# 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 $dom(D)$, and the *co-domain* of $D$, written $com(D)$, where $dom(D) \subseteq com(D)$,

2. and a *definiens* operation: $D : com(D) \to \mathcal{P}(com(D))$.

Objects in $dom(D)$ are referred to as *atoms* and objects in $com(D)$ are referred to as *conditions*.

A natural presentation of a definition is that of a (possibly infinite) system of equations

$$D \left\{ \begin{array}{lcll} a_0 & = & A_0 & \\ a_1 & = & A_1 & \\ & \vdots & & n \geq 0, \\ a_n & = & A_n & \\ & \vdots & & \end{array} \right.$$

where atoms, $a_0, \ldots, a_n, \ldots \in dom(D)$, are defined in terms of a number of conditions, $A_0, \ldots, A_n, \ldots \in com(D)$, i.e., all pairs $(a_i, A_i)$ such that $A_i \in D(a_i)$, and $a_i \in dom(D)$. 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, $D((A \to B)) = \{A, B\}$.

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 $(A \to B)$ 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. $D((A \to B)) = \{A, B\}$.

## 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

$$\{a = b, (c, d) = b, e \to b = b, f = b\}.$$

### 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:

$$X \left\{ \begin{array}{lcl} cm\overline{R_1} & = & \left\{ \begin{array}{lcl} white & = & \left\{ \begin{array}{lcl} cm\underline{R_2} & = & \left\{ \begin{array}{lcl} \epsilon & = & \{white = white \\ \end{array} \right. \\ \end{array} \right. \\ brown & = & \left\{ \begin{array}{lcl} cm\underline{R_2} & = & \left\{ \begin{array}{lcl} \epsilon & = & \{brown = white \\ \end{array} \right. \\ \end{array} \right. \\ \end{array} \right. \\ \end{array} \right. \qquad (1)$$

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 $\epsilon$.

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 $W_1 W_2$ 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 $W : S \Rightarrow X$, meaning "$W : S$ can be computed to $X$". We call $W : S \Rightarrow X$ a *goal*. Depending on the application at hand, we will interpret $X$ and $W : S$ 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 $W : S$ 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, "$W : S$ 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 $M(M) = \{W_1, \ldots, W_n\}, n \geq 1$. $M(M)$ is the definiens of $M$ in the method $M$, that is

$$\{W_i \mid M = W_i \# C_i \in M \wedge C_i(S)\}$$

and $X$ is the result definition

$$X \left\{ \begin{array}{lcl} W_1 & = & X_1 \\ & \vdots & \\ W_n & = & X_n \end{array} \right. .$$

**(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

$$X \left\{ \begin{array}{lcl} A_1 & = & X_1 \\ & \vdots & \\ A_n & = & X_n \end{array} \right. .$$

**(4.2)** If $e = ((A, B) = C)$

$$\frac{W{:}\,(C'/(A,B))S \Rightarrow X}{W\overline{D}{:}\,S \Rightarrow X}\ \overline{D}_V$$

where $C' \in D((A, B))$.

**(4.3)** If $e = ((A \to B) = C)$

$$\frac{W{:}\,S_1 \Rightarrow X_1 \quad W{:}\,S_2 \Rightarrow X_2}{W\overline{D}{:}\,S \Rightarrow X}\ \overline{D}_A$$

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 $e = (A = a)$

$$\frac{W{:}\,S(B/a) \Rightarrow X}{W\underline{D}{:}\,S \Rightarrow X}\ \underline{D}_D$$

where $B \in D(a)$.

**(5.2)** If $e = (A = (B, C))$

$$\frac{W{:}\,S(A/(B,C)) \Rightarrow X_1 \quad W{:}\,S(B/(B,C)) \Rightarrow X_2}{W\underline{D}{:}\,S \Rightarrow X}\ \underline{D}_V$$

where $X$ is the result definition

$$X \left\{ \begin{array}{lcl} A & = & X_1 \\ B & = & X_2 \end{array} \right. .$$

**(5.3)** If $e = (A = (B \to C))$

$$\frac{W{:}\,S' \Rightarrow X}{W\underline{D}{:}\,S \Rightarrow X}\ \underline{D}_A$$

