Visual Compositional–Relational Programming

Andreas Zetterström

June 29, 2010

Abstract

In an ever faster changing environment, software developers not only need *agile methods*, but also agile *programming paradigms* and tools. A paradigm shift towards declarative programming has begun; a clear indication of this is Microsoft's substantial investment in functional programming. Moreover, several attempts have been made to enable *visual* programming. We believe that software development is ready for a new paradigm which goes beyond any existing declarative paradigm: *visual compositional-relational programming*.

Compositional-relational programming (CRP) is a purely *declarative* paradigm—making it suitable for a visual representation. All procedural aspects—including the increasingly important issue of *parallelization*—are removed from the programmer's consideration and handled in the underlying implementation. The foundation for CRP is a theory of higher-order combinatory logic programming developed by Hamfelt and Nilsson in the 1990's.

This thesis proposes a model for visualizing compositional-relational programming. We show that the diagrams are isomorphic with the programs represented in textual form. Furthermore, we show that the model can be used to automatically generate code from diagrams, thus paving the way for a visual integrated development environment for CRP, where programming is performed by combining visual objects in a drag-and-drop fashion. At present, we implement CRP using Prolog. However, in future we foresee an implementation directly on one of the major object-oriented frameworks, e.g. the *.NET* platform, with the aim to finally launch relational programming into large-scale systems development.

Keywords: visual programming, compositional-relational programming, logic programming, declarative programming

Uppsala Universitet
Institutionen för informatik och media
Data- och systemvetenskap

Master thesis, D-level (30 hp)
Spring term 2010
Supervisor: Prof. Andreas Hamfelt

Contents

Acknowledgements

I want to thank the people who have been helpful to me in my work with writing this thesis: Jørgen Fischer Nilsson, Gunnar Dahlberg, Pär Ågerfalk, Jonas Sjöström, Erika Widenkvist, and my supervisor Andreas Hamfelt.

1 Introduction

Globalized and turbulent business environments fused with rapid advancements in technology put new demands on software developing organizations. User requirements are often hard to establish and can seldom be assumed to be stable throughout a project. As a consequence, a class of software development methodology referred to as *agile* has emerged. Agile methods operate on the principle of "just enough method" and are tailored to "embrace change." By adopting principles such as short iterations and test-driven development (TDD), projects are more flexible and better suited to handle changing requirements, even late in the development process [1]. To be successful, agile projects need flexible development tools and environments able to cope with the required pace of change. Object orientation is the dominant paradigm in software development. Unfortunately, it is rooted in imperative problem solving techniques that require the programmer to specify *how* something should be than rather than *what* should be done. For a long time, the dominance of the fundamentally imperative object-oriented paradigm appeared not be broken in any foreseeable future. Now, however, there are clear indications that a paradigm shift is underway in the software development industry. The most evident sign is the current substantial investment in functional programming by Microsoft (*LINQ* and *F#)*.

Functional programming is a *declarative* programming paradigm. Declarative programming is performed at a higher level of abstraction than imperative programming, focusing on *what* the program should do rather than *how*. Declarative programming is not a new phenomenon—functional programming has existed since the 1950's. In the 1970's, another more expressive declarative paradigm emerged: *logic* programming, also called *relational* programming. Theoretically, logic programming has many advantages; however, it also has disadvantages and has not been widely adopted in the software development industry. In existing logic programming languages, e.g. *Prolog*, the programmer has to consider procedural aspects of the program's execution. The *logical* semantics of the program is not identical to the *procedural* semantics of the program.

One reason why object-oriented programming has been the preferred paradigm in software development industry is that it lends itself naturally to a *component-based*, *modular* structure of large-scale programs, where program components can be *re-used*. This *modularity* and *re-usability* are features that to a large extent have been lacking in existing relational programming. Another reason for the unchallenged dominance of object-oriented programming is its suitability for modeling. Class hierarchies are easy to visualize using tools such as design class diagrams in the *Unified Modeling Language* (UML). Logic programming, on the other hand, is still perceived as difficult, or even "strange", by most mainstream systems developers. If declarative programming is to be widely used in mainstream commercial systems development, it has to be easy to use and to visualize. An indication of this is that the only wide-spread declarative programming language is the database management language *Structured Query Language* (SQL), which is built on mathematical set theory and relational algebra. Database modeling, in fact, has a diagrammatic model, namely the *Entity-Relationship* model.

Attempts have been made to enable *visual programming*, e.g. by using UML-diagrams

to generate object-oriented code. The idea behind this is that visual representations of programs are easier to understand to the human mind than textual representations. However, existing visual programming techniques often lead to diagrams that are more complicated than the code itself, something that have been criticized by leading software engineering practitioners [26]. Several types of diagrams—both static and dynamic diagrams—are required to represent a program. Moreover, these visual programming environments often require some manual coding. Attempts have also been made to enable *visual logic programming*, with limited success—the diagrams tend to be more complicated than the code.

A very promising branch of declarative programming is *compositional-relational programming* (CRP), developed by Hamfelt and Nilsson [29, 14, 13, 15, 16, 20, 18, 17, 19]. In comparison to existing logic programming, e.g. Prolog, CRP has multiple advantages such as being *purely declarative* and naturally *compositional*. These properties of CRP enable a unique, unambiguous visualization—a one-to-one relationship between the visualization and the program code. The unique, unambiguous visualization is a prerequisite for high-level visual programming.

One important factor behind the recent interest in declarative programming is *parallelization*. This factor increases in importance with the evolution in hardware. With the multi-core processors available today, programs have to execute in parallel in order to be efficient. Declarative programs are much easier to parallelize than imperative programs. The theoretical properties of CRP enable parallelization to be completely implemented in the underlying framework—the programmer would not have to consider parallelization at all.

1.1 Aim

The aim of this thesis is to develop a design theory [12]—a model—for *visual compositional-relational programming*. To fulfill this aim, we will introduce additional "syntactic sugar" constructs which facilitate visualizing, develop a diagrammatic model for visualizing CRP, and show that the proposed model can be used to automatically generate source code for CRP programs.

1.2 Demarcations

We do not perform any usability study on the visual CRP programming model, nor do we consider other alternative models. The CRP programs we consider are side-effect-free programs. We do not consider programs with side effects (e.g. accessing the computer file system); nor do we consider interaction with existing code libraries (e.g. the *.NET* platform).

1.3 Method

We use a research method that aims at developing a *design theory* in accordance with Gregor and Jones [12]. In their theoretical framework, a design theory is a conceptual model related to information technology (IT), e.g. diagrammatic models, programming paradigms, or systems development methods. When developing a design theory, the *purpose and scope* of the theory must be clearly stated. Furthermore, *principles of form and function* must be declared, *testable propositions* must be made, and *justificatory knowledge* must be provided. *Principles of implementation* must be stated, and proof of concept must be given in the form of *expository instantiations*. We will evaluate our design theory analytically.

1.4 Outline

We begin by describing the various forms of computer programming that currently exist (programming paradigms), with a special focus on relational (logic) programming and compositional-relational programming. We continue by describing the major existing models for visualizing software and information systems. Next, we propose a diagrammatic language—a model—for visual compositional-relational programming, and explore how this model can be used to automatically generate source code from diagrams. In the last chapter we conclude, discuss implications and highlight some interesting areas for future research.

In accordance with the theoretical framework of Gregor and Jones, we state the *purpose and scope* of our theory in chapter 1, we define *principles of form and function* in chapter 5, we address *artifact mutability* in chapter 6.2, we state *testable propositions* in chapter 5, we provide *justificatory knowledge* in chapters 2, 3 and 4, we state *principles of implementation* in chapter 5, in particular 5.6, and we exemplify our design theory with *expository instantiations* in chapter 5 and the Appendices.

2 Programming Paradigms

This chapter provides a background to programming paradigms. It is based on [3] and [31]. If the reader is already familiar with the subject, this chapter can be skipped without loss of continuity.

2.1 Machine Language and Assemblers

At the lowest level, all computer programs consist of sequences of instructions encoded as numeric digits. This form of numerically represented instructions is called *machine language*. An example of an instruction in machine language could be "move the contents of register 3 to register 8", and this instruction is expressed as a binary number (e.g. 10010110). The first computers had to be programmed directly in this way, and machine languages are therefore called *first generation* programming languages.

In the 1940's, notational systems called *assembly languages* or *assemblers* were developed, in which machine language instructions can be expressed as words instead of numbers. A translational system translates the commands to numeric instructions. This was such a large advance in programming that assembly languages are called *second generation* programming languages.

2.2 High-Level Languages

Both first and second generation programming languages are dependent of the properties of a particular machine. The instructions in the language are also restricted to the atomic steps of the machine's execution of the program. The next step in the evolution of programming paradigms were the so-called *high-level* or *third-generation* programming languages that began to emerge in the 1950's.

The instructions in a high-level programming language are expressed at a higher level of abstraction, i.e. several machine-language instructions are bundled together in higher-level constructs such as variable assignment, if-else-statements, loops etc. Furthermore, locations in memory are not referenced directly by their address; they can be given names. These names are known as *variables*. A system called *translator* or *compiler* translates the high-level instructions to machine-language code. Thus, high-level programming languages are *machine-independent*, because as long as there is a compiler for a combination of a particular language and a particular machine, programs written in the language can be executed on the machine.

2.3 Structured Programming

Structured programming was proposed by E.W. Dijkstra et al. in the late 1960's [10, 8], in a reaction to the "spaghetti"-like code which was the result of an abundant use of the *goto*-statement. They proposed that the flow of the program should exclusively be handled by predefined control structures such as loops and if-statements. The code should be organized in named *procedures*. The control flow of the program should be handled by calling these procedures by name, not by ordering a jump to a numbered line in the source code file. The aim was to arrive at cleaner, more maintainable program code.

2.4 Imperative Programming

This section describes the two main imperative programming paradigms: procedural and object-oriented programming.

2.4.1 Procedural Programming

In procedural programming, the program code consists of a sequence of commands that step by step tells the machine how to obtain the desired results. While bundling together

several machine-language instructions into high-level language constructs such as if-else-statements and loops, the basic idea is still the same as in the assembly languages: tell the machine step by step what to do. The first procedural high-level languages gained a huge popularity in the early 60's, the foremost being *FORTRAN* for scientific computing and *COBOL* for business computing. Today, *C* is the most wide-spread procedural language.

2.4.2 Object-Oriented Programming

Today, object-oriented programming (OOP) is the most prominent programming paradigm in the software development industry. In OOP, software is structured in entities called *objects*, reflecting how humans view the real world. For instance, in a system for administrating a university, every course in the real world would be represented by an object in the system. Objects are created from *classes*; in the university administration system, there would be a `Course` class, from which `Course` objects are created.

OOP is still imperative programming, but—and this is why it has replaced procedural programming as the main paradigm—it organizes the imperative code statements more elegantly. This more elegant organization of the code permits huge software systems to be grasped by a human mind. In procedural programming, data structures are kept separate from procedures. In OOP, data structures and procedures—in OOP called *methods*—belong together. A class—and therefore the objects created from that class—contains both *data* and *methods* for performing operations upon that data. For instance, in the university administration system, the `Course` class would contain both data (such as course literature, number of credit points etc.) and methods (such as enrolling a student).

An important principle in OOP is *encapsulation*. This means that a class should expose to other classes only what these need to access; everything else should be hidden from the outside world. In this way, software can be built in a *modular* way, where software components communicate only through strictly defined *interfaces*[1]. Software components should be highly *cohesive*—i.e. have well-defined responsibilities—and they should be loosely *coupled* to one another—i.e. have as few connections to one another as possible.

Another important principle in OOP is *polymorphism*. In the real world, many classes of entities are similar and share attributes. This is reflected in OOP, where we can create class *hierarchies*, using *inheritance*. This means for instance that we can create a class for vehicles (`Vehicle`)—having attributes for production year, owner and color as well as some methods. From this `Vehicle` class (the *superclass*) we can create an inherited class `Car` (a *subclass*) without having to rewrite the code for the attributes in `Vehicle`—they are inherited. What is more, in the superclass we can declare *virtual* properties and methods, which can be altered (*overridden*) in subclasses. Polymorphism means "of several shapes". In our `Vehicle` example, a `VehicleRegister` object could have a collection of vehicles, without knowing which are cars and which are bikes, and could iterate over this collection and invoke a `CalculateTax` method on each vehicle. Each vehicle would then perform this operation according to how the `CalculateTax` method is implemented in the subclass the

[1] In OOP, an *interface* specifies what a class exposes to the outside world.

vehicle belongs to (car, bike, truck etc.).

This modular, component-based structure of OOP is suitable for reuse of code. Huge libraries, often called *frameworks*, have been written, from which the programmer can use already-written—and what is equally important, already *tested*—classes. This suitability for code reuse is a main factor behind the popularity of OOP. The two dominant frameworks in OOP today is the *Java* platform (developed by Sun Microsystems, now belonging to Oracle) and the *.NET* platform (developed by Microsoft). Major object-oriented languages today are *C++*, *Java*, *C#*, *Visual Basic* and *Python*.

2.5 Declarative Programming

Declarative programming stems from mathematical concepts such as set theory, lambda calculus and formalized logic. It has found the most widespread use in the database management systems following the relational model—based on mathematical set theory—proposed by Codd in 1970 [6].

In a declarative paradigm the programmer describes *what* to compute, but does not tell the machine *how* to do it. This description of the desired result is often called an *expression*. Since the computer at the machine level still needs to be told step by step what to do, the logic concerning the program's execution is defined in the underlying implementation—it is hidden "under the hood". The declarative description of the program is translated to imperative statements. This means that declarative programming is performed at a higher level of abstraction than imperative programming.

Another feature common to all declarative paradigms and languages is *statelessness*. In imperative programming, there are variables that can change state (i.e. values). In declarative programming, this is not the case. The only way to change the state of a variable is to create a new variable with a new value. Also, iteration is not handled by loops, but by a technique called *recursion*. Since there are no variables, we cannot have a loop variable that changes value for every step in the loop. Recursion means that the algorithm calls itself with a new changed argument which is the counterpart to the imperative loop variable.

The statelessness of declarative programming has a big impact on an issue that has recently gained significantly in importance: *parallelization*. In imperative programming, making code execute in parallel is difficult, because different threads must be prevented from accessing the so-called *shared state*—i.e. the variables and objects in the program. In declarative programming, on the other hand, there is no shared state. This makes parallelization much less complicated, and this is an important factor behind the recent renewed interest in declarative programming. There are two major branches of declarative programming: *functional* programming and *logic* (or *relational*) programming.

2.5.1 Functional Programming

The functional paradigm dates from the same period (the 1950's) as the first imperative high-level languages. *LISP* is the most prominent functional language of that period. Func-

tional programming has increased in use in recent years, with languages such as *OCaml*, *Erlang* and *Scala*. The year 2010 may signify a turning point for the functional paradigm, with the introduction of *F#* as a fully-fledged language on Microsoft's .NET platform, thus spreading the use of functional programming from only specific domains to mainstream commercial systems development. F# also has an interesting feature—which it shares with several other recent programming languages—namely being a *multi-paradigm* programming language. This feature makes a transition from an object-oriented language such as C# seamless, since existing object-oriented code can be used from inside F#, and F# code can be called from object-oriented C# or Visual Basic code [30].

