# A Graphical Multi-stage Calculus

Stephan Ellner
Rice University
Houston, TX, USA
besan@cs.rice.edu

Walid Taha
Rice University
Houston, TX, USA
taha@cs.rice.edu

## ABSTRACT

While visual programming languages continue to gain popularity in domains ranging from scientific computing to real-time systems, the wealth of abstraction mechanisms, reasoning principles, and type systems developed over the last thirty years is currently available mainly for textual languages. With the goal of understanding how results in the textual languages can be mapped to the graphical setting, we develop the visual calculus PreVIEW. While this calculus visualizes computations in dataflow-style similar to languages like LabVIEW and Simulink, its formal model is based on Ariola and Blom's work on cyclic lambda calculi. We extend this model with staging constructs, establish a precise connection between textual and graphical program representations, and show how a reduction semantics for a multi-stage language can be lifted from the textual to the graphical setting.

## 1. INTRODUCTION

Visual programming languages are finding increasing popularity in a variety of domains, and are often the preferred programming medium for experts in these domains. Examples include data-flow languages like LabVIEW [9, 14], Simulink [21], and Ptolemy [12], a wide range of hardware CAD design environments, spreadsheet-based languages such as Microsoft Excel, or data modeling languages such as UML. Compared to modern text-based languages, many visual languages are limited in expressivity. For example, while they are often purely functional, they generally do not support first-class functions. More broadly, the wealth of abstraction mechanisms, reasoning principles, and type systems developed over the last thirty years is currently available mainly for textual languages. Yet there is real need for migrating many ideas and results developed in the textual setting to the graphical setting.

A specific example can be found in the real-time and embedded setting, where staging constructs [20] and appropriate type systems have been used for generating digital circuits [11], and for distinguishing between the development and the deployment platforms for heap-bounded computations [19]. From the software engineering point of view, such multi-stage languages can provide the expressivity of a general-purpose language and the guarantees of a resource-bounded language. To apply concepts such as staging to mainstream visual languages, we are faced with the question of how to map the ideas and formalisms associated with such concepts from the textual to the graphical setting. One approach is to start with a general-purpose calculus that has both textual and graphical syntax, and then to study what extensions to one representation entail for the other.

But what is the right starting point? The visual programming research literature focuses largely on languages that are accessible to novice programmers and domain-experts, rather than general-purpose calculi. Examples include form-based [4] and spreadsheet-based [2, 8, 10] languages. Citrin et al. give a purely graphical description of an object-oriented language called VIPR [5] and a functional language called VEX [6], but the mapping to and from textual representations is only treated informally. Erwig [7] presents a denotational semantics for VEX using inductive definitions of graph representations to support pattern matching on graphs, but this style of semantics does not preserve information about the syntax of graphs, as it maps syntax to "meaning".

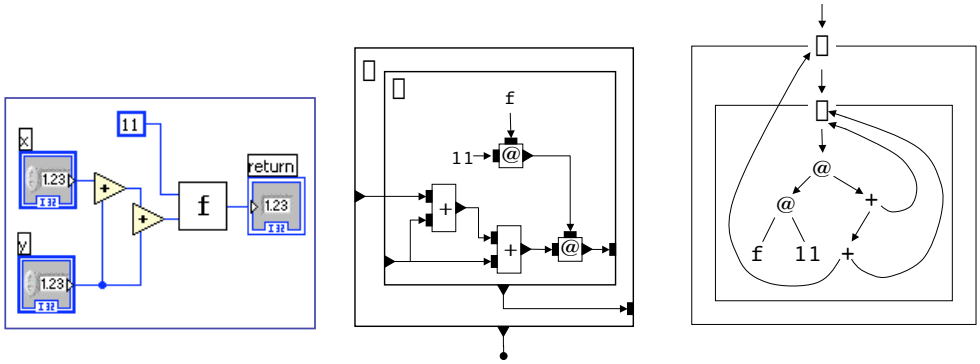
Ariola and Blom [3] provide precisely the starting point we need. Their work on cyclic lambda calculi establishes a formal connection between textual and graph-based program representations. The two representations are not one-to-one because of a subtle mismatch between textual and graphical representations in how they express sharing of values. However, Ariola and Blom define a notion of equivalence for terms that represent the same graph, and establish an isomorphism between graphs and equivalence classes of textual terms. While Ariola's lambda-graphs are essentially abstract syntax "trees", we adopt their formalism to model dataflow-style visual programs, and extend it with staging constructs.

### 1.1 Contributions

The starting point for this paper is the observation that Ariola and Blom's work not only serves as a model for graph-based implementations of functional languages, but that it can also be used as a basis for studying the formal semantics of visual programming languages. The first technical contribution of this paper is to extend Ariola and Blom's calculus with staging constructs typical in textual multi-stage languages [20]. The resulting calculus is based on a one-to-one correspondence between visual programs and a variation of the text-based lambda-calculus. We then use this formal connection to lift the semantics of multi-stage languages to the graphical setting. We show that graph reductions have corresponding reductions at the term level, and similarly, term reductions have corresponding reductions at the graph level.

### 1.2 Organization of this Paper

The rest of the paper is organized as follows. Section 2 explains how the syntax for visual languages such as LabVIEW and




(a) LabVIEW [14] (b) PreVIEW (this paper) (c) Ariola and Blom [3]

**Figure 1: The syntax of PreVIEW as middle-ground between that of LabVIEW and lambda-graphs**

Simulink can be modeled using a variation of Ariola and Blom's cyclic lambda-graphs. Section 3 introduces the syntax for a graphical calculus called PreVIEW. Section 4 defines textual representations for PreVIEW and shows that graphs and terms in a specific normal form are one-to-one. Section 5 describes a reduction semantics for both terms and graphs, and Section 6 concludes. Proofs for the results presented in this paper are available online [1]

## 2. LABVIEW AND LAMBDA-GRAPHS

The practical motivation for the calculus studied in the rest of this paper is to extend popular languages such as LabVIEW or Simulink with higher-order functional and staging features. The main abstraction mechanism in LabVIEW is to declare functions; Figure 1 (a) displays the syntax for defining a function with two formal parameters in LabVIEW. PreVIEW abstracts away from many of the details of LabVIEW and similar languages. We reduce the complexity of the calculus by supporting only functions with one argument and by making functions first-class values. We can then use nested lambda abstractions to model functions with multiple parameters, as illustrated in Figure 1 (b).

Graph (c) illustrates Ariola and Blom's lambda-graph syntax [3] for the same computation. In this representation, lambda abstractions are drawn as boxes describing the scope of the parameter bound by the abstraction. Edges represent subterm relationships in the syntax tree, and parameter references are drawn as back-edges to a lambda abstraction. While the lambda-graph (c) may appear less closely related to (a) than the PreVIEW graph (b), note that the graphs (b) and (c) are in fact dual graphs. That is, by flipping the direction of edges in the lambda-graph (c) to represent data-flow instead of subterm relationships, and by making connection points in the graph explicit in the form of ports, we get the PreVIEW program (b). Based on this observation, we take Ariola and Blom's lambda-graphs as the starting point for our formal development.

## 3. SYNTAX OF PREVIEW

The core language features of PreVIEW are function abstraction and function application as known from the $\lambda$-calculus, and the staging constructs Bracket "$\langle\rangle$", Escape "$\sim$", and Run "!". Brackets are a quotation mechanism delaying the evaluation of an expression, while the Escape construct escapes the delaying effect of a Bracket (and so must occur inside a Bracket). Run executes such a delayed computation. The semantics and type theory for these constructs has been studied extensively in recent years [20]. Before defining the syntax of PreVIEW formally, we give an informal description of its visual syntax. Note that this paper focuses on abstract syntax for both terms and graphs, while issues such as an intuitive concrete syntax and parsing are part of future work (see Section 6).

### 3.1 Visual Syntax

A PreVIEW program is a graph built from the following components:

1. **Nodes** represent function abstraction, function application, the staging constructs Brackets, Escape, and Run, and "black holes". Black holes are a concept borrowed from Ariola and Blom [3] and represent unresolvable cyclic dependencies that can arise in textual languages with recursion. [1] As shown in Figure 2, nodes are drawn as boxes labeled $\lambda$,@,$\langle\rangle$,$\sim$, !, and $\bullet$ respectively. Each lambda node contains a subgraph inside its box which represents the body of the function, and the node's box visually defines the scope of the parameter bound by the lambda abstraction. Bracket and Escape boxes, drawn using dotted lines, also contain subgraphs. The subgraph of a Bracket node represents code being generated for a future-stage computation, while the subgraph of an Escape node represents a computation resulting in a piece of code that will be integrated into a larger program at runtime.

2. **Free variables**, displayed as variable names, represent name references that are not bound inside a given PreVIEW graph.

3. **Ports** mark the points in the graph which edges can connect. We distinguish between *source ports* (drawn as triangles) and *target ports* (drawn as rectangles). As shown in Figure 2, a lambda node provides two source ports: out carries the value of the lambda itself, since functions are first-class values in PreVIEW. When the function is applied to an argument, then bind carries the function's parameter, and the return port receives the result of evaluating the function body, represented by the lambda node's subgraph. Intuitively, the fun and arg ports of an application node receive the function to be applied and its argument respectively, while out carries the value resulting from the application. The out port of a Bracket node

[1] In functional languages, recursion is typically expressed using a letrec-construct. The textual program letrec x=x in x introduces a cyclic dependency that cannot be simplified any further. Ariola and Blom visualize such terms as black holes.

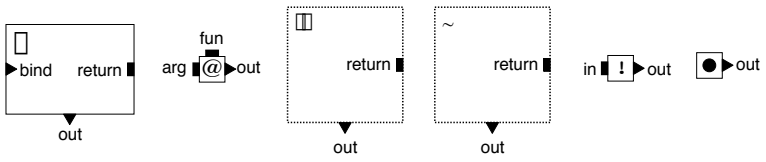



**Figure 2: PreVIEW nodes**

carries the delayed computation represented by the node's subgraph, and return receives the value of that computation when it is executed in a later stage. Conversely, the out port of an Escape node carries a computation that escapes the surrounding Brackets delaying effect, and return receives the value of that computation.

4. **Edges** connect nodes and are drawn as arrows:

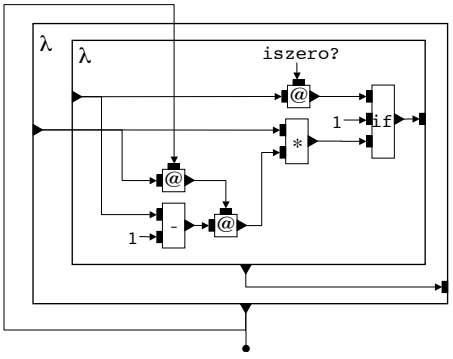

The source of any edge is either the source port of a node or a free variable x. The target of any edge is the target port of some node. The only exception to this is the **root** of the graph. Similar to the root of an abstract syntax tree, it marks the entry-point for evaluating the graph. It is drawn as a dangling edge without a target port, instead marked with a dot.

For convenience, the examples in this paper assume that PreVIEW is extended with integers, booleans, binary integer operators, and conditionals.

EXAMPLE 3.1 (FUNCTIONAL CONSTRUCTS). *Consider the following recursive definition of the power function in OCaml. The function computes the number $x^n$ for two inputs x and n:*

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

*In PreVIEW, this program is expressed as follows:*




*Closely following the textual definition, we visualize the power function as two nested lambda nodes. Consequently, two cascaded application nodes are necessary for the function call* `power x (n-1)`. *Note that the recursive nature of the definition is represented visually by an edge from the* out*-port of the outer lambda node back into the lambda box.*

EXAMPLE 3.2 (MULTI-STAGE CONSTRUCTS). *The power function can be staged by annotating it as follows in MetaOCaml [13]:*[2]

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

*The same program is represented in PreVIEW as shown to the left of Figure 3. As in the text-based program, in PreVIEW we only need to add a few staging "annotations" (in the form of Bracket and Escape boxes) to the unstaged version of the power function.*

EXAMPLE 3.3 (GENERATING GRAPHS). *In MetaOCaml, the staged power function can be used to generate efficient specialized power functions by applying the staged version only to its second input (the exponent). For instance, evaluating the term $M_1$:*

```
.! .<fun x -> .~(power' .<x>. 3)>.
```

*yields the non-recursive function* `fun x -> x*x*x*1`. *Similarly, evaluating the PreVIEW graph in the middle of Figure 3 yields the specialized graph on the right side; the graph in the middle triggers the specialization by providing the staged power function with its second input parameter. Note the simplicity of the generated graph. When applying this paradigm to circuit generation, controlling the complexity of resulting circuits can be essential, and staging constructs were specifically designed to give the programmer more control over the structure of generated programs.*

## 3.2 Formal Syntax

The following syntactic sets are used for defining PreVIEW graphs:

| | | | | | |
|---|---|---|---|---|---|
| *Nodes* | $u, v, w$ | $\in$ | $\mathbb{V}$ | | |
| *Free variables* | $x, y$ | $\in$ | $\mathbb{X}$ | | |
| *Source port types* | $o$ | $\in$ | $\mathbb{O}$ | ::= | bind \| out |
| *Target port types* | $i$ | $\in$ | $\mathbb{I}$ | ::= | return \| fun \| arg \| in |
| *Source ports* | $r, s$ | $\in$ | $\mathbb{S}$ | ::= | $v.o \mid x$ |
| *Target ports* | $t$ | $\in$ | $\mathbb{T}$ | ::= | $v.i$ |
| *Edges* | $e$ | $\in$ | $\mathbb{E}$ | ::= | $(s, t)$ |

As a convention, we use regular capital letters to denote concrete sets. For example, $E \subseteq \mathbb{E}$ stands for a concrete set of edges $e$. We write $\mathcal{P}(V)$ to denote the power set of $V$.