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 $R_2$

$$R_1 \begin{cases} status = direct \\ direct = mucos \\ direct = palpation \\ mucos = mucos\_site \\ mucos = mucos\_col \\ mucos\_site = 112 \\ mucos\_col = white \\ mucos\_col = brown \\ palpation = palp\_site \\ palp\_site = 112 \end{cases} \qquad R_2 \begin{cases} status = direct \\ direct = mucos \\ direct = palpation \\ mucos = mucos\_site \\ mucos = mucos\_col \\ mucos\_site = 232 \\ mucos\_col = white \\ palpation = palp\_site \\ palp\_site = 242 \end{cases} ,$$

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:

$$cm \left\{ \begin{array}{lcll} cm & = & cm\overline{R_1} & \# \quad 'eq \in dom(R_1) \\ cm & = & cm\underline{R_2} & \# \quad eq' \in dom(R_2) \wedge \neg 'eq \in dom(R_1) \\ cm & = & \epsilon & \# \quad otherwise \end{array} \right. . \tag{2}$$

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:

$$X \left\{ cm\overline{R_1} = \left\{ \begin{array}{lcl} white & = & \left\{ cm\underline{R_2} = \left\{ \epsilon = \{white = white \right. \right. \right. \\ brown & = & \left\{ cm\underline{R_2} = \left\{ \epsilon = \{brown = white \right. \right. \right. \end{array} \right. .$$

We also show a derivation. All result definitions are abbreviated with some $X_i$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon : \{brown = white\} \Rightarrow X_6}\,T}{cm : \{brown = white\} \Rightarrow X_4}\,M}{cm\underline{R_2} : \{brown = mucs\_col\} \Rightarrow X_4}\,D_D}{cm : \{brown = mucs\_col\} \Rightarrow X_2}\,M \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\epsilon : \{white = white\} \Rightarrow X_7}\,T}{cm : \{white = white\} \Rightarrow X_5}\,M}{cm\underline{R_2} : \{white = mucs\_col\} \Rightarrow X_5}\,D_D}{cm : \{white = mucs\_col\} \Rightarrow X_3}\,M}{cm\overline{R_1} : \{mucs\_col = mucs\_col\} \Rightarrow X_1}\,\overline{D}_D$$
$$\dfrac{}{cm : \{mucs\_col = mucs\_col\} \Rightarrow X}\,M$$

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

$$cm_{L,R} \left\{ \begin{array}{lcll} cm & = & cm\overline{L} & \# \quad 'eq \in dom(L) \\ cm & = & cm\underline{R} & \# \quad eq' \in dom(R) \wedge \neg 'eq \in dom(L) \\ cm & = & \epsilon & \# \quad otherwise \end{array} \right. .$$

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, $\Sigma$, 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 $(A \to B)$ 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 $\forall i, j(x_i = x_j \to i = j)$. We use $\sigma$, $\tau$, $\phi$, $\sigma_1$,... 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: $D : com(D) \to \mathcal{P}(com(D))$.

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 $\sigma$ is called *$a$-sufficient* if $D(a\sigma)$ is closed under further substitution, that is, for all substitutions $\tau$, $D(a\sigma\tau) = (D(a\sigma))\tau$.

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

$$m_D \begin{cases} m & = & m\overline{D} & \# & {}'eq \in dom(D) \\ m & = & m\underline{D} & \# & eq' \in dom(D) \wedge \neg {}'eq \in dom(D) \\ m & = & D & \# & otherwise \end{cases}$$

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 $\sigma$ 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 $\Gamma$ is a list of goals and $\theta$ 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

$$X' \left\{ \begin{array}{lcl} W_1 & = & X_1 \\ & \vdots & \\ W_n & = & X_n \end{array} \right. .$$

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

$$X \left\{ \begin{array}{lcl} A_1 & = & X_1 \\ & \vdots & \\ A_n & = & X_n \end{array} \right. .$$

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 $e = ((A \to B) = C)$ 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:

$$\begin{array}{ll} Imports & \\ \texttt{method} & m(D_1, \ldots, D_n). \quad n \geq 0 \\ Instantiations & \end{array}$$

$$\begin{array}{rcl} m & = & W_1 \quad \# \quad G_1. \\ & \vdots & \\ m & = & W_m \quad \# \quad G_m. \end{array}$$

*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 $\texttt{\{}e_1, \ldots, e_n\texttt{\}}$ 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 $m\texttt{(}D_1, \ldots, D_n\texttt{)}S$, $n \geq 0$, 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 $\Gamma$ is a list of conditions and $C$ is a condition. The meaning of the query is: "Does $C$ follow from $\Gamma$ 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:

$$D \quad \begin{cases} plus(0,m) & = & m. \\ plus(s(n),m) & = & s(plus(n,m)). \end{cases}$$