In functional programming, everything—including the program itself—is a *function*. A function takes input (so-called *arguments*) and produces output (so-called *return value*). Functions are regarded as values, which means that functions can take other functions as parameters and return a function. A function taking other functions as arguments and/or returning a function is called a *higher-order* function. Using higher-order functions, programs can be written at a high level of abstraction, relieving the programmer of tedious, routine tasks—these tasks are abstracted away from the programmer and encoded as general solutions to a general type of problem.[2] The following example highlights the difference between the imperative and declarative paradigms, with C# and *LINQ* as example language:

```
1   //Find all female customers older than 30 years and return their address
2
3   //Imperative solution
4   //*****************************************************************
5
6   var addressList = new List<Address>();
7   foreach (var customer in customers)
8   {
9           if (customer.Age > 30 && customer.Sex == Sex.Female)
10          {
11                  addressList.Add(customer.Address);
12          }
13  }
14  return addressList;
15
16  //*****************************************************************
17
18
```

[2]In fact, in object-oriented programming these general types of problems are often referred to as *design patterns*. They have a conceptual solution, but this solution needs to be programmed every time in every application, all over again. In functional programming, this can more often than not be replaced by a general encoding of the pattern as *program code*, which can be placed in a library provided to the programmer [30].

```
19  //Functional solution using LINQ
20  //*****************************************************************
21
22  return
23          customers.Where(
24                  c => c.Age > 30 && c.Sex == Sex.Female).Select(
25                          c => c.Address
26          );
27
28  //*****************************************************************
```

In the above example we can clearly see how the imperative solution requires the programmer to create a list object where to store the result, declare a loop over all customers, declare an if-statement to check if the selection conditions hold, add the customer to the result list, and finally return the result list. In the functional solution, on the other hand, all that is required is to declare an expression describing what to return; this description is a composition of two higher-order functions (`Where` and `Select`) which take other functions as arguments.

2.5.2 Logic Programming

Logic programming is based on formal predicate calculus. The most wide-spread logic programming language is *Prolog*, which was invented by Colmerauer and Kowalski in the early 1970's [7, 24]. Logic programming is a paradigm that builds on formal predicate logic. A logic programming language relies on an underlying problem-solving algorithm that can make deductions in a system for predicate logic. For Prolog, this problem-solving algorithm is called *SLD-resolution* (Selective Linear Definite Clause Resolution). A Prolog program consists of predicates, defined by *facts* and *rules*. The program is executed by asking questions to it, either via a Prolog console window, or by other software units (e.g. via a HTTP-request). The following listing gives an example of a simple Prolog program.

```
1   %harry is a man. This is a Prolog "fact".
2   man(harry).
3   %bill is also a man.
4   man(bill).
5   %And so is peter
6   man(peter).
7   %Another fact. harry is bill's parent
8   parent(bill, harry).
9
10  %If Y is a man and is also X's parent, Y is X's father.
11  %This is a Prolog "rule"
12  father(X, Y) :- man(Y), parent(X, Y).
```

The above listing shows a very simple Prolog program. It states the facts that the symbols `harry`, `bill` and `peter` are men. It also states a rule saying that if somebody is a man and a parent of somebody, then he is a father of that somebody. When we load this program into a Prolog engine and asks it questions, it will apply its SLD-resolution problem solving algorithm. If we ask `father(bill, harry)` it will see that this *goal* means that the *subgoals* `man(harry)` and `parent(bill, harry)` must *succeed*. Success of a goal means that Prolog finds a fact or can deduce from the facts and rules in the program that the goal is true. In this case, it has read the source file from top to bottom and matched the question with the rule for `father(X, Y)`, and found that it needs to see if the subgoals `man(Y)` and `parent(X, Y)` succeed. It starts with the first subgoal, `man(harry)` and reads the source file from top to bottom again. This time, it finds a match at `man(harry)` and so this subgoal has succeeded. It moves on to prove `parent(bill, harry)`—and it finds the line saying `parent(bill, harry)`. Thus, `father(bill, harry)` has succeeded. Prolog will now respond "yes".

In the same example program, what happens if we ask Prolog `father(X, Y)`—i.e. we give it *unbound variables* as arguments for the father-relationship (as opposed to the *constants* we gave it before)? As answer to this question, Prolog will give us, one after the other, all father-relationships it can deduce. In our example, there will be just one father-relationship: `X=bill, Y=harry`. Prolog will create a *search tree*, and try to resolve all the subgoals it finds on its way. It will try to *unify* the unbound variables X and Y with the constants found in the program (in this case `harry`, `peter` and `bill`). If a subgoal does not succeed with a particular unification, it will return—*backtrack*—and try to unify the variable with the next constant etc. This is a brief desription of how the underlying problem-solving algorithm in Prolog works.

The fact that Prolog predicates—unlike functions in functional programming—can be used in both directions is called *bi-directionality*. This is a very important concept that makes logic programming more expressive than functional programming. A predicate does not describe a 1-1 mapping between input and output, but any kind of relation between entities. This is why logic programming is also called *relational* programming.

The relational counterpart of recursive functions in functional programming is recursive predicates (or recursive *relations*). This means that the predicate is defined in terms of itself. A classic textbook example is a predicate for the ancestor relation. Say we have a `parent` relation with two arguments (parent and child). Then we can recursively define an `ancestor` relation in terms of the `parent` relation and the `ancestor` relation itself:

```
1
2   %Some facts about parents and children
3   parent(abraham, isaac).
4   parent(isaac, jacob).
5   parent(jacob, judah).
6
7   %Base case. Parents are ancestors.
8   ancestor(A, B) :- parent(A, B).
```

```
9
10  %Recursive case
11  %If there is an X that is parent of someone,
12  %and B is ancestor of this X, then A is
13  %ancestor of B
14  ancestor(A, B) :- parent(A, X), ancestor(X, B).
```

Recursion often involves lists, where each element of the list is processed and the predicate is applied to the rest of the list. The following listing gives an example of a recursively defined list relation, `delete`, that relates a list to another list in which the first occurrence of a given element has been removed:

```
1   /*In Prolog, a list can be constructed and deconstructed
2   with the construct:
3    [Head | Tail] where Head is the first element
4   and Tail is the rest of the list
5   */
6
7   %Base case. If the element to delete is the head of list,
8   %the result is the tail of the list
9   delete(Element, [Element | Tail], Tail).
10
11  %Recursive case. If the head of the list is not
12  %the element to delete, keep it there and apply the predicate itself
13  %to the tail of the list
14  delete(Element, [OtherElement | Tail], [OtherElement | NewTail]):-
15      delete(Element, Tail, NewTail).
```

Negation is a complicated matter in relational programming [5]. In Prolog, negation is implemented as *negation as failure*. This means that to Prolog, something is false as long as it fails to prove it—i.e. that the predicate fails (*closed-world assumption*).

Logic programming in Prolog is not purely declarative, unlike the predicate logic (*Horn clause logic*) which forms the theoretical basis for Prolog programs. The *logical* semantics and the *procedural* semantics of a Prolog program may not be the same. For instance, a predicate can consist of several *clauses*, the order of which does matter. For example, in a recursive predicate, if the base case clause and the recursive clause switch places, the program may not terminate. The following listing exemplifies this:

```
1   %This will execute correctly:
2   ancestor(A, B) :- parent(A, B).
3   ancestor(A, B) :- parent(A, X), ancestor(X, B).
4
5   %Logically, this is the same program,
6   %however it will not execute correctly:
```

```
7   ancestor(A, B) :- parent(A, X), ancestor(X, B).
8   ancestor(A, B) :- parent(A, B).
```

This is just an example of how the programmer must know how the underlying Prolog implementation executes the program. Another aspect the programmer must have knowledge about is how the backtracking mechanism works; in many Prolog programs so-called *cuts* are used to prevent unwanted backtracking.

3 Compositional–Relational Programming

Compositional-relational programming (CRP) was invented by Hamfelt and Nilsson in the late 1990's. This paradigm raises the level of abstraction in relational programming by introducing higher-level control structures, called *combinators* and *operators*, with which predicates (henceforth called *programs*), can be combined and operated upon [16]. In CRP, recursion is not handled by the programmer on an ad-hoc basis in every program definition (as is the case in ordinary logic programming); recursion is conducted using built-in recursion operators. This eliminates all procedural aspects from the programmer's consideration—making CRP a purely declarative programming paradigm.

Ordinary logic programming (e.g. Prolog) is not purely declarative. The programmer has to deal with procedural aspects, which is not in accordance with the fundamental idea of logic programming. Indeed it is reasonable to believe that it is an important factor behind the lack of success of the logic programming paradigm in the software development industry. If the programmer has to control procedural aspects anyway, why not write the program in a main-stream object-oriented language, with all the debugging and other development tools readily available?

In CRP, the programmer does not have to consider procedural aspects of the program's execution; this is a prerequisite for high-level visual component-based programming. CRP introduces structured programming facilities, in the form of pre-defined control structures (*schemes*), similar to the schemes for sequencing, conditionalizing and iterating present in procedural programming or to the higher-order functions present in functional programming. The theoretical basis of CRP is a theory of combinatory logic programming having been proposed and developed by Hamfelt and Nilsson in a series of papers [29, 14, 13, 15, 16, 20, 18, 17, 19]. We will now look into some key aspects of CRP.

3.1 Combilog

We will use the programming language *Combilog*, introduced by Hamfelt and Nilsson [20], for representing compositional-relational programs. Combilog can be implemented in ordinary Prolog using a meta-logic environment, where all programs are represented as arguments to a meta-predicate called `apply`. The following listing shows an example.

```
1   %Ordinary prolog
2   p(X) :- q(X).
```

```
3
4   %Combilog form using the metapredicate "apply"
5   apply(p, [X]) :- apply(q, [X]).
```

The first argument to `apply` is a *program*, or a variable ranging over programs. In order to enable compositional programming, variables ranging over programs are necessary. The second argument to `apply` is a list of arguments that the program is to be applied to. We will henceforth use this Combilog-Prolog form in our program examples. This allows the reader to run and experiment with the programs using ordinary Prolog.

3.2 Variable-Free Form

Definitions of programs in Combilog do not contain any variables. This is called *variable-free form*. Hamfelt and Nilsson [20, 16] have shown that every Prolog program can be rewritten into this variable-free Combilog form, and they also present algorithms for how this is to be done. The fact that the definition of Combilog programs are variable-free is also fundamental for enabling composition of programs. We will now look at what *variable-free form* means, taking as example a simple program for joining two lists: `append`.

```
1   %Ordinary prolog.
2   %The first two arguments are two lists,
3   %The third argument is a list containing
4   %the elements from the first two lists
5   append([],L,L).
6   append([H|T],L2,[H|L3]) :- append(T,L2,L3).
7
8   %Combilog form using the foldright recursion operator
9   %(Recursion operators will be explained subsequently)
10  apply(append, [L1, L2, L3]) :- apply(foldr(cons, id), [L1, L2, L3]).
11
12  %Leaving the Prolog syntax,
13  %"append" written in a pure variable-free Combilog form
14  append :- foldr(cons, id).
15
```

From the above example we can see that in the recursive Prolog predicate definition—consisting of two clauses—variables are needed in the definition. In the Combilog form, the list of arguments on the left-hand side is identical to the list of arguments on the right-hand side. Since this is the case, we can "cancel out" the arguments and write the definition in a completely variable-free form.

3.3 Combinators

CRP programs can be compositionally combined using *combinators*—*and* and *or*—forming a new program which constitutes a combination of the sub-programs. This works similar

to "," and ";" in Prolog. The mechanism of these combinators is standard logical *and* and *or*—if all subcomponents combined with the *and* combinator succeed, the whole combination succeeds, and if any subcomponent combined with the *or* combinator succeeds, the whole combination succeeds. We will now look at an example, first in pure variable-free Combilog syntax and then in Combilog-Prolog syntax of the *and* combinator:

```
1  %Pure variable-free Combilog syntax
2  newProgram :- and(oneComponent, anotherComponent)
3
4  %In a Combilog-Prolog implementation:
5  apply(newProgram, [X]) :-
6          apply(and(oneComponent, anotherComponent), [X]).
7
8  %We have to define the combinator
9  apply(and(P, Q), ArgList) :- apply(P, [X]), apply(Q, ArgList).
```

3.4 Recursion Operators

All iteration in CRP is handled through pre-defined recursion schemes. In Combilog, Hamfelt and Nilsson have introduced two basic recursion operators: foldright (`foldr`) and foldleft (`foldl`).[3] Foldright (also known as *Reduce*) reduces a problem to the base case and then computes the result, whereas foldleft (also known as *Accumulate*) accumulates the result when recursing down to the base case. They have proven a duality theorem regarding these operators—a theorem stating that every program that can be expressed using one of these operators can also be expressed using the other. This in turn leads to a certainty regarding termination criteria for any program using these two recursion operators. Termination can be guaranteed through simple input-output mode analysis (inspecting which arguments are bound and which are unbound) making any other program analysis superfluous [15]. For the time being, there is proof for the duality theorem only for primitive recursive list relations—which can informally be described as recursive relations defined with only one recursive call.[4] Although this is theoretically sufficient, obviously programming praxis and efficiency considerations would require at least one more basic operator: double recursion (binary recursion). If double recursion operators are to be introduced, proof for a corresponding duality theorem should be pursued.

We will now look at the definition of `foldr` and `foldl` in the Prolog implementation of Combilog:

```
1  %Foldright
2  apply(foldr(P, Q), [[], Y, Z]) :- apply(Q, [Y, Z]).
3  apply(foldr(P, Q), [[X | T], Y, W]) :-
```

[3] It is outside the scope of this thesis to provide a full formal explanation of `foldr` and `foldl`; this can be found in [15] and [16].

[4] For full formal description see [15].

```
4         apply(foldr(P, Q), [T, Y, Z]),
5         apply(P, [X, Z, W]).
6
7   %Foldleft
8   apply(foldl(P, Q), [[], Y, Z]) :- apply(Q, [Y, Z]).
9   apply(foldl(P, Q), [[X | T], Y, W]) :-
10        apply(P, [X, Y, Z]), apply(foldl(P, Q), [T, Z, W]).
```

In the above example, we can see that the recursion operators `foldr` and `foldl` are not particularly intuitive or easy to use by the programmer. However, `foldr` and `foldl` could in turn be used to construct more programmer-friendly recursion operators. The three arguments to foldright and foldleft make it possible to use an accumulator argument carrying information during the recursion steps, as well as to present a result when the recursion is finished. It is important to keep in mind that such programmer-friendly recursion operators would merely be "syntactic sugar" for the programmer's convenience: since they would use foldright and foldleft, the duality theorems still hold.

