This is an author produced version of a paper presented at “The 23rd European Japanese Conference On Information Modelling and Knowledge Bases (EJC); 3-7 June 2013; Nara, Japan”. This paper has been peer-reviewed but may not include the final publisher proof-corrections or pagination.

Citation for the published paper:
Pacaci, Görkem and Hamfelt, Andreas
“A Visual System for Compositional Relational Programming”
Publication forthcoming in:
Frontiers in Artificial Intelligence and Applications, IOS Press
(http://www.frontiersinai.com/?q=conferences)
ISSN 0922-6389 (print)
ISSN 1879-8314 (online)

Link to conference website:
http://www.tt.cs.titech.ac.jp/EJC2013/

Access to the published version may require subscription.
A Visual System for Compositional Relational Programming

Görkem Paçacı, Andreas Hamfelt
Information Science, Department of Informatics and Media
Uppsala University
gorkem.pacaci@im.uu.se, andreas.hamfelt@im.uu.se

Abstract. Combilog is a compositional relational programming language that allows writing relational logic programs by functionally composing relational predicates. Higraphs, a diagram formalism is consulted to simplify some of the textual complexity of compositional relational programming to achieve a visual system that can represent these declarative meta-programs, with the final intention to design an intuitive and visually assisted complete development practice. As a proof of concept, an implementation of a two-way parser/visualizer is presented.

Keywords. Compositional Relational Programming, Combilog, Logic Programming, Declarative Programming, Visual Programming, Higraph

1. Introduction

It is a matter of debate whether diagramming is an appropriate method for accurate representation of proofs or theories. The subject is thoroughly analysed with examples in [20]. Shin concludes that diagrams are not only capable of precise visual articulation, they are also favourable for natural reasoning. Following from Shin, we believe application of diagrams to programming is still a path waiting to be exploited.

Recent interest in parallel programming has put declarative programming languages more in focus thanks to their natural characteristics such as modularity and parallelizability [11]. Combilog is a fundamentally relational programming language where programs are composed in meta level. For a thorough reader, details of this language with its abilities and concrete syntax are discussed in [9] and [2]. Although basic principles of this language are simple, its textual representation gets incrementally hard to follow mainly due to subsequent applications of the generalized projection operator (make operator) in a single expression. With this work, we aim to employ the natural reasonability of diagrams to replace or complement the textual representation of Combilog by building a new visual programming language. We believe that as a declarative programming language, Combilog is a strong candidate to benefit from being visualized.

In the following sections we briefly describe Combilog, and present the Higraph diagram formalism we use for the basis of the new visual language. Then we proceed to give an incremental definition of the language with examples.
1.1. Combilog

Since the developed visual system is used to express Combilog programs, first it is necessary to introduce the Combilog language. Building blocks of relational programs are predicates consisting of a name and a list of arguments. A relational programming language requires a way to begin with a few predefined predicates and produces new ones, and eventually, produces complete programs. For this purpose, Combilog follows from Predicate-Functor Logics presented and updated by W. Quine in [19] and [18]. Predicate-Functor Logic is the first-order logic without individual variables. It uses higher-level constructs to reorder, partly eliminate, or populate argument lists of predicates [3]. Combilog merges these argument list modification constructs to a single operator named `make`. The `make` operator is a generalized projection operator that takes two arguments: a list of zero-based indices and a source predicate (or another Combilog expression that can stand as a predicate). It reorders arguments of the source predicate according to the index list to produce a new predicate, adding new unbound arguments or eliminating existing arguments if instructed. Other operators Combilog provides include logic operators `and` and `or`. These logic operators take two or more predicates (or other Combilog expressions which can stand as a predicate) of the same arity\(^1\) and compose a new predicate of that arity which logically only holds true when all(`and`)at least one(`or`) of the component predicates hold. Expressions in Combilog programs are written by applying these (and more) operators in a nested fashion. [10] can be consulted to see how this provides a complete programming model. An example follows: \(^2\)

\[
\text{and}(\text{make}([1, 2, \_], p), \text{or}(\text{make}([\_ , 0, 1], r), k))
\]