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:

$$P \quad \begin{cases} plus(0,m) & = & m. \\ plus(s(n),m) & = & plus(n,m). \end{cases}$$

$$N \quad \{ \quad n \quad = \quad s(n).$$

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:

$$F \quad \begin{cases} F(x) & = & P(x) & \#P(x) \notin Dom(P). \\ F(x) & = & NFP(x) & \#P(x) \in Dom(P). \end{cases}$$

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.

anamnesis = common.
common = drug.
common = allergy.
common = smoke.
drug = no.
allergy = oranges.
smoke = '8 cigarettes/day'.
```

```
definition s2:constant.

anamnesis = common.
common = drug.
common = allergy.
common = smoke.
drug = no.
allergy = lemons.
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).
```

$$\frac{\{\texttt{true} = C\}}{\{\texttt{true} = c\}}\ dr \quad c \in Dom(P), C \in P(c)$$

$$\frac{\{A_1 = C\} \quad \ldots \quad \{A_n = C\}}{\{a = C\}}\ dl \quad a \in Dom(P), P(a) = A_1, \ldots, A_n$$

$$\frac{}{\{a = a\}}\ ax \quad a \in Dom(D)$$

$$\frac{}{\{\texttt{true} = \texttt{true}\}}\ t \qquad \frac{}{\{\texttt{false} = \texttt{false}\}}\ f$$

$$\frac{\{A = B\}}{\{\texttt{true} = A \to B\}}\ ar \qquad \frac{\{\texttt{true} = A\} \quad \{B = C\}}{\{A \to B = C\}}\ al$$

$$\frac{\{\texttt{true} = C_1\} \quad \{\texttt{true} = C_2\}}{\{\texttt{true} = (C_1, C_2)\}}\ vr \qquad \frac{\{C_i = C\}}{\{(C_1, C_2) = C\}}\ vl \quad i \in \{1, 2\}$$

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(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".

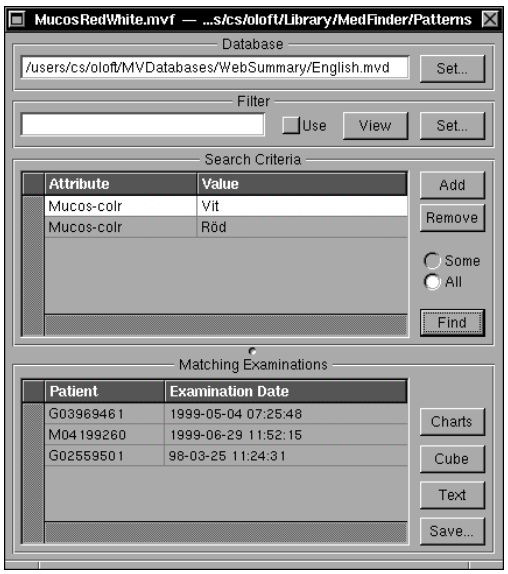


Figure 3: ExaminationFinder search panel.

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 $\Delta$ 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 $\Delta$, a result stack $R$, and a substitution $\theta$.
- A *computation state* is a stack $F; \Phi$ of computation frames. $F$ is the active or topmost frame, and $\Phi$ 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 $\theta$ a substitution, and $no$ which indicates that no answer could be computed.

**(1) Init**

$$M{:}S \rightarrow \langle [M{:}S], [], \emptyset \rangle \ .$$

At the top level only a single method is allowed.

**(2) Success**

$$\langle [], [X], \theta \rangle; \Phi \rightarrow yes(X, \theta) \ .$$

Alternative solutions are computed by restarting the machine from the state $\Phi$.

**(3) Failure**

$$\{\} \rightarrow no \ .$$

**(4) Goal Success**

$$\langle [\epsilon{:}S|\Delta], R, \theta \rangle; \Phi \rightarrow \langle \Delta, [S|R], \theta \rangle; \Phi \ .$$

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 $M(M) = \{W_1,\ldots,W_n\}, n \geq 1$. $M(M)$ is the definiens of the method name $M$ in the method $M$, that is

$$\{W_i \mid M = W_i\#C_i \in M \wedge C_i(S)\}.$$

If $M(M) = \{\}$ 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.