Hamfelt and Nilsson later proposed a more generalized form of the fold operators, where the base case is not confined to match the empty list [16]. This is more expressive, since on many occasions we need a base case that is not restricted to match the empty list—for instance in the classic `ancestor` program in 2.5.2. This is how the more general `foldr` and `foldl` are implemented in Combilog-Prolog:

```
1   %Foldright
2   apply(foldr(P, Q), [L, Y, Z]) :-
3           apply(Q, [L, Y, Z]).
4   apply(foldr(P, Q), [[X | T], Y, W]) :-
5         apply(foldr(P, Q), [T, Y, Z]),
6         apply(P, [X, Z, W]).
7
8   %Foldleft
9   apply(foldl(P, Q), [L, Y, Z]) :- apply(Q, [L, Y, Z]).
10  apply(foldl(P, Q), [[X | T], Y, W]) :-
11        apply(P, [X, Y, Z]), apply(foldl(P, Q), [T, Z, W]).
```

3.5 The *Make* Operator

When combining programs with the combinators and recursion operators described above, making no use of variables, we have to be able to take a program and construct another program with a different number and/or different order of arguments. This means that we need a *projection* operator. Thus the variable-free form can be upheld, even if the combined subprograms do not take the same number of arguments in the same order. The following example shows that if we do not have a possibility for projection of arguments, we find ourselves in a dilemma:

```
1  %First program, taking one argument
2  apply(firstProgram, [Arg1]) :-
3          apply(/*implementation of firstProgram */).
4
5  %Second program, taking two arguments
6  apply(secondProgram, [Arg1, Arg2]) :-
7          apply(/*implementation of secondProgram */).
8
9  %How to combine these without referring to variables?
10 apply(thirdProgram, [/* Which arguments should go here??*/ ]) :-
11         apply(and(firstProgram, secondProgram),
12         [/* Which arguments should go here??*/ ]).
```

The above mentioned dilemma is solved by Hamfelt and Nilsson using a *generalized projection operator*, which they call the *make* operator [20]. The *make* operator takes a list of indeces (we will call it the *index list*) and a program (we will call it the *inside program*) as arguments, thus creating a new program (we will call it the *outside program*). The outside program in turn has its own argument list. The make operator directs the outside program's arguments to the inside program, to the place in the argument list specified in the index list. If the inside program has more arguments than the outside program, the remaining arguments to the inside program will be instantiated with unbound variables. If the inside program has less arguments than the outside program, the remaining arguments to the outside program will not be given to the inside program. The following code listing provides some examples:

```
1  %First program, taking one argument
2  apply(firstProgram, [Arg1]) :-
3          apply(/*implementation of firstProgram */).
4
5  %Second program, taking two arguments
6  apply(secondProgram, [Arg1, Arg2]) :-
7          apply(/*implementation of secondProgram */).
8
9
10 %We construct a new program with two arguments,
11 %which takes the first of its arguments and
12 %gives it to firstProgram.
13 %X1 will be bound to Arg1 in firstProgram,
14 %and X2 is a "dummy" argument which is never used.
15 apply(thirdProgram, [X1, X2]) :-
16         apply(make([1, 2], firstProgram), [X1, X2]).
17
18 %Combination of the two programs
19 apply(fourthProgram, [X1, X2]) :-
```

```
20          apply(and(thirdProgram,
21          secondProgram),
22          [X1, X2]).
23
24  %First and second program
25  %can also be combined directly,
26  %without first declaring thirdProgram
27  apply(fourthProgram, [X1, X2]) :-
28          apply(and(make([1, 2], firstProgram),
29          secondProgram),
30          [X1, X2]).
31
32  %This time, let's give the second argument (X2)
33  %to firstProgram instead.
34  %Here, X1 is a "dummy" argument never used,
35  %and X2 will be bound to Arg1 in firstProgram
36  apply(fifthProgram, [X1, X2]) :-
37          apply(make([2, 1], firstProgram), [X1, X2]).
38
39  /*
40  We need to define all "make" operators that
41  we use
42  */
43  apply(make[1, 2], P), [X1, X2]) :- apply(P, [X1]).
44  apply(make[2, 1], P), [X2, X1]) :- apply(P, [X1]).
```

In the above example we see that the application of the *make* operator to a program makes it possible to "pick" arguments and give it to the program. Thus programs taking different arguments can be combined in the variable-free form. We can also see that in the Combilog-Prolog implementation, a definition of *make* for every combination of indexing and number of arguments needs to be defined.

3.6 Basic Programs

To start with, there has to be a set of basic programs available—predefined basic programs that constitute the basic building blocks in the compositional programs. In Combilog [20], these are *id* (the identity program), *cons* (the list constructor), *true* (the true program) and *const* (program for declaring constants). Much like a machine language needs a few basic instructions, all CRP programs can be built compositionally using only basic programs, the combinators and the operators.[5] The following example shows the Combilog-Prolog

[5]With *cons*, list processing programs can be built. For number processing programs, the *successor* program would be needed. If other data structures than lists, e.g.
trees are to be included, basic programs for constructing these would also be needed.

implementation of the basic programs:

```
1   %The identity program
2   apply(id, [X, X]).
3
4   %The list constructor
5   apply(cons, [H, T, [H | T]]).
6
7   %The true program
8   apply(true, [_]).
9
10  %The constant program.
11  %We need one definition for every constant in the program.
12  apply(const_a, [a]).
13  apply(const_anotherConstant, [anotherConstant]).
```

But why is there a need for a program for constants (const)? The reason is that we need to define every CRP program in the variable-free form discussed above (3.2). The following listing provides an example:

```
1   %Definition of a program with two arguments
2   apply(firstProgram, [X1, X2]) :-
3           /*Implementation of firstProgram */
4
5   %Now we want to call firstProgram with
6   %Arg2 hard-coded as the constant ''a''
7   apply(secondProgram, [Arg1, Arg2]) :-
8           apply(and(make([1, 2], firstProgram),
9           make([2, 1], const_a)), [Arg1, Arg2]).
10
11  %We need a definition for const_a
12  apply(const_a, [a]).
```

In the above example we can see that the variable-free form is preserved—because the variable has been bound to a constant using the constant program.

3.7 Curried Programs

In order to simplify the source code version of a CRP program, so-called *currying* can be applied. This is a "syntactic sugar" construct that can always be re-written to the pure Combilog form. It is used by Hamfelt and Nilsson, (e.g. [16]), and also by us in our example programs in the Appendices. The following listing provides an example:

```
1   %We want to create a curried identity program
2   % that should check if Arg is equal to X.
```

```
3   %With this program available, we can
4   %directly write i.e.\ id([]) or id(1)
5   apply(id(X), [Arg]) :- apply(id, [X, Arg]).
6
7   %If we didn't have the curried version above
8   %we would have to write a program like the following
9   %for everything we need to check for:
10  apply(id([]), X) :- apply(id, [X, []]).
11  apply(id(1), X) :- apply(id, [X, 1]).
12  %Etc...
```

4 Diagrammatic Models

There are countless diagrammatic models for visualizing information. This section will provide a background to the models most relevant to information systems and programming. If the reader is already familiar with the subject, this chapter can be skipped without loss of continuity.

4.1 Euler and Venn Diagrams

Both Euler and Venn diagrams visualize *sets*. Euler diagrams—also called Euler circles—were first presented by Euler in 1768 [11]. An Euler circle divides the plane on which it is drawn in two zones: inside and outside the circle. Everything that is inside the circle belongs to the set, and everything that is outside does not belong to the set. The mathematical set-theoretical notion of *intersection* is represented by letting circles overlap, the notion of a *subset* is represented by letting one circle contain another circle, and the notion of *disjointness* is represented by letting several circles *not* overlap.

Venn [32] criticized Euler diagrams for being too strict in the sense that they cannot deal with imperfect knowledge about the domain. He summarized his criticism as follows:

> The weak point in this [Euler diagrams], and in all similar schemes, consists in the fact that they only illustrate in strictness the actual relation of classes to each other, rather than the imperfect knowledge of these relations which we may possess, or may wish to convey by means of the proposition.[11](p. 510)

In a Venn diagram, all possible combinations are shown even if we do not know anything about them (the "imperfect knowledge"). Areas which we are not interested in are just shaded out. (See figure 1.)

4.2 E–R Diagrams

The Entity-Relationship diagram (ERD) was proposed by Chen in 1976 [4]. The ERD is a diagrammatic model for visualizing the relational database model. The relational database

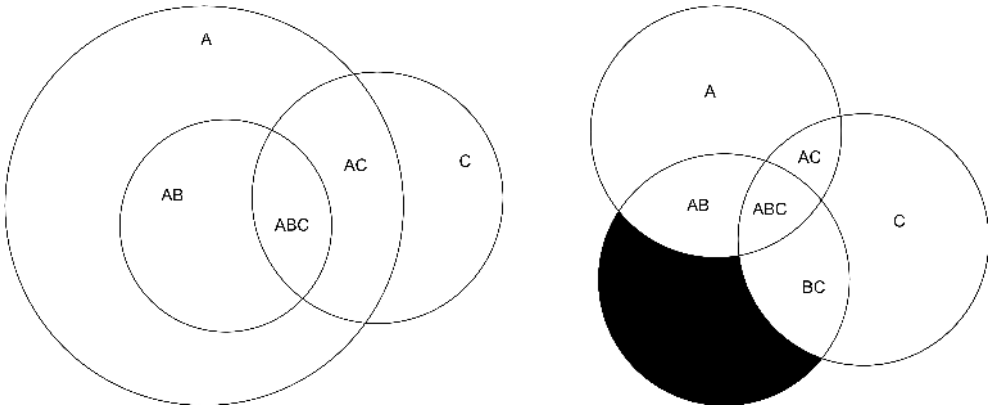

Figure 1: An Euler diagram (left) and a Venn diagram (right).

model has two major components: entities and relationships. Entities are the concepts to be modeled, for example "employees", "aircraft", "products" etc. Relationships denote the relationships between the entities—i.e. "project-worker" may be the relationship between "company" and "employee". (See figure 2.) ERD:s have found a very wide-spread use—mainly for modeling databases, but the concept is also used for object-oriented modeling.

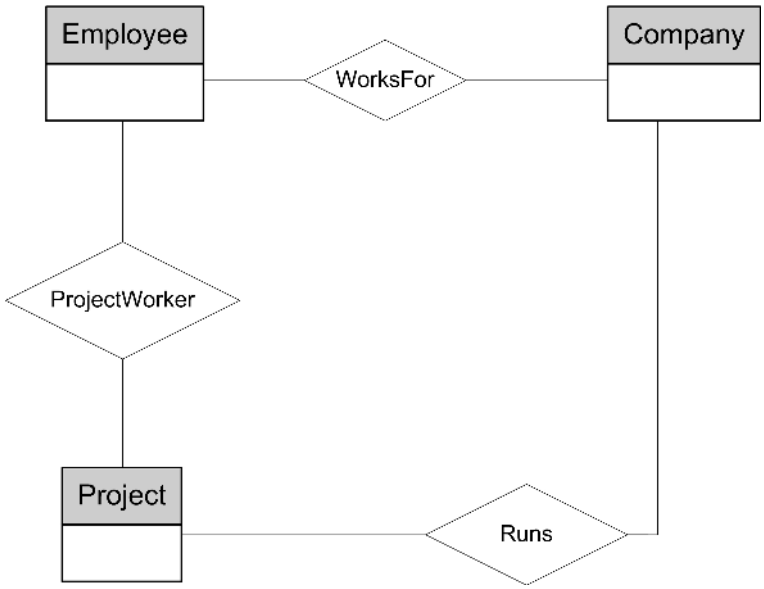

Figure 2: An E–R Diagram depicting the entities *Employee*, *Company* and *Project* and their relations.

4.3 Data Flow Diagrams

In imperative programming there are two traditional views of a program or system: the *static* and the *dynamic* view. The static view is usually depicted by some kind of E–R diagramming; but this only describes how the data is structured. When modeling imperative programs, we also need to depict what is happening when the program executes—how data flows through the system (i.e. what data are passed between procedures, modules or other program components). The traditional way of doing this is *data flow diagrams* (DFD). Despite its name that indicates a focus on data, DFD diagramming focuses on what *activities* are taking place in the program [9]. A data flow diagram shows how data flows between program components (often called *processes*) and data stores. Figure 3 gives an example. DFD:s can also be used to model business processes.

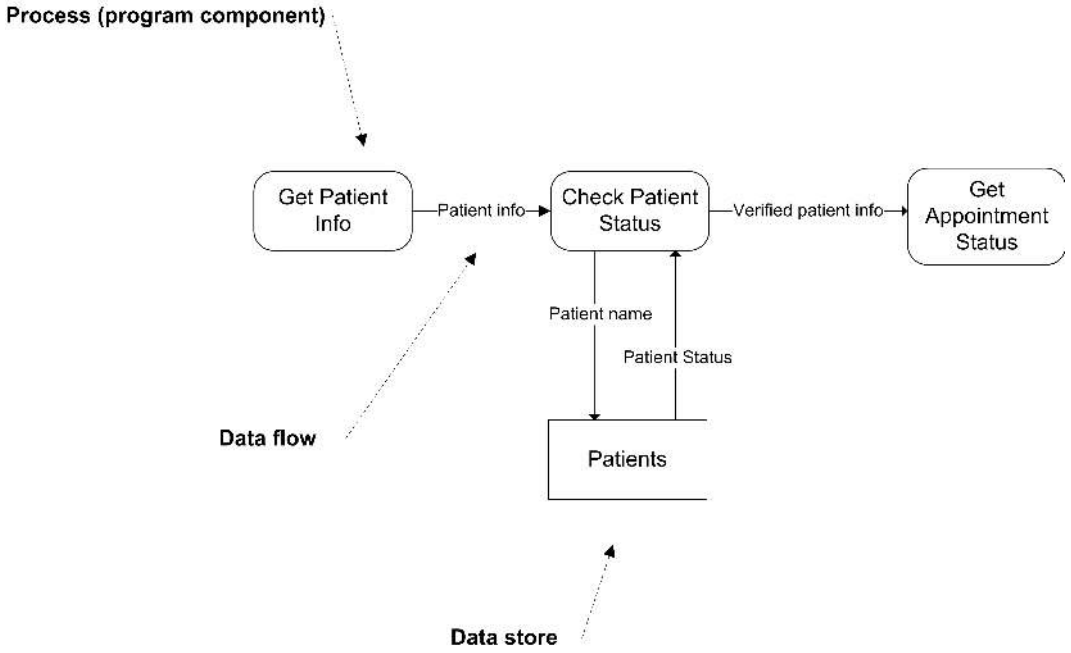

Figure 3: A simple DFD showing how data flows between processes and data stores.

4.4 UML

The *Unified Modeling Language* (UML) was created in the 1990's as a modeling language suitable for object-oriented modeling. It consists of several diagrammatic models which are integrated and used together to replace ERD:s and DFD:s [9]. It would require a whole book to describe all the subtleties of the various UML diagrams.[6] However, the concepts are the same as in ERD:s and DFD:s: there are static diagrams for modeling data (e.g. *design class diagrams*) and dynamic diagrams for modeling activity (e.g. *sequence diagrams*).

[6] A good introduction to UML diagramming and Object-Oriented Analysis and Design (OOAD) is given by Craig Larman in [25].

4.5 Higraphs

The higraphs are proposed by Harel as a suitable visualization for a wide array of applications [21]. The higraph is a general kind of diagramming object, combining set-theoretical diagrams like Venn diagrams with graphs (connections by arrows). He does not allow for showing intersection just by letting two shapes (Harel uses round-corner rectangles) intersect, but he draws a named rectangle inside the intersecting area. In this manner, he arrives at a stunning level of detail and expressiveness, and he suggests that his hi-graphs may be suitable for i.e. database modeling, knowledge representation and statecharts. Figure 4 provides an example.