In the expression above, \(p\), \(r\), and \(k\) are existing predicates in the context this expression lies. The first example of the `make` operator above takes the \(p\) predicate as source and produces a new ternary predicate whose first argument is bound to the second argument of \(p\), second argument is bound to the third argument of \(p\), and third argument is unbound. Note that only by looking at this expression there is no way to tell the arity of predicates \(p\) and \(r\). But it is certain that the matching `make` operators both will produce ternary predicates independent from the arity of the source predicates. The predicate \(k\) has to be ternary for the expression to be valid, since both predicates the `or` operator takes have to be of same arity.

1.2. Higraph

The Higraph is a flexible diagram formalism introduced by Harel in [12]. It represents sets and edges among sets in the form of a hyper-graph. Sets are represented by individual blobs and if a sets contour encompasses other sets then it includes those and only those sets. Atomic sets are defined as sets that do not contain another set. As a part of the formal definition of Higraph, two sets cannot be said to overlap even if their contours do, unless this visual overlap contains another set. Let us interpret the example higraph which is a state diagram from [12]:

\(^1\)Arity of a predicate is the length of its argument list.
\(^2\)In the index list in a `make` operator, we use `\_` (underscore) for indices that are intended to be unbound in the new predicate. We also use zero-based indices. These modifications slightly alter the convention in the original Combilog papers.
This higraph tells us that there are three atomic sets, A, B and C. The set D contains A and C, and B is contained by none. A significant feature in the higraph above is the existence of edges that connect sets with a label, and particularly the existence of hyper-edges that connect atomic sets to non-atomic ones.

2. Background

There are numerous examples in the literature on visual programming, and logic programming also has taken its share from these advances. [6] for example, introduces a visual language named Lograph where visual definitions reflect Prolog predicates. CUBE is another visual language designed in [17] and implemented in [16], also built as a higher-order logic programming language. A distinctive feature of CUBE programs are that they are visualized in 3 dimensional space. While studies like these focus on visualizing programs themselves, some like [14] and [4] work on visualizing execution processes of programs, primarily intending to assist debugging or education.

In a more general perspective, [5] surveys a nicely sampled list of visual programming languages. A comment from this work is that while visual languages are of much use at some aspects, basic text is still useful in a fashion where they complement each other. Similarly, [21] presents a review of Euler-based reasoning systems and diagrams that work with sets. This work is particularly of interest for our work since we also base our visual system on Higraph, introduced earlier, another set-based diagram formalism. Another example of visual systems that use a set-based system with Higraphs is [1] which maintains a one-to-one correspondence to Prolog code.

Similarities between our work and the work cited above exist such as use of Hi-graphs. This is due to the logical connection between set-based diagrams and a set-oriented view of extensions$^3$ of relational predicates.

As a recent example, [13] visualizes Python™ programs. While the visualization they present is straightforward, the overall design of the development practice is neatly complemented with visualization which appropriately suits their educational perspective.

Answer Set Programming is a strong recent trend in Logic Programming. [7] and the following [8] is an attempt to leverage visual programming in favour of ASP, providing an integrated development environment with a visual drawer to compose body graphs.

The closest example to this work is probably [22] where Combilog is the base language in question but the visualization attempt is a theoretical description of how Combilog programs can be visualized rather than a practical visualization application.

$^3$Extension of a predicate is a set of tuples that satisfy the predicate.
While attempts of visualizing functional programming or logic programming are
many, as listed above, visualization of a functionally compositional model still waits to
be fulfilled.