A PreVIEW **graph** is then defined as a tuple $g = (V, L, E, S, r)$ where $V$ is a finite set of **nodes**, $L : V \to \{\lambda, @, \langle\rangle, \sim, !\ , \bullet\}$ is a **labeling function** that associates each node with a label, $E$ is a finite set of **edges**, $S : \{v \in V \mid L(v) \in \{\lambda, \langle\rangle, \sim\}\} \to \mathcal{P}(V)$ is a **scoping function** that associates each lambda, Bracket, and Escape node with a subgraph, and $r$ is the **root** of the graph. When it is clear from the context, we refer to the components $V$, $L$, $E$, $S$, and $r$ of a graph $g$ without making the binding $g = (V, L, E, S, r)$ explicit.

[2] MetaOCaml adds staging constructs to OCaml. Dots are used to disambiguate the concrete syntax: Brackets around an expression $e$ are written as `.<`$e$`>.`, an Escaped expression $e$ is written as `.~`$e$, and `!` $e$ is written as `.!`$e$.

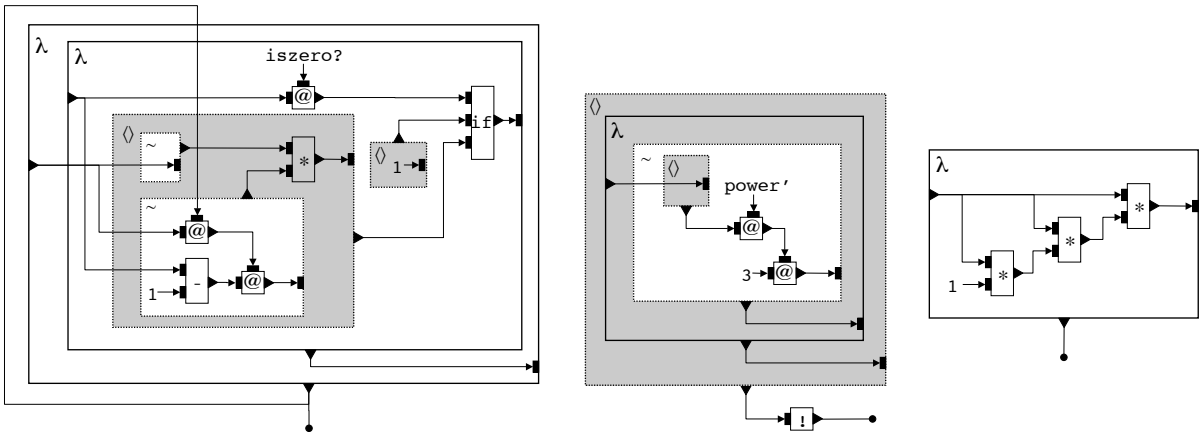



**Figure 3: Generating power functions in PreVIEW**

## 3.3 Auxiliary Definitions

For any PreVIEW graph $g = (V, L, E, S, r)$ we define the following auxiliary notions. The set of **incoming edges** of a node $v \in V$ is defined as $pred(v) = \{(s, v.i) \in E\}$ for any edge targets $i$. Given a set $U \subseteq V$, the set of **top-level nodes** in $U$ that are not in the scope of any other node in $U$ is defined as $toplevel(U) = \{u \in U \mid \forall v \in U : u \in S(v) \Rightarrow v = u\}$. If $v \in V$ has a scope, then the **contents** of $v$ are defined as $contents(v) = S(v) \backslash \{v\}$. For a given node $v \in V$, if there exists a node $u \in V$ with $v \in toplevel(contents(u))$, then $u$ is a **surrounding scope** of $v$. Well-formedness conditions described in the next section will ensure that such a surrounding scope is unique when it exists. A **path** $v \rightsquigarrow w$ in $g$ is an acyclic path from $v \in V$ to $w \in V$ that only consists of edges in $\{(s, t) \in E \mid \forall u : s \neq u.\mathsf{bind}\}$. The negative condition excludes edges starting at a bind port.

## 3.4 Well-Formed Graphs

Whereas context-free grammars are generally sufficient to describe well-formed terms in textual programming languages, characterizing well-formed graphs (in particular with respect to scoping) is more subtle. The well-formedness conditions for the functional features of PreVIEW are taken directly from Ariola and Blom. Since Bracket and Escape nodes also have scopes, these conditions extend naturally to the multi-stage features of PreVIEW. Note however that the restrictions associated with Bracket and Escape are simpler since unlike lambdas these are not binding constructs.

The set $\mathbb{G}$ of **well-formed graphs** is the set of graphs that satisfy the following conditions:

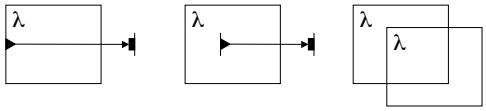
1. **Connectivity** - Edges may connect ports belonging only to nodes in $V$ with the correct port types. Valid *inports* and *outports* for each node type are defined as follows:

| $L(v)$ | *inports*(v) | *outports*(v) |
|---|---|---|
| $\lambda$ | {return} | {bind, out} |
| @ | {fun, arg} | {out} |
| $\langle\rangle$, ~ | {return} | {out} |
| ! | {in} | {out} |
| • | $\emptyset$ | {out} |

   We require that an edge $(v.o, w.i)$ connecting nodes $v$ and $w$ is in $E$ only if $v, w \in V$ and $o \in outports(v)$ and $i \in inports(w)$. Similarly, an edge $(x, w.i)$ originating from a free variable $x$ is in $E$ only if $w \in V$ and $i \in inports(w)$.

   We also restrict the in-degree of nodes: each target port (drawn as a rectangle) in the graph must be the target of exactly one edge, while a source port (drawn as a triangle) can be unused, used by one or shared by multiple edges. Thus we require for any node $v$ in the graph that $pred(v) = \{(s, v.i) \mid i \in inports(v)\}$.

2. **Scoping** - Intuitively, source ports in PreVIEW correspond to bound names in textual languages, and scopes are drawn as boxes. Let $w, w_1, w_2 \in V$ and $v, v_1, v_2 \in dom(S)$ be distinct nodes. By convention, all nodes that have a scope must be in their own scope ($v \in S(v)$). The following three graph fragments illustrate three kinds of scoping errors that can arise:




   A name used outside the scope where it is bound corresponds to an edge from a bind or an out port that *leaves* a scope. We prohibit the first case by requiring that $(v.\mathsf{bind}, t) \in pred(w)$ only if $w \in S(v)$. For the second case, we require that if $w_1 \notin S(v)$ and $w_2 \in S(v)$ and $(w_2.\mathsf{out}, t) \in pred(w_1)$ then $w_2 = v$. Partially overlapping scopes correspond to overlapping lambda, Bracket, or Escape boxes. We disallow this by requiring that $S(v_1) \cap S(v_2) = \emptyset$ or $S(v_1) \subseteq S(v_2) \setminus \{v_2\}$ or $S(v_2) \subseteq S(v_1) \setminus \{v_1\}$.

3. **Root Condition** - The root $r$ cannot be the port of a node nested in the scope of another node. Therefore, the root must either be a free variable ($r \in \mathbb{X}$) or the out port of a node $w$ that is visible at the "top-level" of the graph ($r = w.\mathsf{out}$ and $w \in toplevel(V)$).

# 4. GRAPH-TERM CONNECTION

To develop the connection between PreVIEW graphs and their textual representations, this section begins by defining a term language and a translation from graphs to terms. Not all terms can be generated using this translation, but rather only terms in a specific normal form. A backward-translation from terms to graphs is then defined, and it is shown that a term in normal form represents all terms that map to the same graph. Finally, sets of graphs and normal forms are shown to be in one-to-one correspondence.

## 4.1 From Graphs to Terms

Building on Ariola and Blom's notion of cyclic lambda terms, we use *staged* cyclic lambda terms to represent PreVIEW programs textually, and define them as follows:

$$\begin{array}{rrcl} \textit{Terms} & M \in \mathbb{M} & ::= & x \mid \lambda x.M \mid M\ M \mid \mathsf{letrec}\ d^* \ \mathsf{in}\ M \\ & & \mid & \sim M \mid \langle M \rangle \mid !\ M \\ \textit{Declarations} & d \in \mathbb{D} & ::= & x = M \end{array}$$

**Conventions:** By assumption, all recursion variables $x$ in letrec declarations are distinct, and the sets of bound and free variables are disjoint. We write $d^*$ for a (possibly empty) sequence of letrec declarations $d$. Different permutations of the same sequence of declarations $d^*$ are identified. Therefore, we often use the set notation $D$ instead of $d^*$. Given two sequences of declarations $D_1$ and $D_2$, we write $D_1, D_2$ for the concatenation of the two sequences. We write $M_1[x := M_2]$ for the result of substituting $M_2$ for all free occurrences of the variable $x$ in $M_1$, without capturing any free variables in $M_2$. We use $\equiv_\alpha$ to denote syntactic equality up to $\alpha$-renaming of both lambda-bound variables and recursion variables.

To translate a graph into a term, we define the **term construction** $\tau : \mathbb{G} \to \mathbb{M}$. Intuitively, this translation associates all nodes in the graph with a unique variable name in the term language. These variables are used to explicitly name each subterm of the resulting term. Lambda nodes are associated with an additional variable name, which is used to name the formal parameter of the represented lambda abstraction.

Definition 4.1 (Term construction). *Let $g = (V, L, E, S, r)$ be a well-formed graph in $\mathbb{G}$.*

1. *For every node $v \in V$, we define a unique name $x_v$, and a second distinct name $y_v$ if $L(v) = \lambda$. We then associate a name with each edge source $s$ in the graph as follows:*

$$\mathit{name}(s) = \begin{cases} x_v & \text{if } s = v.\mathsf{out} \\ y_v & \text{if } s = v.\mathsf{bind} \\ x & \text{if } s = x \end{cases}$$

2. *To avoid the construction of empty letrec terms* (letrec _ in $M$) *in the translation, we use the following function:*

$$\mathit{mkrec}(D, M) = \begin{cases} M & \text{if } D = \emptyset \\ \mathsf{letrec}\ D\ \mathsf{in}\ M & \text{otherwise} \end{cases}$$

3. *We construct a term corresponding to each node $v \in V$:*

$$\frac{L(v) = \bullet \quad \mathit{pred}(v) = \emptyset}{\mathit{term}(v) = x_v}$$

$$\frac{L(v) = \lambda \quad \mathit{pred}(v) = \{(s, v.\mathsf{return})\}}{\mathit{term}(v) = \lambda y_v.\mathit{mkrec}(\mathit{decl}(\mathit{contents}(v)), \mathit{name}(s))}$$

$$\frac{L(v) = @ \quad \mathit{pred}(v) = \{(s_1, v.\mathsf{fun}), (s_2, v.\mathsf{arg})\}}{\mathit{term}(v) = \mathit{name}(s_1)\ \mathit{name}(s_2)}$$

$$\frac{L(v) = \langle\rangle \quad \mathit{pred}(v) = \{(s, v.\mathsf{return})\}}{\mathit{term}(v) = \langle \mathit{mkrec}(\mathit{decl}(\mathit{contents}(v)), \mathit{name}(s)) \rangle}$$

$$\frac{L(v) = \sim \quad \mathit{pred}(v) = \{(s, v.\mathsf{return})\}}{\mathit{term}(v) = \sim \mathit{mkrec}(\mathit{decl}(\mathit{contents}(v)), \mathit{name}(s))}$$

$$\frac{L(v) = ! \quad \mathit{pred}(v) = \{(s, v.\mathsf{in})\}}{\mathit{term}(v) = !\ \mathit{name}(s)}$$

4. *We construct letrec declarations for any set of nodes $W \subseteq V$:*

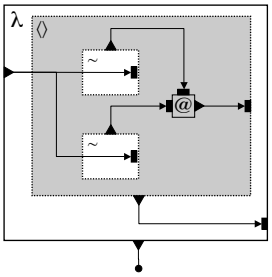
$$\mathit{decl}(W) = \{x_v = \mathit{term}(v) \mid v \in \mathit{toplevel}(W)\}$$

5. *The term construction $\tau$ is then defined as:*

$$\tau(g) = \mathit{mkrec}(\mathit{decl}(V), \mathit{name}(r))$$

The translation $\tau$ starts by computing the set of top-level nodes in $V$ (see Section 3.3), and creates a letrec declaration for each of these nodes. For a node $v$ with no subgraph, the letrec declaration binds the variable $x_v$ to a term that combines the variables associated with the incoming edges to $v$. If $v$ contains a subgraph, then $\tau$ is applied recursively to the subgraph, and $x_v$ is bound to the term that represents the subgraph. The constraint $v \in \mathit{toplevel}(W)$ in the definition of *decl* ensures that exactly one equation is generated for each node: if $v \notin \mathit{toplevel}(W)$, then $v$ is in the scope of a different node $w \in W$, and an equation for $w$ is instead included in $\mathit{term}(w)$.

