoding`, and `NSCopying`, but declares no methods of its own. We have subclasses of `DFTerm` for constants, variables, and compound terms, and subclasses of `DFCondition` for arrow and comma conditions. These classes are very straightforward. The most interesting is perhaps `DFVariable`:

```
@interface DFVariable : DFTerm
{
    long timeStamp;
    id<DFTerm> value;
}
+(id)variable;
...
@end
```

The instance variable `timeStamp` represents the time the variable was bound and is needed to make it possible to undo variable bindings correctly when backtracking occurs. The usage of timestamps like this is standard methodology in implementations of logic programming languages [59].

The implementation is closely related to the description of terms, conditions, equations, and data definitions given in Section 4.1.1. The reason for this is, of course, the idea that it should be possible to use Gisela directly by building data definitions as objects without using any syntactic representation which is parsed and compiled into a program.

7.2.2 Data Definition Classes

We have implemented a number of different data definition classes. All share the methods described in the `DFDefinition` protocol:

```
@protocol DFDefinition <NSObject>

- (NSString *)name;
- (BOOL)inDom:(id)anObject;
- (BOOL)inCom:(id)anObject;
- (NSArray *)def:(id)anObject;
- (id)clause:(id)anObject;
- (NSArray *)def:(id)anObject evaluator:(id)machine
                              operationId:(unsigned)opId
                                 redoable:(BOOL *)hasAlts;
- (id)clause:(id)anObject evaluator:(id)machine
                          operationId:(unsigned)opId
                             redoable:(BOOL *)hasAlts;
...
@end
```

For a data definition class to be valid, the methods in this protocol should implement the behavior given by the abstract description of a data definition given in Section 4.1.1. There are two different versions of the methods for `def` and `clause`. The ones with a single argument may be useful if a definition class is used without the rest of the machinery for definitional computations. The two last methods are for enumerating all possible results. The *D-Machine* described in Section 7.4 treats data definitions as black boxes. All it knows about data definitions is that a definition may be used to find the definiens of an object. It also knows that there may, in general, be more than one result. If `def:evaluator:operationId:redoable` is called multiple times from a machine using the same `opId`, all answers are enumerated.

Currently, all data definition classes inherit from the abstract definition class `DFDefinition` but this is not a requirement. Other base classes for data definitions may be written as long as they implement the `DFDefinition` protocol.

In the general case, computing the definiens of a term with respect to a data definition is a complex operation involving the computation of a-sufficient substitutions. To avoid unnecessary overhead we have implemented several specialized data definition classes handling various simpler definitions. We have also separated the definition classes into static and modifiable definitions since operations may be implemented in a more efficient manner if we know that the definition will not change over time. The most common classes are:

- `DFDefinition`, abstract definition class from which all other definition classes in the framework inherits.
- `DFConstantDefinition`, subclass of `DFDefinition`, suitable to use when the left-hand sides of all equations are constants. This class is used when a data definition is declared `constant` using the syntactic representations of Section 5.1.
- `DFMatchingDefinition`, subclass of `DFDefinition`, suitable to use when matching, and not unification, should be applied in the definiens operation. This class is used when a syntactically represented data definition is declared as `matching`.

- `DFUnifyingDefinition`, subclass of `DFMatchingDefinition`, used for general data definitions. This class also allows specifications which describe how each equation may be used, e.g., matching only. Therefore a unifying definition really subsumes the two classes above.
- `DFGCLAUnifyingDefinition`, subclass of `DFUnifyingDefinition`. For compatibility with GCLA, this class includes all terms in the domain of a data definition and returns `false` instead of the empty set for terms not defined. This is the class used when a syntactically described data definition is declared `gcla`.
- `DFModifiableDefinition`, an abstract subclass of `DFDefinition` which implements common behavior of mutable definitions.

All of the above definitions are created from a list of equations using the method:

```
- (id)initWithName:(NSString *)aString
      andEquations:(NSArray *)someEquations;
```

`DFUnifyingDefinition` also allow directives for how the equations should be handled:

```
- (id)initWithName:(NSString *)aString
         equations:(NSArray *)someEquations
     andDirectives:(NSDictionary *)aDict;
```

When a definition is created, an internal representation of the equations suitable for computing definiens and clause of terms is built. The resources required for building this and the efficiency of the resulting representation are the parameters to consider when deciding what kind of data definition to use.

`DFConstantDefinition` uses a simple hash table to find the definiens of a given constant. `DFMatchingDefinition` and `DFUnifyingDefinition` use one hashtable indexed on the principal functor of a term and then one hashtable indexed on the first argument of a term. Thus, given a definition like

```
f(a) = a.
f(b) = b.
f(c) = c.
```

performing `clause(f(b))` is done by two lookups and leaves no choice points. This is not very complicated. Performing definiens in matching definitions is not particularly complicated either. Currently, to find $D(a)$, a linear search among the equations with the same principal functor as a is performed to collect all matching clauses. The hard part is to implement the definiens operation of `DFUnifyingDefinition` in the general case involving computation of a-sufficient substitutions. Various algorithms for this are described in [7, 30, 38]. To the best of our knowledge, all previous implementations are implemented in Prolog using built-in unification and backtracking. In addition, the descriptions of algorithms are expressed in a manner heavily influenced by the intended implementations.

The algorithm currently in use for computing definiens in the general case is adopted from Algorithm 3, without guards and constraints, in [7]. The general idea of this algorithm is that it is possible to build a representation of the definition which in essence pre-computes all possible a-sufficient substitutions for all terms defined in the definition. There are two advantages with this in the Gisela setting. First, it makes performing definiens more efficient, and second, it makes it possible for a definition object to tell if there are any more alternatives to consider. The backside is that creating the representation is exponential with respect to the number of equations with unifiable heads. For large databases this is not feasible. Therefore, if a term will only be used to the right in equations it is possible to turn off the pre-computation of a-sufficient substitutions using the `restrict right` directive.

A project for the future is to allow guards in unifying definitions and not only matching definitions. This would involve implementing some algorithm similar to Algorithm 3, with guards and constraints, in [7]. The algorithm as such is quite similar to the current pre-computation of a-sufficient substitutions. The hard part would most likely be to extend variables to handle constraints in an efficient manner. The algorithms presented in [7, 38] rely heavily on features of SICStus Prolog to handle constraints on variables.

Finally, the strictly modular construction of Gisela where data definitions are treated as black boxes by the rest of the machinery makes it possible to introduce new improved definition classes without affecting any other part of the framework.

7.3 Implementing Method Definitions

As with data definitions, the implementation of method definitions is closely related to previous descriptions, particularly the one in Section 5.1.2. The reason is the same: it should be possible to build method definitions directly using the various classes in the framework. To achieve this, the framework is built to map the conceptual description of method definitions directly onto a number of classes.

The general structure of a method definition is that it is a sequence of equations

$$M = Word\#Guard$$

where $Word$ is computation condition describing a sequence of operations to perform, and $Guard$ contains restrictions with respect to the current state definition on when the equation may be applied. Guards are built using guard-primitives describing tests with respect to a single equation. In principle methods are implemented through the classes:

- `DFGuardConstraint`, tests on a single condition.

- `DFGuardPrimitive`, tests on a single equation.
- `DFGuard`, tests on a state definition.
- `DFWord`, a sequence of operations.
- `DFMethodScheme` and `DFMethod` for method definitions with and without parameters respectively.

The computation model of Gisela really says nothing more about guards than that they implement tests with respect to the current state definition. Thus, the framework classes implement all the guard functionality described in Section 5.1.2 but there are no restrictions on the possibility of adding new classes.

All that is required of a class to introduce a new guard primitive is that it implements the following protocol:

```
@protocol DFGuardPrimitive<DFMethodObject, NSCopying, NSCoding>
- (BOOL)holds:(DFEquation *)eq;
@end
```

A new guard class must implement the protocol:

```
@protocol DFGuard<DFMethodObject, NSCopying, NSCoding>
- (BOOL)isTrue:(DFStateDefinition *)stateDef;
- (BOOL)isTrue:(DFStateDefinition *)stateDef
       indexes:(NSMutableArray *)indexes;
@end
```

The second method of the `DFGuard` protocol is used to communicate which equations of the given state definition make the guard hold. This is used in computations to help the observer select an equation for reduction.

The class `DFMethod` is a subclass of `DFModifiableDefinition`. What is special about method definitions is that given a method definition M, it is always used to lookup the definiens of the constant M with respect to a given state definition. Therefore, two new methods are added:

```
@protocol DFMethod<DFMethodObject>
- (NSArray *)defWithStateDefinition:(DFStateDefinition *)stateDef;
- (NSArray *)defWithStateDefinition:(DFStateDefinition *)stateDef
                         indexHints:(NSMutableArray *)idxHints;
@end
```

As with guards and many other objects, users using the Gisela frameworks may implement new method definition classes as long as they implement the `DFMethod` protocol. Of course subclassing is also possible.

All the classes of `DFDefinitions` and `DFMethods` implement the protocol `NSCoding`, which means that objects may be archived for permanent storage.

7.4 Implementing a D-Machine

The framework DFComputing provides a rather limited number of classes for definitional computing, most notably the DFDMachine class. A D-machine is an interpreter which takes a query, as described in Section 5.1.3 and evaluates it according to the rewrite rules given in Section 6.1. While an interpreter, we have tried not to make it unnecessarily inefficient. One exception from this, in the current implementation, is the heavy use of objects for everything. For example it would be faster to use C arrays instead of array objects. As the structure of the implementation stabilizes, we expect to move to a lower level and use more pure C code. This can be done piecemeal since C is a subset of Objective-C.

The DFDMachine, as such, is only a shell that is used to set up computations and connections in various ways. The actual computations are performed by an object of the private class DFComputor. A DFDMachine object connects the computor object with its observer, decides whether computations should be run in a separate thread, handles communication between the computor and the calling application etc.

7.4.1 The Computor

Computing a result definition from the initial goal is handled by a DFComputor. After some initializations it starts a loop which runs until the goal is stopped for some reason, e.g., a result is computed or the caller decides that the computation should stop. In a simplified form the loop looks like this:

```
- (id)runMainLoopBreakAtAnswer:(BOOL)returnAnswer {
    while (continueComputing) {
        switch ([self selectRule]) {
            case DFSuccessRule:
                [self performSuccess];
                break;
            ...
            case DFDefiniensRule:
                [self performDefiniens];
                break;
            ...
    }
    return result;
}
```

The current state of the computation is inspected by the method selectRule, and depending on this the correct rule (as described in Section 6.1) is applied. The most important variables describing the state of a computor are:

`timeStamp`	the current timestamp of the computor.
`activeFrame`	a pointer to the current computation frame.
`choicePointStack`	what the name indicates.
`trailStack`	stack with variables which might be undone.
`observer`	a pointer to the current observer.
`csReg`	the element on top of the computation stack
`cs2Reg`	the element below `csReg` .
`wReg`	pointer to currently selected computation condition.
`lcReg`	pointer to last element of `wReg`.
`iReg`	selected equation index.
`eReg`	pointer to selected equation.
`cReg`	pointer to selected condition.

The `choicePointStack` stores `DFChoicePoint` objects containing all the information necessary to create the next frame, in case an alternative should be tried.

7.4.2 Choice Points

When there are alternative paths in a computation, a `DFChoicePoint` object is created and pushed onto the choice point stack. The choice point object stores a copy of the current computation frame, the current timestamp, an index into the trail stack indicating where to undo variable bindings from, what kind of choice caused the choice point, and some extra information depending on the kind of choice point.

A `DFChoicePoint` object knows how to create the sequence of computation frames representing all possible alternatives available from the choice point. These alternatives are enumerated one by one by calling the method `nextAlternative`. When the computor needs a new frame to continue computing it uses its method `popFrameStack`:

```
- (void)popFrameStack {
    BOOL stillBuilding = YES;
    DFFrame *newFrame = nil;
    while (stillBuilding && ![choicePointStack isEmpty]) {
        DFChoicePoint *currentChoicePoint = [choicePointStack top];
        // clear variables bound since current choice
        [self untrail:[currentChoicePoint trailStackPointer]];
        if (newFrame = [currentChoicePoint nextAlternative])
            stillBuilding = NO;
        else
            [choicePointStack pop];
    }
    [self pushActiveFrame:newFrame];
}
```

7.4.3 The Observer

The framework provides a default observer as described in Section 4.3. Implementing the default observer is trivial. The methods an observer must implement are declared in the protocol `DFComputingObserver`. New observers may be created either by subclassing the default observer or by implementing new objects that adhere to the observer protocol.

7.5 Discussion

The use of a proprietary framework such as OpenStep in the development of a programming system like Gisela is somewhat unusual. OpenStep was chosen for two reasons: (i) at the time the implementation was started we believed that MedView would remain based on OpenStep for at least a few years, (ii) OpenStep is arguably one of the most well designed object-oriented frameworks around and provides both access to C and a fully dynamic runtime system. As mentioned above, the design of the implementation is such that it should be easily portable to Java, Ada95 or C++. A port to Java would be easiest, and will most likely be made once the computation model is fixed. So far, Objective-C remains faster than Java though.

One place where many unnecessary computations are performed is in the evaluation of guards in method definitions. Typically, method definitions are written in such a way that at most one equation can be applied to the current state definition. However, all guards are always evaluated. Unnecessary computations could be avoided if it was possible to declare that a method definition was deterministic, meaning that at most one guard could hold.

8 Conclusions

We have presented the Gisela framework for definitional programming. As any reasonably ambitious programming system it is a compromise between different, and, at times, conflicting, requirements. The system has been implemented and a number of applications have been written to test performance and try out programming methodologies. Next, Gisela will be used to re-model the definitional machinery used in the MedView project. Our belief is that, although some refinements will be needed when Gisela is applied to a real-world project such as MedView, the basic computational machinery will remain.

The Gisela framework is our fourth attempt in a series of experiments for finding a definitional programming model which can serve both as a successor to GCLA, allow for new programming methodologies, and be useful for knowledge representation and reasoning in MedView. The overall idea goes back to [22] where a model for computing with definitions radically different from the GCLA approach was described. Another important input were the ideas put forward in [58].

The first system we built was implemented in Prolog and allowed only computations using atoms. It was followed by an implementation in Objective-C using breadth-first search which always computed all answers to queries. Like the Prolog system, it did not use any constructed conditions, but did allow matching in the definiens operation. This second system was discarded due to some fundamental flaws due to misunderstandings of the intended behavior. However, it could be used for things like computing basic separated programs.

A problem during the early phases of development was that it was very unclear how separated programs, and programs doing things like computing definitional similarity measures should be understood. Another was that we tried hard to avoid introducing logical variables. Instead, the vision was to develop a kind of "declarative assembler" on top of which conditions and variables should be programmed. The third prototype developed fixed the problems with the second one and was built on a concept of an abstract search-tree from which different concrete search algorithms, e.g., depth-first search could be derived by subclassing the abstract machinery. Actually, a fair amount of code, in particular most of the code for implementing method definitions and simple data definitions, was inherited into the Gisela framework from this system.

However, the problems of the "declarative-assembler" approach remained. There was something which made it very hard to see how it would be possible to realize the goal of building higher-level programming methodologies on-top of the basic system in a nice manner. Our goal of building a practical system was nowhere near being realized.

We then decided to opt for the definitional computing model that we have described here as the Gisela framework. Compared to the previous attempts, the difference is the presence of logical variables in data definitions and built-in rules for handling constructed conditions. The rules for constructed conditions were modeled after the standard GCLA rules. Finally, we had a system that came reasonably close to our original goals and which we felt would be possible to use for building practical applications.

If we look back at the goals set up in Section 1 and in [22, 58] some things worth noting are:

- Gisela keeps the distinction between declarative and procedural parts used in GCLAII. Programming Gisela is similar enough to programming GCLA to allow reuse of many techniques. On the other hand, Gisela is different enough to allow things like separated programming in a natural way.

- The abstract way in which definitions are introduced solves the problem of the definiens operation being too general. Gisela does provide several different built-in data definition classes of different complexity. Furthermore, the framework is open for the addition of new data definition classes.

- In [58], an important goal was that computations should be able to interact in a natural way with the outside world. The Gisela framework is less oriented towards interactive computations than the original vision. Interaction can mainly be handled through specialized observers. However, the observer only gets called at specific points during computations.

- In [58], it was stated that programs should be compiled to C for portability. What we had in mind was a Gisela to C compiler which would allow easy porting to essentially any platform. The use of Objective-C instead restricts portability but has greatly simplified development. Also, the notion of a compiler does not really apply in the current setting.

- Another goal which has not been realized is to give explicit control of the system's general search behavior. In Gisela depth-first search with backtracking is always used. Providing means for other search-strategies is an area for future work.

There are two things that sets Gisela apart from other systems for declarative programming: (i) Gisela does not attempt to be a general-purpose programming language, rather it is a system for realizing a certain set of definitional models, (ii) Gisela is a framework with a rather loose definition, specifically aimed at allowing experiments and modifications within the general model set up in Section 3. The aim of declarative systems such as Prolog [15], Haskell [35], Mercury [50], Curry [33], and Oz/Mozart [48, 60] is to provide full-fledged programming languages suitable as alternatives to the commonly used imperative and object-oriented ones. The outspoken aim of Mercury is to provide an alternative to C for large scale projects. Mozart is geared towards distributed applications. Being general-purpose languages, they also provide libraries to build GUIs [14, 32]. There is also a need for sophisticated programming environments and software libraries, an area where the mentioned systems so far are not on par with imperative languages. Since Gisela is only aimed at realizing definitional models of systems we have instead focused on simplifying the use of Gisela in combination with object-oriented industrial-strength tools for building GUI-based, user-friendly applications. For our purposes this gives us the most practical set of tools.

Finally, for the future an interesting question is: Will declarative programming ever be a widespread generally used programming paradigm? We believe that a crucial factor for the success of declarative programming is easy integration with commonly used imperative and object-oriented systems and some serious work on programming environments and library modules. Gisela is our attempt at providing a useful declarative programming component for, among other things, future work in the MedView project.

References

[1] Y. Ali, G. Falkman, L. Hallnäs, M. Jontell, N. Nazari, and O. Torgersson. Medview: Design and adoption of an interactive system for oral medicine. In *Proceedings of Medical Informatics Europe (MIE'00), Hannover, Germany, August 2000*, 2000. To appear.

[2] S. Antoy. Lazy evaluation in logic. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 371–382. Springer-Verlag, 1991.

[3] S. Antoy. Definitional trees. In *Int. Conf. on Algebraic and Logic Programming ALP'92*, number 632 in Lecture Notes in Computer Science, pages 143–157. Springer-Verlag, 1992.

[4] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, 1994.

[5] S. Antoy and A. Middeldorp. A sequential reduction strategy. *Theoretical Computer Science*, To Appear.

[6] M. Aronsson. Methodology and programming techniques in GCLA II. In *Extensions of logic programming, second international workshop, ELP'91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[7] M. Aronsson. *GCLA, The Design, Use, and Implementation of a Program Development System.* PhD thesis, Stockholm University, Stockholm, Sweden, 1993.

[8] M. Aronsson. Implementational issues in GCLA: A-sufficiency and the definiens operation. In *Extensions of logic programming, third international workshop, ELP'92*, number 660 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1993.

[9] M. Aronsson, L.-H. Eriksson, L. H. A. Gäredal, and P. Olin. GCLA-generalized horn clauses as a programming language. In *Proceedings of SCAI-89*, 1989.

[10] M. Aronsson, L.-H. Eriksson, A. Gäredal, L. Hallnäs, and P. Olin. The programming language GCLA: A definitional approach to logic programming. *New Generation Computing*, 7(4):381–404, 1990.

[11] M. Aronsson, L.-H. Eriksson, L. Hallnäs, and P. Kreuger. A survey of gcla: A definitional approach to logic programming. In P. Schroeder-Heister, editor,

Extensions of logic programming: Proceedings of a workshop held at the SNS, Universität Tübingen, 8-9 december 1989, number 475 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1991.

[12] L. Augustsson. Compiling Pattern Matching. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, Nancy, France, 1985.

[13] N. Benton, A. Kennedy, and G. Russel. Compiling Standard ML to Java bytecodes. In *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*. ACM Press, 1998.

[14] M. Carlsson and T. Hallgren. Fudgets: A graphical user interface in a lazy functional language. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, 1993.

[15] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[16] L.-H. Eriksson. *Finitary Partial Inductive Definitions and General Logic*. PhD thesis, University of Stockholm, May 1993.

[17] G. Falkman. Program separation as a basis for definitional higher order programming. In U. Engberg, K. Larsen, and P. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory*. Aarhus, 1994.

[18] G. Falkman. Definitional program separation. Licentiate thesis, Chalmers University of Technology, 1996.

[19] G. Falkman. Program separation and definitional higher order programming. *Computer Languages*, 23(2–4):179–206, 1997.

[20] G. Falkman. Similarity measures for structured representations: a definitional approach. In E. Blanzieri and L. Portinale, editors, *EWCBR-2K, Advances in Case-Based Reasoning*, Lecture Notes in Artificial Intelligence. Springer–Verlag, 2000. To appear.

[21] G. Falkman, L. Hallnäs, and O. Torgersson. Program separation in GCLA. In A. Momigliano and M. Ornaghi, editors, *Proceedings of the Post-Conference Workshop on Proof-Theoretical Extensions of Logic Programming*, pages 31–37, June 1994.

[22] G. Falkman, L. Hallnäs, and O. Torgersson. Computing equalities. Manuscript, 1997.

[23] G. Falkman and O. Torgersson. Programming methodologies in GCLA. In R. Dyckhoff, editor, *Extensions of logic programming, ELP'93*, number 798 in Lecture Notes in Artificial Intelligence, pages 120–151. Springer-Verlag, 1994.

[24] G. Falkman and J. Warnby. Technical diagnoses of telecommunication equipment: An implementation of a task specific problem solving method (TDFL) using GCLA II. Research Report SICS R93:01, Swedish Institute of Computer Science, 1993.

[25] L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.

[26] L. Hallnäs. WM94: program separation in GCLA. In *Proceedings of La Wintermöte 94*, pages 93–94. Department of Computing Science, Chalmers University of Technology, 1994.

[27] L. Hallnäs. Classifying algorithms – definitions, intensionality, algorithms, the classification problem. Manuscript, 1997.

[28] L. Hallnäs, M. Jontell, and N. Nazari. MEDVIEW – formalisation of clinical experience in oral medicine and dermatology: The structure of basic data - abstract. In *Proceedings of the Das Wintermöte'96*. Department of Computing Science, Chalmers University of Technology, 1996.

[29] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(2):261–283, 1990. Part 1: Clauses as Rules.

[30] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(5):635–660, 1991. Part 2: Programs as Definitions.

[31] M. Hanus. Combining lazy narrowing and simplification. In *Proc. 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 370–384. Springer LNCS 844, 1994.

[32] M. Hanus. A functional logic programming approach to graphical user interfaces. In *Proc. of the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, volume 1753 of *Lecture Notes in Computer Science*, pages 47–62. Springer-Verlag, 2000.

[33] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[34] M. Hanus and R. Sadre. An abstract machine for Curry and its concurrent implementation in Java. *Journal of Functional and Logic Programming*, 6, 1999.

[35] P. Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[36] P. Kreuger. GCLA II: A definitional approach to control. Licentiate thesis, Chalmers University of Technology, 1992.

[37] P. Kreuger. GCLA II: A definitional approach to control. In *Extensions of logic programming, second international workshop, ELP91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[38] P. Kreuger. *Computational Issues in Calculi of Partial Inductive Definitions.* PhD thesis, Department of Computing Science, University of Göteborg, Göteborg, Sweden, 1995.

[39] H. Kuchen, R. Loogen, J. J. Moreno-Navarro, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proceedings of the Second International Conference on Algebraic and Logic Programming*, number 463 in Lecture Notes in Computer Science. Springer-Verlag, 1990.

[40] D. Larkin and G. Wilson. *Object-Oriented Programming and the Objective C Language.* NeXT Software Inc, 1996.

[41] J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second extended edition, 1987.

[42] J. W. Lloyd. Combining functional and logic programming languages. In M. Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium.* MIT Press, 1994.

[43] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming,PLIP'93*, number 714 in Lecture Notes in Computer Science, pages 184–200. Springer-Verlag, 1993.

[44] NeXT Computer, Inc. OpenStep specification. Available at `http://www.gnustep.org/resources/resources.html`, October 1994.

[45] S. L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[46] S. L. Peyton Jones and D. Lester. *Implementing Functional Languages: A Tutorial.* Prentice Hall, 1992.

[47] H. Siverbo and O. Torgersson. Perfect harmony—ett musikaliskt expertsystem. Master's thesis, Department of Computing Science, Göteborg University, January 1993. In Swedish.

[48] G. Smolka. The definition of kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, 1994.

[49] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Current Trends in Computer Science*, number 1000 in Lecture Notes in Computer Science, pages 441–454. Springer-Verlag, 1995.

[50] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[51] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, second edition, 1994.

[52] P. Tarau. Jinni: a lightweight Java-based logic engine for internet programming. In K. Sagonas, editor, *Proceedings of JICSLP'98 Implementation of LP languages Workshop*, 1998.

[53] P. Tarau. Inference and computation mobility with Jinni. In K. Apt, V. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999.

[54] P. Tarau. Jinni: Intelligent mobile agent programming at the intersection of Java and Prolog. In *Proceedings of PAAM'99*, 1999.

[55] O. Torgersson. Functional logic programming in GCLA. In U. Engberg, K. Larsen, and P. Mosses, editors, *Proceedings of the 6th Nordic Workshop on Programming Theory.* Aarhus, 1994.

[56] O. Torgersson. A definitional approach to functional logic programming. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Extensions of Logic Programming 5th International Workshop, ELP'96*, number 1050 in Lecture Notes in Artificial Intelligence, pages 273–287. Springer-Verlag, 1996.

[57] O. Torgersson. Definitional programming in GCLA: Techniques, functions, and predicates. Licentiate thesis, Chalmers University of Technology and Göteborg University, 1996.

[58] O. Torgersson. A note on declarative programming paradigms and the future of definitional programming. In *Proceedings of Das Wintermöte 96*. Department of Computing Science, Chalmers University of Technology, 1996.

[59] P. Van Roy. 1983–1993: The wonder years of sequential prolog implementation. *Journal of Logic Programming*, 1994.

[60] P. Van Roy and S. Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, 1999.

A Simulating GCLA

Methods making computations similar to standard GCLA

```
// Computation methods corresponding to rules
method true_right.
   true_right = [] # some r:matches(true).

method false_left.
   false_left = [] # some l:matches(false).

method d_right:[D].
   d_right = [r:D] # some r:in_dom(D).

method d_left:[D].
   d_left = [l:D] # some l:in_dom(D).

method axiom:[D].
   axiom = [D].

method v_right:[D].
   v_right = [r:D] # some r:matches((A,B)).


method v_left:[D].
   v_left = [r:D] # some l:matches((A,B)).

method a_right:[D].
   a_right = [r:D] # some r:matches((A->B)).

method a_left:[D].
   a_left = [r:D] # some l:matches((A->B)).
```

```
// Computation methods corresponding to strategies
method left:[D].
   dl = instance(d_left,[D]).
   vl = instance(v_left,[D]).
   al = instance(a_left,[D]).

  left = [dl];[vl];[al].

method right:[D].
   dr = instance(d_right,[D]).
   vr = instance(v_right,[D]).
   ar = instance(a_right,[D]).

   right = [dr];[vr];[ar].

method arl:[D].
   ax = instance(axiom,[D]).
   ls = instance(left,[D]).
   rs = instance(right,[D]).

   arl = [ax];[true_right];[arl,rs];[false_left];[arl,ls].

method lra:[D].
   ax = instance(axiom,[D]).
   ls = instance(left,[D]).
   rs = instance(right,[D]).

   lra = [false_left];[lra,ls];[true_right];[lra,rs];[ax].

method gcla:[D].
   arlD = instance(arl,[D]).

   gcla = [arlD].
```

B Objective-C

Objective-C is an object-oriented extension of ANSI standard C. Compared to other popular object-oriented languages, like C++ and Java, Objective-C can be said to be more “object-oriented” since it is based on the use of *dynamic typing* and *dynamic binding*. Dynamic typing means that the exact type of an object is not decided when a program is compiled but at run time. Dynamic binding, likewise, means that the exact method to use to send a message to an object is decided at run time. This is in contrast to function calls where the compiler decides exactly which function to call from the code.

We give a very brief introduction to Objective-C here. The main purpose is to explain common syntactic constructions. We assume some familiarity with C and will only go into object-oriented extensions to C, such as how classes are defined etc. For a more in-depth description of Objective-C see [40], which is the basis for the presentation given here.

B.1 Classes and Objects

An *object* is an instance of a *class*. An object associates data with the particular operations that can use or affect that data. The operations are known as the object's *methods*, and the data they operate on as the object's *instance variables*. The essence of an object is that it bundles a data structure (instance variables) and a group of procedures (methods) into a self-contained unit.

Objects are defined by defining their class. The class definition is a prototype for a kind of object; it declares the instance variables that become part of every member of the class, and it defines a set of methods that all objects in the class can use. Each object gets its own instance variables but the methods are shared by all objects in the class. Each object of a class is referred to as an *instance* of the class.

B.1.1 Inheritance

Much of the power of object-oriented programming comes from the use of *inheritance*. Class definitions are additive, that is, each new class that is defined is based on another class from which it inherits methods and instance variables. Inheritance links classes together in a hierarchical tree with a single class, the *root* class at its root. Every class (except the root class) has a *superclass* from which it inherits, and any class can be the superclass of any number of *subclasses*.

B.1.2 Defining a Class

A class definition in Objective-C consists of the two parts: the *interface* and the *implementation*, where the interface declares what has to be known to other objects about instances of the class.

The structure of the interface part is

```
#import "MySuperClass.h"

@interface MyClass:MySuperClass
{
   // Instance Variable Declarations
}
   // Method Declarations
@end
```

The meaning of `MyClass:MySuperClass` is that `MyClass` is defined to be a subclass of `MySuperClass`. The syntax for (instance) variable declarations is the same as in C. Worth noting is that all objects are of the general type `id`. This type is defined as a pointer to an object. Thus, if an (instance) variable can be an arbitrary object the declaration

```
id anObject;
```

can be used. If an (instance) variable is known to be of a certain type, it can be statically typed. For instance

```
Rectangle *myRect;
```

declares an object of the `Rectangle` class (or more precisely a pointer to an object of the `Rectangle` class). Each object has a distinguished instance variable `self` which, as the name implies, lets the object refer to itself.

The implementation part has the structure:

```
@implementation MyClass

// Method Definitions

@end
```

To get an object to do something, a *message* is sent to the object telling it to apply a method. Message expressions are enclosed in square brackets

```
[receiver message]
```

where `receiver` is an object and `message` tells it what to do. For example, the following message tells the `myRect` object to perform its `display` method, which causes the object to display itself:

```
[myRect display];
```

The method declaration for the `display` method in the interface part is given as follows:

```
- (void)display;
```

Methods can also take arguments, for instance to set the height and width of `myRect`:

```
[myRect setWidth:10.0 height:5.0];
```

The name of the method in this case is `setWidth:height:` and would be declared as follows in the interface part:

```
- (void)setWidth:(float)w height:(float)h;
```

That arguments are inserted after the colons, breaking the name apart, is intended to make messages more self-documenting. The name of a method usually explains the purpose of all its arguments. Methods can also return values. For example

```
BOOL isFilled;
isFilled = [myRect isFilled];
```

where the declaration of the method `isFilled` is

```
- (BOOL)isFilled;
```

Note that a variable and a method can have the same name. Finally, one message can be nested within another. Here one rectangle is set to the color of another:

```
[myRect setColor:[otherRect color]];
```

where the declarations in the interface of the involved methods would be:

```
- (NSColor *)color;
- (void)setColor:(NSColor *)aColor;
```

B.1.3 Creating Objects

The compiler creates just one accessible object for each class, a *class object* that knows how to build new objects belonging to the class. To create a new instance of a class an `alloc` message is sent to the class object. The following code declares a variable and tells the `Rectangle` class to create a new `Rectangle` instance:

```
Rectangle *myRect;
myRect = [Rectangle alloc];
```

The `alloc` method dynamically allocates a new instance. For an object to be useful, it generally needs to be initialized. Initialization typically follows immediately after allocation:

```
myRect = [[Rectangle alloc] init];
```

Initialization methods often take arguments:

```
myRect = [[Rectangle alloc] initWithWidth:5.0 height:2.0]]:
```

For convenience, classes may provide methods that combine allocation and initialization. Such methods typically start with the name of the class:

```
myRect = [Rectangle rectangleWithWidth:5.0 height:2.0];
```

B.1.4 Naming Conventions

It is common practice to begin class names with an uppercase letter and names of variables and methods with a lowercase letter. All names having the prefix `NS` are part of OpenStep [44], which provides an extensive set of classes to use as a foundation for programming. For instance, the root class is called `NSObject`.