3. Visual representation

To align the Higraph notation with structures of Comblog expressions, it is necessary
to tailor it to a more strict definition. We do this by allowing only atomic sets to exist,
which represent arguments of a predicate. With a slight modification over Higraph, we
represent atomic sets as filled circles rather than rectangles with rounded corners, and we
omit labels that would otherwise lie inside the rectangles. Here we build a customized
higraph for the simplest of Comblog expressions, a single predicate:

![Higraph for a single predicate](image)

The tailored higraph above represents a ternary predicate named $p$. Rectangular set ob-
jects of Higraph are replaced with circles and they each represent an argument of $p$ in
the relational context. Following from this basic example, we move to one with two
predicates of different arities:

![Higraph for two predicates](image)

In the second example, two predicates are shown together. Predicate $p$ is the same as
in the first example and a new predicate $a$ of binary arity is introduced. It is possible
to tell that $a$ is binary because it has only one edge. The reason $a$ does not have its
arguments shown as separate circles is because they are bound with second and third
arguments of $p$. This way of binding arguments of different predicates together is the
main principle of how we compose new predicates. It is the visualization of the **make**
operator of Comblog. Since we wish to represent not only expressions but assignments
as well, it is necessary to clarify a point about the direction of composition. To distinguish
the newly produced predicate from component predicates we exclusively dedicate to it
the colour black (in contrary to gray), place it in the middle of the diagram while its
arguments are ordered from left to right. This also allows us to eliminate edges and
multiple labels for the produced predicate. Here is a more advanced example:

![Higraph for a more advanced example](image)

Diagram above shows that a new quaternary predicate named $p$ is being produced out of
existing predicates $a$, $b$ and $c$. The positioning and direction of the edges display that first
argument of $p$ is bound to the second argument of $b$, second argument of $p$ is bound to
both first arguments of $a$ and $b$, and the third argument of $p$ is bound to second argument
of $a$. First argument of $c$ is bound to second argument of $p$ and the rest is unbound.
$p$ also contains a fourth argument which is unbound. It is important to note that this
diagram above does not only contain three applications of the operator `make`, but it needs to contain an operator that combines these projections into one predicate. In Combilog, this is done by logic operators `and` and `or`. Assuming the composing operator is `and`, the textual code for the diagram above can be written as follows:

\[
\text{and}([1, 0, \_, \_], b), \text{make}([\_, 0, 1, \_], a), \text{make}([\_, 0, \_, \_], c))
\]

The diagram and code above use only one logical operator and use it in the top level. In fact, the representation we describe will always work for a single operation. Nested logic operators are treated as if they were ordinary predicates with names `and/or`. The following is an example of this case:

\[
\text{and}(\text{and}(a, b), \text{make}([1, 0], \text{or}(d, c)), \text{make}([0, 1], d))
\]

Visualization of this code actualized at the top-level `and` looks like this:  

```
\text{or} \quad \text{and} \quad \text{d} \quad \text{or}
```

The interpretation of this diagram reveals that a new binary predicate (let us call it \(p\)) is created. The middle two black circles represent the arguments of \(p\). These two arguments are bound to the arguments of a nested `and` in the same order. At the same time, they are bound to the first two arguments of a predicate \(d\), whose last argument is unbound. Another nested logic operator `or` exists. First argument of `or` is bound to the second argument of \(p\), second argument is bound to the first argument of \(p\) and the third is unbound.

For the user to reveal composition of a nested logic operator, he/she needs to browse into that operator for a visual representation of that level and scope. This is an intended simplification of visual content, since first aim of this visual system is to be used for an interactive visualizer or editor.

As an example of how subsequent projection operators render the textual representation hard to follow, consider this code:

\[
\text{or}(\text{make}([1, 2], \text{and}(\text{make}([0, 1, \_, \_], b), \text{make}([\_, 1, 0, 2], c))), \text{make}([1, 0], d))
\]

This expression above produces a new binary predicate. But it is not intuitive to look at the line above and tell, for example, which argument of \(c\) is bound to the second argument of this new predicate. Let us consider the 2-part visualization of this expression:

```
\text{and} \quad \text{d} \quad \text{or}
```

---

\(^4\)Combilog includes only binary logic operators, but here we use n-ary operators to increase the coverage of a single visual diagram. N-ary operators can trivially be defined in terms of binary ones.