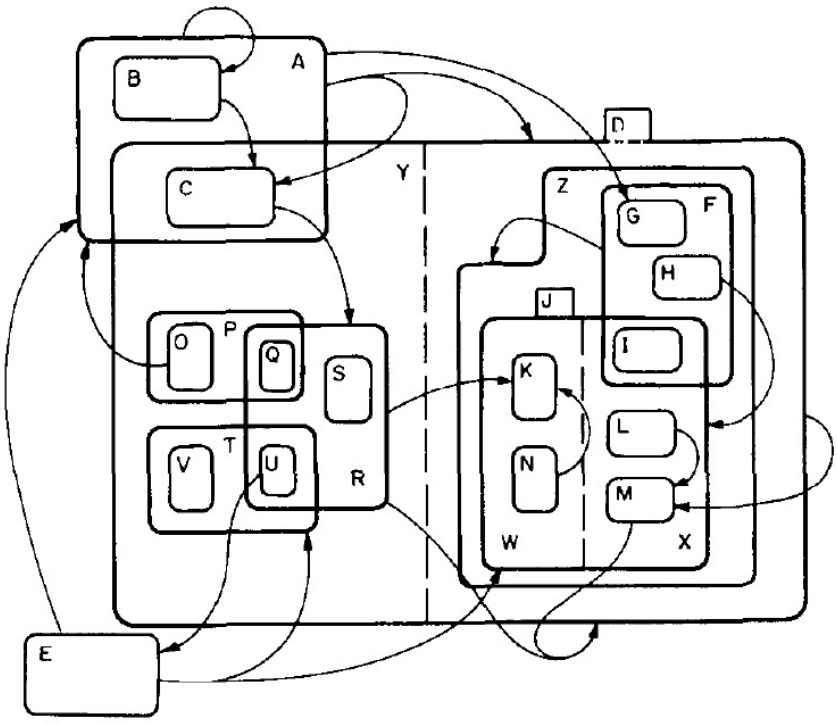

Figure 4: A higraph, taken from [21].

4.6 Visual Object-Oriented Programming Tools

Various attempts have been made for constructing visual programming tools for object-oriented programming, e.g. Model-Driven Architeture [28], Executable UML [27] and various other CASE-tools.[7] These aim at allowing the programmer to model the program visually and automatically generate code from diagrams, much like our approach. However, these approaches are different from ours in that they do not provide an isomorphic relation between the visual model and the code—due to the inherent procedural nature of

[7] *CASE:* Computer-Aided Software Engineering.

the object-oriented paradigm, as opposed to the declarative nature of the visual modeling. They have been criticized by leading software engineering practitioners, e.g. Robert Martin [26], who recommend that they should not be used because the resulting code is in too many cases difficult to understand. Furthermore, most often they also require manual coding, leading to an unfortunate co-habitation of manually written and automatically generated code. However, in some areas of the systems development industry, visual programming tools are widely used, notably for producing classes that represent database entities in object-relational mappers (ORM).[8]

4.7 Previous Attempts at Visualizing Logic Programming

Attempts have been made at visualizing ordinary logic (Prolog) programming, e.g. by Agustí et al. [2]. They use higraphs to visually represent Prolog predicates in the standard Horn clause logic form, and their goal, just as ours, is to provide a visual programming environment. Although this is possible using their approach, their model suffers from several problems. Ordinary logic programming is not compositional like CRP. Recursive predicates cannot be depicted in one diagram but two. Predicates with a different number of arguments cannot be combined. Conjunctions (*and*-combinations) and disjunctions (*or*-combinations) cannot be depicted in one diagram symbol representing the whole construct, but need as many diagrams as there are terms. All this leads to similar problems as with the above mentioned visual programming tools for object-oriented programming: the diagrams are as complicated—if not more—than the corresponding source code.

A visualization of CRP has been attempted by Håkansson et al. [23, 22]. They try to visualize a form of Combilog where recursion is not restricted to built-in recursion schemes. They use a three-dimensional model that looks similar to how molecules (e.g. the DNA molecule) are often visualized in chemistry and physics. However, their model is not complete—they present some ideas for a visualization of CRP, but important issues are left unconsidered. The mapping between the visualization and the Combilog code is not fully elaborated.

5 Towards Visual CRP

We will now explore some important issues on our way towards visual CRP. Some practical examples of our efforts can be seen in Appendix A and B. First, we will add some "syntactic sugar" concerning declaration of constants and addition and removal of arguments without using the *make* operator. Secondly, we will devise strategies for hiding the *make* operator from the programmer. Thirdly, we will devise a visual model—diagrammatic symbols—for CRP. Finally, we will evaluate our model by exploring how an automatic code generator that produces source code from CRP diagrams could be constructed.

[8] An example is the visual designer for Microsoft's *Entity Framework* in the *Visual Studio* IDE.

5.1 Adding Some "Syntactic Sugar"

In order to arrive at a viable visualization of CRP, we will add some "syntactic sugar" constructs that will simplify the visualization. Our goal is to reduce the number of diagrams needed to describe a program.

5.1.1 Declaring Constants and Adding Arguments

It would not be very convenient if the programmer had to declare constants explicitly using the *const* program (see 3.6). Let us recall the example in 3.6:

```
1   %Definition of a program with two arguments
2   apply(firstProgram, [X1, X2]) :-
3           /*implementation of firstProgram */
4
5   %Definition of constant a
6   apply(const_a, [a]).
7
8   %Declaration of constant a using
9   %the const program
10  apply(secondProgram, [Arg1, Arg2]) :-
11          apply(and(make([1, 2], firstProgram),
12          make([2, 1], const_a)), [Arg1, Arg2]).
```

Now we will do the same thing as in the above example; however, we will use constants directly, using the Prolog implementation.

```
1   %Now, we call firstProgram with the constant "a"
2   %as its second argument using the constant directly
3   apply(secondProgram, [Arg1, Arg2]) :-
4           apply(firstProgram, [Arg1, a]).
```

Using this method, every direct use of a constant can be re-written to a variable-free form by means of the *make* operator, the *and* combinator and the *const* program. The direct use of a constant is just "syntactic sugar" on top of the pure variable-free form.

Looking at the previous example, we can see that `Arg2` is a "dummy" argument, meant always to take an unbound variable. In fact, we would probably like to wrap `secondProgram` in a *make* construct:

```
1   %Wrap secondProgram inside another program
2   %which takes 1 argument. Give an unbound "dummy"
3   %value to the second argument of secondProgram,
4   %using the "make" operator
5   apply(thirdProgram, [X1]) :-
6           apply(make([1], secondProgram), [X1]).
```

```
7
8   %Recall the definition of "make" for this case
9   apply(make([1], P), [X1]) :-
10          apply(P, [X1, X2]).
```

From the above example we can see that `thirdProgram` wraps `secondProgram` using the *make* operator. The second argument to `secondProgram` will be given a "dummy" value (righthand-side argument `X2` in the definition of `make`).

Now we will add a "syntatic sugar" construct for adding unbound arguments directly, without using the *make* operator. For clarity, we will give all programs again in this example:

```
1   %Definition of a program with two arguments
2   apply(firstProgram, [X1, X2]) :-
3           /*implementation of firstProgram */
4
5   %Now, we call firstProgram with the constant "a"
6   %as its second argument using the constant directly
7   apply(secondProgram, [Arg1, Arg2]) :-
8           apply(firstProgram, [Arg1, a]).
9
10  %We want to create another program which
11  %takes one argument. It should call
12  %secondProgram with the constant "a"
13  %as the second argument.
14  apply(thirdProgram, [X1]) :-
15          apply(secondProgram, [X1, X2]).
```

In the previous example we can see that there are more arguments on the right-hand side in `thirdProgram` than on the left-hand side. However, every program written in this form can be rewritten to the variable-free form of the previous examples. Finally we will combine the two "syntactic sugar" constructs, using both direct declaration of constants and adding of arguments:

```
1   %Definition of a program with two arguments
2   apply(firstProgram, [X1, X2]) :-
3           /*implementation of firstProgram */
4
5   %We want to create another program which
6   %takes one argument. It should call
7   %secondProgram with the constant "a"
8   %as the second argument. We will now do so directly.
9   apply(secondProgram, [X1]) :-
10          apply(firstProgram, [X1, a]).
```

Clearly this simplifies the source code significantly, and the programs will be easier to visualize because we can eliminate the programs which only declare constants or wrap other programs inside *make* constructs. We will henceforth allow direct declaration of constants and adding new arguments; indeed, we will coin some new terminology for this. We will call the arguments on the left-hand side *outside arguments*, and the arguments on the right-hand side *inside arguments*. When using a program in another program, we care only about the arguments on its left-hand side; the arguments on the right-hand side are "inside" the program, hidden from us. The familiar idea of the "black box" when describing modular software springs to mind (see e.g. [3], chapter 7)—hence the terms *outside* and *inside* arguments.

One could ask if we ever would want to remove outside arguments—i.e. have fewer inside than outside arguments? Although theoretically this would be possible (recall that the *make* operator can both add and remove arguments), we would never have any practical use for it. If we would want to discard an outside argument, there is no need to have it as an outside argument at all. To have a clear, structured, modular design of CRP programs, the programs should expose only the necessary arguments to the outside world—everything else should be hidden inside the program.

5.1.2 Facts

When declaring programs that simply state facts like the following ordinary Prolog facts

```
1   parenthesisPair("{", "}").
2   parenthesisPair("[", "]").
3   parenthesisPair("(", ")").
```

in CRP, it would be tedious to have to use `cons`, `member`[9] to construct the program `parenthesisPair`.[10] For practical reasons, we will introduce a "syntactic sugar" construct for stating facts:

```
1   apply(parenthesisPair, ["(", ")"]).
2   apply(parenthesisPair, ["{", "}"]).
3   apply(parenthesisPair, ["[", "]"]).
```

This construct diminishes the number of programs and thus makes its visualization simpler—however, it can always be rewritten into pure Combilog form.[11] It is used in the example program in Appendix B.

[9] Program that describes the relation between a list and its elements, succeeding if the element is a member of the list.

[10] Use `cons` to put the two characters in a list, then use `member` to see if this list is a member of a hard-coded list of parenthesis pairs.

[11] Of course, the re-writing for the constant declaration "syntactic sugar" construct (see 5.1.1) would also have to be applied.

5.2 Strategies for Handling *Make*

As discussed in 3.5, when applying a combinator to two programs and a list of arguments, we cannot be sure that the two programs needs all of the arguments sent to the combinator. For instance, we could have two programs sent to the combinator, one program which always takes only one argument and the other program taking three arguments. We still want to be able to combine the two programs using the *and* operator. To this end, we use the *make* operator:

```
1   apply(firstProgram, [Arg1]) :- /* firstProgram */
2   apply(secondProgram, [Arg1, Arg2, Arg3]) :- /* Body of secondProgram
3
4   apply(thirdProgram, [Arg1, Arg2, Arg3, Arg4]) :-
5           apply(and(make([1, 2, 3, 4], firstProgram),
6           make([1, 2, 3, 4], secondProgram)), [Arg1, Arg2, Arg3, Arg4]).
```

It is rather obvious that having to write programs like this is not very convenient. Therefore, strategies for hiding the make operator should be devised. For instance, looking at the example program in Appendix A, it is clear that in most cases new arguments inside the program definitions (see 5.1) are added at the rightmost position, i.e. that no re-ordering of arguments is performed. Thus, a simple mapping strategy emerges, which relieves the programmer of tedious, routine work. We will map the inside arguments to the subprograms using a left-to-right strategy (leftmost inside argument bound to leftmost sub-program argument etc.); and if the number of arguments do not match, either arguments are "discarded" or given "dummy" (unbound) values, as discussed in 5.1.

5.2.1 First Strategy—Hiding *Make* Inside the Combinator Implementation

By inserting the *make* operator into the implementation of all combinators a simple strategy for hiding the *make* operator is obtained. We will look at the *and* combinator as an example:[12]

```
1   apply(and(P, Q), [X]) :-
2           apply(make([1], P), [X]),
3           apply(make([1], Q), [X]).
4   apply(and(P,Q), [X1, X2]) :-
5           apply(make([1, 2], P), [X1, X2]),
6           apply(make([1, 2], Q), [X1, X2]).
7   apply(and(P, Q), [X1, X2, X3]) :-
8           apply(make([1, 2, 3], P), [X1, X2, X3]),
9           apply(make([1, 2, 3], Q), [X1, X2, X3]).
10  /* Etc... */
```

[12]This strategy is illustrated in the example program in appendix A.

By defining a basic set of *make* definitions, the Combilog implementation can both remove arguments from the list sent to the combinator as well as add dummy-arguments which are never used. Recall that every *make* construct that we use must be defined. *Make* constructs are distinguished by the index list and the number of outside arguments of the wrapped program; however, in this case we cannot know how many outside arguments the programs (`P` and `Q`) have. We will therefore ascertain that we have defined all make constructs up to a maximum possible number of arguments.[13] The following listing provides an example:

```
1   apply(make([1], P), [X1]) :-
2           apply(P, [X1]).
3   apply(make([1], P), [X1]) :-
4           apply(P, [X1, X2]).
5   apply(make([1], P), [X1]) :-
6           apply(P, [X1, X2, X3]).
7           /* Etc... */
8
9   apply(make([1, 2], P), [X1, X2]) :-
10          apply(P, [X1]).
11  apply(make([1, 2], P), [X1, X2]) :-
12          apply(P, [X1, X2]).
13  apply(make([1, 2], P), [X1, X2]) :-
14          apply(P, [X1, X2, X3]).
15          /* Etc... */
16
17  apply(make([1, 2, 3], P), [X1, X2, X3]) :-
18          apply(P, [X1]).
19  apply(make([1, 2, 3], P), [X1, X2, X3]) :-
20          apply(P, [X1, X2]).
21  apply(make([1, 2, 3], P), [X1, X2, X3]) :-
22          apply(P, [X1, X2, X3]).
23          /* Etc... */
```

5.2.2 Second Strategy—Using *Make* Inside the Program Definitions

Hiding *make* inside the combinator definitions makes the source code more readable; however, there are some drawbacks. It is more difficult to see what goes on in the program. Furthermore, there are unwanted side-effects to this strategy—one example being that combinators cannot be nested, because that results in rather nasty nested *make* constructs such as

```
1   (make([1, 2, 3], make([3, 1], someProgram))
```

[13] In a real-world implementation, this could be done automatically, "on the fly", by a code generator.

which result in bugs when the program executes. This problem can be avoided by prohibiting nested combinators—every combination would then need to be defined as a named program of its own. However, it is not an ideal situation.