B.2 Protocols

Class interfaces declare methods that are associated with a particular class. A *protocol*, on the other hand, declares methods not associated with a class, but which any class, and perhaps many classes, might implement. Protocols free method declarations from dependency on the class hierarchy, so they can be used in ways that classes cannot. Protocols list methods that are (or may be) implemented somewhere, but the identity of the class that implements them is not of interest. What is of interest is whether or not a particular class *conforms* to the protocol, that is, whether it has implementations of the methods the protocol declares. Thus, the use of protocols provides (i) a way to declare properties that an object should have without creating a class, (ii) the possibility for anyone to create a class that conforms to the protocol without knowing anything about any particular class.

A protocol declaration is just a list of method declarations. For instance, a protocol that declares methods related to reference counting could be:

```
@protocol ReferenceCounting
- (void)setRefCount:(int)count;
- (int)refCount;
- (void)decrementCount;
- (void)incrementCount;
@end
```

A class is said to *adopt* a protocol if it agrees to implement the methods the protocol declares. Class declarations list the names of adopted protocols within angle brackets after the superclass name. For example, the following states that the `Rectangle` class implements the `ReferenceCounting` protocol:

```
@interface Rectangle:Shape <ReferenceCounting>
```

A class that adopts a protocol must implement all the methods the protocol declares. Adopting a protocol is somewhat similar to declaring a superclass since both assign methods to the new class. The superclass declaration tells us that an object of the class has all the methods present in the superclass, the adoption of a protocol that it has all the methods declared in the protocol.

An Overview of MedView

Youssef Ali Göran Falkman[*] Lars Hallnäs Mats Jontell[§]
Ulf Mattsson[§] Nader Nazari
Olof Torgersson
Chalmers University of Technology and Göteborg University
S-412 96 Göteborg, Sweden

Abstract

We give an overview of the MedView project and discuss background, current status, and future directions. MedView is a joint project with participants from oral medicine and computer science. The overall aim of the project is to develop models, methods, and tools to support clinicians in their diagnostic work. An important part of this is to be able to efficiently analyze and learn from the monumental amount of information being gathered in clinical records. In the MedView project, clinical data is continuously collected into a large knowledge base of formalized patient examinations. The structure of the knowledge base is based on a formalization of health-care processes and clinical knowledge in oral medicine harmonized within the network SOMNET (Swedish Oral Medicine Network). A number of tools have been built which enable users to extend, view, and analyze the contents of the knowledge base. The system permits immediate analysis of information based on the formal model used. It also well suited for education of dental students. Furthermore, it also provides a basis for distant consultations and generates a solid foundation for multicenter trials and activities.

1 Introduction

The MedView project was initiated in 1995 when some researchers at the clinic of Oral Medicine, faculty of Odontology, Göteborg University, and the department of Computing Science at Chalmers University of Technology, got together and started to discuss their respective research interests. They soon found that their interests had an intersection: The odontologists were looking for ways to use

[*]Department of Computer Science, University of Skövde
[§]Clinic of Oral Medicine, Faculty of Odontology, Göteborg University

computers to improve their daily work and research, and the computer scientists were looking for an area of knowledge on which to apply ideas on knowledge representation and exploration. The solution was obvious, start a joint project aimed at formalizing knowledge in the area of oral medicine, using the mentioned ideas on knowledge representation, and producing computer based tools for use in clinical and analytical work. MedView was born.

Since then several years have passed and many hours of work have been put into the project by various people. The aim of this paper is to give an overview of the work done in the project, its current status, and hint at directions for future areas of research.

Already at the outset one thing was clear: The project would only be of interest from a clinical point of view if it produced tools which improved the daily work of a clinician in oral medicine. Researchers from the clinic of oral medicine had been involved in several attempts to build computerized systems earlier, and deemed them as failures since they did not really do anything for them, neither as clinicians nor as a researchers. They were more or less just systems to put on a digital media what they already had on paper records or photo slides.

A first strategic decision was to not try to build yet another electronic medical record system, but to focus on knowledge gathering and analysis based on a formal description of the concept "examination". This led to the following things to be done, approximately in the given order:

1. Provide a formal framework and methodology to be used.
2. Formalize the knowledge to be gathered based on this methodology in a close cooperation between odontologists and computer scientists.
3. Develop tools for entering the information gathered at an examination into the knowledge base directly in the examination room.
4. Develop tools for viewing the contents of the knowledge base, both for use in the examination room and later for retrospective studies.
5. Develop tools for analyzing and exploring the knowledge base and for adding concepts built on top of the basic formal method.

Today, the first three steps are essentially finished, while the fourth and fifth of course are of the kind where there is always more to be done.

The rest of this article is organized as follows. In Section 2 we give a description of MedView from a medical point of view. Section 3 gives the theoretical model of MedView from a computer scientist's point of view, and mentions some of the areas of computer science to which MedView applies. Section 4 describes the current status of the project. Section 5, finally, contains ideas and directions for future work.

2 MedView and Oral Medicine

Diagnostic work and clinical decision-making are central items in every field of medical practice, where clinical experience, knowledge and judgment are the cornerstones of health care management. In order to achieve increased competence, the clinician is confronted with complex information that needs to be analyzed. There is considerable evidence that the unassisted human mind is challenged when exposed with multiple sets of data [13, 20, 44, 71]. Therefore, the clinician needs tools to improve analysis and visualization of data in the diagnostic and learning processes.

To support the human mind in extracting valuable patterns in clinical information, computer technology has been introduced in several areas of modern medicine with the aim to assist these cognitive processes. The systems provide a broad functionality, from distant consultations of individual patients to intelligent expert systems, where text and image information is collected and analyzed. In the elaboration of a computerized system, several critical problems have to be mastered in order to ensure that conclusions drawn are correct or justified. In this section, we discuss some issues we have confronted in the MedView project. We also describe how they are handled.

2.1 Clinical Experience and Diagnostics

2.1.1 Nomenclature and Definition of Clinical Information

The first step in the diagnostic process, illustrated in Figure 1 below, consists of gathering and storage of clinical information. In order to be meaningful for interpretation, these data must be recorded in such a way that they can be understood and interpreted in a precise manner by all members of the health care system. This means that a *formalized* and *harmonized* health care system is imperative.

The word "formalize", in this context, means to establish and formally define basic health care activities that can provide an explicit structure for intelligent reasoning. This formalization is crucial in order to arrive at a correct diagnosis, based on an explicit definition [43, 106]. The term "harmonization" refers to the process of making the formalized activities adapted within a community [22, 29, 79, 82].

The demand and request for formalization and harmonization is certainly not new, but has frequently been associated with obstacles [43, 56, 86, 120]. Although several international attempts have been made to establish a congruent medical nomenclature not many have been successful.

Today, clinical data are frequently expressed in natural language using terms based on individual subjective assessments or interpretations not defined or harmonized within the health care system [3, 81]. Several terminologies exist, often developed within a specific medical discipline, but they are seldom widespread

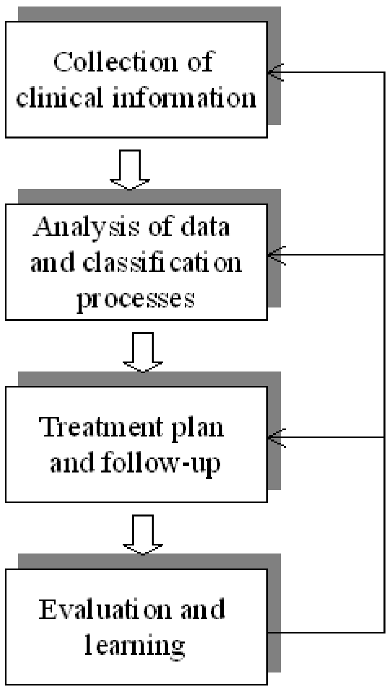

Collection of clinical information
Analysis of data and classification processes
Treatment plan and follow-up
Evaluation and learning

Figure 1: General description of the decision-making process. The process is a chronological sequence where each chain of events present obstacles that have to be controlled in the elaboration of a formalized and harmonized language.

and do not provide a useful international nomenclature system. Furthermore, most terminologies are not related to definitions of terms. Even when definitions exist they can be highly ambiguous. An example is the attempt to define oral leukoplakia. The latest definition reads “a predominantly whitish lesion which can not be diagnosed as any other definable disorder” [7]. This definition is closely related to the clinician’s ability to diagnose all other whitish lesions of the oral mucosa. Thus, to an inexperienced clinician the diagnosis of oral leukoplakia can involve almost any whitish lesion, while the experienced clinician will use it less often.

Evaluation of treatment care and scientific analysis is accordingly not meaningful when registered terms are not precisely defined. New computerized technologies will demand that strategies for clinical registration of data are developed in order to reduce these considerations. Currently, most systems used in the health care sector are not dealing with these problems, but are more focused on transportation and storage of data.

2.1.2 Analysis and Classification of Diseases

The second step in the decision-making process is the analysis of gathered data. The diagnostic process involves the clinician's ability to put the patient into a certain class or group [95, 114]. A diagnosis can be considered as a way of classifying clinical information to facilitate communication between health care providers and to assist in decisions of treatment strategies. The diagnosis is indeed only a common identity of a group of patients with similar clinical information profiles. Consequently, to define a disease it is essential that all patients have an identical information pattern, not shared by any patients who do not have the disease. This is rarely the case [72], and a diagnosis is often based on a description rather than an explicit definition.

The currently used diagnostic system has developed over several centuries. Diagnoses based on pathological anatomy have sometimes been replaced by diagnoses which reflect the introduction of physiology and laboratory research. Patho-anatomical diagnoses as, for example, 'gastric ulcer' was replaced by 'hyperacidity' to denote the patho-physiological dysfunction of this disease. Another problem concerning classification of diseases is that the extension of a disease may change over time along with new discoveries. Lichenoid contact reactions may serve as an example. The diagnosis oral lichen planus was recently split into oral lichen planus and lichenoid contact reaction [12]. However, this subdivision is not yet fully accepted which leaves the diagnostic system in a state of confusion where oral lichen planus may or may not include lichenoid contact reaction.

All in all, the diagnostic systems of today have different backgrounds and there are no rules to promote continuous modification to adapt to new scientific achievements. This lack of harmonization of clinical information will lead to significant problems when new information technologies are to be used in our health care system.

The quality of the analysis and classification process mentioned above is thus influenced by both initial steps in the decision-making process. First, the character and quality of input data will greatly influence our ability to perform subsequent analysis. Second, the classification process is in itself influenced by our ability to adopt adequate and reliable inclusion criteria from input data to a certain disease or diagnosis. Consequently, our knowledge, experience, and treatment strategies of various disorders will be based on conclusions from observations or studies that may not be comparable due to differences in nomenclature or diagnostic criteria. Obviously, it is essential to elaborate routines where these factors are controlled.

2.1.3 Visualization of Clinical Information and Learning

The third step in the decision-making process is the elaboration of treatment strategy and follow-up procedure which emanates as a result of the classification process (Figure 1). From the aspect of treatment strategy, it is common practice to record treatment rendered, but not the diagnostic basis for these treatment decisions. This practice may undervalue diagnosis, but also hamper feedback regarding the effectiveness of treatments relative to specific diagnoses [8].

The fourth and last step can be described as the way we draw conclusions and learn from performed therapies. These experiences are, within the medical community, generally presented as scientific articles or books in order to forward information to increase the knowledge of other clinicians.

Today, the conventional search in index-based volumes has been replaced by computerized databases available to all members of the scientific community. However, complex clinical information stored as images, concepts, videos, etc. is difficult to explore. Concept-based exploration of clinical databases has to be boiled down to a volume of information that is possible to handle. The potential risk with this process is that significant information may be overlooked if it is left outside the search profile. This situation arises especially where clinical hypotheses are to be tested and where the search profile, based on a combination of keywords, will not provide sufficient information. Data with significant interest to the scientist may therefore exist in the database but be left undetected.

In many respects we confront the same problem in clinical research. Information patterns, which may lead to new discoveries, are most likely concealed in large volumes of clinical information stored in non-transparent conventional patient records. Essential information which is not frequently encountered will escape detection as it is hidden in irrelevant information.

Consequently, tools are needed which can intelligently present large volumes of clinical data and where the capacity of the human brain to recognize significant patterns hidden in monumental amount of clinical information is maintained. Therefore, it is important that the capacity of our cognitive function to recognize relevant information and our ability to make rational verdicts is applied, an essential function that computers still are lacking. Computer technology can, however, visualize extractions of complex information as patterns which may initiate associations to new inquires, that may eventually lead to new knowledge.

2.2 What MedView Offers

In short, MedView offers a model for formalization and tools for knowledge gathering, visualization, and analysis of data. The tools are also aimed at improving the everyday work of clinicians in oral medicine. The formalization used was developed in close cooperation between participants from oral medicine and computer science, with the purpose of providing a model suited for both oral medicine and computerized storage and reasoning. The model of the health-care activities and medical expertise involved has evolved through collaboration within the Swedish Oral Medicine Network (SOMNET).

MedView is primarily aimed at increasing the speed by which we may obtain new and valuable information within the field of oral medicine. A formalization

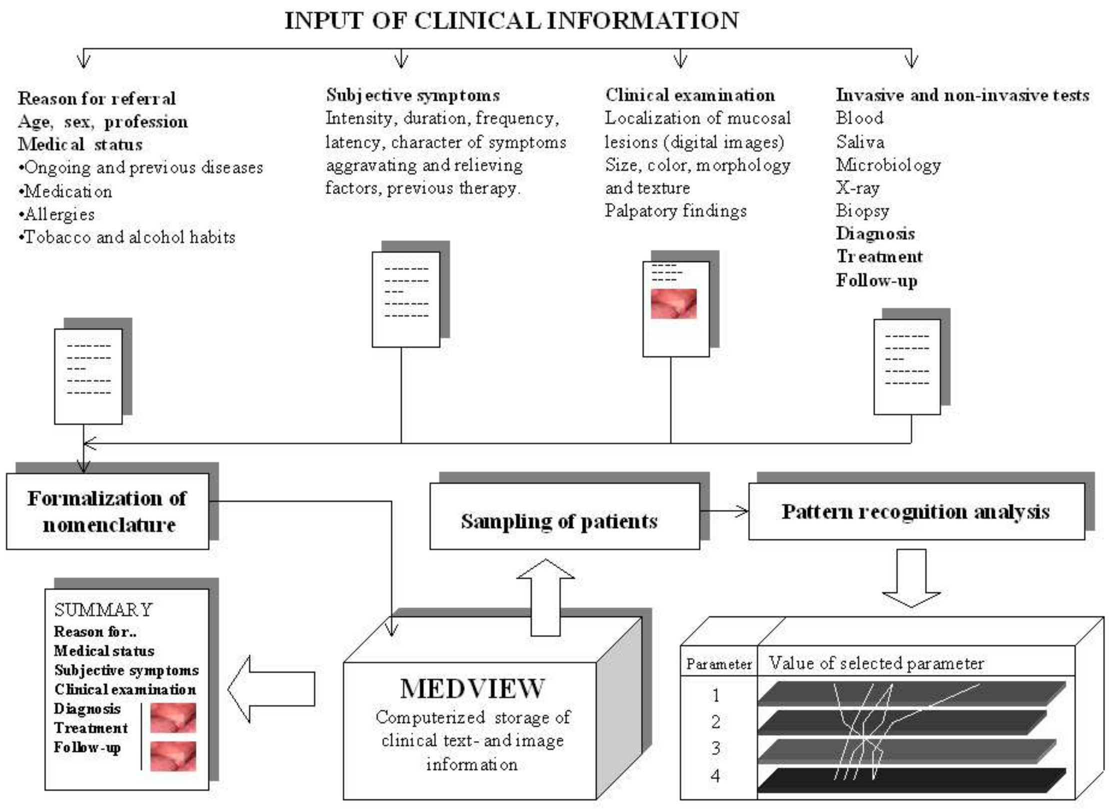

INPUT OF CLINICAL INFORMATION
Reason for referral
Age, sex, profession
Medical status
•Ongoing and previous diseases
•Medication
•Allergies
•Tobacco and alcohol habits
Subjective symptoms
Intensity, duration, frequency, latency, character of symptoms aggravating and relieving factors, previous therapy.
Clinical examination
Localization of mucosal lesions (digital images)
Size, color, morphology and texture
Palpatory findings
Invasive and non-invasive tests
Blood
Saliva
Microbiology
X-ray
Biopsy
Diagnosis
Treatment
Follow-up
Formalization of nomenclature
Sampling of patients
Pattern recognition analysis
SUMMARY
Reason for..
Medical status
Subjective symptoms
Clinical examination
Diagnosis
Treatment
Follow-up
MEDVIEW
Computerized storage of clinical text- and image information
Parameter
Value of selected parameter
1
2
3
4

Figure 2: General description of MedView. MedView is used for formalized registration of clinical text- and image-based information into a knowledge base (top). The registered clinical information is synthesized into a readable text and displayed together with clinical images for each patient (bottom left). The contents of the knowledge base is subsequently used for analysis, evaluation, and learning (bottom right).

of clinical procedures and visualization of information provide a possibility for recognizing new trends and patterns otherwise hidden in large amounts of non-transparent clinical records. With MedView, the knowledge and intuition of the clinician can be combined with the potential of the computer to promote analysis and testing of hypotheses in a favorable environment. MedView is also well suited for educational purposes of dental students and in post-graduate training. It allows distant consultations and generates a solid foundation for multi-center trials and activities.

2.2.1 Formalization and Harmonization

When elaborating the MedView system, great care was taken to determine what clinical information could be defined as useful and constitute the foundation in the knowledge base. The result from these considerations was standardized protocols for input of clinical information, where the nomenclature used was developed in close collaboration between the involved clinics. Case history and all clinical data are entered by use of predefined parameters from the mentioned protocols. Through this process a solid base for subsequent analysis and intelligible reasoning of results is obtained. The nomenclature and information structure is thus *formalized* and *harmonized* within the network. The formalized protocols have a logic interpretation (see Section 3.1), which make them suitable for automated reasoning in a computerized system. At the same time, they are simple enough to have an obvious intuitive reading needing no further explanation.

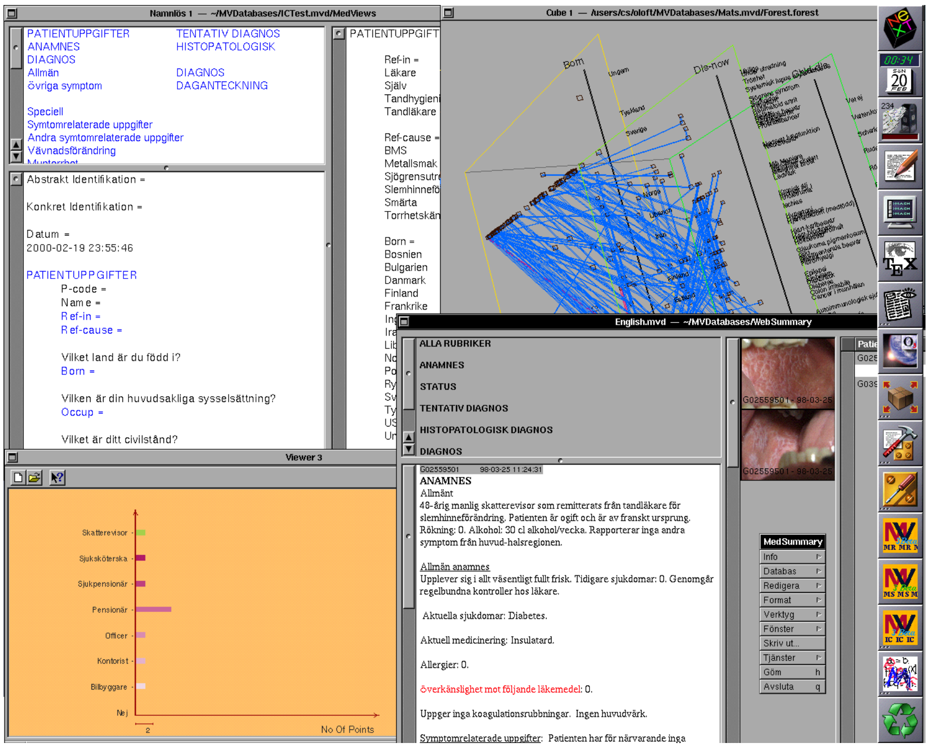
The protocols defined for collection of data are rather extensive including detailed interviews of disease history and protocols for clinical examinations. Existing mucosal lesions are described in terms of localization and clinical appearance. Mucosal lesions are also documented with digital video technique. This technique offers the advantage that the digitized images are immediately accessible in the knowledge base, both for analysis and for distant consultations. Results from biopsies, laboratory tests, and other invasive or non-invasive investigations are included, as are diagnoses, treatment modalities, and clinical outcomes of performed therapies. Additional information not included in the protocols but relevant for the present patient can be included as text.

2.2.2 Everyday Tools

MedView is not all about formalizing and analyzing data. It is also about changing and improving the everyday work of clinicians by providing tools facilitating clinical processes. In part, the development of these tools is a necessity for the success of knowledge gathering: In order to make it possible to collect data in an efficient manner, it must be possible to enter data into the knowledge base during examinations. To motivate this extra work, applications are needed that give immediate feedback in terms of enabling the use of entered data in ways that improve clinical procedures.

The registration of information based on the formalized protocols is done using a specialized input application described further in Section 4.3. Each examination corresponds to a record, including digitized images taken at the examination. Apart from the input application, there are several output applications or *viewers* designed for visualization of obtained information. The viewers are focused on analysis, interpretation, and evaluation, both of individual patients and of groups of patients selected from the knowledge base.

Since an extensive amount of information is collected for each patient, the effect of performing input during examinations is that all information about each



Abstrakt Identifikation =

Konkret Identifikation =

Datum =
2000-02-19 23:55:46

PATIENTUPPGIFTER
P-code =
Name =
Ref-in =
Ref-cause =

Vilket land är du född i?
Born =

Vilken är din huvudsakliga sysselsättning?
Occup =

Vilket är ditt civilstånd?

PATIENTUPPGIFT
Ref-in =
Läkare
Själv
Tandhygieni
Tandläkare

Ref-cause =
BMS
Metallsmak
Sjögrensutr
Slemhinnefö
Smärta
Torrhetskän

Born =
Bosnien
Bulgarien
Danmark
Finland
Frankrike

G02559501 98-03-25 11:24:31
ANAMNES
Allmänt
48-årig manlig skatterevisor som remitterats från tandläkare för slemhinneförändring. Patienten är ogift och är av franskt ursprung. Rökning: 0. Alkohol: 30 cl alkohol/vecka. Rapporterar inga andra symptom från huvud-halsregionen.

Allmän anamnes
Upplever sig i allt väsentligt fullt frisk. Tidigare sjukdomar: 0. Genomgår regelbundna kontroller hos läkare.

Aktuella sjukdomar: Diabetes.

Aktuell medicinering: Insulatard.

Allergier: 0.

Överkänslighet mot följande läkemedel: 0.

Uppger inga koagulationsrubbningar. Ingen huvudvärk.

Symptomrelaterade uppgifter: Patienten har för närvarande inga

Figure 3: Some MedView applications. From top left: Input application, 3D viewer, bar chart viewer, summary application.

patient is immediately available in a well-organized searchable knowledge base. When an examination is completed, MedView can generate a summary where all the contained information is displayed as a readable text, with digitized images of the mucosal lesions shown simultaneously. The character of the generated summary is in its layout in most ways similar to the regular patient record encountered in daily practice. The application and the methods used for generating text are described further in Section 4.4. When a patient comes back for a follow-up the system can synthesize a full medical history together with associated images providing the clinician with all the needed background information. The time that can be gained by letting the system generate patient record text, instead of using the traditional method of dictating and typing the text, more than makes up for the extra time required to gather the information.

2.2.3 Analysis Tools

Once information is gathered the analysis and learning phases begin. Typically, these tasks are performed at the users desktop computer rather than in the examination room.

MedView permits selection of patients from the knowledge base according to any combination of parameters included in the registration protocols. A search may thus be of a simple nature as, for example, finding all patients with confirmed "diabetes mellitus". However, it can also be more complex with several criteria involved, as finding "female patients with bilateral lichenoid reactions in buccal mucosa, treated with local application of clobetasol". The search profile can be decided and directed by the user according to the objective or purpose for the analysis. The system identifies patients that fulfill the chosen criteria and displays them in various ways for instance in a traditional bar chart. A screen with several applications, among them a bar chart viewer, is shown in Figure 3.

The selection of patients can be subjected to pattern recognition analysis. An application enabling a three-dimensional display of a multivariate analysis, where the result is shown in a cube which may be rotated and viewed from different angles by the examiner is discussed in Section 4.7. Another, enabling clustering of patients regarded as similar, in some user-defined way, is described in Section 4.8. The hierarchical clustering of examinations is displayed in a three-dimensional tree. The main purpose of these applications is to visualize patterns in a group of selected patients. They are therefore mainly focused on the possibility of learning and testing various hypotheses within the created knowledge base.

2.2.4 Extensibility

The formal model used in MedView is such that the currently used formalization can be easily extended and modified over time. The protocols used can be seen as a first approximation of the needed knowledge structures. When we learn more, the protocols can be extended to collect more information and describe harmonized nomenclature for a larger part of oral medicine. In fact, this process is in progress all the time. When new values are needed to describe a particular attribute in a protocol, they can be added directly. To keep a high level of harmonization it is of course important to communicate such additions within the network.

Recently, new protocols have been developed which include formalization of new concepts like tooth-status. With these new protocols MedView can be used by a broader category of odontologists. Due to the properties of the formal model, see Section 3.1, old examinations are not made obsolete by the introduction of new protocols. It's just that some knowledge available in new examinations might be missing in the old ones. We hope that it will be possible to formalize and harmonize larger and larger parts of the health care activities within oral medicine in the same manner. Then, broaden the view to other parts of odontology.

The formal model of MedView also makes it easy to introduce more complex concepts based on the basic data collected through the common protocols. For instance, it might be desirable to introduce concepts that group together a number of possible values, or to express a new diagnosis in terms of common observations from a number of cases.

2.3 Discussion

MedView addresses the issue of learning from the complex data which originates from everyday clinical practice in the field of oral medicine. To enable this, MedView was designed with the ambition to meet the demands for formalization and harmonization in the decision-making process. In this section we discuss the project and put it in the context of other medical computer based systems.

2.3.1 Gathering Clinical Information

The first step in the decision-making process constitutes the gathering of clinical information. Conventionally, the clinician collects data, which are then summarized and written down in a non-transparent record. The registration and summary applications of MedView resembles the standard way we collect information in daily clinical practice and the comparison to the conventional record is so far obvious. However, all registrations in MedView are performed using protocols with a *formalized* nomenclature where data are stored in a computer. The relevance of individual terms and parameters in the protocols may be debated, but the stringent use of a formalized language creates a basis for reduced discrepancy in clinical registrations within the network.

If a formalized language can be adopted within a community or network, the question arises if the use of a computerized record will facilitate subsequent use of obtained clinical information. Several studies have documented the usefulness of computerized applications in this context, also within the field of odontology [19, 28, 34, 83, 94, 112]. The computerized records enable quick access to reference and educational information [65, 77] and enhanced storage of structured medical knowledge [53]. The MedView system, using both text- and image-based information, is in agreement with these studies. Computer records are also introduced for quality assurance, replacing the paper record. These systems are aimed at assessment and improvement of patient care at the time of treatment, thus building quality management into the caregiving process [83, 115]. However, to the best of our experience, the vast majority of commercial systems available for computerized patient records within odontology are mainly focused on *individual* patient care and the possibility to analyze the *entire* patient material and visualize clinical patterns is usually rather limited. MedView offers the advantage of *combining* the conventional computer record with the possibility for information visualization and analysis.

2.3.2 Analysis and Classification

The second and third steps of decision-making involve analysis and classification processes with subsequent treatment and follow-up (Figure 1). Computer systems may be characterized as *active* or *passive* in decision-making [30]. Passive support occurs when a computer facilitates access to relevant patient data or clinical knowledge for interpretation by the physician. Active support requires that the computer processes input data to a higher level of information, e.g., a diagnostic or expert system. Such models have been developed to enhance clinical security by facilitating the ability to draw conclusions from background knowledge and diagnostic hypotheses [50, 96]. Most studies in this field have been written within medical research, such as dermatology and a rather limited number within odontology [108]. In [108] it was pointed out that a problem with expert systems in general was the lack of accepted clinical terminology in the medical community.

Systems for computerized support in decision-making processes within odontology can coarsely be classified into four groups:

- The first group consists of studies where a system has been developed as an aid or tool in a very specific clinical situation such as design of removable dentures [25, 26, 47], artificial tooth form selection [100], objective assessment of mucosal lesions [67, 68], or surgical operations [31, 98, 99].
- In the second group, computers for decision-making are used in applications with a somewhat broader perspective. This includes applications where predefined criteria or a questionnaire are used as decision parameters for arriving at a correct diagnosis for an individual patient. Applications like this exist in endodontology [51, 73], oral radiology [15, 27, 113], and oral pathology [60, 61, 84, 97]. This group also includes computerized systems for evaluation of diagnostic performance and therapeutic decisions [35, 74] and studies on how decision analysis in general can be applied to dentistry [69, 70].
- The third group describes computerized expert systems in odontology with the characteristics mentioned above. Such systems have been elaborated in orthodontics [49, 102, 107], endodontology [37], oral pathology [36], cariology [10], and oral radiology [1].
- The fourth and last group consists of systems with the purpose of using the potential of neural networks for analysis of decision-making, therapy planning, and quality assurance [14, 105].

To conclude, the vast majority of these systems are mainly focused on treatment planning and decision-making for *individual* patients, rather than on the possibility to generate further *knowledge* through analysis of continuously obtained clinical information. However, attempts with this purpose are described [45, 48]. Furthermore, in oral medicine there are, to the best of our knowledge, very few papers [104, 121] published with the aim to use computer technology and database engineering for any of the above mentioned purposes or as a tool to increase clinical knowledge.

MedView is mainly a passive support system, primarily focused on facilitating pattern detection where hypotheses can be evaluated in a favorable environment.

The knowledge and intuition of the clinician can be combined with the potential of the computer to promote testing of hypotheses and augment analysis. However, the standardized collection of data definitely provides a future possibility for development of active expert systems.

2.3.3 Evaluation and Learning

The last step in the decision-making process is represented by evaluation and learning and to add to our knowledge and experience. However, the reliability of any analysis or learning process depends on the quality of input data and formalization of nomenclature [87, 119]. The same reasoning also applies to the usefulness of expert systems, which is impaired by incompleteness and inaccuracies of the databases. The need and demand to find appropriate standards and nomenclatures is therefore very important since discrepancies in these fields will always decrease the reliability of the systems [30]. Similar thoughts were expressed in [55] where the development of standardized computerized records as a tool in interdisciplinary communication is advocated. All in all, efforts to draw conclusions from performed observations are highly commendable, as long as we remember that the foundation for our analysis is never better than the quality of input data. Again, *formalization* represents the initial but also fundamental part for analysis, decision-making and harmonization and these processes are facilitated when aided by computer technology. A main purpose with MedView is therefore to act as a hypothesis generator.