Example 4.1 (Term Construction). *The function $\tau$ translates the graph*




*as follows: Let $v_1$ be the lambda node, $v_2$ the Bracket node, $v_3$ and $v_4$ the top and bottom Escape nodes, and $v_5$ the application node in the graph $g$. We associate a variable name $x_j$ with each node $v_j$. In addition, the name $y_1$ is associated with the parameter of the lambda node $v_1$. The result is:*

$$\begin{array}{l} \mathsf{letrec}\ x_1 = \lambda y_1.(\mathsf{letrec}\ x_2 = \langle \mathsf{letrec}\ x_3 = \sim y_1, x_4 = \sim y_1, x_5 = x_3 x_4 \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ x_5 \rangle \\ \qquad\qquad\qquad\qquad \mathsf{in}\ x_2) \\ \mathsf{in}\ x_1 \end{array}$$

*All nodes are in the scope of $v_1$ so it is the only "top-level" node in $g$. We create a letrec declaration for $v_1$, binding $x_1$ to a term $\lambda y_1.N$ where $N$ is the result of recursively translating the subgraph inside $v_1$. When translating the subgraph of the Bracket node $v_2$, note that this subgraph contains three top-level nodes ($v_3$, $v_4$, $v_5$). Therefore, the term for $v_2$ contains three variable declarations ($x_3$, $x_4$, $x_5$).*

## 4.2 Terms in Normal Form

The term construction function $\tau$ only constructs terms in a very specific form. For example, while the graph in the previous example represents the computation $\lambda y_1.\langle \sim y_1\ \sim y_1 \rangle$, the example shows that $\tau$ constructs a different term. Compared to $\lambda y_1.\langle \sim y_1\ \sim y_1 \rangle$, every subterm in the constructed term is explicitly named using letrec. This explicit naming of subterms expresses the notion of value sharing in PreVIEW graphs, where the output port of *any* node can be the source of multiple edges. Such **normal forms** are essentially the same as A-normal form [17], and can be defined as follows:

$$\begin{array}{rrcl} \textit{Terms} & N \in \mathbb{M}_{norm} & ::= & x \mid \mathsf{letrec}\ q^+ \ \mathsf{in}\ x \\ \textit{Declarations} & q \in \mathbb{D}_{norm} & ::= & x = x \mid x = y\ z \mid x = \lambda y.N \\ & & \mid & x = \langle N \rangle \mid x = \sim N \mid x = !\ y \end{array}$$

where $q^+$ is a non-empty sequence of declarations $q$. In normal forms, nested terms are only allowed in function bodies and inside Brackets or Escapes, i.e. only for language constructs that correspond to nodes with subgraphs. All other expressions are explicitly named using letrec declarations, and pure "indirection" declarations of the form $x = y$ with $x \not\equiv y$ are not allowed.

LEMMA 4.1 (NORMAL FORMS ARE TERMS). $\mathbb{M}_{norm} \subseteq \mathbb{M}$.

LEMMA 4.2 ($\tau$ MAPS GRAPHS TO NORMAL FORMS). *If* $g \in \mathbb{G}$ *then* $\tau(g) \in \mathbb{M}_{norm}$.

As we will show, $\tau$ is an injection, i.e. not every term corresponds to a distinct graph. However, we will show that every term has a normal form associated with it, and that these normal forms are one-to-one with graphs. To this end, we define the **normalization function** $\nu : \mathbb{M} \to \mathbb{M}_{norm}$ in two steps: general terms are first mapped to **intermediate forms**, which are then converted into normal forms in a second pass. We define the set $\mathbb{M}_{pre}$ of intermediate forms as follows:

$$\begin{array}{rrcl} \textit{Terms} & N' \in \mathbb{M}_{pre} & ::= & x \mid \mathsf{letrec}\ q'^{*}\ \mathsf{in}\ x \\ \textit{Declarations} & q' \in \mathbb{D}_{pre} & ::= & x = y \mid x = y\ z \mid x = \lambda y.N' \\ & & \mid & x = \langle N' \rangle \mid x = {\sim} N' \mid x = !\ y \end{array}$$

Note that this set consists of normal forms with fewer restrictions: empty letrec terms and indirections of the form $x = y$ are allowed.

DEFINITION 4.2 (TERM NORMALIZATION). *Given the definitions of the translations* $[\![\_]\!]_{pre} : \mathbb{M} \to \mathbb{M}_{pre}$ *and* $[\![\_]\!]_{norm} : \mathbb{M}_{pre} \to \mathbb{M}_{norm}$ *in Figure 4, we define the normalization function* $\nu : \mathbb{M} \to \mathbb{M}_{norm}$ *by composition:* $\nu = [\![\_]\!]_{norm} \circ [\![\_]\!]_{pre}$.

The translation $[\![\_]\!]_{pre}$ maps any term $M$ to a letrec term, assigning a fresh letrec variable to each subterm of $M$. We preserve the nesting of lambda abstractions, Bracket and Escapes by applying $[\![\_]\!]_{pre}$ to subterms recursively. [3] Once every subterm has a letrec variable associated with it, and all lambda, Bracket, and Escape subterms are normalized recursively, the function $[\![\_]\!]_{norm}$ eliminates empty letrec terms and letrec indirections of the form $x = y$ (where $x \not\equiv y$) using substitution. The clause $N' \notin \mathbb{M}_{norm}$ in the definition of $[\![\_]\!]_{norm}$ ensures that normalization terminates: without this restriction we could apply $[\![\_]\!]_{norm}$ to a fully normalized term without making any progress.

EXAMPLE 4.2 (TERM NORMALIZATION). *Given the following terms:*

$$\begin{array}{l} M_1 \equiv \lambda x.\langle {\sim}\ x\ {\sim}\ x \rangle \\ M_2 \equiv \mathsf{letrec}\ y = \lambda x.\langle {\sim}\ x\ {\sim}\ x \rangle\ \mathsf{in}\ y \\ M_3 \equiv \lambda x.\mathsf{letrec}\ y = \langle {\sim}\ x\ {\sim}\ x \rangle\ \mathsf{in}\ y \end{array}$$

*Then* $\nu(M_1)$, $\nu(M_2)$, *and* $\nu(M_3)$ *all yield a term alpha-equivalent to:*

$$\begin{array}{l} \mathsf{letrec}\ y_1 = \lambda x.(\mathsf{letrec}\ y_2 = \langle \mathsf{letrec}\ y_3 = {\sim}\ x, y_4 = {\sim}\ x, y_5 = y_3\ y_4 \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ y_5 \rangle \\ \qquad\qquad\qquad \mathsf{in}\ y_2) \\ \mathsf{in}\ y_1 \end{array}$$

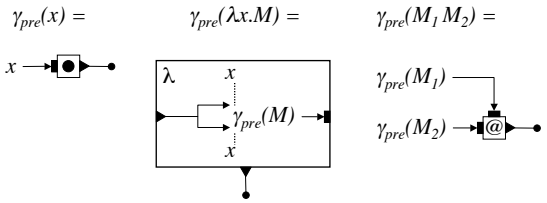
*Note that the basic structure of the original terms (lambda term with Bracket body and application of two escaped parameter references inside) is preserved by normalization, but every subterm is now named explicitly.*

LEMMA 4.3 ($\nu$ MAPS TERMS TO NORMAL FORMS). *If* $M \in \mathbb{M}$ *then* $\nu(M) \in \mathbb{M}_{norm}$.

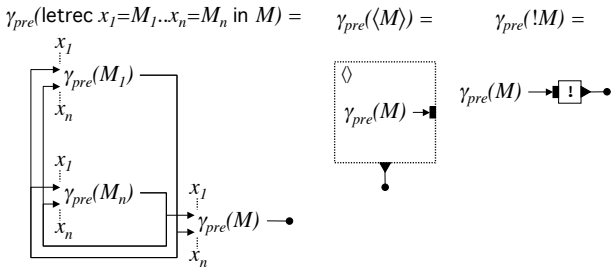
## 4.3 From Terms to Graphs

To simplify the definition of a translation from terms to graphs, we introduce a notion analogous to Ariola and Blom's scoped pre-graphs. The set $\mathbb{G}_{pre}$ of **intermediate graphs** consists of all graphs for which a well-formedness condition is relaxed: nodes with label • may have 0 or 1 incoming edge. Formally, whenever $L(v) = \bullet$ then $pred(v) = \emptyset$ or $pred(v) = \{(s, v.\mathsf{in})\}$. If such a node has 1 predecessor, we call it an **indirection node**. Since free variables are not represented as nodes in PreVIEW, the idea is to associate an indirection node with each variable occurrence in the translated lambda-term. This simplifies connecting subgraphs constructed during the translation, as it provides "hooks" for connecting bound variable occurrences in the graph to their binders. We will also use indirection nodes to model intermediate states in the graph reductions presented in Section 5.2.

We translate terms to PreVIEW graphs in two steps: A function $\gamma_{pre}$ maps terms to intermediate graphs, and a simplification function $\sigma$ maps intermediate graphs to proper PreVIEW graphs. Before defining these translations formally, we give visual descriptions of $\gamma_{pre}$ and $\sigma$.

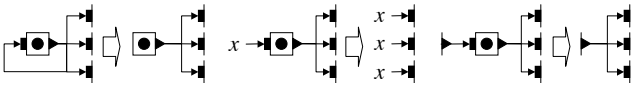



A free variable $x$ is mapped by $\gamma_{pre}$ to an indirection node with $x$ connected to its in port. A lambda term $\lambda x.M$ maps to a lambda node $v$, where the pre-graph for $M$ becomes the subgraph of $v$ and all free variables $x$ in the subgraph are replaced by edges originating at the lambda node's bind port. An application $M_1 M_2$ translates to an application node $v$ where the roots of the pre-graphs for $M_1$ and $M_2$ are connected to the fun and arg ports of $v$.




Given a letrec term (letrec $x_1 = M_1, .., x_n = M_n$ in $M$), $\gamma_{pre}$ translate the terms $M_1$ through $M_n$ and $M$ individually. The root of the resulting pre-graph is the root of $\gamma_{pre}(M)$. Any edge that starts with one of the free variable $x_j$ is replaced by an edge from the root of the corresponding graph $\gamma_{pre}(M_j)$. The cases for $\langle M \rangle$ and $\sim M$ are treated similarly to the case for $\lambda x.M$, and the case for $!M$ is treated similarly to the case for application.

Simplification eliminates indirection nodes from the pre-graph using the following local graph transformations:




Any indirection node with a self-loop (i.e. there is an edge from its out port to its in port) is replaced by a black hole. If there is an edge from a free variable $x$ or from a different node's port $s$ to an indirection node $v$, then the indirection node is "skipped" by replacing all edges originating at $v$ to edges originating at $x$ or $s$. Note that the second and third cases are different since free variables cannot be shared in PreVIEW.

[3] This is similar to the translation $\tau$ from graphs to terms presented above, where lambda, Bracket and Escape *nodes* are translated to terms recursively.

$$\frac{}{[\![x]\!]_{pre} = \text{letrec } \_ \text{ in } x} \qquad \frac{[\![M]\!]_{pre} = N' \quad x_1 \text{ fresh}}{[\![\lambda x.M]\!]_{pre} = (\text{letrec } x_1 = \lambda x.N' \text{ in } x_1)}$$

$$\frac{[\![M_1]\!]_{pre} = \text{letrec } Q_1 \text{ in } x_1 \quad [\![M_2]\!]_{pre} = \text{letrec } Q_2 \text{ in } x_2 \quad x_3 \text{ fresh}}{[\![M_1 M_2]\!]_{pre} = (\text{letrec } Q_1, Q_2, x_3 = x_1 x_2 \text{ in } x_3)}$$

$$\frac{[\![M]\!]_{pre} = \text{letrec } Q \text{ in } y \quad \overrightarrow{[\![M_j]\!]_{pre} = \text{letrec } Q_j \text{ in } y_j}}{[\![\text{letrec } \overrightarrow{x_j = M_j} \text{ in } M]\!]_{pre} = (\text{letrec } Q, \overrightarrow{Q_j, x_j = y_j} \text{ in } y)}$$

$$\frac{[\![M]\!]_{pre} = N' \quad x_1 \text{ fresh}}{[\![\langle M \rangle]\!]_{pre} = (\text{letrec } x_1 = \langle N' \rangle \text{ in } x_1)} \qquad \frac{[\![M]\!]_{pre} = N' \quad x_1 \text{ fresh}}{[\![\sim M]\!]_{pre} = (\text{letrec } x_1 = \sim N' \text{ in } x_1)}$$

$$\frac{[\![M]\!]_{pre} = \text{letrec } Q \text{ in } y \quad x_1 \text{ fresh}}{[\![! \ M]\!]_{pre} = (\text{letrec } Q, x_1 = ! \ y \text{ in } x_1)}$$

$$\frac{}{[\![N]\!]_{norm} = N} \qquad \frac{}{[\![\text{letrec } \_ \text{ in } x]\!]_{norm} = x}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad [\![N']\!]_{norm} = N_1 \quad [\![\text{letrec } y = \lambda z.N_1, Q \text{ in } x]\!]_{norm} = N_2}{[\![\text{letrec } y = \lambda z.N', Q \text{ in } x]\!]_{norm} = N_2}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad [\![N']\!]_{norm} = N_1 \quad [\![\text{letrec } y = \langle N_1 \rangle, Q \text{ in } x]\!]_{norm} = N_2}{[\![\text{letrec } y = \langle N' \rangle, Q \text{ in } x]\!]_{norm} = N_2}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad [\![N']\!]_{norm} = N_1 \quad [\![\text{letrec } y = \sim N_1, Q \text{ in } x]\!]_{norm} = N_2}{[\![\text{letrec } y = \sim N', Q \text{ in } x]\!]_{norm} = N_2}$$

$$\frac{[\![(\text{letrec } Q \text{ in } x)[y := z]]\!]_{norm} = N \quad y \not\equiv z}{[\![\text{letrec } y = z, Q \text{ in } x]\!]_{norm} = N}$$

**Figure 4: The translation functions $[\![\_]\!]_{pre} : \mathbb{M} \to \mathbb{M}_{pre}$ and $[\![\_]\!]_{norm} : \mathbb{M}_{pre} \to \mathbb{M}_{norm}$**

To define these translations formally, we use the following notation: $E[s_1 := s_2]$ denotes the result of substituting any edge in $E$ that originates from $s_1$ with an edge that starts at $s_2$:

$$E[s_1 := s_2] = \{(s,t) \in E \mid s \neq s_1\} \cup \{(s_2,t) \mid (s_1,t) \in E\}$$

$S \backslash u$ stands for the result of removing node $u$ from any scope in the graph: $(S \backslash u)(v) = S(v) \backslash \{u\}$. The substitution $r[s_1 := s_2]$ results in $s_2$ if $r = s_1$ and in $r$ otherwise.