Furthermore, whether the source code is readable or not should not be of significant importance when we have arrived at a visualization viable for use in a visual integrated development environment (IDE). Programming will then be made *visually*, not *textually*. All the same, a default mapping of arguments would be convenient even in a visual environment—relieving the programmer of tedious tasks. The programmer would then when needed visually reorder the arguments, but only when the left-to-right strategy discussed above is not the desired mapping of arguments. Therefore we will devise a second strategy for dealing with mapping of arguments, which we will propose for a visual CRP IDE. In this strategy, every program is "wrapped" inside *make*, with the default left-to-right argument mapping. This is visually shown in the IDE. If a re-ordering of arguments is to be made, this is made visually in the IDE and the list of indeces for the corresponding *make* wrapper in the source code is changed. The following listing provides an example of the default left-to-right argument mapping with *make* used inside the program definitions:

```
1  apply(firstProgram, [Arg1]) :-
2          /*Implementation of firstProgram*/
3  apply(secondProgram, [Arg1, Arg2, Arg3]) :-
4          /* Implementation of secondProgram */
5  apply(thirdProgram, [X1, X2]) :-
6          apply(and(make([1, 2], firstProgram),
7          make([1, 2], secondProgram)), [X1, X2]).
```

5.3 "User-Friendly" Recursion Operators

The *make* operator also needs to be hidden in the recursion operators. The programmer should not have to consider how to apply *make* to make the recursion operators work properly. On the other hand, all recursion in Combilog should be based on fundamental recursion schemes, for which important theorems (see 3.4) have been proven. Therefore, more "user-friendly" recursion operators need to be constructed on top of the basic recursion schemes. In these "user-friendly" recursion operators, appropriate application of *make* takes care of delivering the right arguments to the right programs in the lower-level recursion operators. The following listing exemplifies this:

```
1  /*
2  foldr.
3  Basic recursion scheme (level 0)
4  */
5  apply(foldr(P, Q), [[], Y, Z]) :-
6          apply(Q, [Y, Z]).
7  apply(foldr(P, Q), [[X | T], Y, W]) :-
8        apply(foldr(P, Q), [T, Y, Z]),
```

```
9         apply(P, [X, Z, W]).
10
11  /*
12  More specific recursion, needed when the recursion program
13  needs both the head and the tail of the list.
14  (level 1)
15  */
16   apply(natrec(P, Q), [X, Y]) :-
17           apply(foldr(p(P), q(Q)), [X, _, [Y, _]]).
18  apply(p(P), [X, [V, T], [W, [X | T]]]) :-
19           apply(make([1, 2, 3], P), [[X | T], V, W]).
20  apply(q(Q), [_, [V, []]]) :-
21           apply(make([1], Q), [V]).
22
23  /*
24  User-friendly recursion operator (level 2)
25  */
26    /* Recursion operator which applies P
27    to the head and the tail of List in each step */
28  apply(foreachNatrec(P), [List]) :-
29           apply(make([1], natrec(make([1], P), true)), [List]).
```

5.4 Negation

Although theoretically not necessary, for practical reasons we introduce an operator—*not*—which takes a program and creates its negation. Negation is a complicated matter in relational programming, and the matter of how negation should be handled in CRP should be investigated further. For the time being, we propose that the *not* operator be implemented as *negation as failure*, i.e. that if a program P fails, its negation not(P) succeeds. The following listing shows the Combilog-Prolog definition of *not*:

```
1  %Cancel out double negation
2  apply(not(not(P)), ArgList) :- apply(P, ArgList).
3  %Negation as failure
4  apply(not(P), ArgList) :- \+ apply(P, ArgList).
```

5.5 A Visual Model for CRP

We will now propose a visual model for CRP. We have not performed any usability study of the proposed visual model, nor have we considered other alternative models. Let us again stress that our aim is to show that the proposed model can visualize programs, and that this model can be used for visual programming.

5.5.1 General Structure of Program Symbols

We let a CRP program have the general structure of a simple rectangular box. The program's name is written at the top of the box. In a future visual IDE we will have a button named "i", which will show the documentation for the program when clicked upon. We let the program display a number of "electrical wall sockets", connections that represent the outside arguments. Let us henceforth call these connections *sockets*. These sockets for the outside arguments will have the argument names written on them. If we do not show how a CRP program is constructed internally, its symbol will not contain anything else except its name, the information button (documentation) and sockets for outside arguments. We will call this representation a *closed box*. (See figure 5.)

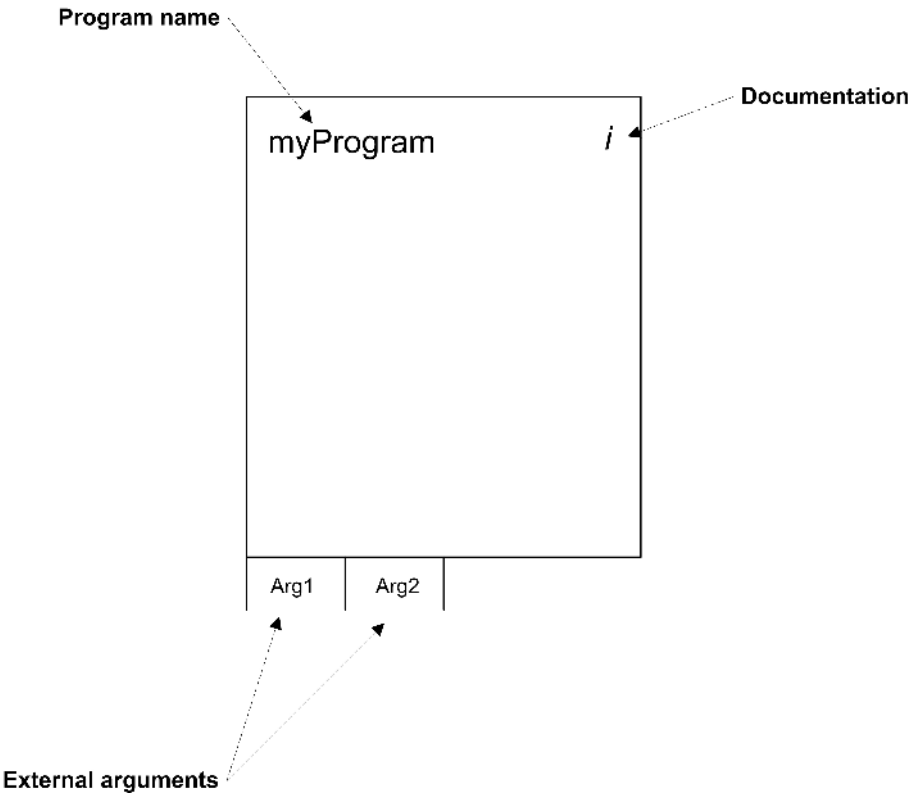

Figure 5: General structure of a CRP program symbol. In this closed box form, we do not show how the program is constructed internally. We only depict what the program exposes to the outside world: program name, documentation and sockets for outside arguments.

5.5.2 Basic Programs

Basic programs will not contain anything else than outside argument sockets, program name and documentation. They do not contain other programs inside; thus they are always represented as closed boxes. Something that should be considered is whether other programs than the fundamental basic programs (see 3.6) should be available as basic

programs—some candidates for this are *member* (member of a list), *length* (length of a list), and arithmetic operators (+, - etc.).

5.5.3 Composed Programs

We let composed programs have additional inside arguments that will be represented by internal sockets on the inside of the program. They can optionally be given hard-coded constants directly—these will then simply be written inside the socket. Every outside argument will also be an inside argument, which is visually shown by letting the outer sockets extend to the inside just as the sockets for the inside arguments. (See figure 6.)

Composed programs will contain subprograms, which are other programs combined by a combinator or a recursion operator. The mapping of inside arguments of the whole program module to outside arguments of each individual subprogram will be done by simply drawing a line from the inside argument to the appropriate outside argument of the subprogram. In a visual IDE we would implement an automatic default mapping according to the left-to-right strategy described in 5.2.2, which the programmer can change by rearranging the lines.

5.5.4 Combinator Programs

As can be seen in figure 6, we let a combinator program have subprograms, which are connected by a bar. On this bar the symbol representing the type of the combinator is drawn: the logical *and* symbol ($\wedge$) for the *and* combinator and the logical *or* symbol ($\vee$) for the *or* combinator. It is reasonable to believe that it would be convenient to be able to combine not just two, but three subprograms—to avoid having to create another program just for combining two of the three subprograms and then combining this new program with the third subprogram. We will include this "syntactic sugar" and use it in the example program in Appendix B. However, we believe that allowing for combination of more than three subprograms would not contribute to clarity—the number of lines for mapping the arguments would just be to large, resulting in a diagrammatic "cobweb". However, considerations like this should be evaluated in a usability study.

5.5.5 Recursive Programs

We let a recursive program have one recursion operator on the inside (see figures 7 and 8). This operator should be one of the "user-friendly" recursion operators proposed in 5.3. The recursion operator has a "saw-tooth" line at the top. The idea behind this is to visualize the *repetitive process* of recursion. We let the recursion operator have two subprograms inside—a subprogram to the left which is the *recursion program* (for the recursive case) and a subprogram to the right which is the *base program* (for the base case). A recursion operator can have one recursive outside argument, whose data type should be a recursive data type (e.g. a list). We let a recursive argument be placed leftmost (see figure 7). We believe that it is important to show that the recursive argument is of a different nature than normal arguments; it needs a special representation, so we let it have a "saw-tooth" line

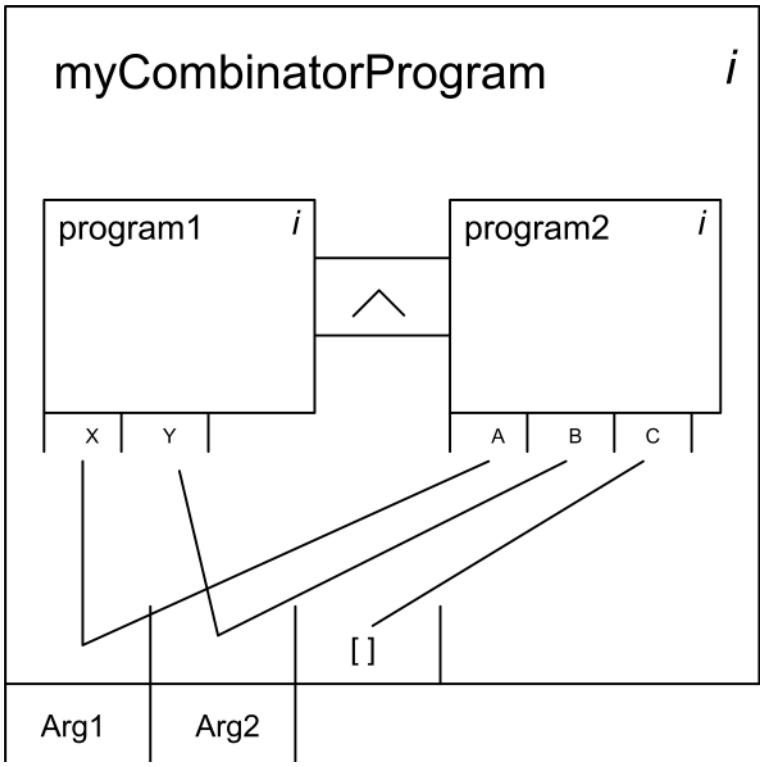

Figure 6: Symbol for a CRP combinator program. This program is an *and*-combination of two subprograms. It has two outside arguments, and one additional internal argument hard-coded as a constant, the empty list ([]). The mapping of the arguments is indicated by lines. The corresponding Combilog-Prolog code for this program is: "apply(myCombinatorProgram, [Arg1, Arg2]) :- apply(and(make([1, 2, 3], program1), make([1, 2, 3], program2)), [Arg1, Arg2, []])."

on its inside socket. Some recursion operators do not use a recursive list argument. In that case, there is no recursive argument; all arguments have the usual socket representation (see figure 8 that visualizes the "ancestor" program).

In some cases of recursion, there are two accumulator arguments that carry information during the recursion steps. The recursion program can describe the relation between the old and new accumulator in each step. When the next recursion step is called, the new accumulator is sent to the argument for the old accumulator. This process is visualized by a dashed arrow that shows how the new accumulator "jumps" to the place of the old accumulator (see figures 7 and 8).

When using a recursion operator, initially its outside arguments are bound to the arguments we give it; however, during the recursion steps, the arguments are bound differently, and when all recursion steps are finished, the initial bindings are valid again. We could think of a recursion operator as a "reactor" performing some kind of process. We give it some arguments, push a "Start" button, and when the process is finished, we get the arguments back again.[14] We will visualize this by letting the inside arguments of the program

[14]For a full, formal explanation of how the foldright and foldleft recursions are unfolded, see [15].

be connected to the outside arguments of the recursion operator by bi-directional arrows instead of lines (see figures 7 and 8).

When constructing the examples found in the Appendices, we found that the most complicated aspect was to get the recursion to work correctly (i.e. use foldright or foldleft and use make to get the right arguments to the right places). We presume that it will be too complicated for the ordinary programmer to use `foldr` and `foldl` directly; however, it is not necessary to understand this in order to construct CRP programs. When we had constructed a "user-friendly" recursion operator for the type of recursion that was needed, we found it easy to construct recursive CRP programs, and we believe that this will also be the case for an ordinary programmer. Therefore, the programmer should be provided with a whole palette of carefully chosen, welldocumented recursion operators, from which he or she can choose one that is "tailor-made" for solving the problem at hand. In a visual CRP IDE, the programmer could drag a recursion operator onto the design surface, and then drag and place two programs onto the recursion operator's placeholders for the recursion program and the basic program.

5.5.6 The *Make* Operator

We let the mapping of arguments—i.e. projection using the *make* operator—be visualized solely through the lines connecting the sockets for the arguments. This is fully sufficient, as every subprogram in a composed program will not appear in its pure form in the source code, but will be wrapped inside a *make* construct according to the mapping of arguments indicated by the lines connecting the sockets in the diagram.

5.5.7 The *Not* Operator

We let the *not* operator be visualized by a box with dashed lines and rounded corners; this box encloses the program to be negated. The logical symbol for negation (¬) is drawn on the box. (See figure 9).

5.5.8 Structure of CRP Diagrams

Subprograms in a composed CRP program are other CRP programs. Thus, given enough space to draw on, the internal implementation of the subprograms could be drawn inside the subprograms, recursively down to the basic programs. Theoretically, just as the CRP program in source code Combilog-Prolog form could be written on one line, the program could also be drawn completely in one diagram. For practical reasons—as in a visual IDE with limited computer screen space—we believe that in general, subprograms in a composed CRP program will be represented by closed boxes, and that these boxes are "opened up" and drawn on another page.

A CRP program has no connection directly to the inside of another program. This is in line with already well-established concepts in software engineering, notably for object-oriented software, where classes and modules communicate through specified interfaces and their internal implementation can stay hidden from the outside world. In our proposed

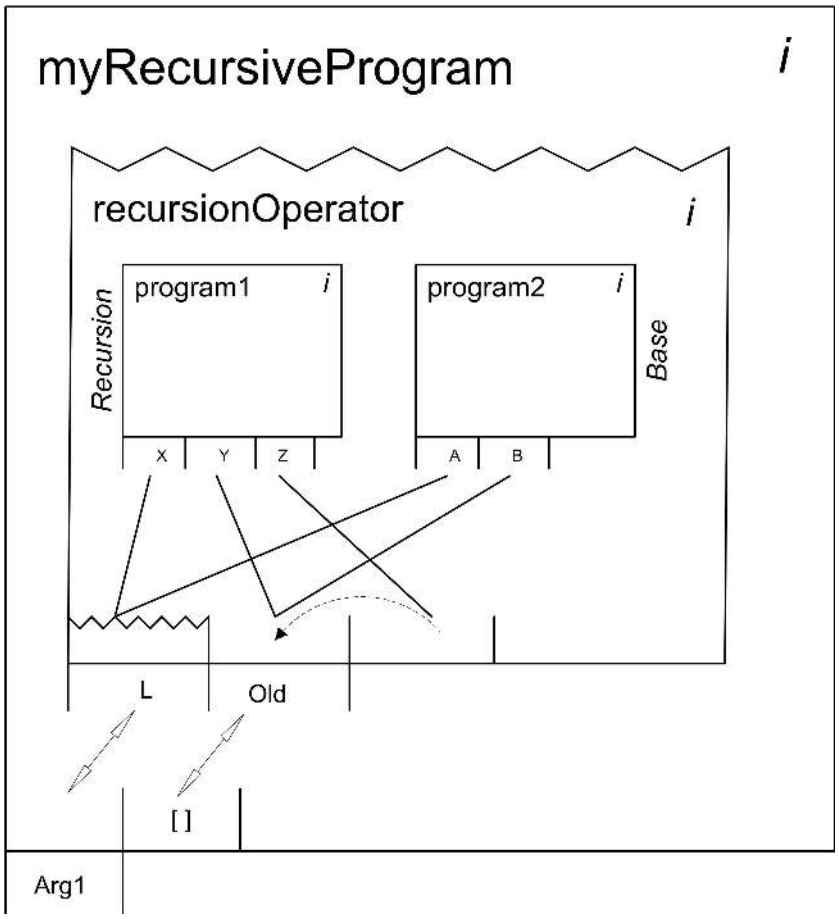

Figure 7: Symbol for a CRP recursive program. This program uses a recursion operator with two subprograms—base program and recursion program. It has a recursive outside argument, which is placed leftmost and represented by a "saw-tooth" line. The other outside argument is an accumulator argument. The internal argument holds the new value for the accumulator argument, and the dashed arrow shows how this new accumulator "jumps" to the place of the old accumulator in each successive recursion step.

visual CRP model, the subprograms in a larger program communicate only through their outside arguments, which in fact is the CRP counterpart for the object-oriented concept of an interface. Since each subprogram in a larger CRP program communicates exclusively through its outside arguments, there is no need for the level of detail present in the higraph model presented by Harel [21].