The use of MedView may also be viewed from the view of education. An individual clinician may not encounter enough cases to develop adequate experience of a certain condition. A network, such as SOMNET, is a way to overcome this problem. All clinics within SOMNET have access to the knowledge base. A multi-center network, with the combined knowledge of individual clinicians creating a knowledge base founded on formalized criteria, increases our ability to reach useful information for education and learning [38, 64, 75, 76]. The formalized protocols generate a possibility for integrated research between clinics. Distant consultations of individual cases have been successful in several tele-medicine applications, among them MedView. Used right, computer technology is most certainly a valuable instrument to increase clinical experience and to promote learning within the field of oral medicine.

3 MedView and Computer Science

The MedView project involves several areas of computer science, mainly knowledge representation, formal reasoning systems, declarative programming, object-oriented programming and software development, artificial intelligence (AI), and human-computer interaction (HCI).

The nature of the project is such that all the above areas are needed and have to be integrated with each other to produce high-quality software tools. These tools are then applied continuously in the everyday work of clinicians and researchers in oral medicine. In addition, hypotheses are directly testable since there is, and has been from the very beginning, an existing userbase. Both applications and knowledge models can be put to the test. If knowledge models cannot be understood and used by the medical experts involved, they are likely not to be of great value to the project. Likewise, applications developed can be introduced and tested. If an application does not provide a useful interface or a meaningful feature set it has to be modified. Examples of applications exist that have been developed but never used. On the other hand, MedRecords described in Section 4.3 has been in use for several years and can be said to be proven a good tool for its task.

A brief description of the basic theoretical model used for knowledge representation in MedView is given in Section 3.1. This model has been used as the basis for a programming system aimed at being the deductive engine of MedView. Currently, the model is not implemented in a uniform manner across applications as discussed in Section 4.2.1.

The various topics mentioned above relates to the MedView project in the following ways:

- Knowledge representation and formal reasoning systems. MedView is based on a theory of definitions [46]. This theoretic model with connections to logic and logic programming is used for all knowledge representation.

- Declarative programming. We believe that declarative programming is a very powerful tool for developing certain kinds of applications, such as symbol-manipulation, knowledge representation, intelligent reasoning etc. Furthermore, declarative programming come very close to the theoretical model used. Consequently, we are developing a declarative programming system [109] based on the theoretical model used in MedView.

- Object Oriented Programming (OOP). OOP is used as a tool in MedView to build applications. It is used since it, in our opinion, is the best existing paradigm for developing modern GUI based applications. The OOP tools used are interfaced with our own frameworks for integrating declarative program components. Thus, we can use OOP programming and declarative programming together and use each paradigm for the task where it's best suited.

- Artificial Intelligence. Knowledge based systems such as MedView is an important part of AI. In particular, case-based reasoning techniques have been studied [32]. Adding more AI-techniques is an area for future research.

- Human-Computer Interaction. The systems developed must interact well with clinicians, students, and researchers in oral medicine. Also, easy to handle administrative tools are needed. This makes studies in HCI an integral part of the project.

Note that both a sound theoretical basis and implementation of knowledge structures, and real working software solutions to be applied in daily work are equally important. The development of working high-quality software is necessary to influence the examination process so that knowledge can be collected and analyzed. We are also interested in using information technology to improve the healthcare process. A well-founded theoretical model is necessary, or the mentioned applications and the data gathered cannot be explained and analyzed in a meaningful manner.

3.1 Theoretical Model

The basic model of clinical information used in MedView is act-oriented. We think of explicit clinical information as resulting from acts of *defining* medical terms in various situations. A clinical diagnosis, an examination record and so on, can all be seen as definitions of collections of specific clinical medical terms [85].

The formalization of definitions as data structures that is used here is based on the idea that a definition generates a local logic, a reasoning model restricted to specific terms. These local logics are then the basis for reasoning using given formal clinical terms.

As a data structure, a definition D can simply be thought of as a collection of equations

$$D \left\{ \begin{array}{lcl} a_0 & = & A_0 \\ a_1 & = & A_1 \\ & \vdots & \\ a_n & = & A_n \end{array} \right. \qquad n \geq 0,$$

where terms, $a_0, \ldots, a_n$, are defined in terms of conditions, $A_0, \ldots, A_n$. The definiens of a term a, $D(a)$, is then the collection of conditions A that define a in D. The local logic of D consists of a relation $A_1, \ldots, A_n \vdash B$, that is, B follows from $A_1, \ldots, A_n$ according to D, where the two constituting rules are

- $A_1, \ldots, a, \ldots, A_n \vdash B$ if $A_1, \ldots, A, \ldots, A_n \vdash B$ for all A defining a in D,
- $A_1, \ldots, A_n \vdash a$ if $A_1, \ldots, A_n \vdash A$ for some A defining a in D.

The logic of D consists of these two rules together with ordinary rules of reasoning for given complex defining conditions built up from atomic terms. A full description is given in [46].

The model we use can be summarized as follows:

- formal clinical data are seen as definitions of clinical terms,
- reasoning is always local to given definitions, there is no single global logic for formal clinical reasoning.

As a concrete example of a definition we show a small part of an examination record:

$$D \left\{ \begin{array}{l} status = direct \\ direct = mucos \\ direct = palpation \\ mucos = mucos\_site \\ mucos = mucos\_col \\ mucos\_site = 112 \\ mucos\_col = white \\ mucos\_col = brown \\ palpation = palp\_site \\ palp\_site = 112 \end{array} \right.$$

In D, the term *status* is defined by the term *direct*, which in turn is defined in terms of *mucos* and *palpation*. Thus, $D(direct) = \{mucos, palpation\}$. All these terms are part of the general structure of an examination record, which is shared by all examinations. In contrast, the term *mucos_col* is defined by the observed values *white* and *brown*, specific for this particular examination record.

4 Current Status

MedView has been developed in an iterative process through close collaboration between experts in oral medicine and computer science, using a mixture of contextual design [11], user oriented design, and logical analysis of the problem and required knowledge. Essentially, the analysis and design of the system can be divided into two sub-problems: knowledge representation and development of applications for gathering and exploring clinical data. Knowledge representation issues are discussed above. In this section we describe the status of the implemented system.

The system is currently in daily use in eight examination rooms at four different clinics. The examination rooms are equipped with a PC on a custom-built table, shown in Figure 4, and a digital video camera. The collected data are stored on a server.

A basic assumption underlying the design of MedView is that of separating the activities of entering information and viewing, or otherwise using, the entered information. The rationale behind this is that the cognitive tasks involved are very different. Thus, specialized applications, each described below, have been developed for each task.

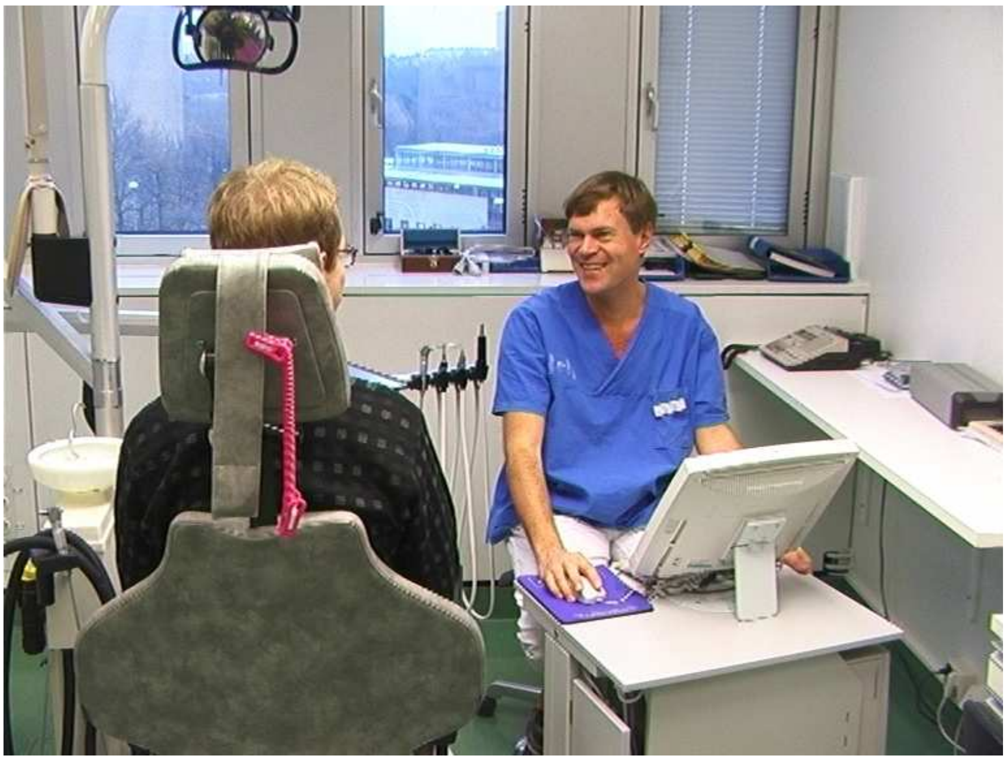


Figure 4: Clinician working with MedView. The computer is placed on a custom-built table aimed at minimizing the interference with the communication between patient and clinician.

Information is collected in a critical situation, namely during examinations. For each examination, values for many different attributes describing an examination must be given. A good deal of effort was therefore put in early in the project to build efficient tools for knowledge gathering consistent with the underlying theoretical model.

4.1 Knowledge Base Contents

Currently, (March 2000) the knowledge base built in the MedView project contains approximately 1500 examination records covering more than 700 cases. The average growth rate is 20 new patients and 30 visits by previously examined patients a week. The main knowledge base is located at the clinic of oral medicine in Göteborg. The various clinics within SOMNET have local knowledge bases containing the examinations made at each clinic. The contents of these local knowledge bases are added regularly to the knowledge base in Göteborg so that the entire amount of data collected can be accessed through one common knowledge base. The clinics within SOMNET will have full remote access to this central knowledge base.

The contents of the knowledge base is mainly used in two ways. First, in the examination room to display the history of the patient under examination. Second, to perform analysis, learn from, and search for patterns in the knowledge base. The second task is typically performed on the clinician's desktop computer. So far, the most used analytical tool is The Cube discussed in Section 4.7. However, building more viewers for exploration, search, knowledge extraction, education, and so on, is an important area for current and future research. We present a number of more specific suggestions in Section 5.

4.2 Applications Overview

We briefly describe the applications currently used within MedView. Some more being under development but not yet taken into use within SOMNET are discussed in [110].

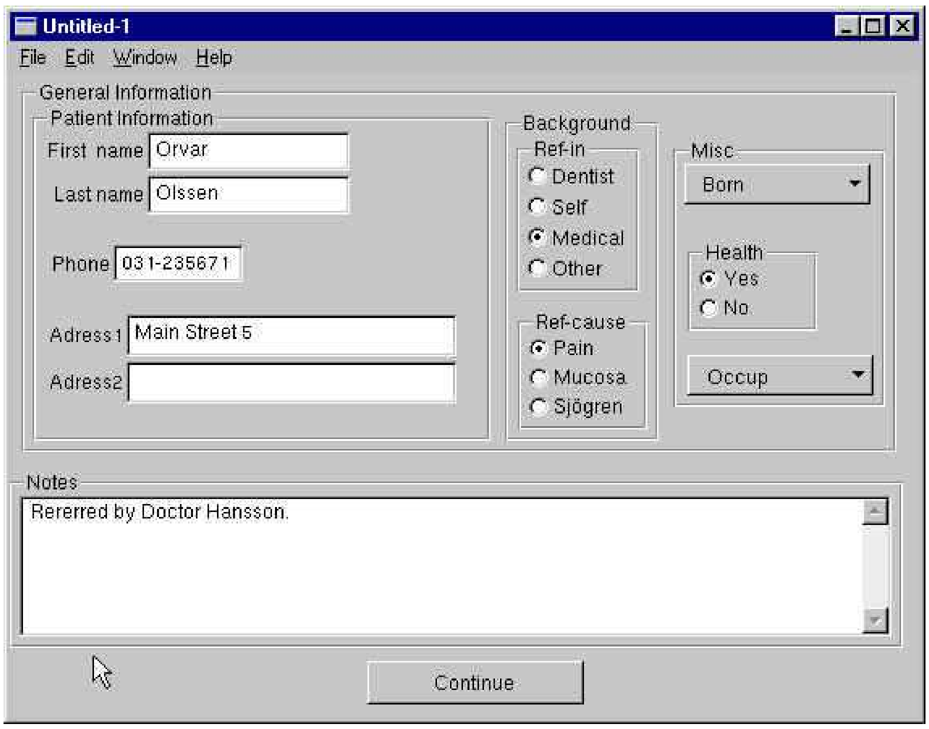
4.2.1 Background

Today, MedView runs on a combination of machines running Windows95/98/NT and some machines running OpenStep/Mach 4.2 and NextStep 3.3. Originally the system was developed using NextStep and GCLAII.

The operating system NextStep, used for the first versions of MedView, was chosen for its advanced GUI, networking capabilities, and object-oriented application development environment. The name NextStep usually denotes both the operating system and the object-oriented application development environment used to build applications for it. The application development environment later evolved to OpenStep available for several platforms including Windows NT.

GCLAII [4, 5, 62, 63] is a definitional programming language developed at the Swedish Institute of Computer Science, (SICS). Due to the similarity with the definitional model used to represent knowledge it was the natural choice for implementing the knowledge base and reasoning part of MedView. However, it was soon discovered that the performance of GCLAII was not sufficient for use in MedView. This led to the development of a simple object-oriented definitional machinery called DefinitionG.

DefinitionG implements the most important features of the definitional knowledge base model and can be subclassed if needed to add more. While DefinitionG lacks both purity, as a definitional representation of the knowledge base, and features, it has nevertheless been crucial for the development of *working* software solutions within MedView. Currently, we are in the process of replacing DefinitionG with a more fully-fledged object-oriented framework for definitional computing called Gisela [109]. One way to view Gisela is as a successor to GCLAII. Gisela has from the start been designed to better fit the demands on a definitional machinery to be applied within MedView.



General Information

Patient Information

First name: Orvar

Last name: Olssen

Phone: 031-235671

Adress1: Main Street 5

Adress2:

Background

Ref-in: Dentist Self [✓ Medical] Other

Ref-cause: [✓ Pain] Mucosa Sjögren

Misc

Born

Health: [✓ Yes] No

Occup

Notes: Rererred by Doctor Hansson.

Figure 5: A traditional form. Data is entered using a number of standardized widgets. Clicking the button labeled Continue will show the next screen.

Originally, MedView had only a small number of users, all active within in SOMNET. With the desire to increase the userbase, it became obvious that it was not possible to require the use of the rare operating system NextStep. Therefore, during 1998 and 1999 a transition was made to the Windows family of operating systems. The development platform NextStep was replaced by OpenStep and in some cases Java to enable the transition.

4.3 MedRecords

MedRecords (MR) is the *input* application used by clinicians to enter detailed formalized examination data during patient visits. Although the version of MR used today has some special features for use within MedView it is best seen as a general-purpose program useful for entering many different kinds of data.

The most common way today to design an application where data needs to be entered is to use forms [39]. The forms are typically built from objects such as text-fields, pull-down lists, and check-boxes. An example of such a form is shown in Figure 5. In MR we have developed a technique for entering data where the forms are replaced by a specialized text-editor, coupled with hypertext links for navigation, and easily scrollable text lists containing possible values.

The design goal behind MR was to create an unobtrusive, easy-to-use, space efficient, and scalable method for entering data, where the forms used can be created by users without requiring any programming knowledge. Here we describe the interaction technique and our experiences from it for entering information about a large number of patient examinations over a period of about two years.

4.3.1 Analysis

MR was conceived as a solution for entering data based on an analysis of the constraints given. The analysis describes a conceptual model of the act of entering data. It also lists external requirements describing the environment in which data is to be entered.

The conceptual model of the knowledge base used in MedView is that of a collection of definitions, where each definition describes one medical examination. Each such definition can be pictured as a collection of equations as described in Section 3.1. Thus, in the MedView setting, entering data is the act of *creating* a definition. Therefore, our goal was to support the act of *defining*, in a precise manner, a particular medical examination. MR aims to mirror this view of the knowledge base, while keeping a non-technical interface to the user.

The environment for which MR was developed introduces a number of requirements that had to be satisfied. Some of the more important ones are:

- Data is entered by the clinician him/herself while a patient is being examined.

- Each record in the knowledge base can have a large number of different attributes and each attribute can have a very large number of possible values.

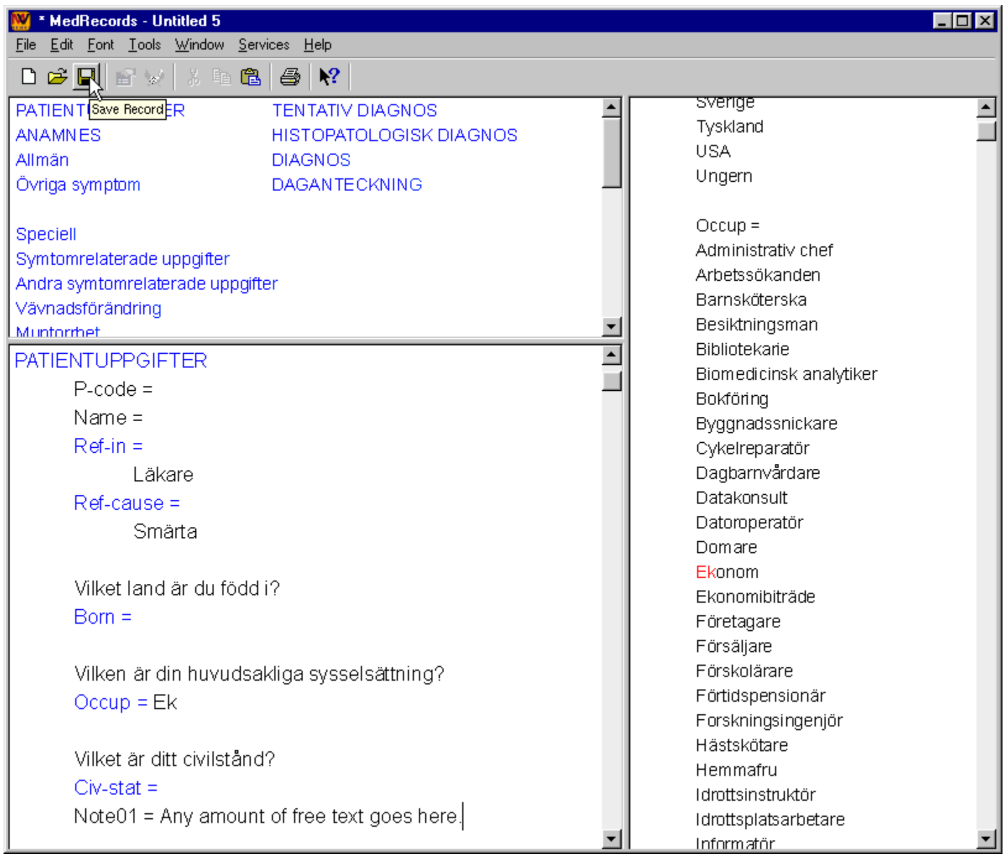
- Values for attributes are most often taken from formalized lists of valid values. However, free text and digitized images may also be included.

- When a new value is encountered it must be easy to add it to the list of valid values.

- The protocols or forms used are developed by the expert users themselves without requiring any programming knowledge.

- The layout of forms should be configurable by each user.

Since the act of entering data is separated from viewing data, MR was designed for entering data only. It is not intended for viewing examination records.


* MedRecords - Untitled 5
File Edit Font Tools Window Services Help
Save Record

PATIENT ER
ANAMNES
Allmän
Övriga symptom

Speciell
Symtomrelaterade uppgifter
Andra symtomrelaterade uppgifter
Vävnadsförändring
Munforrhet

TENTATIV DIAGNOS
HISTOPATOLOGISK DIAGNOS
DIAGNOS
DAGANTECKNING

PATIENTUPPGIFTER
P-code =
Name =
Ref-in =
Läkare
Ref-cause =
Smärta

Vilket land är du född i?
Born =

Vilken är din huvudsakliga sysselsättning?
Occup = Ek

Vilket är ditt civilstånd?
Civ-stat =
Note01 = Any amount of free text goes here

Sverige
Tyskland
USA
Ungern

Occup =
Administrativ chef
Arbetssökanden
Barnsköterska
Besiktningsman
Bibliotekarie
Biomedicinsk analytiker
Bokföring
Byggnadssnickare
Cykelreparatör
Dagbarnvårdare
Datakonsult
Datoroperatör
Domare
Ekonom
Ekonomibiträde
Företagare
Försäljare
Förskolärare
Förtidspensionär
Forskningsingenjör
Hästskötare
Hemmafru
Idrottsinstruktör
Idrottsplatsarbetare
Informatör

Figure 6: MedRecords Form. At the top left is the navigation area which is used to navigate into the appropriate part of the input view at the bottom. To the right is a list of values linked to the attributes in the input view.

4.3.2 Design

Following the analysis of entering information as the act of defining records, MR was designed to display partial definitions and to provide efficient techniques for completing them.

The user interface of MR consists of one window divided into three views as shown in Figure 6. At the top left is a navigation area, below it is the input view where data is entered, and to the right is a list of commonly used values. Apart from auxiliary windows for editing preferences and the like, all work is performed within this single window. The contents of each view is taken from template files in Rich Text Format (RTF). Thus, the contents of a form can be replaced completely without any modification to the program. Further, the layout of each view can be designed using all common features found in word-processors with respect to font, colors, tabbing, etc. In addition, each user may customize the layout.

The interaction paradigm is based on a small number of basic operations found in many applications. The input view at the bottom left works as a specialized text-editor. It displays an incomplete definition or a "form" which is edited when data is entered. The form contains arbitrary lead texts and a number of knowledge base attributes, each followed by an equals sign. The equals sign marks the beginning of an implicit input textfield where the value of the attribute is entered, see Figure 6. Only these implicit input fields in the input view may be edited by users. All other parts of the displayed text are fixed.

Navigating within the input view can be done by tabbing between the different attributes, scrolling, using standard navigation keys, or by following the links in the navigation view at the top. The navigation view typically displays links into all the main sections of the input view. Clicking a link has the expected effect: focus is moved to the corresponding area of the input view.

Values may be entered in several ways. First, by simply typing the value. As a value is being typed, the first matching value in the list to the right is highlighted. Pressing the completion key or clicking on the highlighted value inserts it into the form. Second, by following a link from an attribute to its value list to the right and clicking on the desired value. The value is then inserted into the input view. Additional documents related to a record, such as images, may be included by dragging and dropping them on the input view.

Thus, MR is based on a simple flow of actions from navigation view, to input view, to value list view and back. All the actions involved are simple and well known to most users. Data may be entered in any order and all attributes in the input view are instantly accessible via the navigation view.

4.3.3 Discussion

MR has evolved through a continuous interaction between users and developers. Starting from when the first prototype was stable enough, it has been in use to enter data during clinical examinations. First, a prototype was built to test the concept. Based on the success of the prototype, a more complete application was built together with an editor that is used to create new forms. Today, a third version, which runs on Windows and Mac OS X systems, is in use.

MR has been used as the input application to enter data for all the records in the MedView knowledge base. All this data has been entered by the clinician performing the examination while talking to and examining the patient. The interaction paradigm based on well-known components such as keyboard, mouse, hypertext links, drag & drop, and ordinary text editing works very well.

Current forms consist of about 100 attributes and a large number of values, e.g., lists of different drugs and diseases. The navigation tools are sufficient although some fine-tuning of the systems scrolling behavior is called for.

Compared to traditional form-based interfaces we believe that MR scales very well. Having 100 different readily available attributes in one screen poses no problem. Displaying traditional forms for the same amount of attributes would require navigating between many different screens, typically in some fixed order.

A recent form covering more than 1000 different attributes in the area of oral medicine has been tested. The value lists associated with this form contains more than 12 000 distinct values. The initial experiments with this form indicate that MR works as well as with smaller forms. Another aspect is the simplicity with which new forms may be created. New forms are created using InterfaceMaker, see Section 4.5, an editor comparable to HTML-editors. No programming is required.

The success of MR shows that focusing on simplicity and long-time usefulness instead of elaborate GUIs can be a good thing indeed. Testing the concepts of MR on a large number of different kinds of forms remains an area for future work.

4.4 MedSummary

The first, and so far most used, knowledge base *viewer* is MedSummary (MS). MS is used in conjunction with MR in the examination room, but also to display detailed information during analysis of the material in the knowledge base.

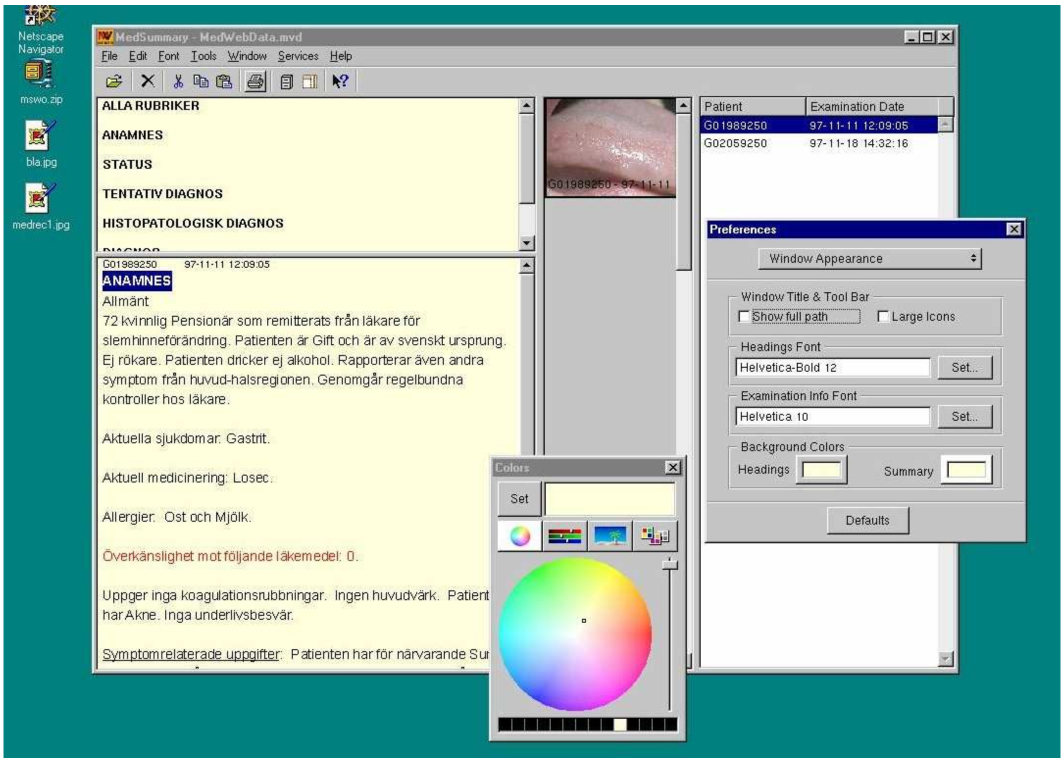
The view of the knowledge base presented by MS is that of a textual summary of one or more examination records together with any associated images as shown in Figure 7. The purpose is to display in a format suitable for viewing the information collected in MR. While it is possible to view data using MR it is not a recommendable way to learn what an examination record is all about. Instead of showing the form or screen used to *collect* data, we use Natural Language Generation (NLG) to generate from the collected data a comprehensible summary of all or parts of the examination(s).

4.4.1 Working with MedRecords and MedSummary

As mentioned earlier, we make a clear distinction between input applications and viewers. While this may sound obvious, the electronic medical record systems we have encountered use the same display to input information and to view it. Consequently, the displays used are optimized neither for entering nor for viewing information.

When working with MR and MS these two activities are separated. New examination data is entered with MR, the contents of existing examination data is viewed using MS. The main window of MS shown in Figure 7 contains a listing of selected examinations to the right, thumbnails of images in the middle and the generated medical record text to the left. Clicking on a thumbnail image will show it full-sized in a separate window. Different texts can be generated by selecting between the headings shown at the top left.

When a previously examined patient comes back for a follow-up, the user



ALLA RUBRIKER

ANAMNES

STATUS

TENTATIV DIAGNOS

HISTOPATOLOGISK DIAGNOS

G01989250 97-11-11 12:09:05

ANAMNES

Allmänt

72 kvinnlig Pensionär som remitterats från läkare för slemhinneförändring. Patienten är Gift och är av svenskt ursprung. Ej rökare. Patienten dricker ej alkohol. Rapporterar även andra symptom från huvud-halsregionen. Genomgår regelbundna kontroller hos läkare.

Aktuella sjukdomar: Gastrit.

Aktuell medicinering: Losec.

Allergier: Ost och Mjölk.

Överkänslighet mot följande läkemedel: 0.

Uppger inga koagulationsrubbningar. Ingen huvudvärk. Patient har Akne. Inga underlivsbesvär.

Symptomrelaterade uppgifter: Patienten har för närvarande Su

Patient	Examination Date
G01989250	97-11-11 12:09:05
G02059250	97-11-18 14:32:16

Preferences

Window Appearance

Window Title & Tool Bar: ☐ Show full path ☐ Large Icons

Headings Font: Helvetica-Bold 12 Set...

Examination Info Font: Helvetica 10 Set...

Background Colors: Headings Summary

Defaults

Figure 7: MedSummary: main and preferences windows. Different texts may be generated by selecting from the headings at the top left. Clicking on a minimized image will show it full-sized in a separate window.

can create a suitable background text. To do this, the user selects the desired previous examinations, and then clicks on a heading to show a summary together with existing images. Once a summary has been created it can be edited as any ordinary text-document if necessary. The text can then be printed and used for things like providing a detailed medical history if the patient is sent to another clinician.

The text-generation used is very flexible and can be adjusted easily both with respect to contents and formatting. Thus, different users may have different summary texts based on the same database if desired. Apart from values of attributes that allow free text, generating summaries in different languages poses no special problems.

4.4.2 Natural Language Generation

Natural Language Generation is the activity of generating text from some kind of sources. A good overview of the area can be found in [23, 24]. In principle, there are two approaches to generation, the *deep* and *shallow* approach. A deep system builds on a deep understanding of linguistics whereas shallow systems use simpler methods to generate text. The advantage of deep text-generators is that they are more domain independent and thus can be applied to various areas with relative ease. However, building a deep system requires a lot of knowledge and resources. Shallow systems are typically specialized for a particular task and need not be more complicated than the task demands. On the other hand, they are less reusable for another task. Some deep systems are described in [9, 42, 80, 92], examples of shallow systems can be found in [17, 40, 91]. Discussions of the two approaches can be found in [17, 23, 89, 90].

Typically, a NLG system is divided into three phases [88]

- Content Determination
- Sentence Planning
- Surface Realization

performed in sequence. Thus, the system first decides what the text should contain, then plan the general structure at sentence level, and finally, realize the desired structure into text. Other approaches are used as well. For instance, [78, 103] propose an integrated constraint-based method that performs all three activities at the same time. The RAGS project [18] is an attempt to develop a reference architecture for NLG systems.

4.4.3 Text Generation In MedSummary

The main focus during the development of MS and the text-generation used has been to create a very flexible system where users can experiment with different texts without having any linguistic expertise. Thus, from a NLG point of view the system is a basic shallow system. Close to a simple mail-merge system, it can be classified as a slot-and-filler, or canned-text with knowledge base references system [17].

Although the text-generator used is very simple it can be clarifying to describe it using the three standard phases mentioned above. Examination summaries have a structure based on the formalization of examinations used in MedView. An examination record forms a tree structure with top level nodes representing the different main tasks from which information is gathered at examinations. Text can be generated for all tasks performed at an examination, for a particular task, or any desired combination of tasks. It is also possible to generate a text covering several examinations. In terms of the phases above:

- Content planning. Depending on the user's choice it is decided what parts of a text template should be used in the resulting text and for which examinations summary text should be created.
- Sentence planning. Depending on which attributes of an examination record have values, it is decided which sentences of the selected template should be included in the text. Sentences for which values are missing are omitted.