Definition 4.3 (Graph construction). *Given the definitions of the translations $\gamma_{pre} : \mathbb{M} \to \mathbb{G}_{pre}$ and $\sigma : \mathbb{G}_{pre} \to \mathbb{G}$ in Figure 5, we define the graph construction $\gamma : \mathbb{M} \to \mathbb{G}$ by composition: $\gamma = \sigma \circ \gamma_{pre}$.*

Lemma 4.4 ($\gamma$ maps terms to well-formed graphs). *For any $M \in \mathbb{M}$, $\gamma(M)$ is defined and is a unique, well-formed graph.*

Using the mappings $\nu$, $\gamma$, and $\tau$, we can now give a precise definition of the connections between terms, graphs, and normal forms. Two terms map to the same graph if and only if they have the same normal form. Thus, normal forms represent equivalence classes of terms that map to the same graph by $\gamma$. The function $\nu$ gives an algorithm for computing such representative terms. Given two well-formed graphs $g, h \in \mathbb{G}$, we write $g = h$ if $g$ and $h$ are isomorphic graphs with identical node labels.

Lemma 4.5 (Soundness of Normalization). *If $M \in \mathbb{M}$. then $\gamma(M) = \gamma(\nu(M))$.*

Lemma 4.6 (Recovery of normal forms). *If $N \in \mathbb{M}_{norm}$ then $N \equiv_\alpha \tau(\gamma(N))$.*

Lemma 4.7 (Completeness of Normalization). *Let $M_1, M_2 \in \mathbb{M}$. If $\gamma(M_1) = \gamma(M_2)$ then $\nu(M_1) \equiv_{alpha} \nu(M_2)$.*

Example 4.3. *In Example 4.2 we showed that the three terms $M_1$, $M_2$, and $M_3$ have the same normal form. By Lemma 4.5, they translate to the same graph. This graph is shown in Example 4.1. By Lemma 4.7, the terms $M_1$, $M_2$, and $M_3$ must have the same normal form since they map to the same graph by $\gamma$.*

Theorem 4.1 (Correctness of Graphical Syntax). *Well-formed graphs and normal forms are one-to-one:*

1. *If $M \in \mathbb{M}$ then $\nu(M) \equiv_{alpha} \tau(\gamma(M))$.*
2. *If $g \in \mathbb{G}$ then $g = \gamma(\tau(g))$.*

# 5. SEMANTICS FOR PREVIEW

This section presents a reduction semantics for staged cyclic lambda terms and graphs, and establishes the connection between the two.

## 5.1 Staged Terms

Ariola and Blom study a call-by-need reduction semantics for the lambda-calculus extended with a letrec construct. In order to extend this semantics to support staging constructs, we use the notion of *expression families* proposed for the reduction semantics of call-by-name $\lambda$-U [18]. In the context of $\lambda$-U, expression families restrict beta-redeces to terms that are valid at level 0. Intuitively, given a staged term $M$, the **level** of a subterm of $M$ is the number of Brackets minus the number of Escapes surrounding the subterm. A term $M$ is **valid at level** $n$ if all Escapes inside $M$ occur at a level greater than $n$.

Example 5.1. *Consider the lambda term $M \equiv \langle \lambda x. \sim (f\langle x \rangle) \rangle$. The variable $f$ occurs at level 0, while the use of $x$ occurs at level 1. Since the Escape occurs at level 1, $M$ is valid at level 0.*

The calculus $\lambda$-U does not provide a letrec construct to directly express sharing in lambda terms. Therefore, we extend the notion of expression families to include the letrec construct as follows:

$$\begin{array}{lcl} M^0 \in \mathbb{M}^0 & ::= & x \mid \lambda x.M^0 \mid M^0 M^0 \mid \text{letrec } D^0 \text{ in } M^0 \\ & \mid & \langle M^1 \rangle \mid ! \ M^0 \\ M^{n+} \in \mathbb{M}^{n+} & ::= & x \mid \lambda x.M^{n+} \mid M^{n+} M^{n+} \mid \text{letrec } D^{n+} \text{ in } M^{n+} \\ & \mid & \langle M^{n++} \rangle \mid \sim M^n \mid ! \ M^{n+} \\ D^n \in \mathbb{D}^n & ::= & \overrightarrow{x_j = M_j^n} \end{array}$$

In order to combine Ariola and Blom's reduction semantics for cyclic lambda-terms with the reduction semantics for $\lambda$-U, we need to account for the difference in beta-reduction between the two formalisms: While $\lambda$-U is based on a standard notion of substitution, Ariola and Blom's beta-rule uses the letrec construct to express a binding from the applied function's parameter to the argument of the application, without immediately substituting the argument for the function's parameter. Instead, substitution is performed on demand by a separate reduction rule. Furthermore, substitution in $\lambda$-U is restricted (implicitly by the $\beta$-rule) to $M^0$-terms. We make this restriction explicit by defining which contexts are valid at different levels:

$$\begin{array}{lcl} C \in \mathbb{C} & ::= & \square \mid \lambda x.C \mid C \ M \mid M \ C \mid \text{letrec } D \text{ in } C \\ & \mid & \text{letrec } x = C, D \text{ in } M \mid \langle C \rangle \mid \sim C \mid ! \ C \\ C^n \in \mathbb{C}^n & = & \{C \in \mathbb{C} \mid C[x] \in \mathbb{M}^n\} \end{array}$$

$$\frac{v \text{ fresh}}{\gamma_{pre}(x) = (\{v\}, \{v \mapsto \bullet\}, \{(x, v.\mathsf{in})\}, \emptyset, v.\mathsf{out})} \qquad \frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\lambda x.M) = (V \uplus \{v\}, L \uplus \{v \mapsto \lambda\}, E[x := v.\mathsf{bind}] \uplus \{(r, v.\mathsf{return})\}, S \uplus \{v \mapsto V \uplus \{v\}\}, v.\mathsf{out})}$$

$$\frac{\gamma_{pre}(M_1) = (V_1, L_1, E_1, S_1, r_1) \quad \gamma_{pre}(M_2) = (V_2, L_2, E_2, S_2, r_2) \quad v \text{ fresh}}{\gamma_{pre}(M_1 M_2) = (V_1 \uplus V_2 \uplus \{v\}, L_1 \uplus L_2 \uplus \{v \mapsto @\}, E_1 \uplus E_2 \uplus \{(r_1, v.\mathsf{fun}), (r_2, v.\mathsf{arg})\}, S_1 \uplus S_2, v.\mathsf{out})}$$

$$\frac{\overrightarrow{\gamma_{pre}(M_j) = (V_j, L_j, E_j, S_j, r_j)} \quad \gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M) = (V \uplus \overrightarrow{V_j}, L \uplus \overrightarrow{L_j}, (E \uplus \overrightarrow{E_j})[\overrightarrow{x_j := r_j}], S \uplus \overrightarrow{S_j}, r)}$$

$$\frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\langle M \rangle) = (V \uplus \{v\}, L \uplus \{v \mapsto \langle\rangle\}, E \uplus \{(r, v.\mathsf{return})\}, S \uplus \{v \mapsto V \uplus \{v\}\}, v.\mathsf{out})}$$

$$\frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(\sim M) = (V \uplus \{v\}, L \uplus \{v \mapsto \sim\}, E \uplus \{(r, v.\mathsf{return})\}, S \uplus \{v \mapsto V \uplus \{v\}\}, v.\mathsf{out})} \qquad \frac{\gamma_{pre}(M) = (V, L, E, S, r) \quad v \text{ fresh}}{\gamma_{pre}(!\ M) = (V \uplus \{v\}, L \uplus \{v \mapsto !\}, E \uplus \{(r, v.\mathsf{in})\}, S, v.\mathsf{out})}$$

$$\frac{\forall v \in V : L(v) = \bullet \Rightarrow \mathrm{pred}(v) = \emptyset}{\sigma(V, L, E, S, r) = (V, L, E, S, r)} \qquad \frac{\sigma(V \uplus \{v\}, L \uplus \{v \mapsto \bullet\}, E, S, r) = g}{\sigma(V \uplus \{v\}, L \uplus \{v \mapsto \bullet\}, E \uplus \{(v.\mathsf{out}, v.\mathsf{in})\}, S, r) = g}$$

$$\frac{s \neq v.\mathsf{out} \quad (v.\mathsf{out}, t) \notin E \quad \sigma(V, L, E \uplus \{\overrightarrow{(s, t_j)}\}, S \setminus v, r[v.\mathsf{out} := s]) = g}{\sigma(V \uplus \{v\}, L \uplus \{v \mapsto \bullet\}, E \uplus \{(s, v.\mathsf{in})\} \uplus \{\overrightarrow{(v.\mathsf{out}, t_j)}\}, S, r) = g}$$

**Figure 5: The translation functions $\gamma_{pre} : \mathbb{M} \to \mathbb{G}_{pre}$ and $\sigma : \mathbb{G}_{pre} \to \mathbb{G}$**

We write $C[M]$ for the result of replacing the hole $\Box$ in $C$ with $M$, potentially capturing free variables in $M$ in the process. Furthermore, we adopt the notation $D \perp M$ from [3] to denote that the set of variables occurring as the left-hand side of a letrec declaration in $D$ does not intersect with the set of free variables in $M$.

Using these families of terms and contexts, we extend Ariola and Blom's reductions as shown in Figure 6. We write $\to$ for the compatible extension of the rules in $\mathcal{R}$, and we write $\to^*$ for the reflexive and transitive closure of $\to$. The idea behind the rules *sub* is to perform substitution on demand after a function application has been performed. In this sense, the reduction rules *sub* and the rule $\beta\circ$ together mimic the behavior of beta-reduction in $\lambda$-U.

## 5.2 Staged Graphs

To define a reduction semantics for PreVIEW, we define similar notions as used in the previous section: the **level** of a node is the number of surrounding Bracket nodes minus the surrounding Escape nodes, and a set of nodes $U$ is **valid at level** $n$ if all Escape nodes in $U$ occur at a level greater than $n$.

Definition 5.1 (Node level). *Given a graph $g = (V, L, E, S, r) \in \mathbb{G}$, a node $v \in V$ has level $n$ if there is a derivation for the judgment $level(v) = n$ defined as follows:*

$$\frac{v \in toplevel(V)}{level(v) = 0} \qquad \frac{surround(v) = u \quad L(u) = \lambda \quad level(u) = n}{level(v) = n}$$

$$\frac{surround(v) = u \quad L(u) = \langle\rangle \quad level(u) = n}{level(v) = n + 1}$$

$$\frac{surround(v) = u \quad L(u) = \sim \quad level(u) = n + 1}{level(v) = n}$$

*We write $level(v_1) < level(v_2)$ as a shorthand for $level(v_1) = n_1 \wedge level(v_2) = n_2 \wedge n_1 < n_2$. A set $U \subseteq V$ is valid at level $n$ if there is a derivation for the judgment $\vdash^n U$ defined as follows:*

$$\frac{\vdash^n v \quad \forall v \in toplevel(U)}{\vdash^n U} \qquad \frac{L(v) \in \{@, \bullet, !\}}{\vdash^n v}$$

$$\frac{L(v) = \lambda \quad \vdash^n contents(v)}{\vdash^n v} \qquad \frac{L(v) = \langle\rangle \quad \vdash^{n+1} contents(v)}{\vdash^n v}$$

$$\frac{L(v) = \sim \quad \vdash^n contents(v)}{\vdash^{n+1} v}$$

Context families and node levels are closely related. In the term reductions presented in the previous section, context families restrict the terms in which a variable may be substituted. In the graph reductions described in this section, determining whether two nodes constitute a redex will require comparing the levels of the two nodes. Furthermore, we can show that the notion of a set of nodes valid at a given level corresponds directly to the restriction imposed on terms by expression families.

Lemma 5.1 (Properties of graph validity).

1. *Whenever $M^n \in \mathbb{M}^n$ and $g = \gamma(M^n)$, then $\vdash^n V$.*
2. *Whenever $g \in \mathbb{G}$ with $\vdash^n V$, then $\tau(g) \in \mathbb{M}^n$.*

When evaluating a graph $g = (V, L, E, S, r)$, we require that $g$ be well-formed (see Section 3.4) and that $\vdash^0 V$. This ensures that $level(v)$ is defined for all $v \in V$.