The visual model of CRP programs (the diagrams) shows—in one and the same diagram—a *complete* and *unambiguous* view of the program. This is in contrast to e.g. UML diagrams, where several diagrams show different views (static and dynamic) of the system. Furthermore, the modular structure of a CRP program is reflected—actually, it is inherent—in its visual representation. Thus, the classic notion of *decomposition* in problem solving is visualized. For instance, the visual representation of the program in the CRP diagrams shows

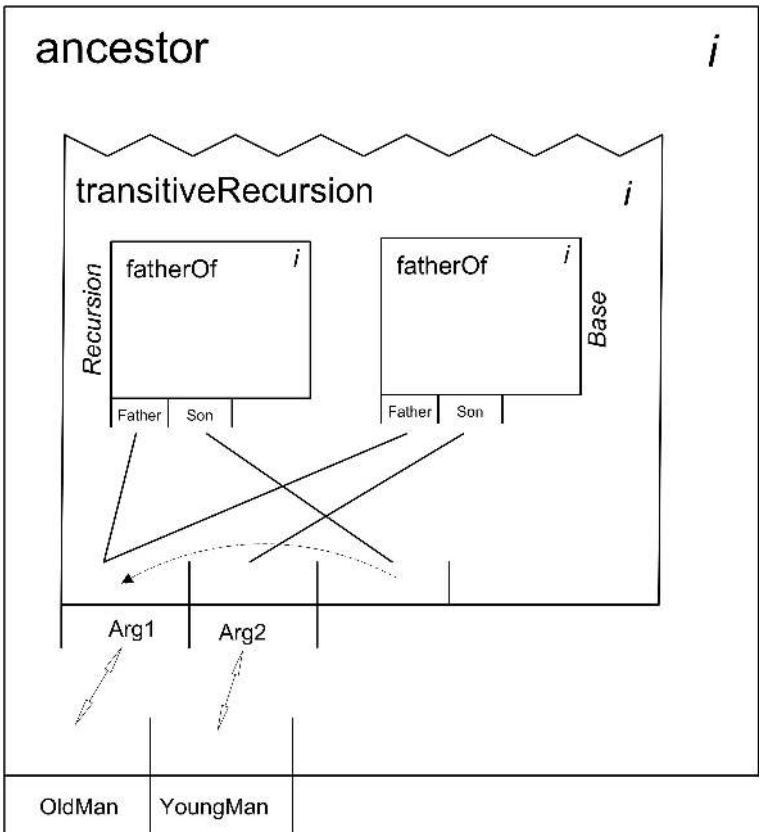

Figure 8: Symbol for a CRP recursive program without a recursive list argument. This program uses a recursion operator called `transitiveRecursion`, with the `fatherOf` program as both recursion program and base program. This visualizes nicely that in order to find out if `OldPerson` is ancestor of `YoungPerson`, we have to keep finding his son and son's sons etc, until we find someone who is the father of `YoungPerson`.

a top-down decomposition of a problem into smaller subproblems. Visual CRP programming can in fact be performed both bottom-up—starting with the basic building blocks, then combining them to ever larger program components—as well as top-down—starting with an empty box representing the whole program, then filling it with other empty boxes combined by combinators and operators, then filling these empty boxes etc. A usability study would shed some light on which of these approaches would be appropriate as a recommended CRP programming practice; however, when programming the example programs found in Appendices A and B, we found that the top-down approach was most adequate.

5.6 Automatic Code Generation

Since the semantical meaning of the diagrams is identical to the semantical meaning of the Combilog source code (e.g. the Combilog-Prolog source code used in the examples in this thesis), it would be possible to construct an automatic code generator—a program that

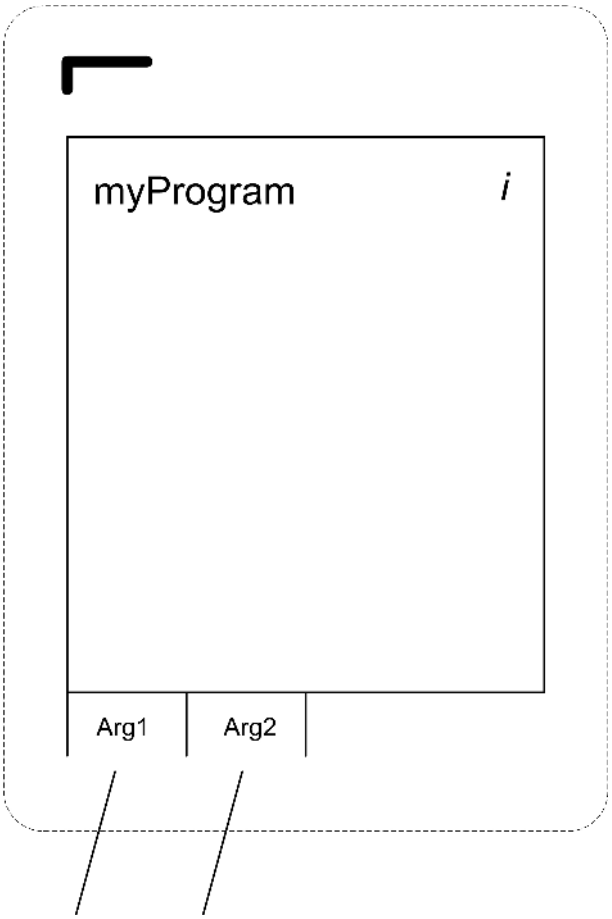

Figure 9: Symbol for a negated CRP program. The program is enclosed by a dashed-line box with rounded corners. The lines for mapping its outside arguments pass right through the box.

generates source code from the CRP diagrams. We will now show that this can be done. We will attempt an object-oriented approach, since the reader is probably most familiar with the object-oriented paradigm.

In our code generator the CRP programs are represented as objects. Every object knows how to produce its source code, using a polymorphic `SourceCode` property. There is a base class for all programs, with a virtual `SourceCode` property. There is a class hierarchy inheriting from this base class: a class for basic programs, another for programs combinating subprograms with the and/or-combinator, other classes for programs built with each available recursion operator, a class for programs wrapping other programs inside a *make* construct, as well as classes for definitions of combinators and operators. We will use C# for the code examples.

5.6.1 Basic Programs

The basic programs are the smallest building blocks of a CRP program—they do not contain other CRP programs and will always be visually represented by closed boxes. Thus, the `SourceCode` of a basic program is just a hard-coded string of text. The following listing shows the definition of the identity program:

```
1  apply(id, [X, X]).
```

We need such lines of source code defining the basic programs in every CRP program (at least for those basic programs which are used in the CRP program).

5.6.2 Wrapping Programs in *Make* Constructs

There is a class for programs wrapped inside a *make* construct. This class has an `IndexList` property for the index list (a list of integers). Its `Name` property could be implemented like this:

```
1  /// <summary>
2  /// The name of the program wrapped in make, e.g.\:
3  /// make([1, 2, 3], programName)
4  /// </summary>
5  public string Name
6  {
7      get
8      {
9          var builder = new StringBuilder();
10         builder.Append("make([");
11         builder.Append(IndexList.ToCommaSeparated<string>());
12         builder.Append("], ");
13         builder.Append(Program.Name);
14         builder.Append(")");
15         return builder.ToString();
16     }
17 }
```

5.6.3 Composed Programs

Composed programs produce source code that represents the subprograms and a combinator/operator. Accordingly, the object representing the composed program has properties for its subprograms (e.g. `Subprogram1` and `Subprogram2` for a and/or-combinator program and `BaseSubprogram` and `RecursiveSubprogram` for a recursive program). Every subcomponent to composed programs should first be wrapped in a *make* construct that represents the mapping of arguments shown with lines connecting the sockets in the diagrams. The following listing gives an example of how programs built with the *and/or* combinator could produce their source code:

```
1   //The following method would provide output like:
2   //"apply(myNewProgram, [Arg1, Arg2]) :-
3   //     apply(or(make([1, 2], anotherProgram),
4   //               make([1, 2], yetAnotherProgram)), [X1, X2])."
5   //
6   //(Subprogram1 and Subprogram2 would be ordinary
7   //      programs wrapped inside a "make" construct.)
8   //
9   public override string SourceCode
10  {
11      get
12      {
13          var builder = new StringBuilder();
14          builder.Append("apply(");
15          builder.Append(Name);
16          builder.Append(", [");
17          builder.Append(ExternalArgs.ToCommaSeparated<string>());
18          builder.Append("]) :- apply(");
19          builder.Append(Type);
20          builder.Append("(");
21          builder.Append(Subprogram1.Name);
22              builder.Append(", ");
23              builder.Append(Subprogram2.Name);
24          if (Subprogram3 != null)
25          {
26              builder.Append(", ");
27              builder.Append(Subprogram3.Name);
28          }
29          builder.Append("), [");
30          builder.Append(InternalArgs.ToCommaSeparated<string>());
31          builder.Append("]).");
32          return builder.ToString();
33      }
34  }
```

Programs consisting of a recursion operator with a base program and a recursive program produce source code using a similar strategy.

5.6.4 Adding Necessary Definitions

When using a *make* construct with a particular number of arguments and index list, an appropriate line of source code defining this needs to be added—but the programmer would not even have to be aware that the line is added. With this method of adding definitions

"on the fly" we avoid unnecessary large source code files. In the object-oriented automatic code generator, these lines could be added in the constructor of the classes that need such definitions, for instance:

```
1   /// <summary>
2   Constructor
3   /// </summary>
4   public Make(IEnumerable<int> indexList, Program program)
5   {
6       IndexList = indexList;
7       Program = program;
8
9       if (!ThisMakeAlreadyExists)
10      {
11          AddThisMake();
12      }
13  }
14
15  private void AddThisMake()
16  {
17      var thisMake = new MakeDefinition
18          (DefinitionName, IndexList, Program.NumberOfExArgs);
19      CRProgram.Instance.Makes.Add(thisMake);
20  }
```

6 Concluding remarks

We will conclude by discussing our design theory and its implications and point out some interesting areas for future research.

6.1 Conclusions

We have added some "syntactic sugar" to the Combilog language, proposed a diagrammatic model that visualizes CRP programs, and explored how source code can be automatically generated from the diagrams. The exploration of an automatic code generator constitutes a first analytical evaluation of our design theory. We have shown how diagrams are unambiguously created from code; with our code generator, we have shown that relationship also holds in the opposite direction: the textual representation of the programs can be automatically generated from the CRP diagrams. Therefore the visual CRP model supports both automatic forward and reverse engineering. We have a diagrammatic model that unambiguously represents the programs; there is a 1–1 relationship between the diagrams and the source code—they are isomorphic. Providing our code generator with a graphical user interface (GUI) would make visual CRP a reality.

6.2 Discussion

Unlike previous attempts at visualizing logic programming (see 4.7), the visual CRP model is fully compositional and compliant with well-established concepts of software engineering, such as modularity and encapsulation. It is reasonable to believe that this will be an advantage for programmers who are trained and experienced in procedural or object-oriented programming, which is the case for the majority of programmers in the world today. Visual CRP can achieve these advantages over previous attempts at visual logic programming because of the properties of the compositional-relational paradigm.

Visual CRP would not be "just another programming language". In fact, the visual CRP paradigm could be *independent of programming languages*. The visual programming model could be the same, but different output in form of source code text could be generated—it would suffice to implement a code generator that generates code in different programming languages. In this thesis, we used Combilog-Prolog as output language—however, the resulting source code could be written in any programming language. In this way, the same conceptual model for visual CRP could be used when the underlying technology changes, in accordance with the concept of *artifact mutability* [12].

Visual CRP programs could be represented in other forms as well as the visual form and various source code forms; an interesting option would be to represent programs in XML[15] form. The diagrams constituting a CRP program could be represented by XML tags, which would facilitate compatibility between different visual CRP implementations.

6.3 Implications and Future Work

If the proposed model for visual CRP programming would prove adequate in a usability study—in a real-world systems development setting—this would most probably have a significant impact on most aspects of systems development. The component-based structure—visualized in the CRP diagrams—together with the pure declarative CRP paradigm would, in our opinion, facilitate agile systems development in an ever-faster-changing world. Following important ideals of software engineering, the internal structure of a software component would remain completely hidden to the outside world. A component could be changed or replaced by another component with no risk of introducing errors. A major advantage of the visual CRP model is that the visualization of the software would not be separate from the software itself—it would represent the exact same thing. We also believe that for the human mind, thinking of software in a visual way would be easier than thinking of software as source code text. If we allow for some speculation, we could even imagine a systems development project where different teams use different visual CRP implementations, i.e. that the various CRP implementations generate source code in different programming languages.

Furthermore, CRP is a purely declarative paradigm. Procedural aspects—*how* the program should execute—do not disturb the programmer from focusing on *what* the program

[15] eXtensible Markup Language.

should do. Today, one increasingly important aspect is the ability of a program to execute in parallel. In CRP, parallelization would not be something the programmer has to consider—after all, executing in parallel is a question of *how* rather than *what*. Instead, the *CRP implementation* could be parallelized: for instance, programs built with a combinator and two subprograms would execute the two subprograms in parallel because the *implementation of the combinator* would be parallelized.

However, much work remains to be done in order to make visual CRP a viable tool for commercial systems development. Pure side-effect free programming is not enough; programs with side-effects need to be included and dealt with in the visual CRP model. Furthermore, we believe that in order to be adopted by the systems development industry, visual CRP has to co-exist with and communicate with existing object-oriented code (e.g. the .NET or the Java platforms)—just as F# is fully integrated on the .NET platform.

Fundamental recursion operators for double recursion would need to be devised, in order to provide adequate tools for dealing with common data structures such as binary trees. A duality theorem corresponding to the one proven for primitive recursive list relations (`foldr` and `foldl`) should be pursued. Additional "user-friendly" recursion operators need to be constructed on top of these new double recursion operators as well as the already existing foldright and foldleft—providing the programmer with a palette of adequate easy-to-use recursion operators tailored for the problem at hand.