- Surface Realization. Depending on the values for attributes in the database particular text-fragments are selected and used to fill slots in sentence templates.

The text-generator takes as input a *template* describing the texts to generate. This template consists of a number of files providing (i) an RTF template text with slots to be filled in depending on the values for attributes in an examination record, (ii) a file describing the connections between slots in the template text and attributes in the knowledge base, (iii) a file that classifies the attributes of the knowledge base into a number of groups, (iv) a file that defines the text-fragments to use as slot-fillers for attribute values. The last file does not simply list value-text pairs, but allows some slightly more complex substitution patterns as well.

The text generator parses the template files into a number of definition objects. Most notably, each attribute gets its own definition object describing text fragments for all possible values of the attribute. Formatting information is kept from the RTF template. To modify the look of generated texts each user may freely change all formatting attributes, font, color, aligning etc. without affecting the actual contents of summary texts.

With the syntax currently used, part of a template could be:

> **§DISEASE HISTORY§**
> Age year old Sex $Occup$ who is referred by $RefIn$ because of a $RefCause$. The patient is $CivStat$ and comes originally from $Born$. $Checkup$.

Now, if part of the definition of an examination is

```
Occup = Lärare.
Ref-in = Tandläkare.
Ref-cause = Slemhinneförändring.
Civ-stat = Gift.
Born = Sverige.
Checkup = Ja.
```

and the value-text maps include the following:

```
Occup:
Lärare = teacher.

Ref-in:
Tandläkare = a general dentist.

Ref-cause:
Slemhinneförändring = mucosal lesion.
```

```
Civ-stat:
Gift = married.

Born:
Sverige = Sweden.

Checkup:
Ja = Attends medical check-ups regularly.
```

the generated text for DISEASE HISTORY becomes:

> **DISEASE HISTORY**
> 58 year old female teacher, who is referred by a general dentist because of a mucosal lesion. The patient is married and comes originally from Sweden. Attends medical check-ups regularly.

Since most values in the MedView knowledge base are in Swedish they have to be given a translation to generate English text. However, if values had been given a neutral language independent coding instead, it would still have been necessary to translate from these codes into English text.

4.4.4 Implementation

MedSummary is written in Objective-C. The text templates used are parsed into a number of definition objects, which were developed as an early part of the Gisela project [109]. There are two versions of the generator, one that uses an RTF template and produces output in RTF format and one that uses an HTML template and produces HTML output. The HTML generator makes it very simple to produce summary texts for web publication, see Section 4.9. The performance of the text-generator is quite sufficient, the desired summary text is displayed immediately.

4.4.5 Discussion

In [90], the term *automatic text generation* (ATG) is used to refer to any computer program that automatically produces texts from some input data. ATG systems are then divided into NLG systems and *template* systems. A template system is defined as a system that simply manipulates character strings using little, if any, linguistic knowledge. From this point of view, the current MS application should be seen as a template system. However, we find it useful to discuss the system in the light of NLG and we are moving towards including more NLG techniques into the system.

As stated earlier, ease of use by non-experts has been deemed more important than producing optimal text quality or using linguistically motivated methods.

The template files forming the basis for generation should best be seen as the user interface to the system for content management. By this we mean that it should not be regarded as *the representation* of the framework used but rather as an *interface* to enter information into the system from which a suitable internal representation can be built. This internal representation could be something with a deeper basis in NLG than the current system. If a more sophisticated system should be introduced it must not be at the expense of the possibility for users to design their own summary texts.

It is interesting to note that several choices made in the development of MS are essentially orthogonal to the approaches suggested for NLG systems. We discuss some issues below. We also mention where NLG techniques would be appropriate in MedSummary and related systems.

Creating a Corpus The corpus-based approach [23] advocates that the first step in the construction of a NLG system is to build a corpus of example texts. This corpus, which should cover the full range of texts the system will produce, is then analyzed for linguistic and information content. Our approach has instead been to build a system where the users, through experiments with given tools, can decide the texts themselves. Actually, an initial prototype for MS was built using something of a corpus-based approach. A number of templates were extracted in collaboration with a domain-expert and realized into an application. However, it was soon discovered that a system that required the assistance of a programmer to alter the contents of summary texts was not appropriate. Therefore, the current system where texts can be continuously refined was developed. By now, after a couple of years of use, it would probably be possible to take a number of generated texts from the system, check them for errors, and use them as a corpus.

Flexibility It is often argued that a major advantage of sophisticated NLG over template systems is that deeper systems are more flexible and easier to maintain. Exactly why this is the case is not always clear. It is interesting that in the development of MS we have selected to use a simple template approach to achieve great flexibility. Of course, this is related to the fact that it is necessary that the *end-users* themselves can modify what the text generated from examination data should be. The text-files used as templates for text-generation are simple enough to be modified by end-users. To expect that they would be able to easily control the workings of a sophisticated NLG system is not realistic.

Text quality Systems building on linguistic knowledge are generally able to produce text of higher quality compared to template based systems. Whether this higher quality is needed depends on what the texts should be used for and on the complexity of the generated texts.

The structure of medical record text is typically very static and uses a rather formal language. Furthermore, there is no need to produce text with great variation. On the contrary, too much variation might be disturbing since clinicians reading the texts expect them to follow certain patterns. Most of the texts generated by MS are read once in the examination room and then discarded. It is also more important that the summary is displayed immediately than that text quality is optimal. This indicates that for the MS application domain a template-based approach is sufficient.

Since the templates used for generation have been refined repeatedly for several years, the quality of the generated texts is in most cases sufficient for their task. In case further refinements are needed, the text may be edited by the user.

Recently, we have put together a web application where summaries, together with images, for selected patients can be viewed. Text can be generated in either English or Swedish. Adding more languages is a simple matter of modifying the text templates. However, in this context it is not possible for users to create new templates. It could therefore be appropriate to use a more complex system since it will be maintained by experts and not by end-users.

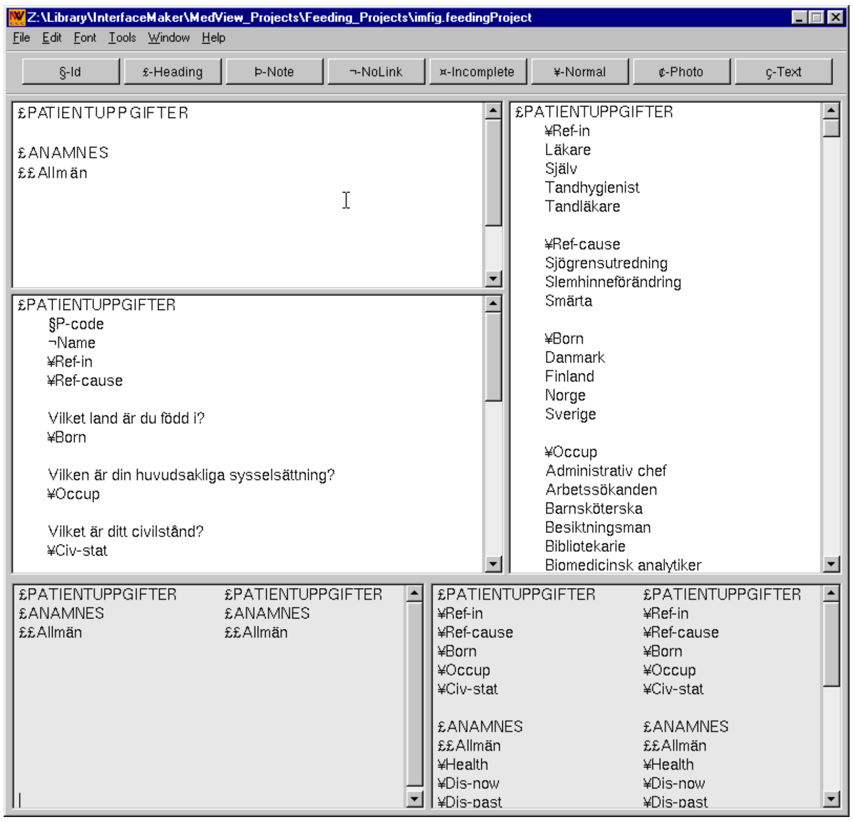
Hybrid Approaches While a deep NLG system does not appear to be needed in MS, using a *hybrid system* would be quite useful. Several hybrid systems have been developed which combine templates with deeper NLG techniques [16, 57, 93].

One obvious technique being a candidate for inclusion in a future version of MS is *aggregation*. Aggregation is used to combine related phrases and sentences together in a linguistically correct manner. Some basic aggregation can be performed in an experimental text-generator we have implemented using the Gisela framework [110]. In MS sentences are either included or omitted depending on whether all attributes needed to generate the sentence have values or not. In the Gisela-based generator the choice is made at a higher level; this, among other things, allows the combination of two sentences into one in certain cases.

Finally, we note that *multi-modality* is of increasing importance in document generation. We need to be able to include diagrams, tables, and other graphics into patient summaries. The images displayed along with the generated text in MS are as important for the clinician as the generated text. Support for tables is present in the Gisela-based generator mentioned above. Creating fully multimodal documents is an interesting challenge for the future.

4.5 InterfaceMaker

Both MR and MS are developed to allow that the contents of the forms or protocols used and the text generated is completely replaceable without changing the application itself. To aid in creating new forms there is a tool called InterfaceMaker (IM). IM is similar to HTML-editors. The user writes the various texts of the new form and adds tags to create links, see Figure 8. IM also supports the creation of text templates for MS.


Z:\Library\InterfaceMaker\MedView_Projects\Feeding_Projects\imfig.feedingProject
File Edit Font Tools Window Help
§-Id
£-Heading
Þ-Note
¬-NoLink
¤-Incomplete
¥-Normal
¢-Photo
ç-Text

```
£PATIENTUPPGIFTER

£ANAMNES
££Allmän
```

```
£PATIENTUPPGIFTER
    §P-code
    ¬Name
    ¥Ref-in
    ¥Ref-cause

    Vilket land är du född i?
    ¥Born

    Vilken är din huvudsakliga sysselsättning?
    ¥Occup

    Vilket är ditt civilstånd?
    ¥Civ-stat
```

```
£PATIENTUPPGIFTER
    ¥Ref-in
    Läkare
    Själv
    Tandhygienist
    Tandläkare

    ¥Ref-cause
    Sjögrensutredning
    Slemhinneförändring
    Smärta

    ¥Born
    Danmark
    Finland
    Norge
    Sverige

    ¥Occup
    Administrativ chef
    Arbetssökanden
    Barnsköterska
    Besiktningsman
    Bibliotekarie
    Biomedicinsk analytiker
```

```
£PATIENTUPPGIFTER      £PATIENTUPPGIFTER
£ANAMNES               £ANAMNES
££Allmän               ££Allmän
```

```
£PATIENTUPPGIFTER      £PATIENTUPPGIFTER
¥Ref-in                ¥Ref-in
¥Ref-cause             ¥Ref-cause
¥Born                  ¥Born
¥Occup                 ¥Occup
¥Civ-stat              ¥Civ-stat

£ANAMNES               £ANAMNES
££Allmän               ££Allmän
¥Health                ¥Health
¥Dis-now               ¥Dis-now
¥Dis-past              ¥Dis-past
```

Figure 8: InterfaceMaker: main window.

IM is more of an administrator tool than a user tool. The general methodology developed within MedView suggests that new forms or protocols are created only when the contents have been formalized and harmonized. Thus, creating new forms should be done by authorized persons only, at the time when a new protocol is to be adopted within the user community.

4.6 Basic Visualizations

The very first visualization of the knowledge base developed was an application that shows ordinary 2D views of data in the knowledge base. The user can view data in a scatter-plot as shown in Figure 9 or as a bar chart. Values for any number of attributes can be displayed simultaneously to let the user look for interesting groups of patients. In Figure 9 the upper left corner shows a cluster of patients born in Sweden who have been referred by their general dentists because of a mucosal lesion.

The application also allows the user to view only a restricted part of the knowledge base by first making a selection based on any combination of the attributes in the knowledge base. Furthermore, details for any particular dot in

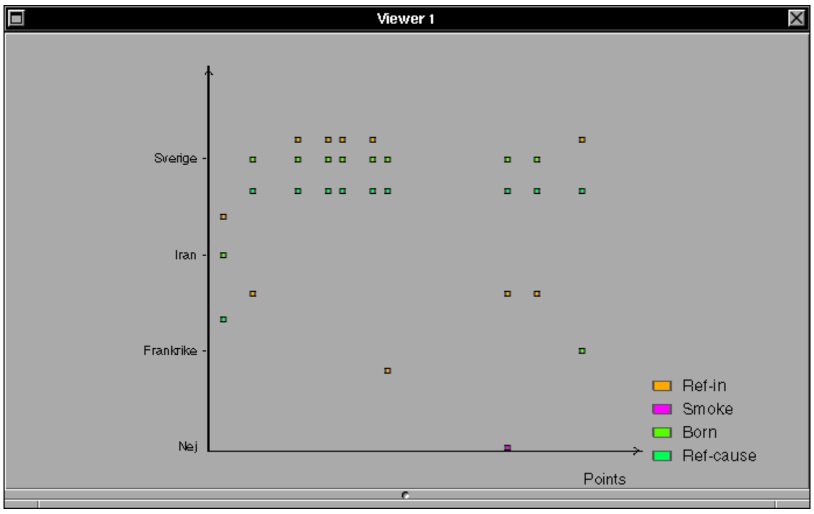


Figure 9: A scatter-plot showing values in the knowledge base.

the display can be shown. So far, the 3D visualizations discussed below have been preferred by the clinicians. We will investigate the reasons for this and ways to build better 2D viewers in the future.

4.7 The Cube

The most used visualization of the knowledge base so far is called The Cube. The Cube has been developed to enhance the clinician's ability to intelligibly analyze the patient material within the knowledge base and to allow for pattern recognition and statistical analysis. The Cube is based on the idea of using parallel coordinates [52] as a solution to multidimensional data analysis [6, 21]. The visualization of parallel coordinates is discussed in [111, 117, 118].

The starting point was the formalization of the notion of a clinical examination as a definition. A formal examination is seen as a set of definitions of specific examination terms. An excerpt of such a definition was shown in Section 3.1.

Clearly, for a given collection of examinations, such a term can be viewed using a simple scatter plot with the x-axis as a sort of time line, e.g. ordered by examination date, and with the values of the term on the y-axis. Thus, if we want an overview of the total set of terms it is natural to think in terms of multiple parallel diagrams (this is similar to the scatter matrix of [21]). This view was then generalized into dynamic 3D parallel diagrams with support for direct manipulation, a 3D cube. The idea is similar to the concept of 3D parallel coordinates, e.g., the casement displays used in [111]. The reason for using 3D was that the notion of 3D parallel diagrams was conceptually very natural from a clinical point of view; it seemed to be a natural model of the raw material of clinical experience. The idea of investigating such a 3D object to learn from

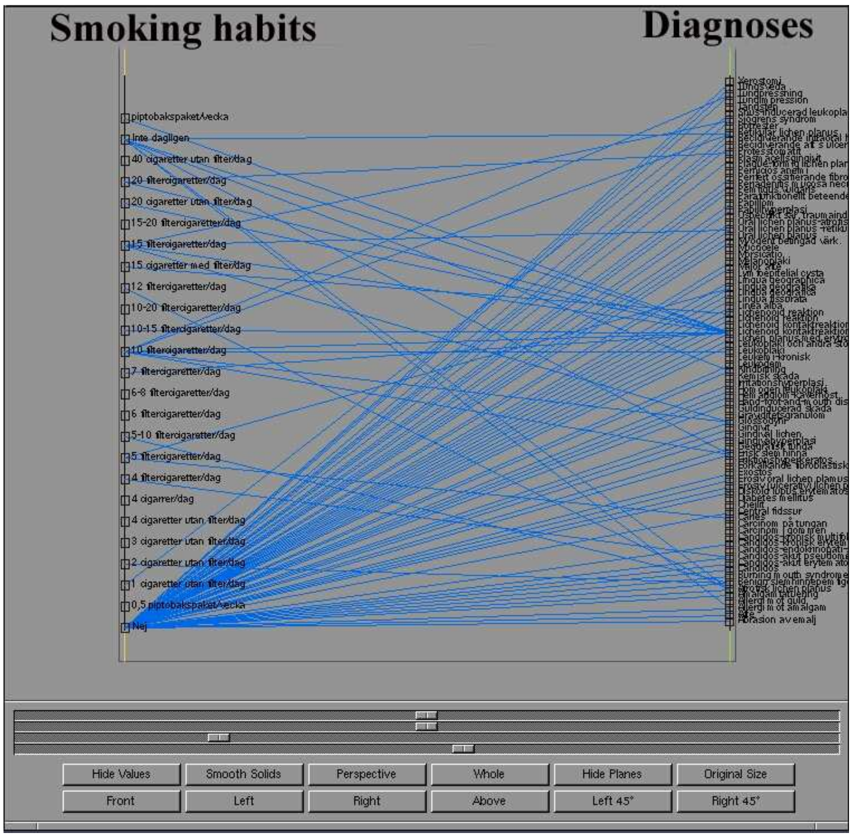


Figure 10: All diagnoses related to the number of cigarettes/day.

clinical data was very appealing.

4.7.1 Defining The Cube

An examination record consists of a number of examination terms (attributes). The user must first decide how many 3D parallel diagrams should be used, and which attribute should be displayed in which diagram.

The total set of attributes in the knowledge base is displayed in a panel and the user simply selects the desired attributes from this list. An attribute can also be marked to be used as the unit on the z-axis.

4.7.2 Viewing The Cube

The Cube consists of a number of planes, one for each attribute that was selected when The Cube was defined. These planes are presented along the x-axis. The z-axis is typically used as a timeline, i.e., as an ordering on the examination identification attribute, but an arbitrary attribute can be used as the unit on this axis. The y-axis then lists the values for the attribute of the corresponding plane in some order, e.g., in alphabetical or numerical order. In any case, each examination is represented by a line connecting individual values in the different attribute planes. If an examination has more than one value for an attribute, the values are connected with a line in the plane. A picture of The Cube with two planes can be seen in Figure 10.

The user can observe The Cube from any desired direction, either by dragging or rotating The Cube with the mouse, or by using the controls at the bottom of the main window.

The appearance of The Cube can be changed in various ways: a strictly parallel projection can be used, the elements of The Cube, e.g., the lines, points, and planes, can temporarily be hidden, the user can change the colors of the elements, elements can be set to be transparent, etc.

If the user finds some lines, i.e., examinations, particularly interesting, these lines can be selected and then opened in a separate window for closer inspection. Similarly, if some lines are blocking the view of others, these lines can temporarily be removed. To get a summary of a number of examinations, the user can select the lines and then open the corresponding examination files in MS.

It is also possible to get statistics about a selected plane: for each value in the plane, the number of examinations with that value is displayed using an ordinary bar chart.

4.7.3 Grouping of Attributes

When data from thousands of examinations are displayed in The Cube, the display will be filled with lines and it may be difficult to recognize clinically meaningful patterns. To solve this problem two techniques can be applied: either showing only a subset of the knowledge base based on a selection made before defining The Cube, or grouping values into classes in a hierarchical manner. For example, a number of diseases can be grouped into viral diseases. Such classifications of attribute values reduce the complexity of the display and facilitate the detection of interesting patterns.

Groups can be created and stored in a library for future use. From a theoretical point of view, a group is simply a definition relating values to groups. Examples of existing simple groups are a division between smokers and non-smokers and between patients with oral lichen planus and patients which do not have oral lichen planus. Combining various groups gives new interesting patterns to explore.

4.7.4 An Example

The Cube is used for finding patterns and correlations. The typical question posed is ”How does a certain set of attributes relate to each other for the entire patient material?” or simply viewing a single attribute. If the patient material is homogenous from the aspect of parameters chosen in the analysis, the lines will appear parallel to each other within The Cube. Heterogeneity and outliers

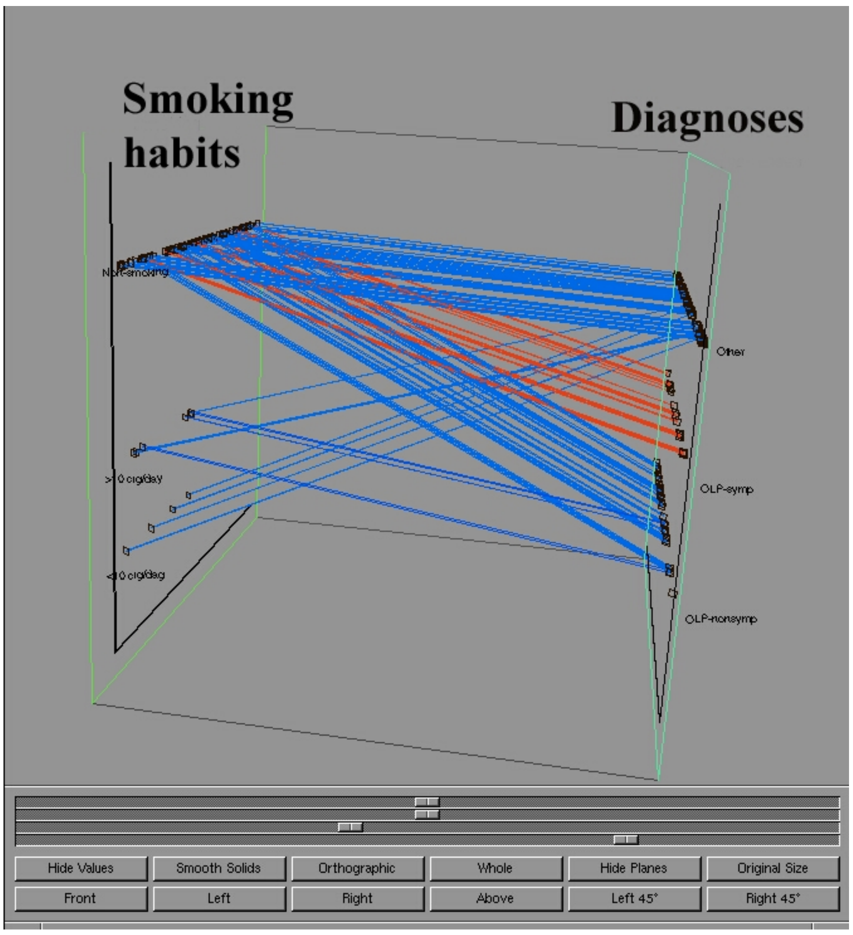


Figure 11: The picture in Figure 10 has been simplified by grouping values.

in the patient material for a certain parameter will, consequently, cause the lines to diverge from each other in the corresponding plane. By using the various selection possibilities, a step-wise procedure may be performed where hypotheses are continuously refined.

Oral lichen planus (OLP) is a disease with unknown etiology that effects the oral mucosa. In its most severe form, the disease presents with erosions and ulcerations, which interfere with, for example, eating of citrus fruits and spicy food. Some of the OLP lesions transform into a malignant disease of the oral mucosa.

In this example, The Cube was used to examine drug and smoking habits for symptomatic (ulcerated) and non-symptomatic (non-ulcerated) OLP, information that has not previously been reported. A cube with two planes was defined: on the first plane the smoking habits of the patients were presented and on the other plane the different diagnoses of the patients were displayed, see Figure 10. The display was then simplified by classification of smoking habits into three groups: non-smokers, patients smoking less than 10 cigarettes/day, and patients smoking more than 10 cigarettes/day, and by changing the color for the two different forms of OLP, see Figure 11.

It was revealed that patients with symptomatic OLP (OLP-symp) were non-smokers (100%) compared to patients affected by non-symptomatic OLP (OLP-nonsymp; 81% non-smokers; 11% more than 10 cigarettes/day; 8% less than 10 but more than 1 cigarette/day). The opposite was found for medication where only 47% of the OLP-nonsymp used drugs compared to 65% of the OLP-symp.

These findings raise thoughts about how different factors may influence the development of the two clinical forms of the disease. The reported observations have now to be statistically evaluated and further investigations by using The Cube have to be conducted to examine if patients with OLP-symp take other types of drugs than patients with OLP-nonsymp.

4.7.5 Discussion

A basic metaphor in MedView is that clinical experience can be viewed geometrically as a space of interconnected atomic points of knowledge. Using The Cube in clinical practice has shown that the tool works well conceptually, as an implementation of this idea.

It is interesting to note that, although 2D visualizations such as scatter-plots and bar charts are more obvious and may appear to be easier to understand, the clinicians prefer working with The Cube. One reason for this is that it is the better tool for viewing the collected clinical experience. It is more of a visualization of the space of interconnected points forming clinical experience.

The grouping, or aggregation, of values has been proven very useful to achieve better results with The Cube.

In the future we need to add more tools for direct manipulation of the, often very complex, displays. Examples of such tools are better and more powerful tools for selecting and, temporarily, discarding or hiding various elements of The Cube, and methods and algorithms for minimizing problems with disorientation and occlusion of elements. Parallel diagrams take practice for users to comprehend [101]. Therefore, the work on a methodology for clinical use of The Cube will be extended and carried further.

4.8 SimVis

Similarity assessments play an important role in most cognitive activities. For example, a clinician examining a patient wants to know if there are previous examination records that are similar to the current one, hoping that these might help him, or her, in the diagnosis of the new patient. However, before we can ask for “similar examination records” we must define what “similar” means. SimVis is a tool designed to allow and encourage clinicians to classify and cluster clinical data in different ways, i.e., a tool that enables them to interactively construct and try out new similarity measures.

Much effort have been spent on studying similarity measures within the medical domain, especially in the area of case-based reasoning (CBR) [41]. This includes work on using CBR-techniques in the retrieval of images from image databases [66] and knowledge mining [54]. In [2] clustering was used to find higher conceptual structures of medical data.

4.8.1 Definitional Similarity Measures

Since examination records are given as definitions, it follows that we must first study how to measure the similarity between these definitions in order to be able to classify clinical data.

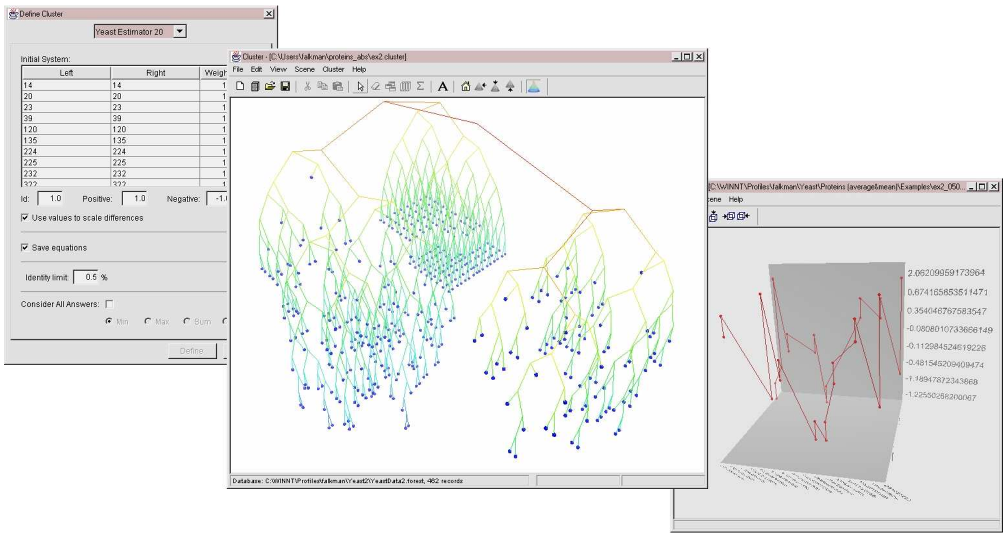
A similarity measure consists of a definition, E, a computation method, M, and a number of definitions, $D_1, \ldots, D_n$. One may think of E as a set of test points, a number of properties of $D_1, \ldots, D_n$, on which the similarity measure is based. The result of the application of M to E and $D_1, \ldots, D_n$, the similarity value, is a new definition, V, which describes both structural and computational similarities between $D_1, \ldots, D_n$. The computation of V is really only the first step in a more general estimation process. If an interpretation of V cannot immediately be found, it can be used as the starting point for further estimation. The result of this second step can, if necessary, be used as the starting point for a third step etc. The process usually ends when V equals a test-definition, S, indicating that the interpretation of V is clear to us. More on definitional similarity measures can be found in [32].

All parts of the similarity measure, including the data structures, the computation method and the estimation process, are given as definitions. Since all parts of the model are given as definitions, the user can, in principle, use the output of the model, i.e., the similarity value, as an input in any other part of the model.

Through experimentation with SimVis, a similarity measure can be found that, for instance, captures the characteristics of a certain diagnosis. This measure can then be used for finding records with this diagnosis, and, indeed, the similarity measure can be said to define this diagnosis.

4.8.2 The SimVis Tool

The general framework for computing similarity measures can be used as a basis for different information visualization models, where each model gives a different visual interpretation of the underlying similarity measures. What is required is a mapping from the results of the estimation process to the visual model. In SimVis a visualization model based on a three-dimensional hierarchical clustering is used. With SimVis, a clinician can interactively construct a similarity measure between examination records, apply the measure to a knowledge base of records, and visualize the resulting classification.



Initial System:

Left	Right	Weigh
14	14	1
20	20	1
23	23	1
39	39	1
120	120	1
135	135	1
224	224	1
225	225	1
232	232	1
322	322	1

Id: 1.0 Positive: 1.0 Negative: -1.

☑ Use values to scale differences

☑ Save equations

Identity limit: 0.5 %

Consider All Answers: ☐

[✓ Min] Max Sum

Database: C:\WINNT\Profiles\falkman\Yeast2\YeastData2.forest, 462 records

Figure 12: SimVis: the panel for constructing similarity measures (left), the visualization of clusters (middle), and the visualization of individual similarity values, i.e., cluster points (right).

SimVis consists of three modules, which are shown in Figure 12. The first module is used for constructing similarity measures and estimations. On the basis of the similarity values, a three-dimensional hierarchical clustering is created, visualized, and examined using the second module of SimVis. The similarity values themselves can be examined in detail using the third module.

Constructing Similarity Measures To construct a similarity measure, the user starts off by choosing one of many predefined computation methods. Each method has its own characteristics and parameters. These can, for instance, be the weights assigned to the different attributes of the examination records, if the length of the estimation process should be taken into account or not etc.

The user then defines which attributes of the examination records that should be taken into account. It is also possible to save the similarity measure for future use.

Visualization Model The user can apply the current similarity measure (or one saved from a previous session) to the knowledge base. From the resulting similarity matrix, a hierarchical cluster is constructed. A 3D visualization of the cluster can then be examined using the second module of SimVis (the middle window in Figure 12). To facilitate the exploration of data, clusters can be visualized in different ways: various parameters controlling the visualization could be modified, color codes could be used, the dynamics of the computation could be simulated by animating the construction of the clusters. If the user finds some sub-clusters particularly interesting, these can be selected and then opened in a separate window for closer inspection. Similarly, if some sub-clusters are blocking the view of others they can be temporarily removed.

The underlying similarity measures can be analyzed in detail as well using 3D parallel diagrams (the right window in Figure 12). The details of this visualization are described in Section 4.7.

4.8.3 Discussion

The theory for similarity measures underlying SimVis could be used as a basis for different classification and visualization models, not just hierarchical clustering. An alternative clustering could be the self-organizing map [58]. Apart from being tested in MedView, SimVis has also proved itself useful in the area of functional genomics, where it has been applied to problems connected to the analysis of expression data from proteome analysis of yeast [33].

Compared to the Cube and the simple 2D visualization tested, SimVis is a step towards a more active system, that is, a system performing tasks for the user, not just an exploration tool. The measures constructed could be used for exploration of data in the knowledge base.