Lemma 5.2 (Node levels in well-formed graphs). *For any graph $g \in \mathbb{G}$ with $\vdash^0 V$ and $v \in V$, we have $level(v) = n$ for some $n$.*

We now define three reduction rules that can be applied to PreVIEW graphs. Each of these rules is applied in two steps: 1) If necessary, we copy nodes to expose the redex in the graph. This step corresponds to using the term reduction rules *sub* or the rules

$$
\begin{array}{rcl}
\text{letrec } x = M^0, D^n \text{ in } C^0[x] & \to_{sub} & \text{letrec } x = M^0, D^n \text{ in } C^0[M^0] \\
\text{letrec } x = C^0[y], y = M^0, D^n \text{ in } M^n & \to_{sub} & \text{letrec } x = C^0[M^0], y = M^0, D^n \text{ in } M^n \\
(\lambda x.M_1^0)\ M_2^0 & \to_{\beta\circ} & \text{letrec } x = M_2^0 \text{ in } M_1^0 \\
\sim \langle M^0 \rangle & \to_{esc} & M^0 \\
! \langle M^0 \rangle & \to_{run} & M^0 \\
\text{letrec } D_1^n \text{ in } (\text{letrec } D_2^n \text{ in } M^n) & \to_{merge} & \text{letrec } D_1^n, D_2^n \text{ in } M^n \\
\text{letrec } x = (\text{letrec } D_1^n \text{ in } M_1^n), D_2^n \text{ in } M_2^n & \to_{merge} & \text{letrec } x = M_1^n, D_1^n, D_2^n \text{ in } M_2^n \\
(\text{letrec } D^n \text{ in } M_1^n)\ M_2^n & \to_{lift} & \text{letrec } D^n \text{ in } (M_1^n\ M_2^n) \\
M_1^n\ (\text{letrec } D^n \text{ in } M_2^n) & \to_{lift} & \text{letrec } D^n \text{ in } (M_1^n\ M_2^n) \\
\text{letrec } D^n \text{ in } \langle M^n \rangle & \to_{lift} & \langle \text{letrec } D^n \text{ in } M^n \rangle \\
\text{letrec } \_ \text{ in } M^n & \to_{gc} & M^n \\
\text{letrec } D_1^n, D_2^n \text{ in } M^n & \to_{gc} & \text{letrec } D_1^n \text{ in } M^n \\
 & & \text{if } D_2^n \neq \emptyset \wedge D_2^n \perp \text{letrec } D_1^n \text{ in } M^n
\end{array}
$$

**Figure 6: Term Reduction Rules**

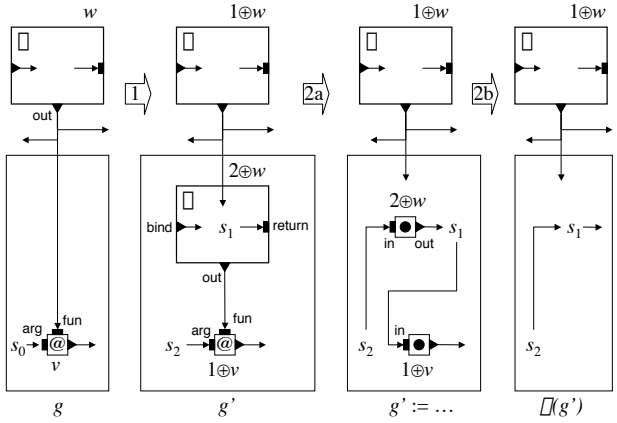



**Figure 7: Beta-reduction for PreVIEW graphs**

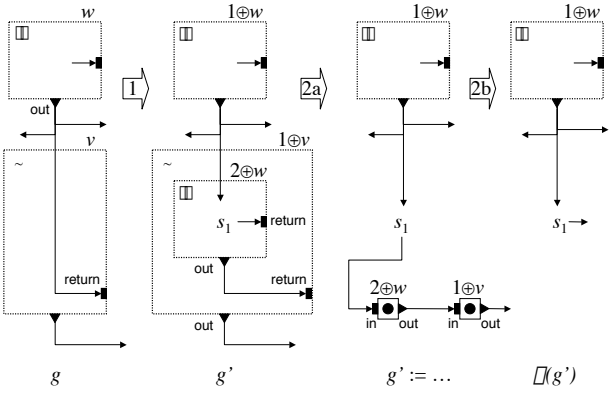



**Figure 8: Escape-reduction for PreVIEW graphs**

*merge*, *lift*, and *gc* (see Figure 6) on the original term. 2) We contract the redex by removing nodes and by redirecting edges in the graph. This step corresponds to performing the actual $\beta\circ$-, *esc*-, or *run*-reduction on a term. In the following, we write $j \oplus V$ for the set $\{j \oplus v \mid v \in V\}$ where $j \in \{1, 2\}$. Furthermore, we write $U \oplus V$ for the set $(1 \oplus U) \cup (2 \oplus V)$.

**Beta** A $\beta\circ$-redex in a PreVIEW graph consists of an application node $v$ that has a lambda node $w$ as its first predecessor. The contraction of the redex is performed in two steps (see Figure 7):

1. Check that the edge ($w$.out, $v$.fun) is the only edge originating at $w$.out, and that the application node $v$ is outside the scope of $w$. If any of these conditions do not hold, copy the lambda node in a way that ensures that the conditions hold for the copy of $w$. The copy of $w$ is called $2 \oplus w$, and the original of $v$ is called $1 \oplus v$. Place $2 \oplus w$ and its scope in the same scope as $1 \oplus v$.

2. Convert $1 \oplus v$ and $2 \oplus w$ into indirection nodes, which are then removed by the graph simplification function $\sigma$ (defined in Section 4.3). Redirect edges so that after simplification, edges that originated at the applied function's parameter ($2 \oplus w$.bind) now start at the root $s_2$ of the function's argument, and edges that originated at the application node's output ($1 \oplus v$.out) now start at the root $s_1$ of the function's body.

DEFINITION 5.2 (GRAPH BETA). *Given a graph $g \in \mathbb{G}$ with $\vdash^0 V$ and $v, w \in V$ such that $L(v) = @$, $L(w) = \lambda$, $(w.\text{out}, v.\text{fun}) \in E$, $\vdash^0$ contents$(w)$, $\vdash^0 \{u \mid u \in S(\text{surround}(v)) \wedge u \rightsquigarrow v\}$, and level$(w) \leq$ level$(v)$ Then the contraction of the $\beta\circ$-redex $v$, written $g \to_{\beta\circ} h$, is defined as follows:*

1. *We define a transitional graph $g' = (V', L', E', S', r')$ using the functions $f_1$ and $f_2$ that map edge sources in $E$ to edge sources in $E'$:*

$$
\begin{array}{lcl}
f_1(x) & = & x \\
f_1(u.o) & = & 1 \oplus u.o \\
f_2(x) & = & x \\
f_2(u.\text{bind}) & = & \begin{cases} 2 \oplus u.\text{bind} & \text{if } u \in S(w) \\ 1 \oplus u.\text{bind} & \text{otherwise} \end{cases} \\
f_2(u.\text{out}) & = & \begin{cases} 2 \oplus u.\text{out} & \text{if } u \in S(w) \backslash \{w\} \\ 1 \oplus u.\text{out} & \text{otherwise} \end{cases}
\end{array}
$$

*Let $s_0$ be the origin of the unique edge in $E$ with target $v$.*arg.

*The components of $g'$ are constructed as follows:*

$$V' = \begin{cases} (V \backslash S(w)) \oplus S(w) & \text{if } |\{(w.\mathsf{out}, t) \in E\}| = 1 \text{ and } v \notin S(w) \\ V \oplus S(w) & \text{otherwise} \end{cases}$$

$$\begin{aligned} E' &= \{(f_1(s), 1 \oplus u.i) \mid 1 \oplus u \in V' \wedge (s, u.i) \in E \wedge u \neq v\} \\ &\cup \{(2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{fun}), (f_1(s_0), 1 \oplus v.\mathsf{arg})\} \\ &\cup \{(f_2(s), 2 \oplus u.i) \mid 2 \oplus u \in V' \wedge (s, u.i) \in E\} \end{aligned}$$

$$L'(j \oplus u) = L(u) \quad \text{for } j \in \{1, 2\}$$

$$\begin{aligned} S'(2 \oplus u) &= 2 \oplus S(u) \\ S'(1 \oplus u) &= 1 \oplus S(u) \quad \text{if } v \notin S(u) \\ S'(1 \oplus u) &= S(u) \oplus S(w) \quad \text{if } v \in S(u) \end{aligned}$$

$$r' = f_1(r)$$

2. *Let $s_1$ and $s_2$ be the origins of the unique edges in $E'$ with targets $2 \oplus w.\mathsf{return}$ and $1 \oplus v.\mathsf{arg}$ respectively. We modify $E'$, $L'$, and $S'$ as follows:*

$$\begin{aligned} (2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{fun}) &:= (s_1, 1 \oplus v.\mathsf{in}) \\ (s_1, 2 \oplus w.\mathsf{return}) &:= (s_2, 2 \oplus w.\mathsf{in}) \\ (s_2, 1 \oplus v.\mathsf{arg}) &:= \textit{removed} \\ L'(1 \oplus v) &:= \bullet \\ L'(2 \oplus w) &:= \bullet \\ S'(2 \oplus w) &:= \textit{undefined} \end{aligned}$$

*Furthermore, any occurrence of port $2 \oplus w.\mathsf{bind}$ in $E'$ is replaced by $2 \oplus w.\mathsf{out}$. The resulting graph $h$ of the $\beta\circ$-reduction is then the simplification $\sigma(g')$.*

**Escape** An *esc*-redex consists of an Escape node $v$ that has a Bracket node $w$ as its predecessor. We contract the redex in two steps (see Figure 8):

1. Check that the edge $(w.\mathsf{out}, v.\mathsf{return})$ is the only edge originating at $w.\mathsf{out}$, and that the Escape node $v$ is outside the scope of $w$. If any of these conditions do not hold, copy the Bracket node in a way that ensures that the conditions hold for the copy of $w$. The copy of $w$ is called $2 \oplus w$, and the original of $v$ is called $1 \oplus v$. Place $2 \oplus w$ (and its scope) in the scope of $1 \oplus v$.

2. Convert $1 \oplus v$ and $2 \oplus w$ into indirection nodes, which are then removed by the function $\sigma$. Redirect edges so that after simplification, edges that originated at the Escape node's output port $(1 \oplus v.\mathsf{out})$ now start at the root $s_1$ of the Bracket node's body.

Definition 5.3 (Graph Escape). *Given a graph $g \in \mathbb{G}$ with $\vdash^0$ $V$ and $v, w \in V$ such that $L(v) = \sim$, $L(w) = \langle\rangle$, $(w.\mathsf{out}, v.\mathsf{return}) \in E$, $\vdash^0$ contents$(w)$, and level$(w)$ < level$(v)$. Then the contraction of the esc-redex $v$, written $g \to_{esc} h$, is defined as follows:*

1. *We define a transitional graph $g' = (V', L', E', S', r')$ where $V'$,$L'$,$S'$, and $r'$ are constructed as in Definition 5.2.* [4] *The set of edges $E'$ is constructed as follows:*

$$\begin{aligned} E' &= \{(f_1(s), 1 \oplus u.i) \mid 1 \oplus u \in V' \wedge (s, u.i) \in E \wedge u \neq v\} \\ &\cup \{(2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{return})\} \\ &\cup \{(f_2(s), 2 \oplus u.i) \mid 2 \oplus u \in V' \wedge (s, u.i) \in E\} \end{aligned}$$

[4] In Definition 5.2, $v$ and $w$ refer to the application- and lambda nodes of a $\beta\circ$-redex. Here, $v$ stands for the Escape node, and $w$ stands for the Bracket node of the *esc*-redex.

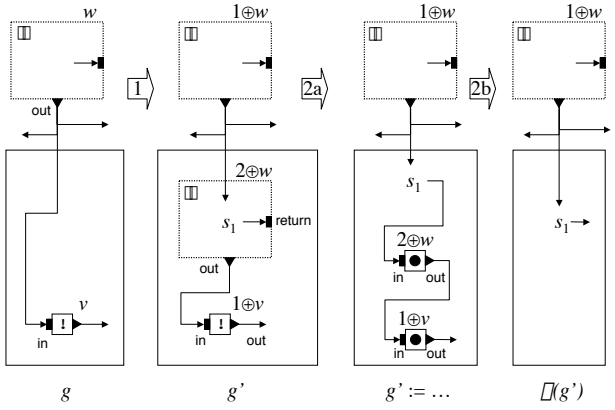



**Figure 9: Run-reduction for PreVIEW graphs**

2. *Let $s_1$ be the origin of the unique edge in $E'$ with target $2 \oplus w.\mathsf{return}$. We modify $E'$, $L'$, and $S'$ as follows:*

$$\begin{aligned} (2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{return}) &:= (2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{in}) \\ (s_1, 2 \oplus w.\mathsf{return}) &:= (s_1, 2 \oplus w.\mathsf{in}) \\ L'(1 \oplus v) &:= \bullet \\ L'(2 \oplus w) &:= \bullet \\ S'(1 \oplus v) &:= \textit{undefined} \\ S'(2 \oplus w) &:= \textit{undefined} \end{aligned}$$

*The resulting graph $h$ of the esc-reduction is $\sigma(g')$.*

**Run** A *run*-redex consists of a Run node $v$ that has a Bracket node $w$ as its predecessor. The contraction of the redex is performed in two steps (see Figure 9):

1. Check that the edge $(w.\mathsf{out}, v.\mathsf{in})$ is the only edge originating at $w.\mathsf{out}$, and that the Run node $v$ is outside the scope of $w$. If any of these conditions do not hold, copy the Bracket node in a way that ensures that the conditions hold for the copy of $w$. The copy of $w$ is called $2 \oplus w$, and the original of $v$ is called $1 \oplus v$. Place $2 \oplus w$ (and its scope) in the same scope as $1 \oplus v$.

2. Convert $1 \oplus v$ and $2 \oplus w$ into indirection nodes, which are then removed by $\sigma$. Redirect edges so that after simplification, edges that originated at the Run node's output port $(1 \oplus v.\mathsf{out})$ now start at the root $s_1$ of the Bracket node's body.

Definition 5.4 (Graph Run). *Given a graph $g \in \mathbb{G}$ with $\vdash^0$ $V$ and $v, w \in V$ such that $L(v) = !$, $L(w) = \langle\rangle$, $(w.\mathsf{out}, v.\mathsf{in}) \in E$, $\vdash^0$ contents$(w)$, and level$(w) \leq$ level$(v)$. Then the contraction of the run-redex $v$, written $g \to_{run} h$, is defined as follows:*

1. *We define a transitional graph $g' = (V', L', E', S', r')$ where $V'$,$L'$,$S'$, and $r'$ are constructed as in Definition 5.2. The set of edges $E'$ is constructed as follows:*

$$\begin{aligned} E' &= \{(f_1(s), 1 \oplus u.i) \mid 1 \oplus u \in V' \wedge (s, u.i) \in E \wedge u \neq v\} \\ &\cup \{(2 \oplus w.\mathsf{out}, 1 \oplus v.\mathsf{in})\} \\ &\cup \{(f_2(s), 2 \oplus u.i) \mid 2 \oplus u \in V' \wedge (s, u.i) \in E\} \end{aligned}$$

2. *Let $s_1$ be the origin of the unique edge in $E'$ with target $2 \oplus w.\mathsf{return}$. We modify $E'$, $L'$, and $S'$ as follows:*

$$\begin{aligned} (s_1, 2 \oplus w.\mathsf{return}) &:= (s_1, 2 \oplus w.\mathsf{in}) \\ L'(1 \oplus v) &:= \bullet \\ L'(2 \oplus w) &:= \bullet \\ S'(2 \oplus w) &:= \textit{undefined} \end{aligned}$$

*The resulting graph $h$ of the run-reduction is $\sigma(g')$.*

## 5.3 Results

Any reduction step on a graph $g = \gamma(M)$ corresponds to a sequence of reduction steps on the term $M$ to expose a redex, followed by a reduction step to contract the exposed redex. Conversely, the contraction of any redex in a term $M$ corresponds to the contraction of a redex in the graph $\gamma(M)$.