\(^5\)At this point, introducing colours for different predicates is favourable for following multiple edges and arguments of a single predicate but this has been intentionally avoided for this written work. The implemented editor (presented later) utilises colours for this purpose.
As seen in two diagrams above, our approach visualizes this code in two levels. Firstly, the visualization of the top-level or is shown on the left. Looking at this diagram, it is straightforward to tell two components, a ternary component \(d\) and a binary composition \(and\) are being made into a new binary predicate. Browsing into\(^6\) the \(and\) composition reveals the diagram on the right, which shows this composition was a product of two predicates: a binary \(b\) and a ternary \(c\). Hence it becomes straightforward to tell which argument of \(c\) is bound to the second argument of the newly produced predicate. Only the circle that represents the first argument of \(c\) aligns with it when the second diagram is displayed right after the first one.

4. Implementation

A prototype of this visual system has been implemented to prove the concept. This prototype allows a user to type in the textual Combilog code to immediately see it rendered as a diagram. Similarly, it also allows the user to modify the diagram on screen by manipulating circles and edges to see the code regenerated accordingly. The interface employs methods to fluently integrate textual and graphical code, such as highlighting part of the text when matching graphical entity is hovered with pointer.

The prototype differs slightly in visualization from the diagrams above, mainly for easy interaction purposes such as highlighting the part of the code corresponding to the part of the diagram being pointed. But the fundamental principles as representation are the same, and can be observed at the screen-shot below.

The prototype containing the visualizer and parser is implemented on a Windows\(^{\text{TM}}\) system with XNA\(^{\text{TM}}\) development tools for graphical geometry generation and rendering with C#\(^{\text{TM}}\) programming language. A screen-shot of the implemented prototype:\(^7\)

\(^6\)Browsing into a component is a matter of interaction, where details are not in the scope of this paper. Clicking with a mouse may be an example of this action.

\(^7\)A video that shows how prototype works can also be viewed at URL http://vimeo.com/50700334, Access password: uppsala, Last accessed: November 2012.
The prototype provides a split-view for separately displaying text and diagram on the same screen. The part of the textual code that is being visualized with the diagram at any point is displayed immediately under the diagram as well.

5. Conclusion

After a brief survey of similar visual languages, Visual Combilog is presented. Building a visual diagram of this sort to represent compositional relational programs is described step-by-step. Following from this visual system, a prototype has been implemented as a proof of concept, including a parser that translates the textual code to visual and a generator that translates the visual code to textual.

We believe there are many advantages of a visual language, such as helping the user to follow and comprehend the code as mentioned earlier. There are also practical advantages in creating diagrams where the very nature of a visual diagram eliminates some of the syntactic errors as discussed in [20] by Shin.

As noted in the Background section, our work includes some similarities with the work done in adjacent fields in the way that we also use Higraphs as a fundamental diagram formalism. On the other hand, Visual Combilog focuses on the compositional aspect where the visualisation is more about the modelling of composition than modelling the final logic programs. We also try to establish a visual system where the number of distinct visual constructs and overall visual complexity are minimal, which is easier compared to other presented work thanks to the variable-free nature of Combilog. We believe this approach will enable us to experiment with more mediums, interaction styles and richer options in incorporating with text.
5.1. Agenda

We will be focusing our work on systems with split-view editors, diagram based notations to complement such systems and particularly on intrinsic features of compositional relational languages in this context.

It is necessary to provide empirical proof for competency of visual programming languages. [15] is a thorough guide for designing such experiments to study usability of visual languages. Visual Combilog as a language is new, and one of the short-term aims is to design a set of experiments to evaluate the visual system in the hands of people outside the development group.

References