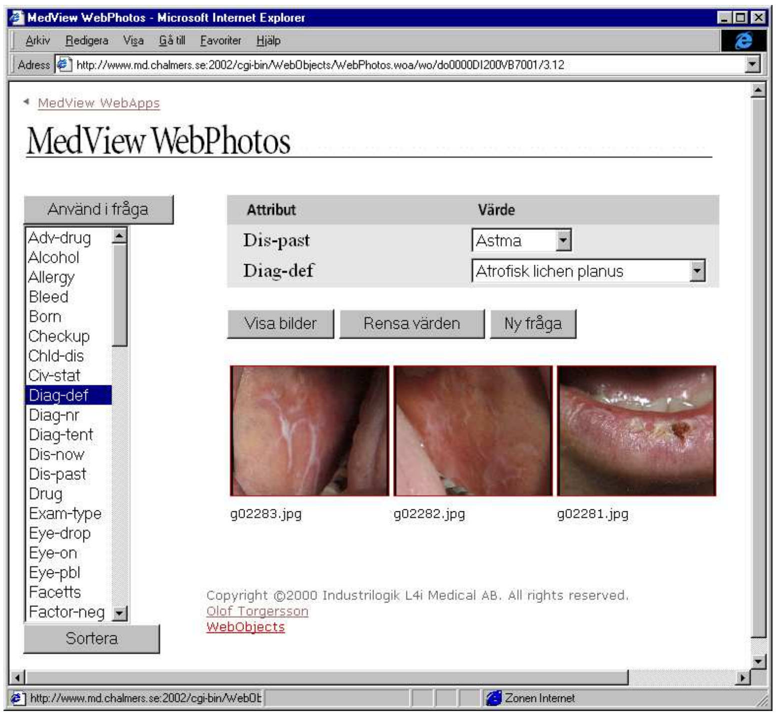
In the future SimVis will be extended into a general case-based reasoning (CBR) system [59, 116] that should provide assistance to clinicians within the field of oral medicine.

4.9 Web Applications

The real treasure of MedView is the knowledge base being built. An obvious step to give clinicians and researchers worldwide access to the data collected is to use the Internet. We are currently considering various web-applications that would allow exploration of the knowledge base using a Web-browser.

To setup a static website with information from the knowledge base would not be very interesting. Instead, we will build dynamic web-applications. Since essentially all the code written so far can be reused, the main problem will be design of an appropriate web-based user-interface.

So far, we have built two simple web applications: One that makes it possible to view patient summaries in the same way as in MedSummary, and one to search the knowledge base for images, see Figure 13. In the search for images each image is indexed by all the information collected at the examination when the image was taken. For example, a query might be "Find all lesions with Mucos-colr red and white and Mucos-txtur plaque". This is possible since images taken at an examination are part of the total knowledge collected at the examination and the knowledge base can be searched for examinations matching any combination of attribute-value pairs.



Attribut	Värde
Dis-past	Astma
Diag-def	Atrofisk lichen planus

Figure 13: Searching for images over the Internet.

5 Future Directions

We believe that MedView is a project that is worth continuing. The foundation for building a large knowledge base in the field of oral medicine has been laid down. Tools that have been put to the test at more than 1500 examinations have been developed and proven useful. Some analysis tools are in use, although in a smaller circle of users. However, to find areas that would be interesting to investigate further is not hard. We mention some of these in no particular order below.

5.1 Foundations

The theoretical model of MedView as a knowledge base containing definitions of examinations is not expected to change. However, some details may need further attention. Examples of such details are the way values are built-up and used. Today all values are atomic. This means that a value, say “2 times a day”, is represented by the atom `'2 x/day'`. There are several reasons for this: First, the basic definitional model used does assume that all values are atomic. Second, `'2 x/day'` does not require any special knowledge about atoms, terms etc. which makes it easier to understand for clinicians. Third, DefinitionG used in some applications to model definitions does only allow atomic values. Of course, an atom like this, which does possess an inherent structure, would be better represented as a compound term that makes it easy to access the components `2` and `x/day`, say `times_day(2)`.

We are currently working on using the Gisela framework as a replacement for DefinitionG as the tool for computing with definitions in MedView. Gisela is a much more flexible tool for programming with definitions than DefinitionG. As such, it is also less efficient and it remains to be seen how much more work is needed before it will reach the level where it can be used as *the* deductive engine and knowledge representation language of MedView.

With Gisela in place, it will be time to look further into knowledge representation and more advanced computations over basic data and knowledge structures built on top of it. Examples are defining new diagnoses based on data in the knowledge base, searching for patterns or similar cases, building a set of useful query filters, such as looking for patients with specific properties instead of examinations etc. Our belief is that Gisela will provide a definitional framework to do the things we need in a sufficiently clean and efficient manner.

5.2 Collaboration

There are several directions in which MedView can benefit from collaborations of various kinds. Some collaborative efforts, considered or ongoing, are mentioned here.

First, extending SOMNET is considered as a way to both increase the expert knowledge within the network, and speeding up knowledge gathering through a larger userbase. Since SOMNET is not about MedView only, this has to be a process where clinics are gradually assimilated. On a related note is letting general practitioners use MedView tools for evaluation and testing. The next extension would be to create an international network building a common knowledge base. Such a venture would of course demand serious efforts in the formalization and harmonization phases.

An international collaboration with Eastman Dental Institute in London is being initialized. There are strong relations between MedView and the work done at Eastman, both theoretically and practically. We hope that this will bring up interesting research opportunities.

Yet another important thing would be to have better cooperation with experts in information visualization, database mining, and pattern recognition. Finally, to further develop the NLG used in MedSummary collaboration with experts in computational linguistics is needed. We hope to be able to start work on this in the near future.

5.3 Applications

As we have mentioned several times before, we are only at the beginning of building tools for exploration of the information that has been collected over the years in MedView. Some ideas for future tools are:

- Database management. The definitional knowledge base model used in MedView needs computerized tools to monitor entered values, add value filters, corrections and so on. Some experiments in this direction are mentioned in [110].

- MedRecords with expert knowledge. Currently, MedRecords simply collects data. An interesting extension would be to add an intelligent agent that aids the user. The agent could provide suggestions for values, verify that entered values are consistent in some manner, or simply rearrange the value-lists so that the values deemed most likely by the agent occurs at the top.

- Combining MedRecords with graphical input devices. Although the basic paradigm used in MedRecords works very well in most cases it is sometimes better to enter data through a graphical user interface. To add a plug-in architecture that would allow various extra input methods would not be very complicated.

- Improved Visualizers. The name Med*View* indicates that viewing visually what is in the knowledge base is an important part of the project's goals. What the best tools for visualizing various aspects of the collected knowledge should be needs some serious work.

- Interactive Distant Consultations. Currently the members of SOMNET send patient information, including images, to each other via email. A better approach would be to build tools for real-time communication using audio/video such that the expert asked for advice can view the patient directly. With both parties having access to the common MedView knowledge base similar cases could be viewed and discussed in relation to the current patient.

- Educational tools. Using the collected material in MedView for education is an obvious application. Educational tools could be of various kinds and directed at different groups, students, graduate students, practitioners, researchers etc. We are currently investigating the possibilities of building an Electronic Handbook of Oral Medicine. An idea of this handbook would be to combine general rules given by experts with actual examples from the MedView knowledge base.

- Web Tools. Related to the above is accessing the MedView database using the World Wide Web. A web application for MedView could combine several of the suggestions above in a MedView web portal. By logging into this the user would have access to distant consultations, image search, patient summaries, electronic handbooks and tutorials, and so on. Once the basics of the various functions are in place, allowing access using the Internet is essentially a matter of programming and user interface design.

- Searching for patterns. Related to the need for visualizations aiding the user is automated database searches. A data-mining program could be constructed to search the knowledge base for patterns that could be reported to an expert for further evaluation. This approach is the dual to letting the user search for patterns using visualizations and direct manipulation.

The list could be made much longer but we stop here for now.

References

[1] L. M. Abbey. An expert system for oral diagnosis. *J Dent Educ*, 51:475–480, 1987.

[2] F. Alte da Veiga. Structure discovery in medical databases: A conceptual clustering approach. *Artificial Intelligence in Medicine*, 8:473–491, 1996.

[3] R. Armstrong, J. Lesiewicz, G. Harvey, L. Lee, K. Spoehr, and M. Zultak. Clinical panel assessment of photodamaged skin treated with isotretinoin using photographs. *Arch Dermatol*, 128:352–356, 1992.

[4] M. Aronsson. Methodology and programming techniques in GCLA II. In *Extensions of logic programming, second international workshop, ELP'91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[5] M. Aronsson. *GCLA, The Design, Use, and Implementation of a Program Development System.* PhD thesis, Stockholm University, Stockholm, Sweden, 1993.

[6] D. Asimov. The grand tour: A tool for viewing multidimensional data. *SIAM J. Sci. Stat. Comp.*, 1(6):128–143, 1985.

[7] T. Axell, J. Pindborg, C. J. Smith, and I. van der Waal. Oral white lesions with special reference to precancerous and tobacco-related lesions: conclusions of an international symposium held in Uppsala, Sweden, May 18-21 1994. International Collaborative Group on Oral White Lesions. *Journal of Oral Pathology*, 25(2):49–54, 1996.

[8] J. D. Bader and D. A. Shugars. A case for diagnoses. *J Am Coll Dent*, (64):44–46, 1997.

[9] J. Bateman. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, (3):15–55, 1997.

[10] D. K. Benn, D. D. Dankel, D. Clark, R. B. Lesser, and A. B. Bridgewater. Standardizing data collection and decision making with an expert system. *J Dent Educ*, 61:885–894, 1997.

[11] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann: San Francisco, CA, 1998.

[12] J. Bolewska, H. Hansen, P. Holmstrup, J. Pindborg, and M. Stangerup. Oral mucosal lesions related to silver amalgam restorations. *Oral Surg Oral Med Oral Pathol*, 70:55–58, 1990.

[13] B. Brehmer. Dynamic decision making human control of complex systems. *Acta Psychol Amst*, 81:211–241, 1992.

[14] M. R. Brickley, J. P. Shepherd, and R. A. Armstrong. Neural networks: a new technique for development of decision support systems in dentistry. *J Dent*, 26:305–309, 1998.

[15] S. L. Brooks. Computed tomography. *Dent Clin North Am*, 37:575–590, 1993.

[16] B. Buchanan, J. Moore, D. Forsythe, G. Carenini, and S. Ohlsson. Using medical informatics for explanation in a clinical setting. Technical Report 93-16, Intelligent Systems Laboratory, University of Pittsburg, 1994.

[17] S. Busemann and H. Horacek. A flexible shallow approach to text generation. In E. Hovy, editor, *Proceedings of the Nineth International Natural Language Generation Workshop (INLG'98)*, pages 238–247, 1998.

[18] L. Cahill, C. Doran, R. Evans, C. Mellish, D. Paiva, M. Reape, D. Scott, and N. Tipper. Towards a reference architecture for natural language generation systems. Technical Report ITRI-99-14, University of Brighton, March 1999.

[19] J. Chasteen. A computer database approach for dental practice. *J Am Den Assoc*, 123:26–33, 1992.

[20] V. Chinburapa, L. Larson, M. Brucks, J. Draugalis, J. Bootman, and C. Puto. Physician prescribing decisions: the effects of situational involvement and task complexity on information acquisition and decision making. *Soc. Sci. Med.*, 36:1473–1482, 1993.

[21] T. Chomut. Exploratory data analysis in parallel coordinates. Research report, IBM Los Angeles Scentific Center, 1987.

[22] J. J. Cimino, P. D. Clayton, G. Hripcsak, and S. B. Johnson. Knowledge-based approaches to the maintenance of a large controlled medical terminology. *J. Am. Med. Inform. Assoc.*, 1:35–50, 1994.

[23] R. Dale and E. Reiter. Building applied natural-language generation systems. *Journal of Natural Language Engineering*, 3:55–87, 1997.

[24] R. Dale and E. Reiter. *Building Natural-Language Generation Systems.* Cambridge University Press, 2000.

[25] J. C. Davenport and P. Hammond. The acquisition and validation of removable partial denture design knowledge. I. Methodology and overview. *J Oral Rehabil*, 23:152–157, 1996.

[26] J. C. Davenport, P. Hammond, and P. Hazlehurst. Knowledge-based systems, removable partial denture design and the development of RaPiD. *Dent Update*, 24:227–233, 1997.

[27] D. N. Davis and D. Forsyth. Knowledge-based cephalometric analysis a comparison with clinicians using interactive computer methods. *Comput Biomed Res*, 27:210–228, 1994.

[28] M. Diehl. Developing the american dental association concept model for the standard computer-based oral health record. *J Am Coll Dent*, 62:30–32, 1995.

[29] C. Dragula and G. Burin. International harmonization for the risk assessment of pesticides results of an IPCS survey. *Regul. Toxicol. Pharmacol.*, 20:337–353, 1994.

[30] R. B. Elson and D. P. Connelly. Computerized decision support systems in primary care. *Prim Care*, 22:356–384, 1995.

[31] G. Enislidis, I. V. Wagner, O. Ploder, and R. Ewers. Computed intraoperative navigation guidance–a preliminary report on a new technique. *Br J Oral Maxillofac Surg*, 35:271–274, 1997.

[32] G. Falkman. Similarity measures for structured representations: a definitional approach. In E. Blanzieri and L. Portinale, editors, *EWCBR-2K, Advances in Case-Based Reasoning*, Lecture Notes in Artificial Intelligence. Springer–Verlag, 2000. To appear.

[33] G. Falkman, J. Norbeck, L. Hallnäs, and A. Blomberg. The use of similarity measures and three-dimensional hierarchical clustering for the analysis of expression data from proteome analysis of *Saccharomyces cerevisiæ*. Presented at *Bioinformatics'99, Lund, Sweden, June 1999*, June 1999.

[34] C. Farr. Million-dollar files: adding up the benefits of computerized patient records. *Dent Today*, 14:72–79, 1995.

[35] A. R. Firestone, D. Sema, T. J. Heaven, and R. A. Weems. The effect of a knowledge-based image analysis and clinical decision support system on observer performance in the diagnosis of approximal caries from radiographic images. *Caries Res 32*, 32:127–134, 1998.

[36] F. J. Firriolo and B. A. Levy. Computer expert system for the histopathologic diagnosis of salivary gland neoplasms. *Oral Surg Oral Med Oral Pathol Oral Radiol Endod*, 82:179–186, 1996.

[37] F. J. Firriolo and T. Wang. Diagnosis of selected pulpal pathoses using an expert computer system. *Oral Surg Oral Med Oral Pathol*, 76:390–396, 1993.

[38] D. Fontaine, C. Riou, and C. Jacquelinet. An intelligent computer-assisted instruction system for clinical case teaching. *Methods Inf Med*, 33:433–445, 1994.

[39] M. R. Frank and P. Szekely. Adaptive forms: An interaction paradigm for entering structured data. In *Proceedings of the 1998 International Conference user interfaces*, San Fransisco, CA USA, 1998.

[40] S. Geldof and W. V. de Velde. An architecture for template based (hyper)text generation. In *Proceedings of the 6th European Workshop on Natural Language Generation-EWNLG'97*, pages 28–37, 1997.

[41] L. Gierl, M. Bull, and R. Schmidt. CBR in medicine. In *Case-based reasoning technology: From foundations to applications*, volume 1440 of *Lecture Notes in Artificial Intelligence*, pages 273–297. Springer-Verlag, 1998.

[42] E. Goldberg, N. Driedger, and R. Kittredge. Using natural-language processing to produce weather forecasts. *IEEE Expert*, 9(2):45–53, 1994.

[43] W. T. Gossen, P. J. Epping, and I. L. Abraham. Classification systems in nursing: Formalizing nursing knowledge and implications for nursing information systems. *Methods Inf Med*, 35(1):59–71, 1996.

[44] L. Green and M. Becker. Physician decision making and variation in hospital admission rates for suspected acute cardiac ischemia. A tale of two towns. *Med. Care*, 32:1086–1097, 1994.

[45] T. A. Gregg and D. H. Boyd. A computer software package to facilitate clinical audit of outpatient paediatric dentistry. *Int J Paediatr Dent*, 6:45–51, 1996.

[46] L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.

[47] P. Hammond, J. C. Davenport, and A. J. Potts. Knowledge-based design of removable partial dentures using direct manipulation and critiquing. *J Oral Rehabil*, 20:115–123, 1993.

[48] R. M. Hammond and T. J. Freer. Application of a case-based expert system to orthodontic diagnosis and treatment planning: a review of the literature. *Aust Orthod J*, 14:150–153, 1996.

[49] R. M. Hammond and T. J. Freer. Application of a case-based expert system to orthodontic diagnosis and treatment planning. *Aust Orthod J*, 14:229–234, 1997.

[50] V. Hasselblad and D. McCrory. Meta-analytic tools for medical decision making: a practical guide. *Med Decis Making*, 15:81–96, 1995.

[51] J. J. Hyman and W. Doblecki. Computerized endodontic diagnosis. *J Am Dent Assoc*, 107:755–758, 1983.

[52] A. Inselberg. The plane with parallel coordinates. *The Visual Computer*, 1:69–91, 1985.

[53] R. A. Jenders, M. Morgan, and G. O. Barnett. Use of open standards to implement health maintenance guidelines in a clinical workstation. *Comput Biol Med*, 24:385–390, 1994.

[54] I. Jurisica, J. Mylopoulos, J. Glasgow, and H. Shapiro. Case-based reasoning in IVF: Prediction and knowledge mining. *Artificial Intelligence in Medicine*, 12(1):1–24, 1998.

[55] D. Kalra. Electronic health records the European scene. *Bmj*, 309:1358–1361, 1994.

[56] M. A. Kamrin. Environmental risk harmonization federal/state approaches to risk assessment and management. *Regul Toxicol Pharmacol*, 25:158–165, 1997.

[57] A. Knott, C. Mellsih, J. Oberlander, and M. O'Donnell. Sources of flexibility in dynamic hypertext generation. In *Proceedings of the International Workshop on Natural Language Technique*, pages 64–71, 1996.

[58] T. Kohonen. *Self-organizing maps.* Springer-Verlag, 2nd edition, 1997.

[59] J. L. Kolodner. An introduction to case-based reasoning. *Artificial Intelligence Review*, 6(1):2–34, 1992.

[60] I. Kramer, N. el Labban, and S. Sokodi. Further studies on lesions of the oral mucosa using computer-aided analyses of histological features. *Br J Cancer*, 29:223–231, 1974.

[61] I. R. Kramer. Computers in clinical and laboratory diagnosis. *Int Dent J*, 30:214–225, 1980.

[62] P. Kreuger. GCLA II: A definitional approach to control. In *Extensions of logic programming, second international workshop, ELP91*, number 596 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1992.

[63] P. Kreuger. GCLA II: A definitional approach to control. Licentiate thesis, Chalmers University of Technology, 1992.

[64] M. J. Lincoln, C. W. Turner, P. J. Haug, J. W. Williamson, S. Jessen, R. M. Cundik, and H. R. Warner. Iliad's role in the generalization of learning across a medical domain. In *Proc Annu Symp Comput Appl Med Care*, pages 174–178, 1992.

[65] H. J. Lowe, B. G. Buchanan, G. F. Cooper, and J. K. Vries. Building a medical multimedia database system to integrate clinical information an application of high-performance computing and communications technology. *Bull Med Libr Assoc*, 83:57–64, 1995.

[66] R. Makura and T. Makura. MACRAD: Radiology image resource with a case-based retrieval system. In M. Veloso and A. Aamodt, editors, *Case-based reasoning research and development: Proceedings of the first international conference, ICCBR-95*, volume 1010 of *Lecture Notes in Artificial Intelligence*, pages 43–54. Springer-Verlag, 1995.

[67] U. Mattson, G. Heyden, A. Chodorowski, T. Gustavsson, M. Jontell, and F. Bergquist. Computer analysis in oral lichenoid reactions. *Acta Odont Scand*, 52:86–92, 1994.

[68] U. Mattson, A. Jönsson, M. Jontell, and J. Cassuto. Digital image analysis (DIA) of colour changes in human skin exposed to standardized thermal injury and comparison with laser doppler measurements. *Comput Methods Programs Biomed*, 50:31–42, 1996.

[69] A. M. McCreery and E. Truelove. Decision making in dentistry. Part I: A historical and methodological overview. *J Prosthet Dent*, 65:447–451, 1991.

[70] A. M. McCreery and E. Truelove. Decision making in dentistry. Part II: Clinical applications of decision methods. *J Prosthet Dent*, 65:575–585, 1991.

[71] J. Merz, M. Small, and P. Fishbeck. Measuring decision sensivity: A combined Monte Carlo-logistic regression approach. *Med Decis Making*, 12:189–196, 1992.

[72] N. Mohl and R. Ohrbach. Clinical decision making for temporomandibular disorders. *J Dent Educ*, 56:823–833, 1992.

[73] B. D. Monteith. Computerized expert system for the diagnosis of pulp-related pain. *Int J Prosthodont*, 4:30–36, 1991.

[74] R. Mulligan and G. J. Wood. A controlled evaluation of computer assisted training simulations in geriatric dentistry. *J Dent Educ*, 57:16–24, 1993.

[75] E. Munch. P.A.I.S., a personal medical information system. A comprehensive medical knowledge base. *Hno*, 42:355–373, 1994.

[76] L. Nicholson and I. Beaulieu. Interactive multimedia learning systems in oral medicine: experiences at the Faculty of Oral Medicine of Laval University. *J Can Dent Assoc*, 63:819–821, 1997.

[77] C. Nielson, C. S. Smith, D. Lee, and M. Wang. Implementation of a relational patient record with integration of educational and reference information. In *Proc Annu Symp Comput Appl Med Care*, pages 125–129, 1994.

[78] J. Nivre and T. Lager. Constraint-based text realization. Submitted to Natural Language Engineering, 1999.

[79] S. P. Nolan. The search for standards. *J .Heart .Valve. Dis.*, pages 7–9, 1995.

[80] Penman. The Penman documentation. Technical report, USC/Information Sciences Institute, 1989.

[81] D. Perednia, J. Gaines, and A. Rossum. Variability in physician assessment of lesions in cutaneous images and its implications for skin screening and computer-assisted diagnosis. *Arch Dermatol*, 128:357–364, 1992.

[82] P. E. Petersen, L. B. Christensen, I. J. Moller, and K. S. Johansen. Continuous improvement of oral health in Europe. *J Ir Dent Assoc*, 40(4):105–107, 1994.

[83] L. C. Peterson, D. S. Cobb, and D. C. Reynolds. ICOHR: intelligent computer based oral health record. *Medinfo*, 2, 1995.

[84] R. W. Priddy and L. Yip. ORPAMS: a data-management system for oral pathology. *Oral Surg Oral Med Oral Pathol*, 61:590–596, 1986.

[85] A. L. Rector and W. A. Nowlan. The GALEN project. *Computer Methods and Programs in Biomedicine*, 45:75–78, 1993.

[86] J. L. Reginster. Harmonization of clinical practice guidelines for the prevention and treatment of osteoporosis and osteopenia in Europe a difficult challenge. *Calcif Tissue Int*, 59:24–29, 1996.

[87] Y. Reisman. Computer-based clinical decision aids. A review of methods and assessment of systems. *Med Inf*, 21:179–197, 1996.

[88] E. Reiter. Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In *Proc of the Seventh International Workshop on Natural Language Generation (INLGW-1994)*, pages 163–170, Kennebunkport, Maine, USA, 1994.

[89] E. Reiter. NLG vs templates. In *Proc of the fifth European Workshop on Natural-Language Generation*, Leiden, The Netherlands, 1995.

[90] E. Reiter. Shallow vs. deep techniques for handling linguistic constraints and optimisations. In *Proceedings of the KI-99 Workshop on May I Speak Freely: Between Templates and Free Choice in Natural Language Generation*, 1999.

[91] E. Reiter, C. Mellish, and J. Levine. Automatic generation of technical documentation. *Applied Artificial Intelligence*, 9, 1995.

[92] E. Reiter, C. Mellsih, and J. Levine. Automatic generation of on-line documentation in the IDAS project. In *Proceedings of the Third Conference on Applied Natural Language Processing (ANLP-1992)*, pages 64–71, 1992.

[93] E. Reiter, R. Robertson, and L. Osman. Types of knowledge required to personalise smoking cessation letters. In *Artificial Intelligence and Medicine: Proceedings of AIMDM-1999*, pages 389–399, 1999.

[94] P. R. Rhodes. The computer-based oral health record exploring a new paradigm. *J Calif Dent Assoc*, 22:29–33, 1994.

[95] D. Rizzi. Medical diagnosis. *Ugeskr Laeger*, 153:694–697, 1991.

[96] G. Rolfe. Science, abduction and the fuzzy nurse an exploration of expertise. *J Adv Nurs*, (25):1070–1075, 1997.

[97] J. L. Rudin. DART (diagnostic aid and resource tool): a computerized clinical decision support system for oral pathology. *Compendium*, 15, 1994.

[98] S. Seipel, I. V. Wagner, S. Koch, and W. Schneider. Three-dimensional visualization of the mandible: a new method for presenting the periodontal status and diseases. *Comput Methods Programs Biomed*, 46:51–57, 1995.

[99] S. Seipel, I. V. Wagner, S. Koch, and W. Schneider. A virtual reality environment for enhanced oral implantology. *Medinfo*, 1995.

[100] P. N. Sellen, D. C. Jagger, and A. Harrison. Computer-generated study of the correlation between tooth, face, arch forms, and palatal contour. *J Prosthet Dent*, 80:163–168, 1998.

[101] B. Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Longman, Inc, Reading, MA, USA, 3rd edition, 1998.

[102] J. H. Sims-Williams, I. D. Brown, A. Matthewman, and C. D. Stephens. A computer-controlled expert system for orthodontic advice. *Br Dent J*, 163:161–166, 1987.

[103] H. Somers, B. Black, J. Ellman, L. Giardoni, T. Lager, A. Multari, J. Nivre, and A. Rogers. Multilingual generation and summarization of job adverts: The TREE project. In *Proceedings of the Fifth Conference on Applied Natural Language Processing*, 1997.

[104] S. T. Sonis and K. A. Costello. A database for mucositis induced by cancer chemotherapy. *Eur J Cancer B Oral Oncol*, 31B:258–260, 1995.

[105] P. M. Speight, A. E. Elliot, J. A. Jullien, M. C. Downer, and J. M. Zakzrewska. The use of artificial intelligence to identify people at risk of oral cancer and precancer. *Br Dent J*, 179:382–387, 1995.

[106] J. Stempczynska and E. Kacki. Problems of knowledge acquisition automation in medical expert systems. *Medinfo*, 1:857–860, 1995.

[107] C. D. Stephens, N. Mackin, and J. H. Sims-Williams. The development and validation of an orthodontic expert system. *Br J Orthod*, 23:1–9, 1996.

[108] S. E. Stheeman, P. F. van der Stelt, and P. A. Mileman. Expert systems in dentistry: Past performance—future prospects. *J Dent*, 20:68–73, 1992.

[109] O. Torgersson. Gisela—a framework for definitional programming, 2000.

[110] O. Torgersson. MedView and Gisela, 2000.

[111] J. W. Tukey. *Exploratory Data Analysis.* Addison-Wesley, Reading, MA, USA, 1977.

[112] J. A. Valenza. Medical risk report: improving patient management and record keeping through a problem-oriented approach. *J Gt Houst Dent Soc*, 65:46–48, 1994.

[113] P. F. van der Stelt. Computer-assisted interpretation in radiographic diagnosis. *Dent Clin North Am*, 37:683–696, 1993.

[114] V. Velanovich. Bayesian analysis in the diagnostic process. *Am J Med Qual*, 9:158–161, 1994.

[115] I. V. Wagner and W. Schneider. Computer based decision support in dentistry. *J Dent Educ*, 1991.

[116] I. D. Watson. An introduction to case-based reasoning. In I. Watson, editor, *Progress in case-based reasoning*, volume 1020 of *LNAI*, pages 3–16. Springer-Verlag, 1995.

[117] E. J. Wegman. Hyperdimensional data analysis using parallel coordinates. *Journal of the American Statistical Association*, 85(411):664–675, 1990.

[118] E. J. Wegman. The grand tour in k-dimensions. In *Computing Science and Statistics. Statistics of Many Parameters: Curves, Images, Spatial Modes. Proceedings of the 22nd Symposium on the Interface*, pages 127–136, New York, 1992. Springer-Verlag.

[119] S. C. White. Decision-support systems in dentistry. *J Dent Educ*, 60:47–63, 1996.

[120] P. J. Whitehouse, C. G. Sciulli, and R. M. Mason. Dementia drug development use of information systems to harmonize global drug development. *Psychopharmacol Bull*, 33:129–133, 1997.

[121] N. Zhizhina, A. A. Prokhonchukov, I. M. Rabinovich, and V. Pelkovskii. A computerized automated system for the differential diagnosis and treatment of oral mucosal diseases. *Stomatologiia*, 77(1):55–61, 1998.

MedView and Gisela

Olof Torgersson
Department of Computing Science
Chalmers University of Technology and Göteborg University
S-412 96 Göteborg, Sweden
oloft@cs.chalmers.se

Abstract

In the MedView project knowledge representation is based on a theory of definitions. In the initial phases of the project, definitions and computations over definitions have been represented using classes customized for a special task. Gisela is an object-oriented framework for definitional programming suitable for knowledge representation in MedView. We show how it can be used in MedView to replace the current ad-hoc implementations of definitions and reasoning.

1 Introduction

In the MedView project [1], clinical examination data are collected at examinations for subsequent use for analysis and learning. The collected data from each examination is stored into a knowledge base currently containing about 1500 records. Apart from using the knowledge base for analytical studies, it is also used at examinations to view disease history for the patient being treated.

The information in the knowledge base is stored in a formalized format to facilitate computerized reasoning. The formalization used is based on a theory of definitions [10]. Each examination performed by a clinician results in a new definition, or examination record, which is added to the knowledge base. Apart from data from examinations, the knowledge base also contains additional definitional knowledge structures not related to any particular patient or examination.

An early decision in the MedView project was to build applied software, for use in the clinical setting, in parallel with the development of theory and implementation of tools for knowledge representation and reasoning. Another early decision was to use an industrial-strength object-oriented commercial development environment (NextStep) to build the applications aimed for use in the clinical setting. These decisions made it possible to implement a system that has been in use for several years for gathering data during examinations.

While the object-oriented development environment used provided excellent support for rapid application development, it did, of course, lack support for the definitional knowledge representation model used in MedView. Therefore, specialized classes were developed to handle exactly the needed definitional computations. In the long run it was obvious that another, more general, solution was needed for knowledge representation and reasoning.

A necessary requirement for a suitable knowledge representation tool for the MedView project was that it should be possible to integrate the tool in a seamless manner into the object-oriented application development environment used in the project. Another, that it should be as flexible as possible, since the definitional models used are still under development. The result of the efforts to produce such a tool is the Gisela framework for definitional computing [17]. In the present setting Gisela is best viewed as an object-oriented framework that makes it possible to represent the MedView knowledge base in a natural way and to build various knowledge structures and search methods on top of the basic database of examination records. With Gisela in place it is time to start looking into how it can be used to implement knowledge representation and reasoning in MedView in a coherent manner. In this paper we will discuss how this can be done. We will describe both things that have already been implemented and general ideas for future use of Gisela in the MedView project. In addition, some details about the implementation of the current MedView system will be given as well. However, it should be noted that the work on applying Gisela in the MedView project is still very much work in progress. Thus, what is given here is not a complete description of how to use Gisela in MedView but rather a collection of ideas and examples.

The rest of this paper is organized as follows. In Section 2 we discuss knowledge representation issues in MedView and how the knowledge base can be realized using Gisela. Section 3 concerns finding information in the knowledge base in different ways. In Section 4 the general architecture of applications in the MedView system is described. In Section 5 we give a more detailed overview of how Gisela is, or can be, used in a number of applications related to MedView. Section 6 concerns an application of functional logic programming methodology in Gisela and describes a program that can be used to generate summaries in natural language from examination records. Finally, in Section 7 we give some concluding remarks and discuss a number of problems encountered.

2 Knowledge Representation

MedView is based on a definitional formalization of medical data and knowledge. Entering data into the knowledge base is an act of creating definitions. Retrieving information involves computations using definitional structures. Using definitions to formulate both data and computations is intended to make the basic model conceptually simple and uniform. The definitions used have a precise interpretation, which make them suitable for automated reasoning in a computerized system. At the same time, they are simple enough to have an obvious intuitive reading needing no further explanation.