THEOREM 5.1 (CORRECTNESS OF GRAPHICAL REDUCTIONS). *Let $g \in \mathbb{G}$, $\delta \in \{\beta\circ, esc, run\}$, $M_1^0 \in \mathbb{M}^0$ and $g = \gamma(M_1^0)$.*

1. *Graph reductions preserve well-formedness:*

$$g \rightarrow_\delta h \text{ implies } h \in \mathbb{G}$$

2. *Graph reductions are sound:*

$$g \rightarrow_\delta h \text{ implies } M_1^0 \rightarrow^* M_2^0 \rightarrow_\delta M_3^0$$
$$\text{for some } M_2^0, M_3^0 \in \mathbb{M}^0 \text{ such that } h = \gamma(M_3^0)$$

3. *Graph reductions are complete:*

$$M_1^0 \rightarrow_\delta M_2^0 \text{ implies } g \rightarrow_\delta h \text{ for some } h \in \mathbb{G}$$
$$\text{such that } h = \gamma(M_2^0)$$

# 6. CONCLUSIONS AND FUTURE WORK

With the goal of better understanding how to extend visual languages with programming constructs and techniques available for modern textual languages, this paper studies and extends a graph-text connection first developed by Ariola and Blom. While the motivation for Ariola and Blom's work was the graph-based compilation of functional languages, only minor changes to their representations and visual rendering were needed to make their results a suitable starting point for our work. We extended this formalism with staging constructs, thereby developing a formal model for generative programming in the visual setting.

In this paper we only presented an abstract syntax for PreVIEW. In the future, it will be important to develop a more user-friendly concrete syntax with features such as multi-parameter functions or color shading to better visualize stage distinctions. This step will raise issues related to parsing visual languages, where we expect to be able to build on detailed previous work on layered [16] and reserved graph grammars [22].

Another important step in developing the theory will be lifting both type checking and type inference algorithms defined on textual representations to the graphical setting. Given the interactive manner in which visual programs are developed, it will also be important to see whether type checking and the presented translations can be incrementalized so that errors can be detected locally and without the need for full-program analysis.

# 7. REFERENCES


[1] `http://www.cs.rice.edu/~besan/proofs.pdf`.

[2] Robin Abraham and Martin Erwig. Header and unit inference for spreadsheets through spatial analyses. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.

[3] Z. M. Ariola and S. Blom. Cyclic lambda calculi. *Lecture Notes in Computer Science*, 1281:77, 1997.

[4] M. Burnett, J. Atwood, R. Walpole Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11:155–206, March 2001.

[5] W. Citrin, M. Doherty, and B. Zorn. Formal semantics of control in a completely visual programming language. In Allen L. Ambler and Takayuki Dan Kimura, editors, *Proceedings of the Symposium on Visual Languages*, pages 208–215, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[6] W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In Volker Haarslev, editor, *Proc. 11th IEEE Int. Symp. Visual Languages*, pages 294–301. IEEE Computer Society Press, 5–9 September 1995.

[7] M. Erwig. Abstract syntax and semantics of visual languages. *Jounral of Visual Languages and Computing*, 9:461–483, October 1998.

[8] Martin Erwig and Margaret M. Burnett. Adding apples and oranges. In *4th International Symposium on Practical Aspects of Declarative Languages*, pages 173–191, 2002.

[9] National Instruments. *LabVIEW Student Edition 6i*. Prentice Hall, 2001.

[10] S. Peyton Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in Excel. *ICFP*, pages 165–176, 2003.

[11] Oleg Kiselyov, Kedar Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *the International Workshop on Embedded Software (EMSOFT '04)*, Lecture Notes in Computer Science, Pisa, Italy, 2004. ACM.

[12] Edward A. Lee. What's ahead for embedded software? *IEEE Computer*, pages 18–26, September 2000.

[13] MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.

[14] National Instruments. LabVIEW. Online at `http://www.ni.com/labview`.

[15] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from `ftp://cse.ogi.edu/pub/tech-reports/README.html`.

[16] Jan Rekers and Andy Schuerr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[17] Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Rice University, August 1994.

[18] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [15].

[19] Walid Taha, Stephan Ellner, and Hongwei Xi. Generating Imperative, Heap-Bounded Programs in a Functional Setting. In *Proceedings of the Third International Conference on Embedded Software*, Philadelphia, PA, October 2003.

[20] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *The Symposium on Principles of Programming Languages (POPL '03)*, New Orleans, 2003.

[21] The MathWorks. Simulink. Online at `http://www.mathworks.com/products/simulink`.

[22] Da-Qian Zhang, Kang Zhang, and Jiannong Cao. A context-sensitive graph grammar formalism for the specification of visual languages. *The Computer Journal*, 44(3):186–200, 2001.

# APPENDIX

# A. PROOFS

**Lemma 4.1 (Normal forms are terms)** $\mathbb{M}_{norm} \subseteq \mathbb{M}$.

**Proof** Let $N \in \mathbb{M}_{norm}$. We proceed by structural induction on $N$. If $N \equiv x$, then we are done since $x \in \mathbb{M}$. If $N \equiv \mathsf{letrec}\ Q\ \mathsf{in}\ x$, then we consider the different cases for declarations in $Q$, and show that the right-hand side of any such declaration is in $\mathbb{M}$.

- $x = x$. Clearly, $x \in \mathbb{M}$.
- $x = \lambda x.N_1$ where $N_1 \in \mathbb{M}_{norm}$. By the inductive hypothesis we have $N_1 \in \mathbb{M}$. It follows that $\lambda x.N_1 \in \mathbb{M}$.
- $x = y\ z$. Trivially holds since $y\ z \in \mathbb{M}$.
- $x = \langle N_1 \rangle$ where $N_1 \in \mathbb{M}_{norm}$. By the inductive hypothesis we have $N_1 \in \mathbb{M}$. It follows that $\langle N_1 \rangle \in \mathbb{M}$.
- $x = \sim N_1$ where $N_1 \in \mathbb{M}_{norm}$. By the inductive hypothesis we have $N_1 \in \mathbb{M}$. It follows that $\sim N_1 \in \mathbb{M}$.
- $x = !\ y$. Clearly, $!\ y \in \mathbb{M}$.

**Lemma 4.2 ($\tau$ maps graphs to normal forms)** *If $g \in \mathbb{G}$ then $\tau(g) \in \mathbb{M}_{norm}$.*

**Proof** By definition, $\tau(g) = mkrec(decl(V), name(r))$. We first show that for any $U \subseteq V$ the set of letrec declarations $decl(U)$ is in normal form, i.e. $decl(U) \subseteq \mathbb{D}_{norm}$. By the definition of *mkrec* it follows that $\tau(g)$ is a normal form. We prove this claim by strong induction on the size of $U$.

We assume that for all $W \subset U$ we have $decl(W) \subseteq \mathbb{D}_{norm}$. The set $decl(U)$ consists of one letrec declaration $x_u = term(u)$ for each top-level node $u \in U$. By a case analysis on the label of $u$, we show that each such declaration has the form specified by the grammar for normal forms. It follows that $decl(U) \subseteq \mathbb{D}_{norm}$.

- $L(u) = \bullet$: The declaration for $u$ is $x_u = x_u$ which is a declaration in normal form.
- $L(u) = \lambda$: The declaration for $u$ is $x_u = \lambda y_u.mkrec(decl(contents(u)), name(s))$ where $s$ is the root of the subgraph of $u$. We have $contents(u) \subset U$ and so by the inductive hypothesis we get $decl(contents(u)) \subseteq \mathbb{D}_{norm}$. Since $name(s)$ is just a variable name, the term $mkrec(decl(contents(u)), name(s))$ is itself a normal form. This establishes that the letrec declaration for $u$ is in normal form.
- $L(u) = @$: The declaration for $u$ is $x_u = name(s_1)\ name(s_2)$ where $s_1$ and $s_2$ are the sources of the incoming edges to the fun and arg ports of $u$ respectively (Well-formedness guarantees that $u$ has exactly two incoming edges). The terms $name(s_1)$ and $name(s_2)$ are just variable names, which establishes that the declaration for $u$ is in normal form.
- $L(u) = \langle\rangle$: The declaration for $u$ is $x_u = \langle mkrec(decl(contents(u)), name(s))\rangle$ where $s$ is the root of the subgraph of $u$. We have $contents(u) \subset U$ and so by the inductive hypothesis we get $decl(contents(u)) \subseteq \mathbb{D}_{norm}$. Since $name(s)$ is just a variable name, the term $mkrec(decl(contents(u)), name(s))$ is itself a normal form. This establishes that the letrec declaration for $u$ is in normal form.
- $L(u) = \sim$ : Similar to the case for $L(u) = \langle\rangle$.
- $L(u) = !$ : The declaration for $u$ is $x_u = !\ name(s)$ where $s$ is the source of the incoming edge to the in port of $u$ (Well-formedness guarantees that $u$ has exactly one such edge). The term $name(s)$ is just a variable name, which establishes that the declaration for $u$ is in normal form.

**Lemma 4.3 ($\nu$ maps terms to normal forms)** *If $M \in \mathbb{M}$ then $\nu(M) \in \mathbb{M}_{norm}$.*

**Proof** Let $M \in \mathbb{M}$. Since $\nu = [\![\_]\!]_{norm} \circ [\![\_]\!]_{pre}$, we prove the claim separately for $[\![\_]\!]_{pre}$ and $[\![\_]\!]_{norm}$.

1. We first show by structural induction on $M$ that $[\![M]\!]_{pre} \in \mathbb{M}_{pre}$.
   - $M \equiv x$. By definition, $x \in \mathbb{M}_{pre}$.
   - $M \equiv \lambda x.M_1$. We have $[\![\lambda x.M_1]\!]_{pre} = (\mathsf{letrec}\ x_1 = \lambda x.[\![M_1]\!]_{pre}\ \mathsf{in}\ x_1)$. By induction, $[\![M_1]\!]_{pre} \in \mathbb{M}_{pre}$. From the grammar for $\mathbb{M}_{pre}$ it follows that $(\mathsf{letrec}\ x_1 = \lambda x.[\![M_1]\!]_{pre}\ \mathsf{in}\ x_1)$ is in $\mathbb{M}_{pre}$.
   - $M \equiv M_1 M_2$. We have $[\![M_1 M_2]\!]_{pre} = (\mathsf{letrec}\ Q_1, Q_2, x_3 = x_1 x_2\ \mathsf{in}\ x_3)$ where $[\![M_1]\!]_{pre} = (\mathsf{letrec}\ Q_1\ \mathsf{in}\ x_1)$ and $[\![M_2]\!]_{pre} = (\mathsf{letrec}\ Q_2\ \mathsf{in}\ x_2)$. By induction, $(\mathsf{letrec}\ Q_1\ \mathsf{in}\ x_1)$ and $(\mathsf{letrec}\ Q_2\ \mathsf{in}\ x_2)$ are both in $\mathbb{M}_{pre}$. From the grammar for $\mathbb{M}_{pre}$ it follows that $(\mathsf{letrec}\ Q_1, Q_2, x_3 = x_1 x_2\ \mathsf{in}\ x_3)$ is in $\mathbb{M}_{pre}$.
   - $M \equiv \mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M$. We have $[\![\mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M]\!]_{pre} = (\mathsf{letrec}\ Q, \overrightarrow{Q_j, x_j = y_j}\ \mathsf{in}\ y)$ where $[\![M_j]\!]_{pre} = (\mathsf{letrec}\ Q_j\ \mathsf{in}\ y_j)$ and $[\![M]\!]_{pre} = (\mathsf{letrec}\ Q\ \mathsf{in}\ y)$. By induction, $(\mathsf{letrec}\ Q_j\ \mathsf{in}\ y_j)$ and $(\mathsf{letrec}\ Q\ \mathsf{in}\ y)$ are all in $\mathbb{M}_{pre}$. From the grammar for $\mathbb{M}_{pre}$ it follows that $(\mathsf{letrec}\ Q, \overrightarrow{Q_j, x_j = y_j}\ \mathsf{in}\ y)$ is in $\mathbb{M}_{pre}$.
   - $M \equiv \langle M_1 \rangle$. Similar to the case for $\lambda x.M_1$.
   - $M \equiv \sim M_1$. Similar to the case for $\lambda x.M_1$.
   - $M \equiv !\ M_1$. Similar for the case for $M_1 M_2$.
2. We now show that $[\![N']\!]_{norm} \in \mathbb{M}_{norm}$ for any $N' \in \mathbb{M}_{pre}$. We proceed by induction over the derivation of the judgment $[\![N']\!]_{norm} = N$ and by a case analysis on the last rule applied.
   - $[\![N]\!]_{norm} = N$. By the premise of this rule, $N \in \mathbb{M}_{norm}$.
   - $[\![\mathsf{letrec}\ \_\ \mathsf{in}\ x]\!]_{norm} = x$. By definition, the variable $x$ is a normal form.
   - $[\![\mathsf{letrec}\ y = \lambda z.N', Q'\ \mathsf{in}\ x]\!]_{norm} = N_2$ where $[\![N']\!]_{norm} = N_1$ and $[\![\mathsf{letrec}\ y = \lambda z.N_1, Q'\ \mathsf{in}\ x]\!]_{norm} = N_2$. By assumption, $(\mathsf{letrec}\ y = \lambda z.N', Q'\ \mathsf{in}\ x) \in \mathbb{M}_{pre}$. This can only be the case if $N' \in \mathbb{M}_{pre}$ and $Q' \subseteq \mathbb{D}_{pre}$. By induction, we get $N_1 \in \mathbb{M}_{norm}$. By a similar argument to the proof of Lemma 4.1, we can show that $\mathbb{M}_{norm} \subseteq \mathbb{M}_{pre}$. Therefore we have $N_1 \in \mathbb{M}_{pre}$ and $(\mathsf{letrec}\ y = \lambda z.N_1, Q'\ \mathsf{in}\ x) \in \mathbb{M}_{pre}$. We can then apply the inductive hypothesis again, and get $N_2 \in \mathbb{M}_{norm}$.
   - $[\![\mathsf{letrec}\ y = \langle N' \rangle, Q'\ \mathsf{in}\ x]\!]_{norm} = N_2$. Similar to the third case.
   - $[\![\mathsf{letrec}\ y = \sim N', Q'\ \mathsf{in}\ x]\!]_{norm} = N_2$. Similar to the third case.
   - $[\![\mathsf{letrec}\ y = z, Q'\ \mathsf{in}\ x]\!]_{norm} = N$ where $y \not\equiv z$ and $[\![(\mathsf{letrec}\ Q'\ \mathsf{in}\ x)[y := z]]\!]_{norm} = N$. By assumption, $(\mathsf{letrec}\ y = z, Q'\ \mathsf{in}\ x) \in \mathbb{M}_{pre}$. This implies that the term $(\mathsf{letrec}\ Q'\ \mathsf{in}\ x)[y := z]$ is also in $\mathbb{M}_{pre}$. We apply the inductive hypothesis and get $N \in \mathbb{M}_{norm}$.