In order to implement a commercially viable CRP environment, several other issues need to be considered as well. Concerning datatypes: should CRP be strongly typed (unlike the Combilog-Prolog implementation used in the examples in this thesis)? How should the visual CRP language be implemented—i.e. what should the output of the automatic code generator be? How should this implementation be optimized for efficiency? How should the visual IDE be designed—using color schemes, sound effects etc.?

We argue that the most promising attempt to make CRP viable for large-scale use in commercial systems development would be to implement it on the .NET platform. At first an implementation in F# seems natural, since many issues regarding side-effects and cooperation with existing object-oriented code have already been handled in that language.

References

[1] P. Abrahamsson, K. Conboy, and X. Wang, *'lots done, more to do': the current state of agile systems development research.*, European Journal of Information Systems, 18 (2009), pp. 281–284.

[2] J. Agustí, J. Puigsegur, and D. Robertson, *A visual syntax for logic and logic programming*, Journal of Visual Languages & Computing, 9 (1998), pp. 399 – 427.

[3] J. Brookshear, *Computer Science — an Overview*, Pearson Education, Boston, MA., 2008.

[4] P. Chen, *The entity–relationship model—toward a unified view of data*, ACM Transactions on Database Systems, 1 (1976), pp. 9–36.

[5] K. L. Clarke, *Negation as Failure*, Plenum Press, New York, USA, 1978, pp. 293–322.

[6] E. F. Codd, *A relational model of data for large shared data banks*, ACM, 13 (1970), pp. 377–387.

[7] A. Colmerauer, H. Kanoui, M. v. Caneghem, R. Pasero, and P. Roussel, *Un système de communication homme-machine en français*, Univ. Aix-Marseilles, II (1973).

[8] O.-J. Dahl, E. Dijkstra, and C. Hoare, *A case against the go to statement*, Communications of the ACM, 11 (1968).

[9] A. Dennis, B. Haley Wixom, and R. M. Roth, *Systems Analysis and Design*, John Wiley & Sons, Inc., Hoboken, NJ, USA, 2006.

[10] E. W. Dijkstra, *A case against the go-to statement*, Communications of the ACM, 11 (1968), pp. 147–148.

[11] L. Euler, *Lettres à une princesse d'Allemagne*, l'Academie Imperiale des Sciences, St. Petersburg, 1768.

[12] S. Gregor and G. Jones, *The anatomy of a design theory*, Journal of the Association for Information Systems, 8 (2004), pp. 312–335.

[13] A. Hamfelt and J. F. Nilsson, *Constructing logic programs with higher order predicates*, in Procs. of GULP-PRODE'95, the Joint Conference on Declarative Programming 1995, M. Alpuente and M. I. Sessa, eds., Università degli Studi di Salerno, 1995, pp. 307–312.

[14] A. Hamfelt and J. F. Nilsson, *Inductive metalogic programming*, in Procs. of Workshop on Inductive Logic Programming, S. Wrobel, ed., 237, Bad Honnef, Bonn, 1995, GMD-Studien.

[15] A. Hamfelt and J. F. Nilsson, *Declarative logic programming with primitive recursive relations on lists*, Maher, P. (ed.) Proceedings of the Joint International Conference and Symposium on Logic Programming, (1996), pp. 230–243.

[16] A. Hamfelt and J. F. Nilsson, *Towards a logic programming methodology based on higher-order predicates*, New Generation Computing, 15 (1997), pp. 421–448.

[17] A. Hamfelt and J. F. Nilsson, *Inductive logic programming with well-modedness constaints*, in The 8th Int. Workshop on Functional and Logic Programming, Grenoble, 1999, Laboratoire LEIBNIZ, Centre National de la Recherce Scientifique.

[18] A. Hamfelt and J. F. Nilsson, *Inductive synthesis of logic programs by composition of combinatory program schemes*, in Procs. of Workshop on Logic Based Program Transformation and Synthesis 1998, P. Flener, ed., vol. Lecture Notes in Computer Science of 1559, Springer Verlag, 1999.

[19] A. Hamfelt, J. F. Nilsson, and N. Oldager, *Logic program synthesis as problem reduction using combining forms*, Journal of Automated Software Engineering, 8 (2001), pp. 165–191.

[20] A. Hamfelt, J. F. Nilsson, and A. Vitoria, *A combinatory form of pure logic programs and its compositional semantics*, unpublished, http://www.anst.uu.se/andhamlt/pub/Combilog.ps, (1998).

[21] D. Harel, *On visual formalisms*, Communications of the ACM, 31 (1988), pp. 514–530.

[22] A. Håkansson, *Graphic Representation and Visualisation as Modelling Support for the Knowledge Acquisition Process*, PhD thesis, Uppsala University, Computer Science, 2003.

[23] A. Håkansson, L. Oestreicher, T. Jonsson, and A. Hamfelt, *Vicoll - a visual compositional logic language*, in Human-Centric Computing Languages and Environments, 2001. Proceedings IEEE Symposia on, 2001, pp. 394 –395.

[24] R. Kowalski, *The early years of logic programming*, Communications of the ACM, 3 (1988).

[25] C. Larman, *Applying UML and Patterns*, Pearson Education, Upper Saddle River, NJ, USA, 2005.

[26] R. C. Martin and M. Martin, *Agile Principles, Patterns, and Practices in C#*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[27] S. J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Jacoboson, Ivar.

[28] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, *Model-Driven Architecture*, vol. 2426/2002 of Lecture Notes in Computer Science, Springer Verlag, Berlin/Heidelberg, 2002, pp. 233–239.

[29] J. F. Nilsson, *Combinatory logic programming*, in Procs. of the 2nd Workshop on Metaprogramming in Logic, M. Bruynooghe, ed., K U Leuwen, april 1990, pp. 187–202.

[30] T. Petricek and J. Skeet, *Real-World Functional Programming—With examples in F# and C#*, Manning Publications, Greenwich, CT., 2010.

[31] P. V. Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*, MIT Press, Cambridge, MA, USA, 2004.

[32] J. Venn, *Symbolic logic*, McMillan, London, 1881.

Appendix A

An eight-queens solver in Combilog

This is an example program for solving the eight-queens problem, written in a Prolog-based implementation of Combilog. The program itself is found at line 83 ff. Lines 1–82 constitute the Combilog implementation.

```
1   apply(true, [_]).
2   apply(id, [X, X]).
3   apply(id(X), [X]) :- apply(id, [X, X]).
4
5   /*
6   Definitions of combinators
7   */
8
9   apply(and(P,Q), [X]) :-
10      apply(make([1], P), [X]), apply(make([1], Q), [X]).
11  apply(and(P,Q), [X1, X2]) :-
12      apply(make([1, 2], P), [X1, X2]),
13      apply(make([1, 2], Q), [X1, X2]).
14  apply(and(P, Q), [X1, X2, X3]) :-
15      apply(make([1, 2, 3], P), [X1, X2, X3]),
16      apply(make([1, 2, 3], Q), [X1, X2, X3]).
17  /* Etc... */
18
19  apply(or(P,Q), [X]) :-
20      apply(make([1], P), [X]); (make([1], Q), [X]).
21  apply(or(P,Q), [X1, X2]) :-
22      apply(make([1, 2], P), [X1, X2]);
23      apply(make([1, 2], Q), [X1, X2]).
24  apply(or(P, Q), [X1, X2, X3]) :-
25      apply(make([1, 2, 3], P), [X1, X2, X3]);
26      apply(make([1, 2, 3], Q), [X1, X2, X3]).
27
28  /* Definition of "make" */
29
30  apply(make([1], P), [X1]) :- apply(P, [X1]).
31  apply(make([1], P), [X1]) :- apply(P, [X1, X2]).
32  apply(make([1], P), [X1]) :- apply(P, [X1, X2, X3]).
33
34  apply(make([1, 2], P), [X1, X2]) :- apply(P, [X1]).
35  apply(make([1, 2], P), [X1, X2]) :- apply(P, [X1, X2]).
36  apply(make([1, 2], P), [X1, X2]) :- apply(P, [X1, X2, X3]).
```

```
37
38  apply(make([1, 2, 3], P), [X1, X2, X3]) :- apply(P, [X1]).
39  apply(make([1, 2, 3], P), [X1, X2, X3]) :- apply(P, [X1, X2]).
40  apply(make([1, 2, 3], P), [X1, X2, X3]) :- apply(P, [X1, X2, X3]).
41
42  apply(make([1, 3, 2], P), [X1, X3, X2]) :- apply(P, [X1, X2]).
43
44  apply(make([3, 1, 2], P), [X3, X1, X2]) :- apply(P, [X1, X2]).
45
46
47
48  /* Recursion operators */
49  apply(foldr(P, Q), [[], Y, Z]) :- apply(Q, [Y, Z]).
50  apply(foldr(P, Q), [[X | T], Y, W]) :-
51        apply(foldr(P, Q), [T, Y, Z]), apply(P, [X, Z, W]).
52
53  apply(foldl(P, Q), [[], Y, Z]) :- apply(Q, [Y, Z]).
54  apply(foldl(P, Q), [[X | T], Y, W]) :-
55        apply(P, [X, Y, Z]), apply(foldl(P, Q), [T, Z, W]).
56
57
58  apply(natrec(P, Q), [X, Y]) :-
59      apply(foldr(p(P), q(Q)), [X, _, [Y, _]]).
60  apply(p(P), [X, [V, T], [W, [X | T]]]) :-
61      apply(make([1, 2, 3], P), [[X | T], V, W]).
62  apply(q(Q), [_, [V, []]]) :- apply(make([1], Q), [V]).
63
64  /* Recursion operator that applies P to all elements in List */
65  apply(foreachRecursion(P), [List]) :-
66      apply(make([1], foldr(make([1, 2, 3], P), true)), [List]).
67
68  /* Recursion operator which applies P to the head and the tail of List */
69  apply(foreachNatrec(P), [List]) :-
70      apply(make([1], natrec(make([1], P), true)), [List]).
71
72  apply(append, [L1, L2, R]) :-
73      apply(foldr(cons, id), [L1, L2, R]).
74
75  /* This is not implemented using pure Combilog */
76  apply(member, [X, L]) :- member1(X, L).
77  member1(X, [X | L]).
78  member1(X, [Y | L]) :- member1(X, L).
79
```

```
80  apply(length, [List, R]) :-
81                length(List, R).
82
83  /*********************************************
84  EightQueens as a Combilog program
85  */
86
87  /*
88  (This uses Prolog built-in predicates)
89  */
90  apply(notInCheck, [X/Y, X1/Y1]) :-
91  X =\= X1, Y =\= Y1, abs(Y1-Y) =\= abs(X1-X).
92
93  /*
94  Succeeds if Queen does not hold X1/Y1 in check.
95  */
96  apply(notInCheckWith(Queen), [X1/Y1]) :-
97      apply(notInCheck, [Queen, X1/Y1]).
98
99  /*
100 Succeeds if X/Y is on the board (8x8 squares)
101 */
102 apply(isOnBoard, [X/Y]) :-
103     apply(and(make([1, 3, 2], member),
104     make([3, 1, 2], member)), [X, Y, [1,2,3,4,5,6,7,8]]).
105
106  /*
107  Succeeds if Queen is not in check with any queen in Queens.
108  */
109  apply(nocheck, [Queen, Queens]) :-
110                 apply(foreachRecursion(notInCheckWith(Queen)), [Queens]).
111
112  /*
113  Succeeds if Queen is on the board an not in check
114  with any queen in Rest.
115  */
116 apply(queenOnBoardAndOkWithRest, [[Queen | Rest]]) :-
117     apply(and(isOnBoard, nocheck), [Queen, Rest]).
118
119 /*
120 Succeeds if Queens is a legal placement of queens on the board.
121 */
122 apply(legal, [Queens]) :-
```

```
123       apply(foreachNatrec(queenOnBoardAndOkWithRest), [Queens]).
124
125   /*
126   Succeeds if Queens is a valid placement of 8 queens on the board.
127   */
128   apply(eightQueens, [Queens]) :-
129       apply(and(length, legal), [Queens, 8]).
```

Appendix B

A matching parentheses parser

Now we will look at another example program in CRP. First we will show it visually, using the proposed visual model. Subsequently we will give the corresponding Combilog-Prolog code. In the code example, the program itself is found at line 111 ff. Lines 1–110 constitute the Combilog implementation.

Visual representation

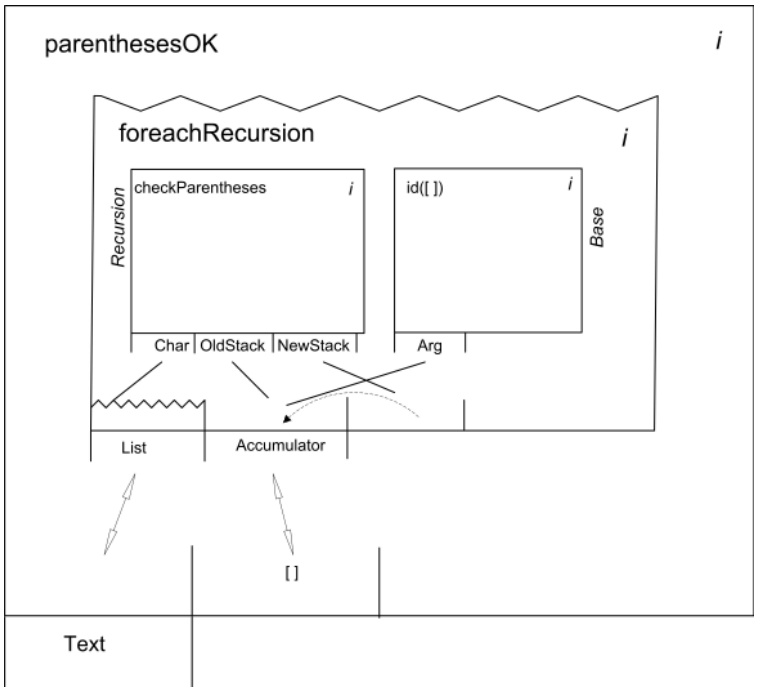

Figure 10: parenthesesOK. The top-level program. (The corresponding code is found at line 145.)

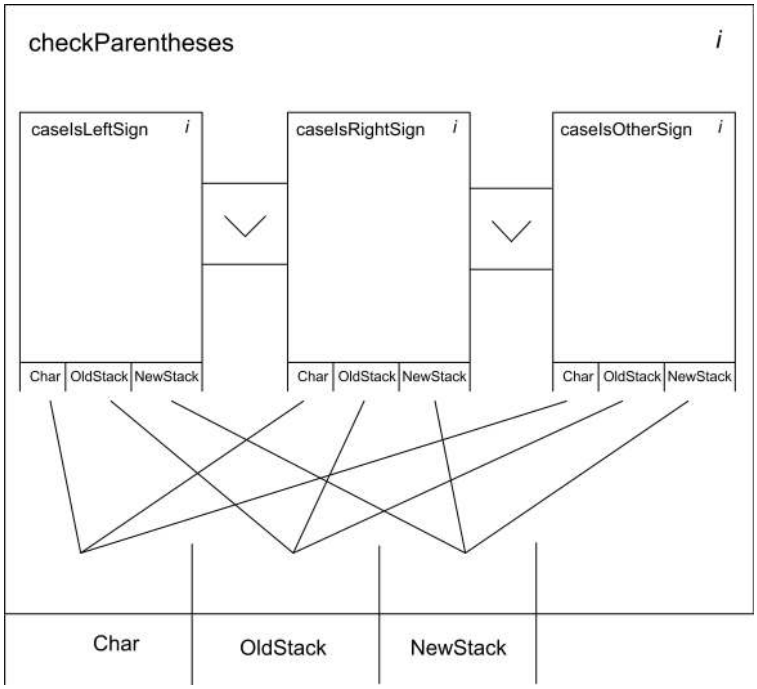

Figure 11: checkParentheses. One of the subprograms to parenthesesOK. (Line 136.)

