PreVIEW: A Model for Supporting Higher-order, Generative Programming in Visual Languages

Gregory Malecha
gmalecha@rice.edu

Walid Taha
taha@rice.edu

RICE

Introduction

Despite the popularity of visual programming languages in many fields, the tools used to develop systems in these languages have not received as much attention as more traditional textual programming languages have received. This is especially true in the aspects of static type systems and program verification tools. We look to develop a more direct correspondence between the textual and visual representations of programs in order to be able to use existing tools when working with graphical languages. This requires us to formalize a textual representation for graphs as well as systems which can translate between the two formats.

Problem

- Syntax needed for both the visual and textual language.
- Establish a direct correspondence between the textual and visual representations of programs.

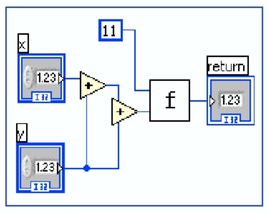
Approach

In order to make the correspondence between graph and text complete, we are developing an integrated development environment (IDE) which allows direct translation between the two representations as well as the ability to modify either representation. This required several problems to be addressed:

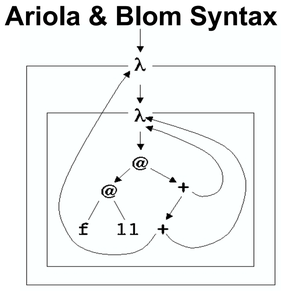
- The development of concrete syntax for the graphical and textual languages
- The implementation of algorithms to translate between the textual and graphical representations
- The development of a system for rendering the visual representation and allowing direct editing of its entities

Concrete Syntax

LabVIEW Syntax



Ariola & Blom Syntax



PreVIEW Syntax

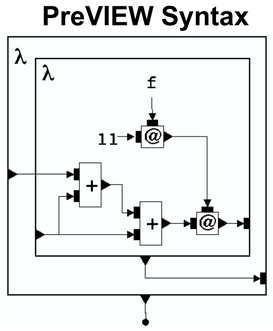
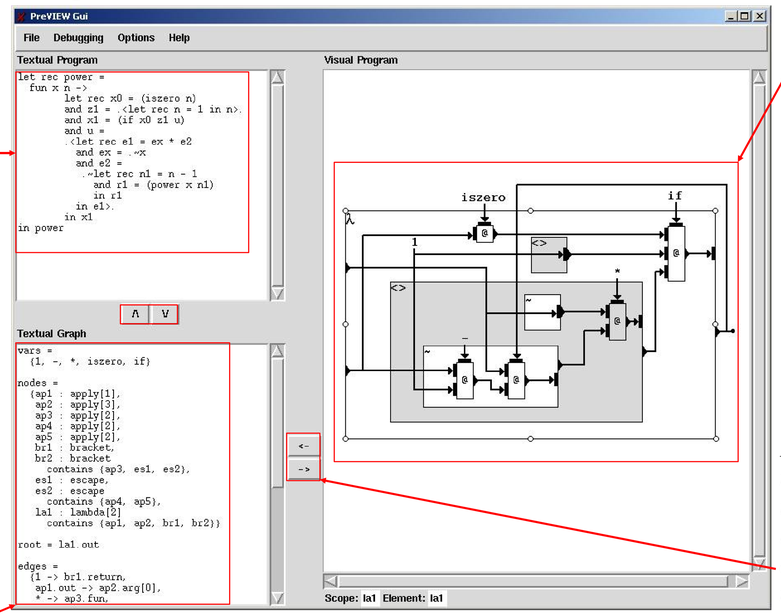


Figure 1: In choosing the syntax for the visual representation, we chose a mixture of the syntax developed by Ariola & Blom and the LabVIEW syntax. Note that the PreVIEW graph and the lambda-graph (Ariola & Blom) are dual graphs.

Translation Algorithms

The textual form of the program is written in a functional normal form in which all sub-computations are named. This allows unambiguous representations of programs as graphs.



The visual form of the graph allows direct editing with the mouse.

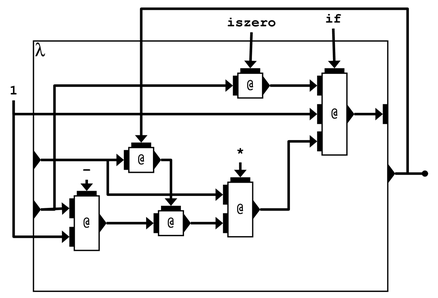
Arrows represent translations between the different representations

Textual representation of the graph provides an intermediate form which can be easily saved and efficiently expresses the data in the graph.

Multi-stage Programming and Program Generation

Generators allow programmers to write general algorithms in a natural way and then generate instances of those algorithm to handle special cases. This allows programmers to use abstraction without paying for it in runtime-overhead. An illustrative example of using multi-stage programming for program generation is exponentiation.

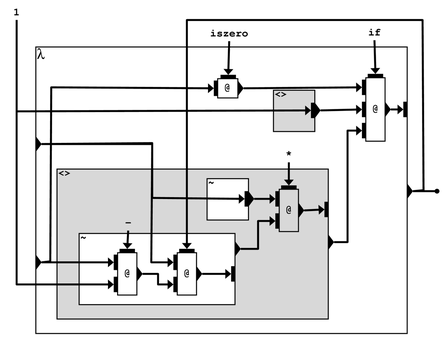
Unstaged Power Function




```
let rec power = fun x n ->
  if iszero? n then 1
  else
    x * (power x (n-1))
```

A simple implementation of power can be written in PreVIEW and visualized as a graph.

Power Function with Staging Annotations

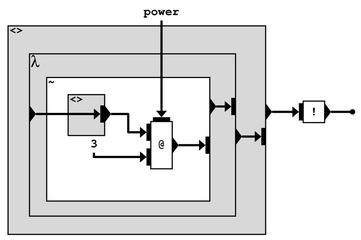



```
let rec power = fun x n ->
  if iszero? n then .<1>.
  else
    .<.~x * .~(power x (n-1))>.
```

PreVIEW adds multi-stage programming techniques which are shown in the graph using different shades of gray to show the escape level that a term is at.

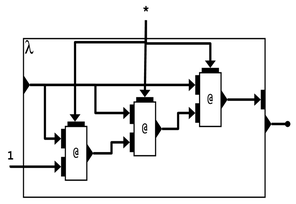
Visualizations generated from PreVIEW and manually laid out.

Generator for power(-,3)



Generates

Generated Code for Cube




```
let cube = .~fun x ->
  !(power .<x>. 3)
```

Once written with multi-stage annotations, the programming language can generate code for special cases of the function.

```
let cube = fun x ->
  1 * x * x * x
```

The interpreter or compiler generates a special form of the power function to compute the cube of a number. The generated code removes the inefficiencies allowing the programmer to utilize abstraction without paying a price at runtime.

Future Work

- More sophisticated graph layout using routing and positioning algorithms developed by Eschbach, Gunther, and Becker at VLSID '05
- Develop static type systems for the visual language
- Implementation of reduction semantics and execution engine
- Incremental translations between graphical and textual representations