**Lemma 4.4 (Graphing)** *For any $M \in \mathbb{M}$, $\gamma(M)$ is defined and is a unique, well-formed graph.*

**Proof** Given a term $M \in \mathbb{M}$, we show separately that 1) $\gamma(M)$ is defined, 2) $\gamma(M)$ is a well-formed graph, and 3) $\gamma$ is deterministic.

1. We show that $\gamma$ terminates for any input term $M \in \mathbb{M}$ by showing that both $\gamma_{pre}$ and $\sigma$ terminate. The mapping $\gamma_{pre}(M)$ is defined recursively. Each recursive call is performed on a term strictly smaller than $M$. Therefore every chain of recursive applications of $\gamma_{pre}$ is finite, eventually reaching the base case for variables. The only condition in the rules defining

$\gamma_{pre}$ is that node names $v$ be fresh. This requirement can always be satisfied, and therefore $\gamma_{pre}$ is always defined.

To show that $\sigma(g)$ terminates, we observe that each application of the second or third rules in the definition of $\sigma$ removes exactly one indirection node from the intermediate graph $g$. If there is at least one such node in the graph, then one of these two rules can be applied: the side conditions in the premises of these rules only enforce correct pattern matching. This ensures progress towards the well-defined base case for $\sigma$, a graph with no indirection nodes.

2. According to the definition of well-formedness for graphs, we need to consider the following three aspects:
   - Connectivity: By a simple inspection of the rules in the definition of $\gamma_{pre}$, we can verify that these rules only create edges connecting nodes in the graph with the appropriate port types, thereby avoiding dangling edges. Furthermore, for each created node $v$, the function $\gamma_{pre}$ generates exactly one edge for each target port of $v$, and makes $v$ the root of the resulting graph. This ensures that each target port in the graph has exactly one incoming edge associated with it. The definition of $\sigma$ ensures that for any indirection node that is removed, the incoming and outgoing edges are merged into one edge that "skips" the indirection node. Therefore, $\sigma$ does not introduce any dangling edges.
   - Scoping: The construction of a graph for the term $\lambda x.M$ in the definition of $\gamma_{pre}$ creates the subgraph $\gamma_{pre}(M)$ recursively, and places it entirely in the scope of a new lambda node. No edges of the subgraph leave the lambda node. In fact, the only "edge" of the subgraph that is affected by the construction is the root of the subgraph: it is connected to the return port of the new lambda node. Similarly, we can show that the constructions for Bracket and Escape nodes generate well-scoped intermediate graphs. The remaining cases follow by a simple induction on the structure of the term being translated. The mapping $\sigma$ only affects indirection nodes, and therefore does not impact the scoping of the respective graph.
   - Root condition: Each rule in the definition of $\gamma_{pre}$ generates a new node and marks this node as the root of the resulting graph. The only cases in which the root is placed in the scope of a node are the constructions of lambda, Bracket, and Escape nodes. However, in these cases the root is placed only in its *own* scope. This matches the requirements for correct scoping. The only rule in the definition of $\gamma_{pre}$ that does not involve the creation of a new node is the rule for letrec. Here, the claim holds by considering the construction of each subgraph inductively. Again, the mapping $\sigma$ does not affect scoping, and therefore does not impact the root condition.
3. The mapping $\gamma_{pre}$ is defined by induction on the structure of the input term $M$. For any term, there is exactly one rule that can be applied. This ensures that $\gamma_{pre}$ is deterministic.

   The rules in an application of $\sigma$ can be applied in non-deterministic order. However, the final result of a graph without indirection nodes is unique for a given intermediate graph, and therefore $\sigma$ is deterministic.

**Lemma 4.5 (Soundness of Normalization)** *If $M \in \mathbb{M}$ then $\gamma(M) = \gamma(\nu(M))$.*

**Proof** Let $M \in \mathbb{M}$. Since $\nu = [\![\_]\!]_{norm} \circ [\![\_]\!]_{pre}$, we prove the claim separately for $[\![\_]\!]_{pre}$ and $[\![\_]\!]_{norm}$.

1. We first show by structural induction on $M$ that $\gamma(M) = \gamma([\![M]\!]_{pre})$.
   - $M \equiv x$. We have $[\![x]\!]_{pre} = (\mathsf{letrec}\ \_ \ \mathsf{in}\ x)$ and $\gamma_{pre}(x) = \gamma_{pre}(\mathsf{letrec}\ \_ \ \mathsf{in}\ x)$. Because $\gamma_{pre}$ and $\sigma$ are deterministic, we get $\gamma(x) = \gamma([\![x]\!]_{pre})$.
   - $M \equiv \lambda x.M_1$. We have $[\![\lambda x.M_1]\!]_{pre} = (\mathsf{letrec}\ x_1 = \lambda x.[\![M_1]\!]_{pre}\ \mathsf{in}\ x_1)$ We now describe the construction of the graphs $g = \gamma_{pre}(\lambda x.M_1)$ and $h = \gamma_{pre}(\mathsf{letrec}\ x_1 = \lambda x.[\![M_1]\!]_{pre}\ \mathsf{in}\ x_1)$, and then show that $g$ and $h$ simplify to the same graph.
     The graph $g$ consists of a lambda node $v$ as its root. The root of the graph $\gamma_{pre}(M_1)$ is connected to $v$'s return port, and all nodes in this subgraph are placed in the scope of $v$. The graph $h$ represents the term $(\mathsf{letrec}\ x_1 = \lambda x.[\![M_1]\!]_{pre}\ \mathsf{in}\ x_1)$. It contains an indirection node $w$ for $x_1$ as its root, and a lambda node $u$ for the subterm $\lambda x.[\![M_1]\!]_{pre}$. The nodes $u$ and $w$ are connected by an edge $(u.\mathsf{out}, w.\mathsf{in})$. The root of the graph $\gamma_{pre}([\![M_1]\!]_{pre})$ is connected to to $u$'s return port, and all nodes in this subgraph are placed in the scope of $u$.
     From the inductive hypothesis we get $\sigma(\gamma_{pre}(M_1)) = \sigma(\gamma_{pre}([\![M_1]\!]_{pre}$ Since $\sigma$ also removes the indirection node $w$ from $h$, this shows that $g$ and $h$ simplify to the same graph.
   - $M \equiv M_1 M_2$. We have $[\![M_1 M_2]\!]_{pre} = (\mathsf{letrec}\ Q_1, Q_2, x_3 = x_1 x_2\ \mathsf{in}\ x_3)$ where $[\![M_1]\!]_{pre} = (\mathsf{letrec}\ Q_1\ \mathsf{in}\ x_1)$ and $[\![M_2]\!]_{pre} = (\mathsf{letrec}\ Q_2\ \mathsf{in}\ x_2)$. We now describe the construction of the graphs $g = \gamma_{pre}(M_1 M_2)$ and $h = \gamma_{pre}([\![M_1 M_2]\!]_{pre})$, and then show that $g$ and $h$ simplify to the same graph.
     The graph $g$ consists of an application node $v$ as its root. The roots of the graphs $\gamma_{pre}(M_1)$ and $\gamma_{pre}(M_2)$ are connected to $v$'s fun and arg ports respectively. The graph $h$ represents the term $(\mathsf{letrec}\ Q_1, Q_2, x_3 = x_1 x_2\ \mathsf{in}\ x_3)$. It contains an indirection node $w$ for $x_3$ as its root, and one application node $u$ for the subterm $x_1 x_2$. The nodes $u$ and $w$ are connected by an edge $(u.\mathsf{out}, w.\mathsf{in})$. The roots of the graphs $\gamma_{pre}(\mathsf{letrec}\ Q_1\ \mathsf{in}\ x_1)$ and $\gamma_{pre}(\mathsf{letrec}\ Q_2\ \mathsf{in}\ x_2)$ are connected to $u$'s fun and arg ports respectively. From the inductive hypothesis we get $\sigma(\gamma_{pre}(M_1)) = \sigma(\gamma_{pre}(\mathsf{letrec}\ Q_1\ \mathsf{in}\ x_1))$ and $\sigma(\gamma_{pre}(M_2)) = \sigma(\gamma_{pre}(\mathsf{letrec}\ Q_2\ \mathsf{in}\ x_2))$. Since $\sigma$ also removes the indirection node $w$ from $h$, this shows that $g$ and $h$ simplify to the same graph.
   - $M \equiv \mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M$. We have $[\![\mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M]\!]_{pre} = (\mathsf{letrec}\ Q, \overrightarrow{Q_j, x_j = y_j}\ \mathsf{in}\ y)$ where $[\![M_j]\!]_{pre} = (\mathsf{letrec}\ Q_j\ \mathsf{in}\ y_j)$ and $[\![M]\!]_{pre} = (\mathsf{letrec}\ Q\ \mathsf{in}\ y)$. We now describe the construction of the graphs $g = \gamma_{pre}(\mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M)$ and $h = \gamma_{pre}([\![\mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M]\!]_{pre})$, and then show that $g$ and $h$ simplify to the same graph.
     The graph $g$ consists of the graphs $\gamma_{pre}(M_j)$ and $\gamma_{pre}(M)$, where the root of $\gamma_{pre}(M)$ is the root of $g$, and each free variable occurrence $x_j$ in the subgraphs $\gamma_{pre}(M_j)$ and $\gamma_{pre}(M)$ is replaced by the root of the respective graph $\gamma_{pre}(M_j)$.
     The graph $h$ represents the term $(\mathsf{letrec}\ Q, \overrightarrow{Q_j, x_j = y_j}\ \mathsf{in}\ y)$. It consists of the graphs $\gamma_{pre}(\mathsf{letrec}\ Q_j\ \mathsf{in}\ y_j)$ and $\gamma_{pre}(\mathsf{letrec}\ Q\ \mathsf{in}\ y)$ where the root of $\gamma_{pre}(\mathsf{letrec}\ Q\ \mathsf{in}\ y)$ is the root of $h$, and each free variable occurrence $x_j$ in the graph is replaced by the root of the respective graph $\gamma_{pre}(\mathsf{letrec}\ Q_j\ \mathsf{in}\ y_j)$.
     From the inductive hypothesis we get $\sigma(\gamma_{pre}(M_j)) = \sigma(\gamma_{pre}(\mathsf{letrec}\ Q$ and $\sigma(\gamma_{pre}(M)) = \sigma(\gamma_{pre}(\mathsf{letrec}\ Q\ \mathsf{in}\ y))$. This shows that $g$ and $h$ simplify to the same graph.
   - $M \equiv \langle M_1 \rangle$. Similar to the case for$\lambda x.M_1$.
   - $M \equiv \sim M_1$. Similar to the case for $\lambda x.M_1$.

- $M \equiv\ !\ M_1$. Similar to the case for $M_1 M_2$.