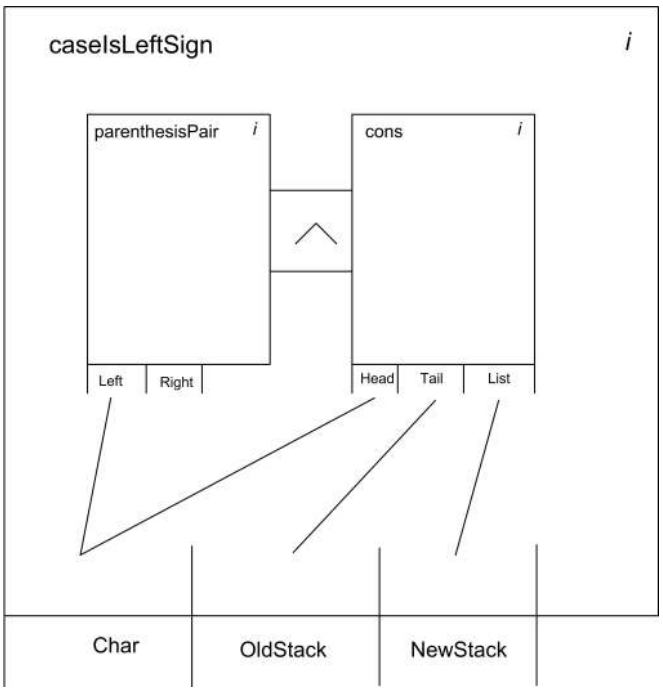

Figure 12: caseIsLeftSign. One of the subprograms to checkParentheses. (Line 129.)

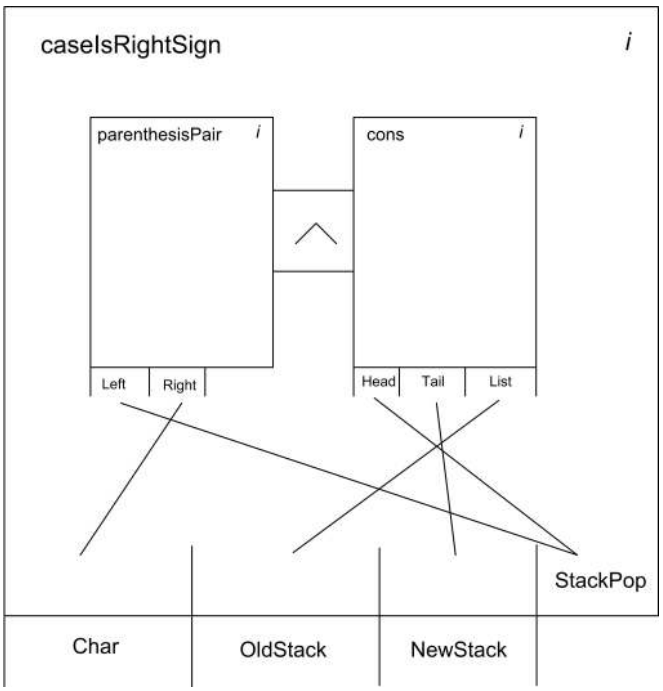

Figure 13: caseIsRightSign. One of the subprograms to checkParentheses. (Line 126.)

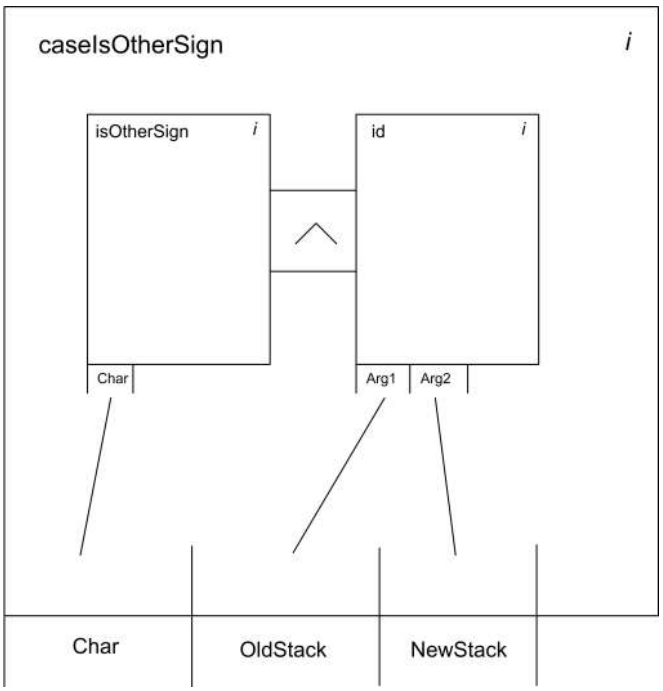

Figure 14: caseIsOtherSign. One of the subprograms to checkParentheses. (Line 132.)

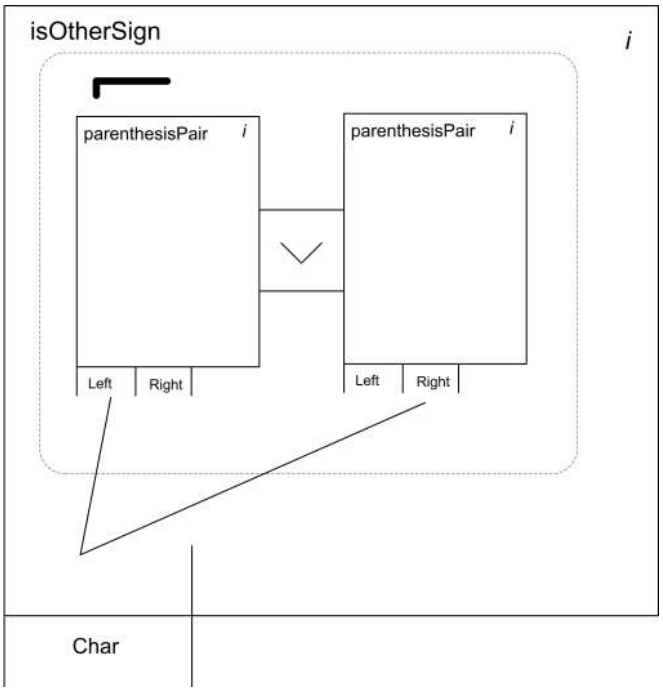

Figure 15: isOtherSign. One of the subprograms to caseIsIOtherSign. (Line 122.)

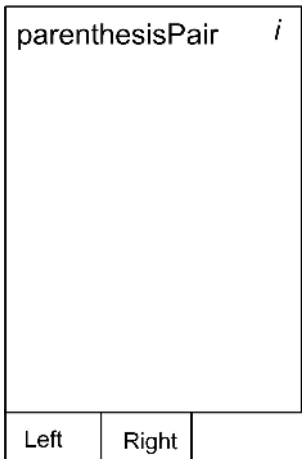

Figure 16: parenthesisPair. This is a basic program stating facts, namely which parenthesis pairs we handle in our matching parenthesis parser. (Lines 118–120.)

Textual representation

```
1   apply(true, [_]).
2   apply(id, [X, X]).
3   apply(id(X1), [X2]) :- apply(id, [X1, X2]).
4   apply(cons, [Head, Tail, [Head | Tail]]).
5   apply(not(not(P)), ArgList) :- apply(P, ArgList).
6   apply(not(P), ArgList) :- \+ apply(P, ArgList).
7
8   /*
9   Definitions of combinators
10  */
11
12  apply(and(P, Q), ArgList) :-
13      apply(P, ArgList), apply(Q, ArgList).
14  apply(and(P, Q, R), ArgList) :-
15      apply(P, ArgList), apply(Q, ArgList), apply(R, ArgList).
16
17  apply(or(P, Q), ArgList) :-
18      apply(P, ArgList); apply(Q, ArgList).
19  apply(or(P, Q, R), ArgList) :-
20      apply(P, ArgList); apply(Q, ArgList); apply(R, ArgList).
21
22  /* Implementation of "make" */
23
24  /*
25  General "makes"
26  */
27  apply(make([1], P), [X1]) :- apply(P, [X1]).
28  apply(make([1], P), [X1]) :- apply(P, [X1, X2]).
29  apply(make([1], P), [X1]) :- apply(P, [X1, X2, X3]).
30  apply(make([1], P), [X1]) :- apply(P, [X1, X2, X3, X4]).
31
32  apply(make([1, 2], P), [X1, X2]) :- apply(P, [X1]).
33  apply(make([1, 2], P), [X1, X2]) :- apply(P, [X1, X2]).
34  apply(make([1, 2], P), [X1, X2]) :- apply(P, [X1, X2, X3]).
35  apply(make([1, 2], P), [X1, X2]) :-
36      apply(P, [X1, X2, X3, X4]).
37
38  apply(make([1, 2, 3], P), [X1, X2, X3]) :- apply(P, [X1]).
39  apply(make([1, 2, 3], P), [X1, X2, X3]) :- apply(P, [X1, X2]).
40  apply(make([1, 2, 3], P), [X1, X2, X3]) :-
41      apply(P, [X1, X2, X3]).
```

```
42  apply(make([1, 2, 3], P), [X1, X2, X3]) :-
43      apply(P, [X1, X2, X3, X4]).
44
45  apply(make([1, 2, 3, 4], P), [X1, X2, X3, X4]) :- apply(P, [X1]).
46  apply(make([1, 2, 3, 4], P), [X1, X2, X3, X4]) :- apply(P, [X1, X2]).
47  apply(make([1, 2, 3, 4], P), [X1, X2, X3, X4]) :-
48      apply(P, [X1, X2, X3]).
49  apply(make([1, 2, 3, 4], P), [X1, X2, X3, X4]) :-
50      apply(P, [X1, X2, X3, X4]).
51
52  /*
53    Specific "makes"
54  */
55  apply(make([2], P), [X2]) :- apply(P, [X1, X2]).
56  apply(make([2], P), [X2]) :- apply(P, [X1, X2, X3]).
57  apply(make([1, 3], P), [X1, X3]) :- apply(P, [X1, X2, X3]).
58  apply(make([2, 1], P), [X2, X1]) :- apply(P, [X1]).
59  apply(make([2, 1], P), [X2, X1]) :- apply(P, [X1, X2]).
60  apply(make([2, 1], P), [X2, X1]) :- apply(P, [X1, X2, X3]).
61  apply(make([2, 3], P), [X2, X3]) :- apply(P, [X1, X2, X3]).
62  apply(make([1, 3, 2], P), [X1, X3, X2]) :- apply(P, [X1, X2, X3]).
63  apply(make([1, 3, 2], P), [X1, X3, X2]) :- apply(P, [X1, X2]).
64  apply(make([3, 2, 1], P), [X3, X2, X1]) :- apply(P, [X1, X2, X3]).
65  apply(make([3, 2, 1], P), [X3, X2, X1]) :- apply(P, [X1, X2]).
66  apply(make([2, 1, 3], P), [X2, X1, X3]) :- apply(P, [X1]).
67  apply(make([2, 1, 3], P), [X2, X1, X3]) :- apply(P, [X1, X2, X3]).
68  apply(make([3, 1, 2], P), [X3, X1, X2]) :- apply(P, [X1, X2, X3]).
69  apply(make([4, 3, 2, 1], P), [X4, X3, X2, X1]) :-
70      apply(P, [X1, X2, X3, X4]).
71  apply(make([2, 3, 4, 1], P), [X2, X3, X4, X1]) :- apply(P, [X1, X2]).
72  apply(make([4, 3], P), [X4, X3]) :- apply(P, [X1, X2, X3]).
73  apply(make([3, 4, 1], P), [X3, X4, X1]) :- apply(P, [X1, X2, X3]).
74
75
76  /* Recursion operators */
77
78  apply(foldr(P, Q), [L, Y, Z]) :- apply(Q, [L, Y, Z]).
79  apply(foldr(P, Q), [[X | T], Y, W]) :-
80        apply(foldr(P, Q), [T, Y, Z]), apply(P, [X, Z, W]).
81
82  apply(foldl(P, Q), [L, Y, Z]) :- apply(Q, [L, Y, Z]).
83  apply(foldl(P, Q), [[X | T], Y, W]) :-
84        apply(P, [X, Y, Z]), apply(foldl(P, Q), [T, Z, W]).
```

```
85
86
87  /*
88  User-friendly recursion operators
89  */
90
91  /*
92  Recurse over a list, using an accumulator argument during the recursion.
93
94  Arguments:
95  List: List to recurse over
96  Accumulator: an accumulator argument which changes during the recursion
97  (hence OldAccumulator and NewAccumulator for predicate P)
98
99  Predicates:
100 P: Predicate with 3 arguments(Element, OldAccumulator, NewAccumulator).
101 Q: Base case predicate with 1 argument, to be applied to Accumulator.
102
103 */
104 apply(foreachRecursion(P, Q), [List, Accumulator]) :-
105     apply(foldl(make([1, 2, 3], P),
106         and(make([1, 2, 3], id([])),
107             make([2, 1, 3], Q))), [List, Accumulator, NewAccumulator]).
108
109
110
111 /*******************************************
112 A matching parentheses parser in Combilog:
113 */
114
115 /*
116 The allowed parenthesis signs, mapped to each other
117 */
118 apply(parenthesisPair, ["(", ")"]).
119 apply(parenthesisPair, ["{", "}"]).
120 apply(parenthesisPair, ["[", "]"]).
121
122 apply(isOtherSign, [Char]) :-
123     apply(not(or(make([1], parenthesisPair),
124         make([2], parenthesisPair))), [Char]).
125
126 apply(caseIsRightSign, [Char, OldStack, NewStack]) :-
127     apply(and(make([2, 3, 4, 1], parenthesisPair),
```

```
128           make([4, 3, 2, 1], cons)), [Char, OldStack, NewStack, StackPop]).
129   apply(caseIsLeftSign, [Char, OldStack, NewStack]) :-
130       apply(and(make([1, 3, 4], parenthesisPair),
131           make([1, 2, 3], cons)), [Char, OldStack, NewStack]).
132   apply(caseIsOtherSign, [Char, OldStack, NewStack]) :-
133       apply(and(make([1, 2, 3], isOtherSign),
134           make([3, 1, 2], id)), [Char, OldStack, NewStack]).
135
136   apply(checkParentheses, [Char, OldStack, NewStack]) :-
137     apply(or(make([1, 2, 3], caseIsLeftSign),
138         make([1, 2, 3], caseIsRightSign),
139         make([1, 2, 3], caseIsOtherSign)), [Char, OldStack, NewStack]).
140
141   /*
142     Check whether parentheses match in a text,
143     eg. a C# code file.
144   */
145   apply(parenthesesOK, [Text]) :-
146       apply(foreachRecursion(checkParentheses, id([])), [Text, []]).
```