2.1 Preliminaries

We review the basics of the definitional model used here. For a more complete description see [17].

2.1.1 Definitions

In both MedView and Gisela the concept of a definition is given by:

1. two sets: the *domain* of D, written $\mathit{dom}(D)$, and the *co-domain* of D, written $\mathit{com}(D)$, where $\mathit{dom}(D) \subseteq \mathit{com}(D)$,
2. and a *definiens* operation: $D : \mathit{com}(D) \rightarrow \mathcal{P}(\mathit{com}(D))$.

It is natural to present a definition as a system of equations

$$D \left\{ \begin{array}{lcl} a_0 & = & A_0 \\ a_1 & = & A_1 \\ & \vdots & \\ a_n & = & A_n \end{array} \right. \qquad n \geq 0,$$

where *atoms*, $a_0, \ldots, a_n \in \mathit{dom}(D)$, are defined in terms of a number of *conditions*, $A_0, \ldots, A_n \in \mathit{com}(D)$. Note that the equation $a = A$ is just a notation for A being a member of $D(a)$.

2.1.2 Computation Methods

A *computation method* is a definition that contains procedural knowledge. A computation method describes how the declarative knowledge in definitions in the knowledge base should be used to perform computations. All computation methods presented in this paper will be of the form

$$\texttt{method} \quad m(D_1, \ldots, D_n). \quad n \geq 0$$

$$\begin{array}{lcccc} m & = & C_1 & \# & G_1 \\ & & \vdots & & \\ m & = & C_k & \# & G_k \end{array}$$

where m is the name of the computation method, $D_1, \ldots, D_n$, parameters representing the actual definitions used in computations, each C_i is a computation condition describing a number of operations to perform, and the G_is are guards restricting the applicability of equations.

2.1.3 Definitional Computations

A computation is a transformation of an initial *state* definition into a final *result* definition. To compute a query, a computation method is applied to an initial state definition. We will write the initial definition as a sequence of equations. The general form of a query is

$$m\{e_1, \ldots, e_n\}.$$

where m is a computation method and each e_i an equation. If the computation method applied cannot be used to transform the initial state definition into an acceptable result definition the computation fails. If the computation succeeds we take the result definition and any bindings of variables in the initial state definition as the answer to the query. Note that the computation method m provides the particular definitions describing declarative knowledge to use in the computation. Depending on the context the result of a computation can be interpreted in different ways. One commonly used method in this paper is to test if some definitions fulfill criteria given by the initial state definition by testing if an answer can be computed from the initial state definition using the given definitions.

2.2 Knowledge Base Structure

The structure of knowledge representation in MedView is based on the assumption that a natural basic concept to use is that of an "examination". Thus, each examination generates a unique definition, which is entered into the knowledge base. The general idea behind this approach is that a collection of examinations correspond closely to the gathered medical expertise of an experienced clinician. The mind of the clinician is, so to speak, thought of as being filled with a cloud of points representing distinct examinations. Of course, the expert knowledge of a clinician also consists of generalizations made from the clinicians total clinical experience. Consequently, definitions expressing other kinds of knowledge, not related to any particular examination record, can also be stored in the knowledge base. In addition, we make a distinction between declarative and procedural knowledge, where the procedural knowledge describes how to perform computations, or retrieve facts from the knowledge base. Both declarative and procedural knowledge is expressed in terms of definitions, as mentioned above.

The general structure of the knowledge base is pictured in Figure 1. It consists of the following:

- A collection of examination records, where each examination is represented by a definition.
- Additional definitions describing different kinds of general knowledge.

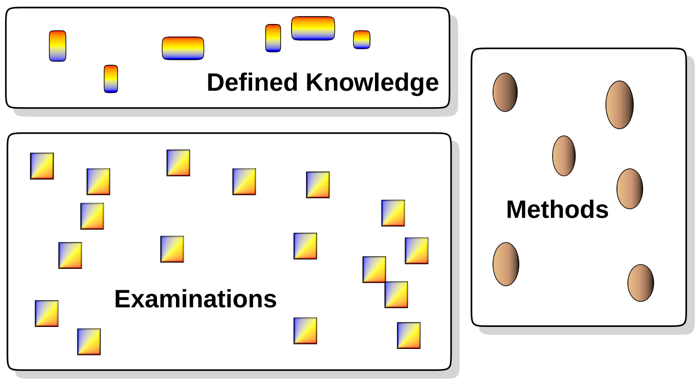


Figure 1: Schematic view of the MedView knowledge base. The knowledge base consists of a collection of examination records on top of which extra knowledge may be added. To perform computations, methods, shown to the right, are needed.

- Procedural knowledge represented by definitions in terms of computation methods.

Furthermore, the knowledge base contains a large number of digitized images taken at examinations. Each image is associated with a particular examination and can be retrieved by searching the collection of examination records.

2.2.1 Representing Basic Data

The formalization of basic data in MedView is centered around the concept "examination". Assembling information at examinations is modeled as defining a series of descriptive parameters, such as disease history (anamnesis), status, diagnosis, and so on. Thus, each examination generates a definition containing a number of equations describing the examination.

All examination definitions share a common structure, which so to speak defines the basic concept "examination". A small part of this structure is:

$$E \left\{ \begin{array}{l} \mathit{examination} = \mathit{anamnesis} \\ \mathit{examination} = \mathit{status} \\ \mathit{anamnesis} = \mathit{common} \\ \mathit{status} = \mathit{direct} \\ \mathit{common} = \mathit{drug} \\ \mathit{common} = \mathit{smoke} \\ \mathit{direct} = \mathit{mucos} \\ \mathit{direct} = \mathit{palpation} \\ \mathit{mucos} = \mathit{mucos\_site} \\ \mathit{mucos} = \mathit{mucos\_col} \\ \mathit{palpation} = \mathit{palp\_site} \end{array} \right. \tag{1}$$

As can be seen from the example, the general structure is hierarchical. An examination consists of *anamnesis*, *status*, etc. Each of these in turn consist of a number of parts, which consist of a number of parts, and so on until we reach the actual *attributes* for which values are collected at an examination. The attributes in (1) are, *drug*, *smoke*, *mucos_site*, *mucos_col*, and *palp_site*.

Important things to note are:

- All values are atomic.
- All examination records share the same structure.
- Not all attributes must be given values. A missing value simply indicates that we know nothing about it.
- The structure may be changed as long as it is *extended*, since then old records will remain valid, it is simply that they may have a larger number of attributes without values.

It is natural to view an examination record as being the sum of two definitions. One definition, E, describing the concept examination, and one definition, R, providing data collected at a particular patient visit. Thus, a complete examination is given by $E + R$.

For instance, a set of equations that together with (1) define a particular examination could be

$$R \begin{cases} drug = losec \\ drug = dermovat \\ smoke = no \\ mucos\_site = l122 \\ mucos\_col = white \end{cases}$$

Note that there is no value given for the attribute *palp_site* since nothing is known about it.

2.2.2 Additional Knowledge Structures

On top of the basic collection of examination records additional definitions may be created to represent different kinds of knowledge. While a large number of examinations have been collected in MedView, the work on additional knowledge structures is still in an early phase. In part, this depends on that, prior to the development of the Gisela framework, each kind of new definitional computation to be performed required the development of new specialized procedures for computing with definitions. Accordingly, trying out new ideas in practice was a cumbersome process that required a lot of work. With the Gisela framework different kinds of definitions and computation methods can be expressed easily, something we hope will speed-up the process of trying out new ideas on how to explore the MedView knowledge base.

A number of different additional knowledge structures have been tested however. We describe some of them here. The definitions are presented using the syntax for equational representations of definitions used in Gisela. Algorithms, that is, computation methods, for computing based on the presented structures in conjunction with examination definitions are given in Section 3.

Value Corrections In the MedView project new values for attributes may be freely added by any user. This is necessary since we cannot anticipate all possible values. Also, it is not possible to wait for approval when a new value is encountered, since data is entered during examinations. There are of course at least two major problems with this practice: (i) letting all users add new values might lead to confusion and a less harmonized terminology within the network of users involved, (ii) misspelled words may be introduced into the lists of valid values.

One solution to the problem is to monitor the values in the knowledge base regularly and add extra definitions that can be used to find replacements for incorrect values. These definitions can then be used by other applications to ensure that a harmonized terminology is used. It is natural to use one definition with corrections for each attribute in the general examination structure that has incorrect values. Which values are correct and which are not is decided by the network of clinicians working with MedView. Note that solving the problem by making changes in the examination definitions directly is not a viable solution for several reasons, one being the general rule stating that medical record information may not be changed.

As an example, when the values used for the attribute `'Chld-dis'` (Child Disease) were inspected, it was found that both "Mässlingen" and "Mässling" were used to denote the same disease (Measles). It was also found that in a number of examinations the disease "Röda Hund" (Rubella) was misspelled "Ruda Hund".

A Gisela definition that describes how to correct values for the attribute `'Chld-dis'` is the following:

```
definition 'Chld-dis':constant.

'Mässling' = 'Mässlingen'.
'Ruda hund' = 'Röda Hund'.
```

It was decided that "Mässlingen" was preferred over "Mässling". Therefore, the value `'Mässling'` is defined to be equal to the correct value `'Mässlingen'`. Note that only values that are regarded as incorrect are defined in this definition. For simplicity, the name of the definition is the same as the name of the attribute it gives corrections for. That a definition is declared as being `constant` means that all left-hand sides are constants and that the domain and co-domain of the definition are given by its equational presentation. An application that can be used to create value corrections, or localizing all values to different languages, is discussed in Section 5.5.1.

Value Classes As the number of examinations in the knowledge base grows, it becomes increasingly important to group related values into classes in a hierarchical manner. For example, diseases such as Herpes labialis, Herpetic gingivostomatis, Shingles etc., can be classified into viral diseases. Such classifications facilitate the detection of interesting patterns in the data. Value classes, or groups, are given as definitions and can be stored in the knowledge base for future use. The use of grouping, or aggregation, of values has proven invaluable in the MedView project to achieve better results with visualization tools such as The Cube [6, 8]. Examples of existing simple value classes are a division between smokers and non-smokers and between patients with oral lichen planus and patients that do not have oral lichen planus.

A Gisela definition that classifies smoking habits into three groups is:

```
definition smoke_3:constant.

'1 cigaretter utan filter/dag' = '< 10 cigarettes/day'.
'4 cigaretter utan filter/dag' = '< 10 cigarettes/day'.
'6 filtercigaretter/dag' = '< 10 cigarettes/day'.
'10 filtercigaretter/dag' = '> 10 cigarettes/day'.
'10-15 filtercigaretter/dag' = '> 10 cigarettes/day'.
'40 filtercigaretter/dag' = '> 10 cigarettes/day'.
'Nej' = 'non smoking'.
```

Of course, as the knowledge base grows so will the number of equations in the definition `smoke_3`. To further categorize smoking habits into smokers and non-smokers another definition can be used:

```
definition smoke_2:constant.

'< 10 cigarettes/day' = smoking.
'> 10 cigarettes/day' = smoking.
```

Note that in the definition `smoke_2` it is assumed that smoking habits have already been grouped using `smoke_3`. A complete value class definition is derived by adding together `smoke_2` and `smoke_3`, that is, conceptually $smoke = smoke\_2 + smoke\_3$.

More on computations using value classes and definitions of value classes is given in Section 3.2.

Search Patterns As mentioned, the collection of examination definitions is viewed as a space of clinical experience where each point represents a particular examination. A natural structure can be imposed on this space, based on classifications of the given points, e.g., the notion of a *patient*, different *diagnostic patterns* etc. Classifications like these can be given by defining a pattern where particular values for some subset of the total attributes are given. The examinations belonging to a certain class are the ones for which the actual values for attributes match those given in the pattern. As any other knowledge structure a pattern is expressed using a definition.

A very basic pattern is to classify the patient material with respect to each examination's personal identification code. For instance

```
definition 'G10029110':constant.

'P-code' = 'G10029110'.
```

identifies examinations with the personal identification code `'G10029110'`. A slightly more interesting pattern might be patients born in Sweden having a mucosal lesion colored red (Röd) or white (Vit):

```
definition sample_pattern:constant.

'Born' = 'Sverige'.
'Mucos-col' = 'Röd'.
'Mucos-col' = 'Vit'.
```

To find really interesting patterns it is necessary to provide means that let experts experiment with different patterns in a flexible way. A tool that, among other things, is intended to facilitate the process of finding interesting patterns in the MedView knowledge base is SimVis [6, 7]. Different ways to interpret search patterns computationally are given Section 3.1.

Combined Attributes In certain cases it might be desirable to view several attributes as one to find everything that can be derived from these attributes. If the attributes are found in the same branch of the general examination structure we can use this structure to find values by looking at a higher level. If we wish to combine two attributes that are not close to each other in the general examination structure this is not a good approach. The problem that occurs is that if attributes are far apart in different branches of the hierarchical examination structure, a very large part of this structure, containing irrelevant information, has to be searched.

A possible alternative is to introduce a new definition, which so to speak defines a new attribute in terms of existing ones.

For instance, in the general examination structure there are four different attributes `'Diag-tent'`, `'Diag-def'`, `'Diag-hist'`, and `'Diag-nr'`, which are used to define various aspects of the notion *diagnosis*. Now, if we want to find, say all images related to lichen planus in some way, it would be easier to just state the query “find examinations with diagnosis lichen planus” than specifying that all the attributes related to diagnosis should be examined. Therefore, we create a new definition `diagnosis` to bring together all the related attributes. Expressed as a Gisela definition we get:

```
definition diagnosis:constant.

diagnosis = ‘Diag-tent’.
diagnosis = ‘Diag-hist’.
diagnosis = ’Diag-def’.
diagnosis = ’Diag-nr’.
```

Now, instead of using all the various attributes to form a query, a simple query such as

```
m(exam){lichen_planus = diagnosis}.
```

can be used to examine if any of the diagnosis-attributes are related to lichen planus. Of course, a definition that groups different kinds of lichen planus is also needed, something like:

```
definition lichen_planus:constant.

’Atrofisk lichen planus’ = lichen_planus.
’Retikulär lichen planus’ = lichen_planus.
...
```

More on computations using combined attributes can be found in Section 3.2.3.

Miscellaneous The basic model of data used in MedView is very much one where all definitions are simple systems of equations with atoms both to the left and to the right in equations. In the early object-models used so far in the project this was also the only kind of definition supported by the computational machinery. With Gisela however, compound conditions and definitions being similar to ordinary logic or functional programs can be used as well. The introduction of these new possibilities means that it is time to reconsider the current model. This is an area for future work.

As an example of a more complex definition we show a definition that can be used to find values for an attribute with restrictions with respect to gender and age. The definition consists of a query predicate and some auxiliary definitions.

```
definition sample_query.

query(Attr, Gender, Min, Max, Date, Value) =
    db(’P-code’, PC),
    db('Datum', Date),
    db(Attr, Value),
    get_gender(PC, Gender),
    get_age(PC, Date, Age),
    between(Min, Max, Age).

get_gender(PC, Gender) =
    gender(PC) -> Gender.

get_age(PC, Date, Age) =
    age(PC, Date) -> Age.

between(Min, Max, Mean) =
    Min =< Mean,
    Mean =< Max.

db(Att, Val) =
    Val -> Att.
```

Note the resemblance with an ordinary logic program. Worth noting in this example is that `gender/1` and `age/2` are defined in another definition written using Gisela's Objective-C API. A possible query could be

```
m(gender_age, exam){true = query('Allergy', male, 40, 60, D, V}.
```

where `gender_age` defines `gender/1` and `age/2` and `exam` is an examination record. An example of how the definition `gender_age` can be implemented is given in Appendix B.

2.2.3 Computation Methods

The various classes of definitions discussed above define declarative knowledge useful in the MedView domain. To be able to perform computations using these definitions it is also necessary to define procedural knowledge that describes how to use the declarative knowledge to perform computations. Previously in the MedView system all procedural knowledge has been expressed using object-oriented programming. In principle, a specialized search algorithm, or inference machine, has been implemented for each kind of computation needed.

In Gisela procedural knowledge is expressed using computation methods. Computation methods are definitions just like everything else. Generally, there can be computation methods expressing common search strategies and computation methods specialized to handle a particular kind of query. We will present a number of different computation methods in Section 3. Since the Gisela framework provides all the basic machinery needed to perform definitional computations different kinds of procedural knowledge can be expressed with ease.

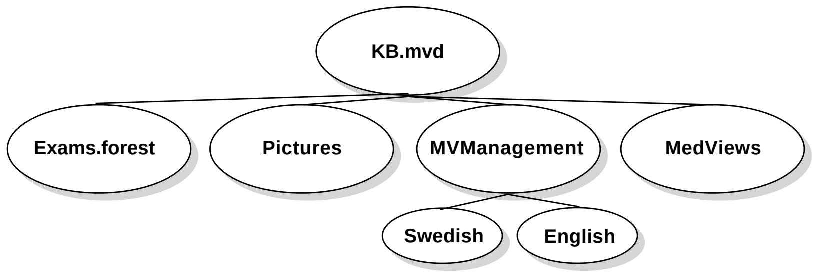


Figure 2: General file structure of current knowledge base.

Here we demonstrate the general nature of computation methods with an example method that can be used in conjunction with basic search patterns. The method is called `sli` (for some-left-identity) and takes one definition parameter `D`:

```
method sli:[D].

sli = [sli, l:D] # some l:in_dom(D) &  all not(identity).
sli = [] # some identity.
```

In the MedView setting, the computation method `sli` can be used to test if an examination record fulfills a simple search pattern. For instance, does the examination `exam` have the value `'G10039110'` for the attribute `'P-code'`:

```
sli(exam){'P-code' = 'G10039110'}.
```

The computation method replaces some left-hand side in the state definition according to the examination definition given for the parameter `D` until some equation in the state definition is an identity. If no equation in the state definition can be reduced and no equation is an identity the computation fails, which means that the examination definition does not match the given pattern.

2.3 Current Realization

The general organization of the knowledge base was shown in Figure 1. Currently, the file-system is used to implement the knowledge base as shown in Figure 2. A knowledge base is stored into a directory with the following main sub-directories:

- `Exams.forest` contains all the examination data. Each examination is stored into its own text-file representing a definition.
- `Pictures` contains all images related to the examinations in `Exams.forest`.
- `MedViews` contains RTF-files that are the actual forms completed during examinations. This part of the knowledge base is obsolete since the data stored in it can be retrieved from the information in `Exams.forest`.

- `MVManagement` contains correction definitions for all attributes that have corrections. It also contains some other files related to administrative tasks.

The MedView system permits the use of several different knowledge bases. Therefore, it is most practical not to store definitions such as value classes and computation methods directly into the knowledge base. The general organization of how data is stored and administered is subject to change.

For historical reasons examination definitions are stored in what is called a "tree-file". Part of such a file might be:

```
NPATIENTUPPGIFTER
NP-code
LM04119410##
NName#
NRef-in
LTandläkare##
NRef-cause
LSlemhinneförändring##
NBorn
LSverige##
NOccup
LSjukpensionär##
NCiv-stat
LGift##
NNote01##
```

Each tree-file encodes both the structure common to all examinations and the individual values for the present examination. A tree-file reflects the hierarchical tree-structure of examination definitions. Lines starting with an `N` are nodes in the tree and lines starting with an `L` are leaves, that is, values for some attribute registered at the examination.

2.3.1 Definitions as Objects

The conceptual model of the MedView knowledge base is that of a large collection of definitions of different kinds. These definitions are abstract objects as described in Section 2.1. So far, we have only shown equational representations of definitions. However, in the implementation of MedView applications it is often better to work with definitions as objects directly. The Gisela framework provides a number of different definition classes appropriate for various tasks. Among them, the class `DFTreeDefinition` and its subclass `DFLeafDefinition` can be used to read examination data stored in tree-files into objects suitable for definitional computations.

Since the structure is shared among all records, the Gisela representation of tree-files by default only contains attribute-value pairs. The standard syntactic representation as a Gisela definition of the above tree-file without structural information is:

```
definition M04119410_990604102247:constant.

'P-code' = 'M04119410'.
'Ref-in' = 'Tandläkare'.
'Ref-cause' = 'Slemhinneförändring'.
'Born' = 'Sverige'.
'Occup' = 'Sjukpensionär'.
'Civ-stat' = 'Gift'.
```

In the following, when we speak of definitions representing examination records we generally mean a definition like the one above containing only attribute-value pairs.

Definition objects can be created either by parsing syntactic representations or by programming directly in Objective-C using the Gisela API. Furthermore, Gisela is open for modification, which means that new definition classes can be created at will. From the view of definitional computations, new definitional classes are no different from the predefined ones, as long as they adhere to the restrictions set up by the framework.

2.3.2 The MVDatabase Class

Each definition used in the conceptual model of knowledge is represented as an object of some definition class in a Gisela realization. An object of the class `MVDatabase` is used to read examination data from disk and represent them in a manner suitable for definitional computations. The current realization of this class is the result of the need to represent examination records in various applications developed within MedView.

An `MVDatabase` is initialized by providing the path to the directory where the knowledge base is stored. The database object creates one Gisela definition object of the class `DFLeafDefinition` for each examination record in the knowledge base. It also does some basic indexing of the examinations needed to present the contents and iterate over examinations. While these could be defined in terms of Gisela computations it is a more pragmatic approach to include this functionality in the database class. Some of the more important methods provided by an `MVDatabase` object are:

- `- (NSArray *)allExaminations`, returns an array with all definition objects representing the examinations in the knowledge base.
- `- (unsigned int)personCount`, returns the number of distinct patients currently in the knowledge base.

- `- (NSString *)pCodeAtIndex:(unsigned)index`, returns the value for the attribute giving the patient identification code of the examination at `index`. The index must be less than the number of different persons in the knowledge base.

- `- (NSArray *)examinationsForPCode:(NSString *)aPCode`, returns all examination definitions sharing the value `aPcode` for the patient identification code attribute.

- `- (NSArray *)allAttributes`, returns an array with all the attributes, that is, all leaves of the general examination structure.

- `- (NSArray *)formalizedAttributes`, returns an array with all attributes that have formalized values. This essentially excludes free text values, document links, and image names.

- `- (NSArray *)topLevelAttributes`, returns an array with all attributes at the top level of the general examination structure. This can be useful for unfolding the examination structure.

- `- (DFDefinition *)correctionsForAttribute:(NSString *)attrib`, returns the correction definition for the attribute `attr`, or `nil` if the attribute has no correction definition.

The main use of an `MVDatabase` object is to read examinations in a knowledge base, use `allExaminations` to iterate over the examinations in the knowledge base in definitional search queries (Section 3), and to use some of the other methods to be able to present GUI, for instance, a list of all persons or attributes.

2.4 Discussion

The general model of the knowledge base as a large collection of definitions is a very flexible one and makes it easy to add knowledge structures and to combine existing definitions in different ways. Compared to the MedView system in clinical use, the introduction of Gisela as a representation framework for definitions facilitates the implementation of the basic knowledge model and makes it much easier to express additional structures and procedural knowledge in a coherent way.

An open issue in the current implementation is how large the knowledge base can become before Gisela gets too slow. At some point we will have to start looking into new definition classes, more tuned towards database management. One solution could be to implement definition classes which act as covers for high-performance relational database systems, that is, as a definitional database adaptor, thus giving the relational database a definitional programming interface.

A related problem is that the realization with one definition for each examination record is inherently inefficient in many ways. To find records fulfilling some given criteria it is necessary to perform a linear search among all the records. This problem is not easily solved by using another media for permanent storage. Instead it will be necessary to revise the general model for knowledge representation. This is an area for future work though.

3 Structured Search

In the previous section we presented the basic organization of knowledge and data used in MedView. Here we build on this material and present how various kinds of definitions can be used to retrieve information and search the knowledge base. Generally, this means to program Gisela to search for definitions fulfilling some given criteria. Of course, this might be done in many different ways, some examples are given here.

3.1 Using Search Patterns

In Section 2.2.2 we discussed how to classify the examination material by defining various search patterns. Schematically, a search pattern is a number of equations specifying the specific criteria that identifies the group of examinations defined by the pattern:

$$\begin{array}{rcl} \mathit{Attribute}_1 & = & Value_1 \\ & \vdots & \\ \mathit{Attribute}_n & = & Value_n \end{array}$$

In the sequel we call a definition like this a *pattern definition*. The intuitive understanding of a pattern definition is that it identifies a number of examination records where the value given for $\mathit{Attribute}_i$ is equal to $Value_i$. Note that it is not required that $\mathit{Attribute}_i \neq \mathit{Attribute}_j$ for $i \neq j$, nor is it required that $Value_i \neq Value_j$. While a set of equations is a natural conceptual view of a pattern definition, it is not clear if *all* or *some* equations must be fulfilled or whether $Value_i$ should be the *only* value given for $\mathit{Attribute}_i$. In addition, using a pattern definition as is, as the initial state definition in a query, is not likely to correspond to the most intuitive understanding of the pattern.

Instead, we will use the pattern definition as the basis to form a *query definition* that is used to pose queries to the system. To make this clear, we give a description in definitional terms of different ways that a pattern definition can be used to identify a certain class of examinations.

As mentioned above, the set of equations given in a pattern definition leaves us with two choices that must be handled:

1. Any or all equations must be fulfilled,

2. For any attribute, either some or exactly the values given are required.

Obviously, this gives us four combinations, which we will call `all-all`, `all-some`, `some-all`, and `all-all`. The equations of a pattern definition will be used to build a customized query definition through which we can pose all the four possible queries represented by the equations.

As our running example we will use the following pattern definition that was also given in Section 2.2.2:

```
definition pattern_sample:constant.
'Born' = 'Sverige'.
'Mucos-colr' = 'Röd'.
'Mucos-colr' = 'Vit'.
```

To build the query definition we first group together equations which have the same left-hand side. Thus, if the pattern definition contains the equations

$$\begin{array}{rcl} a & = & v_1 \\ & \vdots & \\ a & = & v_n \end{array}$$

we get the new equations

$$\begin{array}{rcl} a_{vals} & = & v_1 \\ & \vdots & \\ a_{vals} & = & v_n \end{array}$$

in the query definition. We then replace each set of equations $a = v_1, \ldots, a = v_n$ in the pattern definition with a single equation $a = a_{vals}$. In our example we then get

```
'Born' = 'Sverige'.
'Mucos-colr' = mucos_vals.
```

and add the equations below to the query definition:

```
mucos_vals = 'Röd'.
mucos_vals = 'Vit'.
```

When the equations in the pattern definition have been grouped together, we create the conditions $a_i \rightarrow w_i$ and $w_i \rightarrow a_i$ for each equation $a_i = w_i$ in the modified set of equations. Finally, we put different combinations of these conditions into the query definition to form the basis for the four kinds of queries describe above, thus:

$$\begin{array}{lcl}
q(\texttt{all}-\texttt{all}) & = & a_1 \rightarrow w_1, \ldots, a_m \rightarrow w_m. \\
q(\texttt{all}-\texttt{some}) & = & w_1 \rightarrow a_1, \ldots, w_m \rightarrow a_m. \\
q(\texttt{some}-\texttt{all}) & = & a_1 \rightarrow w_1. \\
 & \vdots & \\
q(\texttt{some}-\texttt{all}) & = & a_m \rightarrow w_m. \\
q(\texttt{some}-\texttt{some}) & = & v_1 \rightarrow a_1. \\
 & \vdots & \\
q(\texttt{some}-\texttt{some}) & = & v_n \rightarrow a_n.
\end{array}$$

Note that for `some-some` queries the original equations from the pattern definition are used to form a number of conditions.

The complete query definition for our example becomes:

```
definition qdef:matching.

q(all-all) = 'Born' -> 'Sverige', 'Mucos-colr' -> mucos_vals.
q(all-some) = 'Sverige' -> 'Born', mucos_vals -> 'Mucos-colr'.
q(some-all) = 'Born' -> 'Sverige'.
q(some-all) = 'Mucos-colr' -> mucos_vals.
q(some-some) = 'Sverige' -> 'Born'.
q(some-some) = 'Röd' -> 'Mucos-colr'.
q(some-some) = 'Vit' -> 'Mucos-colr'.

mucos_vals = 'Röd'.
mucos_vals = 'Vit'.
```

That a definition is declared as `matching` means the definiens operation is only implemented for ground terms. The domain of consists of all terms with the same principal functor as some term occurring as a left-hand side in an equation.

To look for examinations matching search criteria defined in the manner shown above, a computation method with two parameters, the query definition, Q, and an examination record, E, is used:

$$qdm(Q, E)\{\texttt{true} = q(Pattern)\}.$$

The method *qdm* expands the initial state definition according to the definitions Q and E and the built-in computation rules for constructed conditions (such as the arrow). The method is defined in such a way that a computation succeeds if all equations can be reduced to the identity equation. A possible definition of the computation method *qdm* is:

```
method qdm:[QDef,Exam].

// Stop when identity.
```

```
qdm = [] # some identity.

// Exam definition left, replace and continue.
qdm = [qdm, l:Exam] # some l:in_dom(Exam) & all not(identity).

// Exam definition right, replace and continue.
qdm = [qdm, r:Exam] # some r:in_dom(Exam) & all not(identity).

// Constructed conditions, expand and continue.
qdm = [qdm, r:QDef] # some r:matches((A,B));r:matches((C->D)).

// QDef definition left. Will only be used for grouped values.
qdm = [qdm, l:QDef] # some l:in_dom(QDef) & all not(identity).

// QDef definition right. Computation will start here..
qdm = [qdm, r:QDef] # some r:in_dom(QDef) & all not(identity).
```

The nature of the intended queries and the constructed query definition is such that the state definition only contains a single equation. Searching for records fulfilling the query is performed by iterating over all examination definitions in the knowledge base and collecting the ones for which the query succeeds.

The most important difference between the different query types is whether the attribute A occurs to the left or right of the arrow in the query definition. If it occurs to the left the reading is “all values of A must be identical to the given value”, if it occurs to the right the reading is “some value of A must be identical the given value”.

To illustrate, we show the expansion of the query

```
qdm(qdef,exam){true = q(all-some)}.
```

where `qdef` is our example query definition and `exam` an examination record containing the equations:

```
'Born' = 'Sverige'.
'Mucos-colr' = 'Röd'.
'Mucos-colr' = 'Vit'.
'Mucos-colr' = 'Gul'.
```

To save space some names have been abbreviated.

$$\dfrac{\dfrac{\dfrac{\dfrac{\{sv = sv\}}{\{sv = born\}}}{\{\texttt{true} = sv \to born\}} \qquad \dfrac{\dfrac{\dfrac{\{rd = rd\}}{\{rd = mucolr\}} \quad \dfrac{\{vt = vt\}}{\{vt = mucolr\}}}{\{muvals = mucolr\}}}{\{\texttt{true} = muvals \to mucolr\}}}{\{\texttt{true} = sv \to born, muvals \to mucolr\}}}{\{\texttt{true} = q(\texttt{all} - \texttt{some})\}}$$

The technique described above can be summarized as follows: Using a number of equations is an intuitive way to define a pattern. To get meaningful queries from a number of equations a query definition is created describing the possible queries. We then use the possibility in Gisela to combine several definitions to use the query definition in conjunction with an examination record to pose a query. How computations should be performed on the basis of these two definitions is described in a specialized computation method.

3.2 Using Value Classes

In Section 2.2.2 value classes and various other kinds of definitions for combining values or attributes were discussed. Here we will look closer into how such definitions can be used in computations. We will only study queries where the initial state definition contains a single equation $V = A$. V is the value we are looking for and A is some kind of attribute, not necessarily part of the basic examination structure. An examination record fulfills the demands of the query if the single equation in the initial state definition can be reduced to identity.

3.2.1 Using Basic Value Class Definitions

In its most basic form a value class definition is simply a number of equations that collects together different values into groups:

$$\begin{array}{rcl} a & = & v_1 \\ b & = & v_1 \\ c & = & v_2 \\ d & = & c_2 \\ & \vdots & \end{array}$$

As discussed in Section 2.2.2, we can have any number of such value class definitions and they can be combined in different ways to form more complex groups.

To test if an examination definition fulfills a simple equation $V = A$ we can use the computation method

```
method sri:[Exam].

sri = [sri, r:Exam] # some r:in_dom(Exam) &  all not(identity).
sri = [] # some identity.
```

in a query. For instance

```
sri(exam){'Sverige' = 'Born'}
```

will succeed if the examination `exam` contains the value `'Sverige'` for the attribute `'Born'`. To use value classes we instead use a computation method with two parameters: the examination record and the definition of value classes to use. It is natural to think of this definition as a filter that maps values into groups. A possible method definition is:

```
method srfi:[Exam,Filter].

srfi = [] # some identity.
srfi = [srfi, r:Exam] # some r:in_dom(Exam) &
                          all not(identity).
srfi = [srfi, r:Filter] # some r:in_dom(Filter) &
                          all not(identity) &
                          all not(r:in_dom(Exam)).
```

The meaning of the equations in `srfi` is: (i) if there is an equation with identical left- and right-hand sides, the computation is finished, (ii) if some attribute can be reduced using `Exam`, reduce it and continue, and (iii) if a value can be grouped using `Filter`, do that and continue.

Now, if we have a definition classifying countries into geographical regions

```
definition geo_regs.

'Sverige' = western_europe.
'Norge' = western_europe.
'Tyskland' = western_europe.
...
'Polen' = eastern_europe.
...
```

we can run a query such as

```
m(geo_regs,exam){western_europe = 'Born'}
```

to check if a patient is born in western Europe. Computationally, `'Born'` is first replaced with the country where the patient is born, which can be found in the definition `exam`, and then this country is replaced by the geographical region as given by the definition `geo_reg`.

Grouped values can be combined into value classes, which can be combined into new value classes etc. For example:

```
definition continents.
western_europe = europe.
eastern_europe = europe.
.....
```

The definitions `geo_regs` and `continents` can be used with a computation method with three parameters in queries such as

```
m(geo_regs, continents, exam){europe = 'Born'}.
```

to check if a patient is born in Europe. Alternatively, several value class definitions can be added together to form more complex groups.

Another approach is to view the value class definition as an extension to the examination record. In this case, checking if a patient is born in Europe can be done with the query:

```
sri(exam+geo_regs+continents){europe = 'Born'}.
```

That is, the definitions `exam`, `geo_regs`, and `continents` are added together to a new definition that is used in the computation.

3.2.2 Using Custom Value Class Definitions

A problem with using basic value class definitions is that all values must be listed to create a group. For some kind of values, typically numerical, this is not feasible. To solve this problem somewhat more complex value class definitions are needed. For instance, to group values for the attribute `'Vas-life'`, which can be any number between `0.0` and `10.0`, we could use:

```
definition vas_groups.
X = vas_group(X).

definition vas_group:matching.

vas_group(X) = 0.0 =< X, X < 3.0, bad.
vas_group(X) = 3.0 =< X, X < 6.0, ok.
vas_group(X) = 6.0 =< X, X =< 10.0, nice.
```

The definition `vas_groups` needs some explanation. It cannot be written textually as shown in Gisela, since a variable is not allowed as the right-hand side of an equation in any of the definition classes the framework provides. However, since Gisela is a framework open for extensions, and the only requirements on definitions are that they implement certain operations, a new kind of definition with the desired properties can be created.

In this case, what we want is a definition whose domain is all numbers in the proper range and where `D(N) = {vas_group(N)}` for any element in the domain. Building on the classes of the framework this can be accomplished with a few lines of Objective-C code. Once this is done we can run queries like

```
vgroup(vas_groups, vas_group, exam){ok = 'Vas-life'}.
```

The definition `vas_groups` can be avoided by writing a different query, but that is another story. The interesting thing here is that when the definition classes provided by the Gisela framework are not sufficient, the framework can easily be extended with new definition classes that implement the desired behavior.

The computation method `vgroup` could be implemented as follows:

```
method vg.
import_definition(g3arithmetics).

vg = [] # some r:matches(true).
vg = [] # some identity.
vg = [vg, r:g3arithmetics] # some r:matches((A,B)).
vg = [vg, r:g3arithmetics] # some r:in_dom(g3arithmetics).

method vgroup:[D1, D2, E].

vgroup = [vg, r:D2, r:D1, r:E] # some r:in_dom(E).
```

The computation method `vgroup` checks that the right-hand side of the equation is in the domain of the definition `E` (the examination record). If that is the case, the right-hand side is replaced using the definitions `D1` and `D2` and then the auxiliary method `vg` is used to handle the rest of the computation.

Of course, having one specialized computation method for each kind of query might be considered somewhat cumbersome. Below we show a more general method, `group`, that could be used for all queries shown so far in this section. For the method to work correctly the initial state definition used in the query must contain exactly one equation.

```
method group:[A1,A2,E].
import_definition(g3arithmetics).

group = [] # some r:matches(true).
group = [] # some identity.
group = [group, r:g3arithmetics] # some r:matches((_,_)).
group = [group, r:g3arithmetics] # some r:in_dom(g3arithmetics).
group = [group, r:E] # some r:in_dom(E) &
                      all not(identity).
group = [group, r:A1] # some r:in_dom(A1) &
                       all not(identity).
group = [group, r:A2] # some r:in_dom(A1) &
                       all not(identity).
```

To use the method `group` to check if a patient is born in Europe we use the query:

```
group(continents, geo_regs, exam){europe = 'Born'}.
```

To check if a patient's value for `'Vas-life'` is `ok` the query to use becomes:

```
group(vas_group, vas_groups, exam){ok = 'Vas-life'}.
```

3.2.3 Combined Attributes

In Section 2.2.2 we showed an example where several different attributes related to diagnosis were grouped into one. This combination of attributes used in conjunction with a value class as proposed in Section 2.2.2 is from a computational point of view the same as the use of two value class definitions discussed in the previous section. Therefore, the computation method `group` can be used in a query like:

```
group(diagnosis,lichen_planus,exam){lichen_planus = diagnosis}.
```

If efficiency is important, and we know that all relevant values in the knowledge base for lichen planus are found in the definition `lichen_planus`, it is possible to hard-wire the exact sequence of operations that are performed in a successful computation into a new computation method:

```
method combine:[A1, E, A2]

combine = [E, r:A2, r:E, r:A1];
```

The proper query to check if the examination `exam` is connected to lichen planus then becomes:

```
combine(diagnosis,lichen_planus,exam){lichen_planus = diagnosis}.
```

3.3 Further Examples

In the current MedView knowledge base *all* values are atomic. Whether this is a flaw or a feature is not something we will go into here. Instead, we look into ways of finding the implicit subparts of atomic values.

Values for the attribute `'P-code'` encode in an atom, unique for each patient, the identity of the clinician who performed the examination, when the patient was born, and whether the patient is male or female. To find, for instance, all female patients we have to take this atom apart. In Section 2.2.2 we showed a definition written in a logic programming style that could be used to make selections based on age and gender. Furthermore it used a definition class written in Objective-C to inspect the implicit components of a personal code.

Here we show an alternative solution which does not use Objective-C and is more in-line with the general style of most examples. What we do is to introduce a new defined concept `gender` which will enable queries like

```
m(gender,exam){female = gender}.
```

A possible definition is shown below. The definition makes use of a number of built-in functions for converting between atoms and strings:

```
definition gender.

gender = PC -> 'P-code', get_gender(PC, G), G.

get_gender(PC, G) =
    constant_string(PC) -> PS,
    explode_string(PS) -> [_,_,_,_,_,_,_,_,C],
    string_constant(C) -> CC,
    gender_code(CC, G).

gender_code(0, female).
gender_code(1, male).
```

We show a partial expansion of a query using this definition:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathtt{PC}=\mathtt{a19231}}{\{\mathtt{PC}=\mathtt{a19231},\mathtt{fm}=\mathtt{a19231}\}}}{\{\mathtt{PC}=\mathtt{pcode},\mathtt{fm}=\mathtt{pcode}\}}}{\{\mathtt{fm}=\mathtt{PC}\to\mathtt{pcode}\}} \quad \dfrac{\dfrac{\begin{array}{c}\underline{\mathtt{G}=\mathtt{fm}}\\ \vdots\end{array}}{\{\mathtt{fm}=\mathtt{get\_gender}(\mathtt{a19231},\mathtt{G})\}} \quad \{\mathtt{fm}=\mathtt{fm}\}}{\{\mathtt{fm}=\mathtt{get\_gender}(\mathtt{PC},\mathtt{G}),\mathtt{G}\}}}{\{\mathtt{fm}=\mathtt{PC}\to\mathtt{pcode},\mathtt{get\_gender}(\mathtt{PC},\mathtt{G}),\mathtt{G}\}}}{\{\mathtt{fm}=\mathtt{gender}\}}$$

Computing the age of a patient can be done in a similar way but will use the *Date* attribute of an examination also.

3.4 Discussion

A thorough investigation of various search methods compared both to relational databases and to deductive databases, e.g., Datalog, is something which must be done in the future. At the same time we expect to (re-)organize the knowledge base structure and methodology for building new concepts on top of basic data. With Gisela we have the tools needed to carry out such an investigation.

4 System Architecture

The initial version of the MedView system was implemented in Objective-C using the NextStep application program interface (API). The current system is also written in Objective-C, but most NextStep applications have been ported to use the OpenStep [13] APIs instead. OpenStep is a more platform independent successor to NextStep.

The implementation is divided into a number of applications for specific tasks and a number of frameworks containing things that are commonly used by several applications. The main part of these are the ones that implement Gisela. The current frameworks and their tasks are:

- `DFDefinitions` implements definitions,
- `DFMethods` implements computation methods,
- `DFComputing` implements definitional computations,
- `MVDatabase` implements classes to manage MedView knowledge bases,
- `MVAppKit` implements extensions to OpenStep's AppKit used to build applications. This includes things such as common user interface elements etc.

All classes defined in these frameworks can be reused to build different kinds of MedView software, e.g., command line tools, desktop applications, web applications etc.

4.1 Using Gisela in Applications

When Gisela is used for definitional computations in Objective-C programs the interface to the definitional machinery is an object of the `DFDMachine` class. This object performs definitional computations by providing answers to queries posed by some other object. A `DFDMachine` requires an observer to handle certain choices in computations. If no observer object is given, a default observer is used. If the definitional computation should be run in a separate thread, a `DFDMachine` object sends its answers to a delegate object. The delegate is also responsible for ending the computation etc. The API for using a `DFDMachine` is very simple. In the most basic case it consists of the methods:

```
// Create a machine that uses the default observer.
- (id)initWithDelegate:(id)anObject;

// Set the query to evaluate.
- (void)setQuery:(DFQuery *)aQuery;

// Returns the next answer if there is one or nil otherwise.
// Use this as an enumerator to get all answers.
- (DFAnswer *)nextAnswer;

// Returns an  array with all the possible
// answers for the current query.
- (NSArray *)findAllAnswers;
```

If the machine should run its computations in a separate thread, the API becomes somewhat more complicated.

4.2 Model-View-Controller

The general design approach used in most applications is to use the Model-View-Controller (MVC) paradigm. MVC is a commonly used object-oriented software development methodology. When MVC is used, the *model*, that is, data and operations on data, is kept strictly separated from the *view* displayed to the user. The *controller* connects the two together and decides how to handle user actions and how data obtained from the model should be presented in the view. Applied to the MedView and Gisela setting, the model of what an application should do is implemented using definitional programming in Gisela. The view displayed to the user can be of different kinds, desktop applications, web applications etc. In between the view presented to the user and the Gisela machinery there is a controller object which manages communication between the two parts. One advantage of this approach is, of course, that different views may be used without changing the model.

4.3 Application Architecture

An application developed using the OpenStep API is really a directory `Name.app`, which contains the application binary and a resources folder containing all sorts of resources needed by the application. The items in the resources folder can be easily loaded into the application at run time using methods provided by the API. Applied to the use of Gisela in MedView this means that it is straightforward to have text-files in the resources folder representing definitions and computation methods. These text-files can be parsed into definition objects at run time using the API provided by the Gisela framework. Consequently, any Gisela program developed using equational presentations can smoothly be integrated into an Objective-C application.

To make things a little bit more precise we will show an example model that provides data suitable for drawing simple visualizations of the MedView knowledge base. We also demonstrate how definitional resources are loaded into applications.

4.3.1 An Example Model Class

A basic visualization of the knowledge base is to display ordinary bar charts and scatter-plots. An application based on Gisela that does this is discussed in Section 5.4.1. Part of the general organization of this application is shown in Figure 3. At the center is a `Document` object, which manages an on-screen window displaying graphs and all resources needed to perform computations. In Figure 3 the view is represented by the object at the top-left. All definitional computations are embedded into a model consisting of an object of the `DataProvider` class. An object of this class creates a `DFDMachine` to perform computations and fetches data from an `MVDatabase`. The `MVDatabase` in turn is provided by an

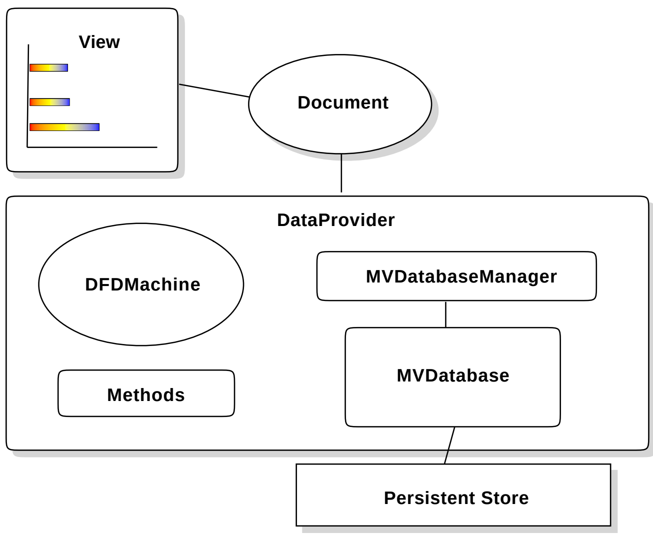


Figure 3: An example data model.

`MVDatabaseManager`, which is responsible for things like loading databases and sharing them among objects.

To access data, that is, to perform definitional computations, the following methods are provided by a `DataProvider` object:

```
// Collect, group and count all values for attr
- (NSCountedSet *)barChartValuesForAttribute:(NSString *)attr;

// Collect, group and count all values for attr.
- (NSCountedSet *)barChartValuesForAttribute:(NSString *)attr
                               withGroups:(NSDictionary *)groupDefs;

// Provide data for a scatter-plot
- (NSDictionary *)scatterValuesForAttribute:(NSString *)attr;

// Provide data for a scatter-plot using groups
- (NSDictionary *)scatterValuesForAttribute:(NSString *)attr
                              withGroups:(NSDictionary *)groupDefs;
```

To be able to draw a bar chart we need to find all values for a certain attribute, and count them. A suitable data structure for this is a counted set, also known as a bag or a multi-set. Note that methods that use value classes (groups) are provided.

When a new `DataProvider` object is created it must be initialized with the path to the knowledge base to use. It then creates a `DFDMachine` object and loads the required computation methods:

```
-(id)initWithPath:(NSString *)dbPath {
  if (self = [super init]) {
      database = [MVDatabaseManager databaseAtPath:dbPath];
      dMachine = [[DFDMachine alloc] initWithDelegate:self];
      [self loadComputationMethods];
  }
  return self;
}
```

More details on how a `DataProvider` object performs its definitional computations are given in Section 5.4.1.

All details about how to compute data to draw graphs is encapsulated in the model. Therefore, it can be reused without any modifications, with another controller and view, to write, for instance, a web application displaying graphs in a web-browser.

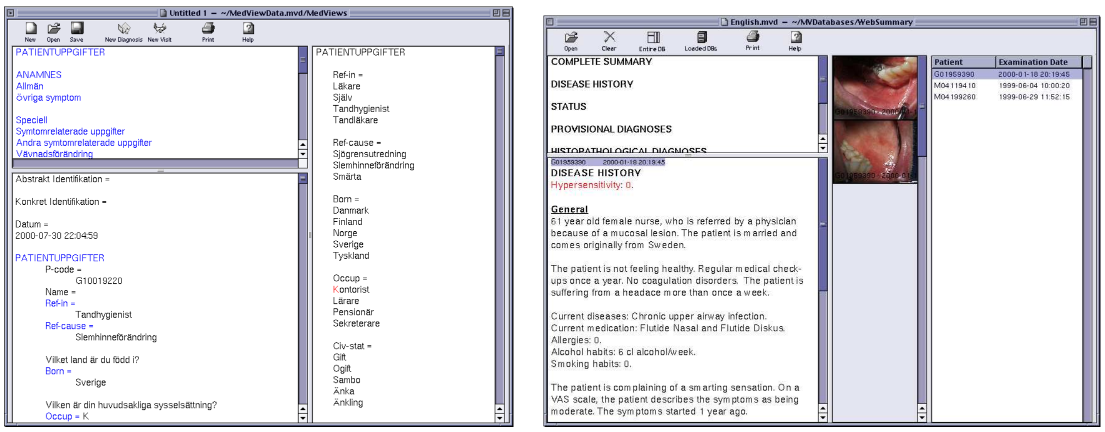
4.3.2 Loading Definitional Resources

OpenStep provides an API to read files stored in an application's resources folder at run time. The Gisela framework in turn, provides an API to parse equational representations of definitions and computation methods into the corresponding object representations. Thus, very little code is required to load definitional resources for use in applications. The following code is enough to read some method definitions stored in the file `methods.dxt` in an application's resources folder:

```
- (void)loadComputationMethods {
    NSString *tmpString;
    DFMethodParser parser = [[DFMethodParser alloc] init];

    // Get full path to methods.dxt
    tmpString = [[NSBundle mainBundle]
                             pathForResource:@"methods"
                                      ofType:@"mxt"];
    // Read methods.dxt
    tmpString = [NSString stringWithContentsOfFile:tmpString];
    // Parse methods.dxt into an array of method objects.
    methods = [parser methodsFromString:tmpString];
    // Dispose parser object.
    [parser release];
}
```

What the example illustrates is the relative ease with which definitional resources can be loaded for use in applications. Once loaded, definitional objects can be used like any other object in programs.



PATIENTUPPGIFTER

ANAMNES
Allmän
övriga symptom

Speciell
Symtomrelaterade uppgifter
Andra symtomrelaterade uppgifter
Vävnadsförändring

Abstrakt Identifikation =

Konkret Identifikation =

Datum =
2000-07-30 22:04:59

PATIENTUPPGIFTER
P-code =
G10019220
Name =
Ref-in =
Tandhygienist
Ref-cause =
Slemhinneförändring

Vilket land är du född i?
Born =
Sverige

Vilken är din huvudsakliga sysselsättning?
Occup = K

PATIENTUPPGIFTER

Ref-in =
Läkare
Själv
Tandhygienist
Tandläkare

Ref-cause =
Sjögrensutredning
Slemhinneförändring
Smärta

Born =
Danmark
Finland
Norge
Sverige
Tyskland

Occup =
Kontorist
Lärare
Pensionär
Sekreterare

Civ-stat =
Gift
Ogift
Sambo
Änka
Änkling

COMPLETE SUMMARY

DISEASE HISTORY

STATUS

PROVISIONAL DIAGNOSES

HISTOPATHOLOGICAL DIAGNOSES

G01959390 2000-01-18 20:19:45
DISEASE HISTORY
Hypersensitivity: 0.

General
61 year old female nurse, who is referred by a physician because of a mucosal lesion. The patient is married and comes originally from Sweden.

The patient is not feeling healthy. Regular medical check-ups once a year. No coagulation disorders. The patient is suffering from a headace more than once a week.

Current diseases: Chronic upper airway infection.
Current medication: Flutide Nasal and Flutide Diskus.
Allergies: 0.
Alcohol habits: 6 cl alcohol/week.
Smoking habits: 0.

The patient is complaining of a smarting sensation. On a VAS scale, the patient describes the symptoms as being moderate. The symptoms started 1 year ago.

Patient	Examination Date
G01959390	2000-01-18 20:19:45
M04119410	1999-06-04 10:00:20
M04199260	1999-06-29 11:52:15

Figure 4: Left: MedRecords main window. Right: MedSummary main window.

4.4 Discussion

To use a state-of-the-art object-oriented programming environment was an early decision in MedView. The choice was obvious since object-oriented programming provides the natural setting today for the development of a system heavily dependent on advanced graphical user interfaces and the obvious need for interfaces to various standards for data and knowledge bases etc.

On the other hand, knowledge representation and reasoning is a key part of the project, an area where declarative languages are the natural choice. Therefore, an integration of the two paradigms was deemed to be the best approach. While the necessary declarative programming tools were being developed temporary solutions were used to implement definitional computations. By now, the efforts are beginning to pay off. A number of useful applications have been developed and the declarative system is complete enough to start replacing older versions.

The division of code into frameworks simplifies the reuse of components in different kinds of applications. Of course, this is standard software engineering technique but it is anyway a good thing to notice how much less effort is needed to build new software today, as compared to the early days of the project.

5 Applications

In this section we discuss how existing MedView applications would have to be modified to use the Gisela framework. We also give brief presentations of some tools that have been developed using Gisela but that have not yet been used in the clinical setting.

5.1 MedRecords

MedRecords (MR) is the application used to enter data at examinations (Figure 4, left picture). In definitional terms, MR is best thought of as displaying an incomplete examination definition. The task of entering examination data is thus to complete this examination definition. The result of each examination is a new tree-file that is stored into the knowledge base. The current version of MR does not use the Gisela framework in any way. Possible future plans for including Gisela features are:

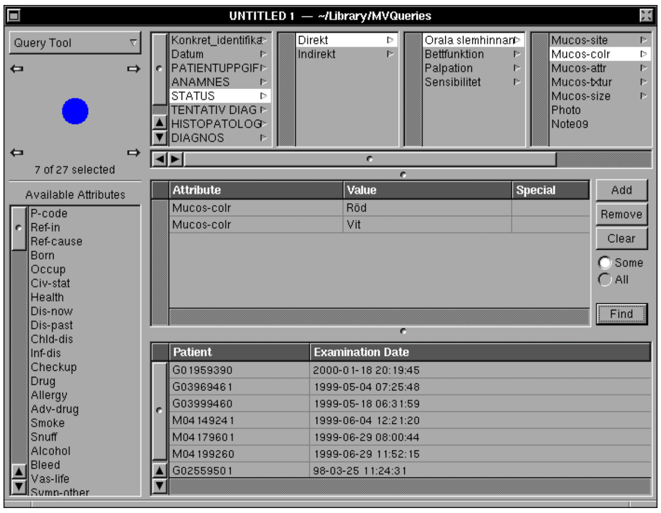
- To change the storage format used to native Gisela syntax instead of tree-files.
- To store the result from an examination as a binary definition object. This requires no special coding since all definition objects provided by the Gisela framework can be archived (serialized).
- To use Gisela to build an intelligent agent that can help the user. The idea behind such an agent is that it should monitor the contents of the knowledge base and the user's actions and come up with suggestions, warnings etc. Since the entire knowledge representation theory is based on definitional terminology it should be easier to code such an agent in Gisela than in Objective-C.

5.2 MedSummary

In the MedView project an application called MedSummary (Figure 4, right picture) is used to generate natural language summaries of patient data from the formalized examination definitions in the knowledge base [1]. The code in use to generate summaries is based on an early version of the definition classes found in the Gisela framework. However, the definitions are used only as a kind of high-level data containers in what is otherwise an Objective-C program. More specifically, all examination data and the various parts of the text-templates used are represented using definition objects, but all communication between these objects, and also the top-level text-generation machinery, is written in Objective-C.

There are at least two ways that the Gisela framework could be used to improve MedSummary:

- Replace the definition classes used with the Gisela framework. This reduces the amount of code to maintain and will be done in a near future.
- Use the Gisela framework to implement a better text-generator based on declarative programming methods. A prototype for such a text-generator is discussed in Section 6.



Attribute	Value	Special
Mucos-colr	Röd	
Mucos-colr	Vit	

Patient	Examination Date
G01959390	2000-01-18 20:19:45
G03969461	1999-05-04 07:25:48
G03999460	1999-05-18 06:31:59
M04149241	1999-06-04 12:21:20
M04179601	1999-06-29 08:00:44
M04199260	1999-06-29 11:52:15
G02559501	98-03-25 11:24:31

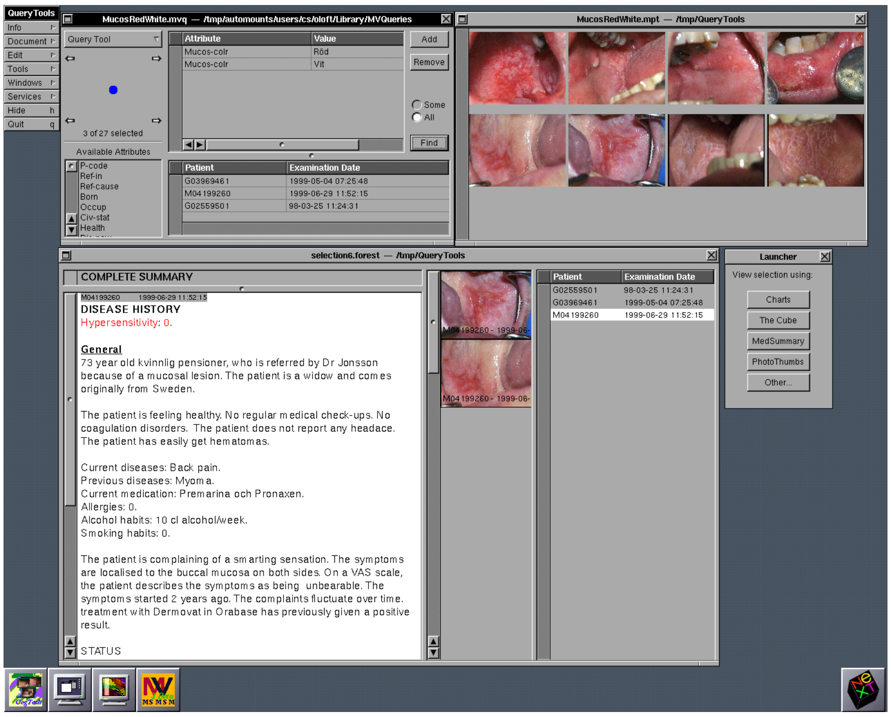
Figure 5: QueryTools in browser mode. The path from the left-most to the right-most column illustrates a part of the examination structure.

5.3 Query Tools

QueryTools is an application that was developed to simplify testing of different ways to enter search queries and find matching examination records. An example of a search window is shown in Figure 5. To the left is a list of all available attributes in the knowledge base and a circle indicating how many records are currently selected. Clicking on an attribute will generally insert it into a query. Depending on what kind of tool is used for entering queries, the view to the right varies. In Figure 5 a query tool that displays the formal structure of an examination at the top of the window, using a browser, is shown. The user can navigate through the browser and add attribute-value pairs to the query by clicking on leaves in the browser. QueryTools was designed to make it straightforward to add more views to test new query devices.

Any number of search windows may be open at the same time and queries found useful can be saved for future use. Each search window has an associated `DFDMachine` object handling definitional computations over the knowledge base. The search mechanisms used are those described in Section 3. The selected examinations may be opened for further study using some other application.

Figure 6 shows an example of how QueryTools may be used in conjunction with other applications. At the top left is a QueryTools window where a previously created query has been opened and used to select a number of examinations. At the top right, the selection is viewed using an application that shows all images related to a number of examinations. At the bottom, the selection has been opened using MedSummary, and a text summarizing all the known information about one of the examinations has been generated.



MucosRedWhite.mvq — /tmp/automounts/users/cs/oloft/Library/MVQueries

Attribute	Value
Mucos-colr	Röd
Mucos-colr	Vit

Patient	Examination Date
G03969461	1999-05-04 07:25:48
M04199260	1999-06-29 11:52:15
G02559501	98-03-25 11:24:31

MucosRedWhite.mpt — /tmp/QueryTools

selection6.forest — /tmp/QueryTools

COMPLETE SUMMARY

M04199260 1999-06-29 11:52:15

DISEASE HISTORY

Hypersensitivity: 0.

General

73 year old kvinnlig pensioner, who is referred by Dr Jonsson because of a mucosal lesion. The patient is a widow and comes originally from Sweden.

The patient is feeling healthy. No regular medical check-ups. No coagulation disorders. The patient does not report any headace. The patient has easily get hematomas.

Current diseases: Back pain.
Previous diseases: Myoma.
Current medication: Premarina och Pronaxen.
Allergies: 0.
Alcohol habits: 10 cl alcohol/week.
Smoking habits: 0.

The patient is complaining of a smarting sensation. The symptoms are localised to the buccal mucosa on both sides. On a VAS scale, the patient describes the symptoms as being unbearable. The symptoms started 2 years ago. The complaints fluctuate over time. treatment with Dermovat in Orabase has previously given a positive result.

STATUS

Patient	Examination Date
G02559501	98-03-25 11:24:31
G03969461	1999-05-04 07:25:48
M04199260	1999-06-29 11:52:15

Figure 6: QueryTools in use.

5.4 Visualization Tools

Information visualization is an integral part of the MedView project. Some of the work done is described in [1, 6, 7, 8]. The applications developed so far do not use the Gisela framework. Instead, specialized classes are implemented to collect answers to queries. We have started work on how Gisela can be used to retrieve data for visualizations from the knowledge base. We describe the required computations here. Most of the actual drawing code can be reused from earlier implementations.

5.4.1 Basic Graphs

Figure 7 shows Charts, an application that can be used for displaying bar charts, scatter-plots, and the like, visualizaing MedView data. Charts is based on one of the first applications developed in MedView but uses the Gisela framework to perform definitional computations.

The views of the knowledge base provided by Charts are bar charts, pie charts and scatter-plots. All parts of the definitional computations performed in the application are embedded in an `DataProvider` object, which was introduced briefly in Section 4.3.1. The application allows the use of correction definitions, as well as value class definitions to enhance visualizations.

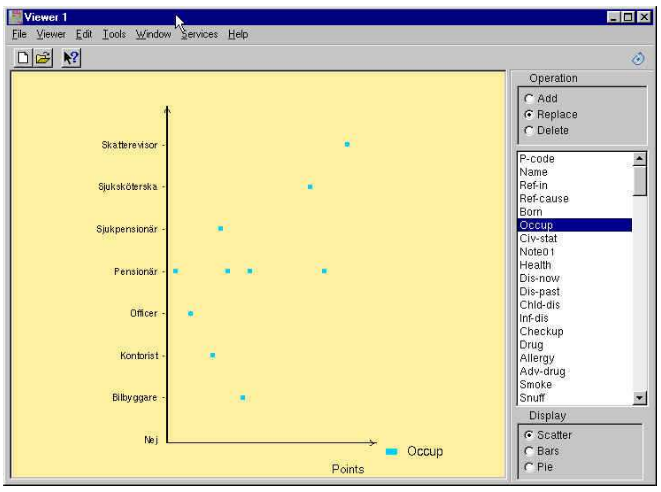


Figure 7: Gisela enabled basic visualization of data.

Computing Bar Chart Data To provide data suitable for drawing a bar chart, all that has to be done is to retrieve all values for an attribute in the knowledge base and count them. A query suitable for finding values for an attribute, say `'Drug'`, is

```
ra(exam){X = 'Drug'}.
```

The computation method `ra` is very simple. It replaces the right-hand side of the single equation in the initial state definition according to the current examination and then unifies the left and right-hand sides:

```
method ra:[Exam].

ra = [Exam, r:Exam].
```

If the attribute given in the right-hand side of the initial state definition has no value in `Exam` the query fails. All values are collected by iterating over all examinations in the knowledge base.

Charts uses the correction definition for an attribute if there is one. When corrections are used the query to ask for values instead becomes

```
filter(exam, drugs){X = 'Drug'}.
```

where `drugs` is a definition with corrections for the attribute `'Drug'`. The computation method currently used in Charts for this kind of query is:

```
method f1:[F].

f1 = [F,r:F] # some r:in_dom(F).
```

```
f1 = [F] # all  not(r:in_dom(F)).

method filter:[E,F].

f = instance(f1,[F]).

filter = [f, r:E].
```

The method `filter` uses the auxiliary method `f1`. The idea is that `filter` is used to look up a value for the current attribute. If a value is found the method `f1` will apply the correction if there is one. Otherwise the left and right-hand sides are unified with each other.

To compute bar chart data using an extra definition defining value classes it is possible to use the same computation method as for corrections. The only difference is that a value class definition is used instead of a definition giving corrections.

Scatter-plot Data To compute values for display in a scatter-plot some more information is needed. Charts displays patient identification codes along the x-axis and all values for the desired attribute along the y-axis. To be able to retrieve the examination in the knowledge base corresponding to a certain dot in the plot we also need the value for the attribute `'Date'` which is unique for each examination. The easiest way to retrieve the desired values is to use the methodology with a query definition discussed in Section 3. The `DataModel` object performing calculations in Charts uses the following query definition, which is read from the application's resources folder at run time:

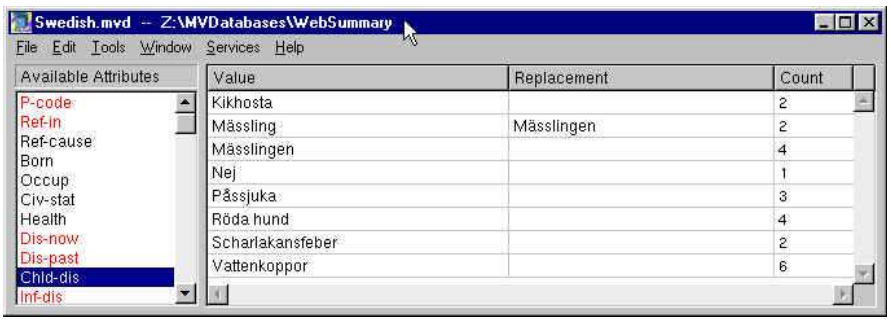
```
definition scatter_data.

query(Attribute, Value, PCode, Date) =
    Value -> Attribute,
    PCode -> 'P-code',
    Date -> 'Date'.
```

The method definition for general query definitions shown in Section 3 provides the necessary procedural knowledge.

5.4.2 The Cube

The Cube [8] offers a 3D visualization of a number of attributes in the knowledge base. The visualization consists of a number of planes, where each plane in principle displays a scatter-plot of the values for an attribute. The current implementation uses a hard-wired definition object capable of performing the required computations.


Swedish.mvd -- Z:\MVDatabases\WebSummary
File Edit Tools Window Services Help
Available Attributes

P-code
Ref-in
Ref-cause
Born
Occup
Civ-stat
Health
Dis-now
Dis-past
Chld-dis
Inf-dis

Value	Replacement	Count
Kikhosta		2
Mässling	Mässlingen	2
Mässlingen		4
Nej		1
Påssjuka		3
Röda hund		4
Scharlakansfeber		2
Vattenkoppor		6

Figure 8: MVDManager: The value “Mässlingen” is used as correction for “Mässling”.

If Gisela is used instead the values for each plane can be computed using the methods used in Charts described above. The use of value classes in the current version of The Cube is implemented by adding a number of extra hash-tables to the machinery for retrieving values from the knowledge base. With Gisela, value classes can be cleanly represented as definitions, which are invoked in computations through the use of a suitable computation method.

We see that the definitional computations required to be able to display basic graphs are very simple. The only issue is for how large data-sets the performance of Gisela will be sufficient.

5.5 Administrative Tools

There is an obvious need for applications aimed at administrative tasks. For instance, it is necessary to be able to monitor the values entered into the knowledge base and create suitable correction definitions. Another important task is to be able to create value classes in an easy way. Here we describe a simple tool to manage correction definitions and discuss the issue of creating value classes.

5.5.1 MVDManager

In Section 2.2.2 we discussed how extra definitions could be used to model corrections of values in the knowledge base. Even if no modifications are needed, it is important to be able to regularly inspect new values added to the knowledge base. To enable this practice we have written a simple tool called MVDManager (MM).

The tool reads a MedView knowledge base and displays information about its contents as shown in Figure 8. To the left is a list of all attributes in the knowledge base. If any new values have been added since the last time the knowledge base was opened in MM the attribute is highlighted (e.g., shown in red). When a user selects an attribute all values for this attribute are listed to the right together with any defined corrections. In Figure 8 both “Mässlingen” and “Mässling” denote the same disease (Measles). We decide that “Mässlingen” is the correct value and add a correction. This information is then stored as a correction definition into the knowledge base.

MM makes rather little use of the Gisela framework. The only definitional computation used is to read all values for each attribute in the knowledge base. The same method `ra` as in Charts is used for this purpose. However, the interesting thing about MM is the output, which provides important information for Gisela computations in other applications.

MM can also be used for localizing the values in a knowledge base to different languages. In this case we simply add replacements for all values that need to be localized.

5.5.2 Creating Value Classes

Creating values classes is in many ways similar to creating correction definitions. A first prototype for a tool to aid the user in the process of creating value classes has been implemented. It is important to be able to organize value classes in some way that makes it easy to have a library of useful groups for different tasks. It is also important to be able to easily combine different value classes into new ones. Furthermore, different kinds of definitions are needed , e.g., for numerical values it should be possible to create value classes by providing ranges. As with MM, the most interesting thing from a definitional point of view with a tool for creating value classes is not the computations it performs but the resulting definitions.

6 Generating Examination Summaries

In this section we describe an implementation of a Gisela program for generating texts in natural language from examination records. The text generator has been written to test functional logic programming methodology in Gisela on a larger example. Texts can be created in HTML or LaTeX format. The generator is written using equational presentations of definitions that can be parsed into definition objects using classes present in the framework. The generator does not modify or extend the framework in any way through the Objective-C API. Thus, in this application, we can say that Gisela is used as a definitional programming language. We present the methods used and some problems encountered.

6.1 Another View of the Knowledge Base

The functional logic programming methodology used to build the text generator is based on functional logic programming methodologies developed for GCLA

[15, 16]. When this methodology is used, function calls will always appear in the left-hand sides of equations in state definitions and predicates in right-hand sides.

Typical examples of simple functions, predicates, and combinations of both are:

```
definition samples.

len([]) = zero.
len([_|Xs]) = (len(Xs) -> N) -> s(N).

length(Xs, N) = (len(Xs) -> N).

member(X, [X|_]).
member(X, [_|Xs]) = member(X, Xs).

mem(Xs) = member(X, Xs) -> X.
```

Note how the predicate `length` calls the function `len` to compute a value and how the non-deterministic function `mem` uses the predicate `member` to enumerate all members of a list.

When a summary text is to be generated from an examination record we typically want to find *all* values of an attribute. That is, if an examination record contains the equations

```
'Drug' = 'Losec'.
'Drug' = 'Dermovat'.
```

we want to generate something like “Currently, the patient takes Losec and Dermovat”.

However, there is really no way to pick up all values defining an attribute which works in conjunction with the functional logic programming methodology used. Since the main objective of our text-generating program was to write a larger functional logic style application we decided to view the knowledge base in another way instead of trying to solve this problem.

Describing the view of data in the knowledge base used is very simple. Each set of equations with the same head

$$\begin{array}{rcl} a & = & v_1. \\ & \vdots & \\ a & = & v_n. \end{array}$$

occurring in an examination record definition, is replaced by a single equation where all the values are put in a list, thus:

$$a \quad = \quad [v_1, \ldots, v_n].$$

On the one hand, with this representation it is easy to find all values for a given attribute. On the other hand, the hierarchical structure of examination records becomes somewhat hidden. Reading the tree-files used to store examinations into this kind of definition can easily be done by subclassing the `DFLeafDefinition` class in the Gisela framework which is otherwise used to represent examination records.

6.2 Describing Summaries

The basis for any Natural Language Generation (NLG) system is to study what kinds of texts should be generated [4, 5]. In MedView, the MedSummary application has been in use over a period of several years to generate patient overviews. During this period, the users have gradually refined the structure and contents of the texts being generated. The Gisela generator described here is based on these experiences. In the following, it should be kept in mind that the purpose of the described system is to test functional-logic programming in Gisela on some larger example, applying sophisticated NLG techniques is a project for the future.

6.2.1 Text Structure

We have tried to distinguish textual building blocks which could form the basis for an authoring tool that could be used to create various text templates. The idea behind such a tool would be to provide a hierarchical view of the building blocks. A similar idea is presented in [12]. It should then be possible to move parts around, and add and group blocks to form complete text templates for different purposes. Essentially, the building blocks form a tree structure with canned text in the nodes, something which is in line with several more ambitious projects [3, 9, 14]. We believe that the division into reusable building blocks would be useful also in a system with a more sophisticated NLG kernel.

We have chosen to structure summary texts into the following:

- Summary. A *summary* is the top level unit that creates a complete document from a number of components. The idea is that a user should be able to select from a number of different summary documents tailored for different tasks.

- Component. A *component* is a reusable unit, approximately corresponding to a section of text. A component is built from a number of blocks.

- Block. A *block* is another reusable unit that corresponds to a paragraph of text. A block is made up of a number of segments.

- Segment. A *segment* is yet another reusable unit. Each segment has a number of associated attributes. These attributes are used to find information from the examination record needed to generate the appropriate text for the segment. Segments are built from sentences and phrases.

- Sentence. A *sentence* is built from canned text and strings from phrases which depend on the database.
- Phrase. A *phrase* consists of an attribute-value pair. It is replaced with a string that depends on the particular value present. Generally, a value can be a list of atoms. The string can be a simple replacement for the value or an arbitrarily complex string created in some way, for instance by inserting the value into a template or combining a number of values into a string.

To construct a summary we define all the various parts in Gisela definitions. When these are complete, a text may be generated using some generator functions and data taken from an examination record. Note that only the two last categories above depend on the language used for generation.

6.2.2 Formatting

In MedSummary a text-template in Rich Text Format (RTF) is used as the basis for generation. The generator parses the template into a number of sentences but keeps all formatting information from the original template and uses it to correctly format the resulting text. While this approach makes it very simple for users to modify the look of the template text it relies heavily on the particular classes used to represent text in the implementation and is not very general.

In the Gisela generator we have instead used logical text-formatting which does not depend on the particular format of the resulting text. That is, when a template is defined, a set of predefined commands are used to describe formatting information. These commands do not describe the exact result, but are of the kind "emphasize this part of the text" or "make this a first level heading". The actual output is defined in a separate *formatting* definition. We have implemented formatters for HTML and LaTeX. Other formats like XML, RTF, or plain text are possible, of course.

Examples of commands available for formatting are:

- `heading1(Item)`,...,`heading5(Item)`, different levels of headings.
- `new_paragraph`, start a new paragraph.
- `emphasis(Item)`, `strong(Item)`, `small(Item)`, `underline(Item)`, some font related commands.
- `unordered_list(Items)`, `ordered_list(Items)`, `list_item(Item)`, used to build lists.

There are also commands for things like building tables. The number and complexity of formatting commands would be larger in a complete system.

6.2.3 Methodology

The system described implies a certain methodology for developing summary templates. Text generated from examination records can be used for several different purposes, for example for a quick overview in the examination room, a complete description with all details to use if the patient should be sent to another clinician, a text tailored towards education, a text intended for the patient to read etc. Although these are not identical they are likely to share some components. These components may be of varying size, from a component providing a full disease history, to a table which displays values from blood-tests. Since texts are split into subparts, which are split into subparts, and so on, all different parts can be stored and combined in new ways for a different purpose. The quality of text resulting from combining subparts in new orders might vary. A deeper NLG system would inspect the parts and try to improve text quality, e.g., by combining two sentences into one.

To write a definitional description of a summary is a very time-consuming activity and requires too much programming knowledge to be considered as a user-friendly approach. However, as with MedSummary used in MedView today, very little formal linguistic knowledge is needed. Also, the code used to generate text is rather simple and follows certain patterns again and again. It should therefore be possible to build, as discussed above, a GUI where users can create new blocks and combine existing ones in new ways. This authoring tool could then generate Gisela definitions. Such a tool could also allow for users with more linguistic knowledge to create more sophisticated templates for generation. As is the case with the text-generation itself, this is a project for the future.

6.3 Defining a Summary

As mentioned, to define a summary we have to give definitions for all the parts the summary is built from. We start with the top level:

```
% summary_components(SummaryName, Components)
summary_components(anamnes, [anamnes]).
summary_components(anamnes_diary, [anamnes,diary]).
```

Here we we have given the components for two summary types. Next, the components are defined by declaring which blocks they are built from:

```
% component(ComponentName, Title, ShowsTitle, Blocks)
component(anamnes, Title, titled, [introblock,commonblock]) =
    component_title(anamnes, Title).
component(diary, Title, titled, [diary_block]) =
    component_title(diary, Title).
```

The title of a component is defined separately to simplify localization into different languages. Along the same lines, blocks are defined by declaring the segments they are built from:

```
% block(BlockName, Segments)
block(introblock, [intro0,intro1]).
block(commonblock, [common1,common2]).
block(diary_block, [diary1,diary2,diary3]).

% segment(SegmentName, Attributes)
segment(intro0, [adv_drug]).
segment(intro1, [pc_age,pc_sex,occupation,ref_in,ref_cause]).
segment(diary1, [vis_cause]).
```

Note that at the segment level, the set of attributes in the knowledge base needed to generate text are given. Thus, the segment `intro0` needs the value of the attribute `adv_drug`.

When an attribute is missing no text can be generated for the part of the text that depends on it. When a segment has only one attribute this means that the empty string is returned. When there are many attributes and only some are missing, as many as possible are used to provide as much information as possible. As an example, for the segment `intro0` there are only two cases:

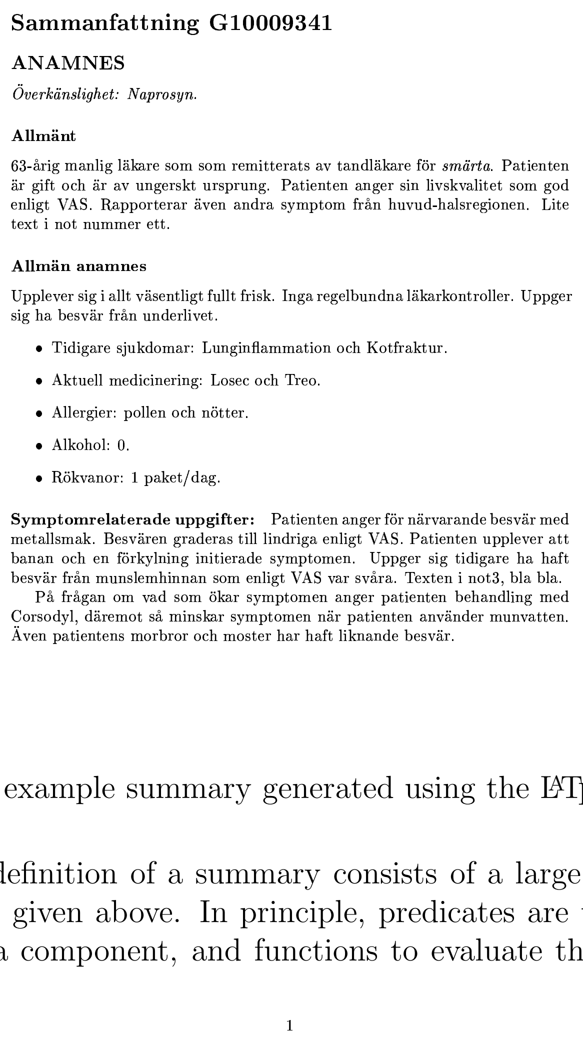
```
% segment_intro0(Attributes, Values, Text)
segment_intro0([adv_drug], [AD],
   concat(text_color(red,
            sentence(["Hypersensivity: ", phrase(adv_drug, AD)])),
          new_paragraph)).
segment_intro0([], [], empty_string).

% adv_drug_phrase(Value, Text)
adv_drug_phrase(['Nej'], "0").
adv_drug_phrase(['Inga'], "0").
adv_drug_phrase(Xs, S)  = value_list(Xs) -> S.
```

The third argument of the predicate `segment_intro0` is an expression which is evaluated to a string on request. The actual strings to use for different values are given by `adv_drug_phrase`. In the third clause of this predicate the function `value_list` is used to create a comma-separated string from a list of atoms. A small part of the definition of `phrase/1` is:

```
phrase(adv_drug, Xs) = adv_drug_phrase(Xs, S) -> S.
```

which illustrates a connection between the predicate `adv_drug_phrase/2` and the function `phrase/1`.



Sammanfattning G10009341

ANAMNES

Överkänslighet: Naprosyn.

Allmänt

63-årig manlig läkare som som remitterats av tandläkare för *smärta*. Patienten är gift och är av ungerskt ursprung. Patienten anger sin livskvalitet som god enligt VAS. Rapporterar även andra symptom från huvud-halsregionen. Lite text i not nummer ett.

Allmän anamnes

Upplever sig i allt väsentligt fullt frisk. Inga regelbundna läkarkontroller. Uppger sig ha besvär från underlivet.

- Tidigare sjukdomar: Lunginflammation och Kotfraktur.
- Aktuell medicinering: Losec och Treo.
- Allergier: pollen och nötter.
- Alkohol: 0.
- Rökvanor: 1 paket/dag.

Symptomrelaterade uppgifter: Patienten anger för närvarande besvär med metallsmak. Besvären graderas till lindriga enligt VAS. Patienten upplever att banan och en förkylning initierade symptomen. Uppger sig tidigare ha haft besvär från munslemhinnan som enligt VAS var svåra. Texten i not3, bla bla.

På frågan om vad som ökar symptomen anger patienten behandling med Corsodyl, däremot så minskar symptomen när patienten använder munvatten. Även patientens morbror och moster har haft liknande besvär.

1

Figure 9: An example summary generated using the LaTeX formatter.

The complete definition of a summary consists of a large number of clauses similar to the ones given above. In principle, predicates are used to store data, like the blocks of a component, and functions to evaluate the stored units into text.

6.4 Generating Text

To actually compute something using a definition we must have a method which specifies how Gisela should proceed to evaluate a query. In our text generator we use a slightly extended version of the method for functional-logic programming given in [17]. The method has three parameters, a summary definition, a formatting definition, and an examination record. To compute a short summary in LaTeX format from the examination `exam1` we ask the query

```
medsum(sum,latex,exam1){summary(short_summary) = S}.
```

which will give the summary as a binding of the variable `S`. If we instead want a summary in HTML format for `exam2` we ask the query:

```
medsum(sum,html,exam2){summary(short_summary) = S}.
```

The function `summary/1` is just a cover for the function `mk_summary/2` which starts the generation process:

```
summary(Name) = mk_summary(Name, no_navigation).

mk_summary(Name, IncludeNavigation) =
    (summary_components(Name, Components),
     anchor_names(Components, AnchorTitles))
    -> mk_summary(IncludeNavigation, AnchorTitles, Components).
```

For use in hyper-text formats, it is possible to generate navigation links to the different parts of a text. The function `summary` turns off this feature. The function `mk_summary/2` picks up values from the predicates `summary_components` and `anchor_names` before evaluation continues with `mk_summary/3`. Then it is simply a matter of unfolding the parts down to the bottom level and combining the results together. For instance, to generate text for all components:

```
make_components([]) = "".
make_components([C|Cs]) =
    concat(make_component(C), make_components(Cs)).
```

To make a segment it is necessary to first find the values for the attributes required by the segment:

```
make_segment(Name) =
    (segment(Name, Attributes),
     values(Attributes, FoundAtts, AttVals),
     segment_text(Name, FoundAtts, AttVals, Text))
    -> Text.

restrict values/3:right.
values([], [], []).
values([A|As], [A|As1], [V|Vs]) =
    (db_vals(A) -> V),
    values(As, As1, Vs).
values([A|As], As1, Vs) =
    values(As, As1, Vs).
```

Here `make_segment` is a function and `values` a predicate. The predicate `values` has a problem: If the second clause succeeds, the third should not be tried. In Prolog a cut would be inserted into the second clause. In Gisela there is really no way to express what we want. To make Gisela more efficient a solution to this problem is needed in the future.

The function `db_vals`, used to find values from an attribute, is defined in a separate definition:

```
definition dbase:matching.
db_vals(A) = A.
```

Evaluation of this function is hard-wired in the computation method `medsum`:

```
medsum = [medsum, E, l:E, l:dbase] # some l:matches(db_vals(A)).
```

The purpose of the above is to make sure that the examination record `E` given as an argument to `medsum` is used to find the values of an attribute. It works as follows: When the left-hand side of an equation matches `db_vals(A)`, the definition `dbase` is used to start evaluation of `db_vals`. To complete evaluation, the computation continues using the definition `E` to find values in the examination record. The left-hand side is then unified with the right-hand-side of the equation and the computation continues using the method `medsum`.

Finally, we show some HTML formatting functions. Most HTML environments are defined using the function `generic_block/3`, which in turn calls `concat/2`:

```
heading1(Heading) = generic_block("<H1>", Heading, "</H1>").

emphasis(Item) = generic_block("<EM>", Item, "</EM>").

generic_block(StartTag, Contents, EndTag) =
    concat(StartTag, concat(Contents, EndTag)).

concat(X, Y) =
    ((X -> X1),
    (Y -> Y1))
    -> g3strings_append(X1, Y1).
```

What `concat` does is to force evaluation of its arguments to canonical values (strings). It then calls `g3strings_append/2`, defined in the definition `g3strings`, to concatenate the resulting strings. The built-in definition `g3strings` defines a number of useful string operations.

To sum it all up, we have as an experiment implemented a complete summary text generator as a functional logic program in Gisela. Due to the large number of attributes and values present, the complete definition of this generator consists of approximately 2300 lines of code (including comments and blank lines). The program runs well but not really fast enough—generating a complete examination record text takes several seconds.

6.5 Alternative Approaches

Declarative programs are often a compromise between beauty and efficiency. It is not unusual that one has to rewrite code in a less declarative way to increase performance. Looking at the text generator in hindsight, from a programming point of view, we can see that there are a number of such cases, mostly related to whether to express something using a function or a predicate.

We have used a number of predicates to define all subparts of the texts and then a number of functions working on these structures to create a text. One reason for the use of predicates instead of functions in some places is not that a predicate is more natural, but that a predicate is assumed to be more efficient. The reason for this is the complexity of the full definiens-operation [2, 11]. However, Gisela allows us to state that a function will only be used using matching, not unification, in which case a lot of the complexity can be avoided. Many predicates could therefore be replaced by functions using matching.

In principle, the following might be a better general structure:

```
anamnes = gen([block1,block2,block3]).

block1 = gen([seg1, seg2]).

seg1 = db([a,b,c], V) -> seg1(V).   % V list [Att-VList...

seg1([A-V]) =
    concat(text_color(red,
               sentence(["Hypersensivity: ", phrase(A, V)])),
           new_paragraph).
seg1([]) = empty_string.
```

where `gen` is a generator function that expands all parts of a list and combines them into a string. This approach is also much more similar to the way grammar rules are usually written which is an advantage from a conceptual point of view.

The main advantage of the generator we have written using Gisela compared to the one in use in MedSummary is that it is completely implemented using definitional programming. For experimenting with different text generation techniques we feel that high-level declarative programming is more appropriate than Objective-C used in MedSummary.

7 Discussion

One of the main objectives behind the development of Gisela has been to provide a framework that can be used for knowledge representation and reasoning in MedView applications. So far, it has not been introduced in the applications actually in use in clinical work. However, the experiments carried out indicate that Gisela integrates well with the tools used in the MedView project to build GUI applications, and that it makes the definitional parts of applications clearer and easier to implement.

We plan to gradually introduce Gisela into the real-world applications developed within MedView to model definitional content. However, we have currently no plans to extend Gisela to handle sophisticated interaction with the user. Instead, we advocate an approach with a definitional model programmed in Gisela and an interface part programmed using other, more suitable, tools. This approach is in line with the Model-View-Controller paradigm, where Gisela is used to build the model, and the controller and view are constructed using standard programming tools.

Most of the applications developed so far are very simple. A question could be whether an application like MVDManager, which only uses Gisela to read a database into memory is of any interest. In accordance with the above discussion, it is definitely relevant since it is an example of an application that uses Gisela to implement the model part of an MVC-based application. Also, using MVDManager results in new definitional structures useful for other applications.

As the database is approaching a size where it might be meaningful to apply data mining techniques to search for patterns it will become increasingly important to have a tool that is tailored for definitional computing.

A problem with the programming techniques for search discussed in Section 3 is that it is difficult to combine computations from within Gisela. Typically, we have to provide the connections between computations externally. For instance, we cannot easily describe things like "find all examinations having property a and then use aggregate b to find the examinations among these having property c". Instead we have to run three separate queries connected through glue-code written in Objective-C. How to handle this issue is an important problem to solve. The essential question is whether the basic knowledge representation with separate definitions for each examination should be changed in some way, or whether there is a clean way to extend Gisela to solve the problem. On a related note, in the MedView database all values are atomic with no internal structure. Since Gisela can handle compound terms and variables, introducing more structure into values should be considered.

For now, we conclude that Gisela does reasonably well for its intended tasks in MedView. However, some performance improvements are needed or we cannot justify the use of pure Gisela for applications such as the text-generation discussed in Section 6.

References

[1] Y. Ali, G. Falkman, L. Hallnäs, M. Jontell, N. Nazari, and O. Torgersson. Medview: Design and adoption of an interactive system for oral medicine. In *Proceedings of Medical Informatics Europe (MIE'00), Hannover, Germany, August 2000*, 2000. To appear.

[2] M. Aronsson. *GCLA, The Design, Use, and Implementation of a Program Development System.* PhD thesis, Stockholm University, Stockholm, Sweden, 1993.

[3] S. Busemann and H. Horacek. A flexible shallow approach to text generation. In E. Hovy, editor, *Proceedings of the Nineth International Natural Language Generation Workshop (INLG'98)*, pages 238–247, 1998.

[4] R. Dale and E. Reiter. Building applied natural-language generation systems. *Journal of Natural Language Engineering*, 3:55–87, 1997.

[5] R. Dale and E. Reiter. *Building Natural-Language Generation Systems*. Cambridge University Press, 2000.

[6] G. Falkman. Information visualization in clinical odontology: multidimensional analysis and interactive data exploration. *Artificial Intelligence in Medicine*. Accepted for publication.

[7] G. Falkman. SimVis: an interaction model for exploring clinical data. In G. Szwillus and T. Turner, editors, *CHI 2000 Extended Abstracts. Conference on Human Factors in Computing Systems, 1–6 April 2000, The Hague, The Netherlands*, pages 319–320. ACM Press, New York, 2000.

[8] G. Falkman, M. Jontell, and N. Nazari. Information visualisation in clinical medicine using 3D parallel diagrams: a case history. In *Proceedings of Medical Informatics Europe (MIE'00), Hannover, Germany, August 2000*, August 2000. To appear.

[9] S. Geldof and W. V. de Velde. An architecture for template based (hyper)text generation. In *Proceedings of the 6th European Workshop on Natural Language Generation-EWNLG'97*, pages 28–37, 1997.

[10] L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.

[11] L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(5):635–660, 1991. Part 2: Programs as Definitions.

[12] G. Hirst, C. DiMarco, E. Hovy, and K. Parsons. Authoring and generating health-education documents that are tailored to the needs of the individual patient. In *Proceedings of the Sixth International Conference on User Modeling*, 1997.

[13] NeXT Computer, Inc. OpenStep specification. Available at `http://www.gnustep.org/resources/resources.html`, October 1994.

[14] E. Reiter, R. Robertson, and L. Osman. Types of knowledge required to personalise smoking cessation letters. In *Artificial Intelligence and Medicine: Proceedings of AIMDM-1999*, pages 389–399, 1999.

[15] O. Torgersson. A definitional approach to functional logic programming. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Extensions of Logic Programming 5th International Workshop, ELP'96*, number 1050 in Lecture Notes in Artificial Intelligence, pages 273–287. Springer-Verlag, 1996.

[16] O. Torgersson. Definitional programming in GCLA: Techniques, functions, and predicates. Licentiate thesis, Chalmers University of Technology and Göteborg University, 1996.

[17] O. Torgersson. Gisela—a framework for definitional programming, 2000.

A Method Used by Text Generator

```
import_definition(canonicals).
import_definition(g3arithmetics).
import_definition(g3strings).
import_definition('dbase.dxt').

method medsum:[S, F, E].
// Right rules tried only when left empty (true).

// stop if true.
medsum = [] # some l:matches(true) & some r:matches(true).

// something proved false.
medsum = [] # some l:matches(false) & some r:matches(false).

// axiom only when left-hand side atom. X = a not accepted.
medsum = [canonicals] # some l:in_dom(canonicals).

// constructed conditions to the left.
medsum = [medsum, l:canonicals] # some l:matches((A,B));
                                     l:matches((C->D)).

// constructed conditions to the right
medsum = [medsum, r:canonicals] # some l:matches(true) &
                                 some r:matches((A,B));
                                      r:matches((C->D)).
// Remaining, definitionals
// first left
medsum = [medsum, E, l:E, l:dbase] # some l:matches(db_vals(A)).
medsum = [medsum, l:g3arithmetics] # some l:in_dom(g3arithmetics).
medsum = [medsum, l:g3strings]     # some l:in_dom(g3strings).
medsum = [medsum, l:S]             # some l:in_dom(S).
medsum = [medsum, l:F]             # some l:in_dom(F).
```

```
// then right
medsum = [medsum, r:g3arithmetics] # some l:matches(true) &
                                     some r:in_dom(g3arithmetics).
medsum = [medsum, r:g3strings]     # some l:matches(true) &
                                     some r:in_dom(g3strings).
medsum = [medsum, r:S]             # some l:matches(true) &
                                     some r:in_dom(S).
medsum = [medsum, r:F]             # some l:matches(true) &
                                     some r:in_dom(F).
```

B Implementation of GenderAgeDefinition

```
#import <DFDefinitions/DFDefinition.h>

// GenderAgeDefinition is a subclass of DFDefinition.
@interface GenderAgeDefinition : DFDefinition
{

// Returns a shared GenderAgeDefinition object.
+ (id)sharedGenderAgeDefinition;

}
@end

@implementation GenderAgeDefinition

+ (id)sharedGenderAgeDefinition {
   // Details omitted
}

// Initialization methods are omitted.

// The domain consists of all terms with principal
// functor gender/1 or age/2.
- (BOOL)inDom:(id)anObject {
    id refered = [anObject dereference];
    if ([refered isKindOfClass:[DFTerm class]]) {
        NSString *termName = [refered fullname];
        return [termName isEqualToString:@"gender/1"] ||
                 [termName isEqualToString:@"age/2"];
    }
    return NO;
}
```

```
// def is defined in terms of clause.
- (NSArray *)def:(id)anObject {
    return [NSArray arrayWithObject:[self clause:anObject]];
}

- (id)clause:(id)anObject {
    id refered = [anObject dereference];
    if ([refered isKindOfClass:[DFCompoundTerm class]]) {
        NSString *termName = [refered fullname];
        if ([termName isEqualToString:@"gender/1"]) {
            return [self getGender:[refered argumentAtIndex:0]];
        }
        else if ([termName isEqualToString:@"age/2"]) {
            return [self getAge:refered];
        }
    }
    return [DFFalseCondition falseCondition];
}


// Find gender from P-code and return male or female.
- (id)getGender:(id)anObject {
    // Get string representation.
    NSString *pnr =  [[anObject dereference] stringValue];
    if ([pnr length] > 8) {
        if([[pnr substringWithRange:NSMakeRange(8,1)] intValue] % 2)
            return [DFConstant constantWithName:@"male"];
        else
            return [DFConstant constantWithName:@"female"];
    }
    return [DFFalseCondition falseCondition];
}

- (id)getAge:(id)anObject {
   // Details omitted
}
```