2. We now show that for a given $N' \in \mathbb{M}_{pre}$ we have $\gamma(N') = \gamma(\llbracket N' \rrbracket_{norm})$. We proceed by induction over the derivation of the judgment $\llbracket N' \rrbracket_{norm} = N$ and by a case analysis on the last rule applied.
   - $\llbracket N \rrbracket_{norm} = N$. In this case, the claim trivially holds since $\gamma$ is deterministic.
   - $\llbracket \mathsf{letrec}\ \_\ \mathsf{in}\ x \rrbracket_{norm} = x$. We have $\gamma_{pre}(\mathsf{letrec}\ \_\ \mathsf{in}\ x) = \gamma_{pre}(x)$. Since $\sigma$ is deterministic, we get $\gamma(\mathsf{letrec}\ \_\ \mathsf{in}\ x) = \gamma(x)$.
   - $\llbracket \mathsf{letrec}\ y = \lambda z.N', Q'\ \mathsf{in}\ x \rrbracket_{norm} = N_2$. The premises are $\llbracket N' \rrbracket_{norm} = N_1$ and $\llbracket \mathsf{letrec}\ y = \lambda z.N_1, Q'\ \mathsf{in}\ x \rrbracket_{norm} = N_2$. Applying the inductive hypothesis to these premises yields $\gamma(N') = \gamma(N_1)$ and $\gamma(\mathsf{letrec}\ y = \lambda z.N_1, Q'\ \mathsf{in}\ x) = \gamma(N_2)$. When constructing a graph $g$ for the term $(\mathsf{letrec}\ y = \lambda z.N', Q'\ \mathsf{in}\ x)$, the function $\gamma$ recursively constructs a graph for $N'$ which is then made the subgraph of the lambda node representing the term $\lambda z.N'$. Since $\gamma(N') = \gamma(N_1)$, it follows that $g$ is the same graph as $\gamma(\mathsf{letrec}\ y = \lambda z.N_1, Q'\ \mathsf{in}\ x)$ which in turn is equal to $\gamma(N_2)$.
   - $\llbracket \mathsf{letrec}\ y = \langle N' \rangle, Q'\ \mathsf{in}\ x \rrbracket_{norm} = N_2$. Similar to the third case.
   - $\llbracket \mathsf{letrec}\ y = \sim N', Q'\ \mathsf{in}\ x \rrbracket_{norm} = N_2$. Similar to the third case.
   - $\llbracket \mathsf{letrec}\ y = z, Q'\ \mathsf{in}\ x \rrbracket_{norm} = N$. The important premise of this rule is $\llbracket (\mathsf{letrec}\ Q'\ \mathsf{in}\ x)[y := z] \rrbracket_{norm} = N$. Applying the inductive hypothesis yields $\gamma((\mathsf{letrec}\ Q'\ \mathsf{in}\ x)[y := z]) = \gamma(N)$. The graph $g = \gamma_{pre}(\mathsf{letrec}\ y = z, Q'\ \mathsf{in}\ x)$ has one more indirection node than the graph $h = \gamma_{pre}((\mathsf{letrec}\ Q'\ \mathsf{in}\ x)[y := z])$. This indirection node represents the right-hand side $z$ of the declaration $y = z$. However, the function $\sigma$ removes this indirection node, and therefore both $g$ and $h$ simplify to the same graph $\gamma(N)$.

**Lemma 4.6 (Recovery of normal forms)** *If $N \in \mathbb{M}_{norm}$ then $N \equiv_\alpha \tau(\gamma(N))$.*

**Proof** We proceed by structural induction over $N$.

- $N \equiv x$. $\gamma(x)$ is a graph with no nodes and the free variable $x$ as its root. This graph is translated back by $\tau$ to the term $x$.
- $N \equiv \mathsf{letrec}\ Q\ \mathsf{in}\ x$. The function $\gamma$ constructs a graph $g$ by mapping letrec declarations in $Q$ to nodes in $g$. The translation $\tau$ maps nodes in $g$ back to letrec declarations. We assume that $\tau$ associates with any node $u_z$ the name $x_{u_z}$, and with any lambda node $u_z$ the additional name $y_{u_z}$. By a case analysis on the different possibilities for declarations in $Q$, we can show that $N \equiv_{alpha} \tau(g)$.
  - $x = x$. This equation is translated by $\gamma$ to a black hole $v_x$. The black hole is mapped back by $\tau$ to a letrec declaration $x_{v_x} = x_{v_x}$.
  - $x = y\ z$. This equation is translated by $\gamma$ to an application node $v_x$ with two incoming edges $(s_y, v_x.\mathsf{fun})$ and $(s_z, v_x.\mathsf{arg})$. The edge sources $s_y$ and $s_z$ are the roots of the subgraphs for $y$ and $z$ respectively. The application node is mapped back by $\tau$ to a letrec declaration $x_{v_x} = \mathit{name}(s_y)\ \mathit{name}(s_z)$. Since the mapping from node names to variable names in $\tau$ is deterministic, *all* occurrences of $s_y$ and $s_z$ in the resulting term will be mapped to the variables $\mathit{name}(s_y)$ and $\mathit{name}(s_z)$ respectively. Therefore, the declaration $x_{v_x} = \mathit{name}(s_y)\ \mathit{name}(s_z)$ is alpha-equivalent to $x = y\ z$ with respect to both full terms.
  - $x = \lambda y.N_1$. Let $U$ and $s$ be the set of nodes and the root of the graph $\gamma(N_1)$ respectively. The letrec declaration is translated by $\gamma$ to a lambda node $v_x$ with an incoming edge $(s, v_x.\mathsf{return})$ and the set $U$ placed in the scope of $v_x$, i.e. $U = \mathit{contents}(v)$. The lambda node is mapped back by $\tau$ to a letrec declaration $x_{v_x} = \lambda y_{v_x}.\mathit{mkrec}(\mathit{decl}(\mathit{contents}(v)), \mathit{name}($ We observe that $\mathit{mkrec}(\mathit{decl}(\mathit{contents}(v)), \mathit{name}(s)) = \mathit{mkrec}(\mathit{decl}(U), \mathit{name}(s)) = \tau(\gamma(N_1))$. Therefore the resulting letrec declaration is $x_{v_x} = \lambda y_{v_x}.\tau(\gamma(N_1))$. By induction we have $N_1 \equiv_\alpha \tau(\gamma(N_1))$. Furthermore, *all* references to node $v_x$ will be mapped to the variables $x_{v_x}$ (normal reference) or $y_{v_x}$ (lambda parameter reference) in the resulting term. This establishes that the initial and resulting letrec declarations are alpha-equivalent.
  - $x = \langle N_1 \rangle$. Similar to the case for $x = \lambda y.N_1$.
  - $x = \sim N_1$. Similar to the case for $x = \lambda y.N_1$.
  - $x = !\ y$. This equation is translated by $\gamma$ to a run node $v_x$ with an incoming edge $(s_y, v_x.\mathsf{in})$. The edge source $s_y$ is the root of the subgraph for $y$. The run node is mapped back by $\tau$ to a letrec declaration $x_{v_x} = !\ \mathit{name}(s_y)$. Since *all* occurrences of $s_y$ in the resulting term will be mapped to the variable $\mathit{name}(s_y)$, the declaration $x_{v_x} = !\ \mathit{name}(s_y)$ is alpha-equivalent to $x = !\ y$ with respect to both full terms.

**Lemma 4.7 (Completeness of Normalization)** *Let $M_1, M_2 \in \mathbb{M}$. If $\gamma(M_1) = \gamma(M_2)$ then $\nu(M_1) \equiv_{alpha} \nu(M_2)$.*

**Proof** We proceed in two steps. We first show that $\gamma(M) = g$ implies $\nu(M) \equiv_{alpha} \tau(g)$. The main claim follows trivially.

1. Let $\gamma(M) = g$ and $N = \nu(M)$. By Lemma 4.5 we have $\gamma(M) = \gamma(\nu(M))$ and therefore $\gamma(N) = \gamma(\nu(M)) = \gamma(M) = g$. Furthermore, by Lemma 4.6 we have $N \equiv_{alpha} \tau(\gamma(N))$. It follows that $\nu(M) = N \equiv_{alpha} \tau(\gamma(N)) \equiv_{alpha} \tau(g)$.
2. Let $\gamma(M_1) = \gamma(M_2) = g$. From the first part of this proof we know that $\nu(M_1) \equiv_{alpha} \tau(g)$ and that $\nu(M_2) \equiv_{alpha} \tau(g)$. Then clearly $\nu(M_1) \equiv_{alpha} \nu(M_2)$.

**Theorem 4.1 (Main)** *The sets of well-formed graphs and normal forms are one-to-one:*

1. *If $M \in \mathbb{M}$ then $\nu(M) \equiv_{alpha} \tau(\gamma(M))$.*
2. *If $g \in \mathbb{G}$ then $g = \gamma(\tau(g))$.*

**Proof** We prove both claims separately.

1. Let $N \in \mathbb{M}_{norm}$. From Lemma 4.6 we know that $N \equiv_{alpha} \tau(\gamma(N))$. If we let $N = \nu(M)$, then we get $\nu(M) \equiv_{alpha} \tau(\gamma(\nu(M)))$. From Lemma 4.5 it follows that $\nu(M) \equiv_{alpha} \tau(\gamma(M))$.
2. We map the graph $g$ to a normal form $N$ by $\tau$ which is then mapped to a graph $h$ by $\gamma_{pre}$. We show that $\sigma(h) = g$. To this end, we investigate the effects of $\tau$ and $\gamma_{pre}$. The translation $\tau$ maps any top-level node $v$ in $g$ to a letrec declaration of the form $x_v = \mathit{term}(v)$. The function $\gamma_{pre}$ maps subterms of $N$ to nodes in $h$. More precisely, for every declaration $x_v = \mathit{term}(v)$ in $N$, $\gamma_{pre}$ creates a node that we call $\mathit{node}(x_v)$. We can show that after simplification, the graph $h$ only contains the nodes $\{\mathit{node}(x_v) \mid v \in V\}$ where each $\mathit{node}(x_v)$ corresponds to exactly one $v \in V$. Furthermore, these nodes are labeled, connected, and scoped in the same way as the nodes in $g$. We proceed by a case analysis on the label of $v$:
   - $L(v) = \bullet$. The black hole $v$ is mapped by $\tau$ to the declaration $x_v = x_v$, which is mapped back by $\gamma_{pre}$ to an indirection node $\mathit{node}(x_v)$ pointing to itself. The simplification function $\sigma$ will change $\mathit{node}(x_v)$ into a black hole.

- $L(v) = \lambda$. The well-formedness condition ensures that there is exactly one edge of the form $(s, v.\mathsf{return})$ in $g$. The declaration in $N$ for $v$ is then $x_v = \lambda y_v.mkrec(decl(contents(v)), name(s))$. The function $\gamma_{pre}$ maps this declaration to a lambda node $node(x_v)$ and an indirection node placed between the image of $s$ and the return port of $node(x_v)$. The function $\sigma$ removes this indirection node, so that there is an edge from the image of $s$ to $v.\mathsf{return}$. All nodes in $contents(v)$ are mapped by $\tau$ to declarations nested inside the lambda term for $x_v$, which are in turn mapped by $\gamma_{pre}$ to nodes in $contents(node(x_v))$.
- $L(v) = @$. The well-formedness condition ensures that there are exactly two edges in $g$ of the form $(s_1, v.\mathsf{fun})$ and $(s_2, v.\mathsf{arg})$. The declaration in $N$ for $v$ is then $x_v = name(s_1)\ name(s_2)$. This declaration is mapped by $\gamma_{pre}$ to an application node $node(x_v)$ with an indirection node placed between the image of $s_1$ and the fun port of $node(x_v)$, and with a second indirection node placed between the image of $s_2$ and the arg port of $node(x_v)$. These indirection nodes are removed by $\sigma$ so that the images of $s_1$ and $s_2$ are directly connected to the fun and arg ports of $node(x_v)$.
- $L(v) = \langle\rangle$. Similar to the case for $L(v) = \lambda$.
- $L(v) = \sim$. Similar to the case for $L(v) = \lambda$.
- $L(v) = !$. The well-formedness condition ensures that there is exactly one edge in $g$ of the form $(s, v.\mathsf{in})$. The declaration in $N$ for $v$ is then $x_v = !\ name(s)$. This declaration is mapped by $\gamma_{pre}$ to a Run node $node(x_v)$ with an indirection node placed between the image of $s$ and the in port of $node(x_v)$. This indirection node is removed by $\sigma$ so that the image of $s$ is directly connected to the in port of $node(x_v)$.

**Lemma 5.1 (Properties of graph validity)**

1. *Let $M^n \in \mathbb{M}^n$ and $g = \gamma(M^n)$. Then $\vdash^n V$.*
2. *Let $g \in \mathbb{G}$ with $\vdash^n V$. Then $\tau(g) \in \mathbb{M}^n$.*

**Proof** We prove the two claims separately.

1. Let $M^n \in \mathbb{M}^n$ and $g = \gamma(M^n)$. We show that $\vdash^n V$ by structural induction on $M^n$.
   - $M^n \equiv x$. The graph $\gamma(x)$ contains no nodes, and we have $\vdash^n \emptyset$.
   - $M^n \equiv \lambda x.M_1^n$. Let $V_1$ be the set of nodes for the graph $\gamma(M_1^n)$. From the inductive hypothesis we get $\vdash^n V_1$. The graph $g = \gamma(\lambda x.M_1^n)$ consists of a lambda node $v$ with the nodes in $V_1$ placed in the scope of $v$, i.e. $V = V_1 \uplus \{v\}$ and $contents(v) = V_1$. By the definition of graph validity, $L(v) = \lambda$ and $\vdash^n contents(v)$ imply $\vdash^n V$.
   - $M^n \equiv M_1^n M_2^n$. Let $V_1$ and $V_2$ be the sets of nodes for the graphs $\gamma(M_1^n)$ and $\gamma(M_2^n)$ respectively. The graph $g = \gamma(M_1^n M_2^n)$ consists of an application node $v$ with the roots of $\gamma(M_1^n)$ and $\gamma(M_2^n)$ connected to $v$'s fun and arg ports respectively. From the inductive hypothesis we get $\vdash^n V_1$ and $\vdash^n V_2$. This means that $\vdash^n u$ for all nodes $u$ in $toplevel(V_1)$ and in $toplevel(V_2)$. By definition of graph validity, $L(v) = @$ implies $\vdash^n v$. Since $V = V_1 \uplus V_2 \uplus \{v\}$, this establishes that $\vdash^n u$ for all nodes $u$ in $toplevel(V)$, and therefore $\vdash^n V$.
   - $M^n \equiv \mathsf{letrec}\ \overrightarrow{x_j = M_j^n}\ \mathsf{in}\ M^n$. Let $V_j$ and $U$ be the sets of nodes for the graphs $\gamma(M_j^n)$ and $\gamma(M^n)$ respectively. The graph $g = \gamma(\mathsf{letrec}\ \overrightarrow{x_j = M_j^n}\ \mathsf{in}\ M^n)$ consists of all nodes in the sets $V_j$ and $U$, i.e. $V = V_j \uplus U$. From the inductive hypothesis we get $\vdash^n V_j$ and $\vdash^n V$. This means that $\vdash^n u$ for all nodes $u$ in $toplevel(V_j)$ and in $toplevel(U)$, and therefore $\vdash^n V$.
   - $M^n \equiv \langle M_1^{n+1}\rangle$. Let $V_1$ be the set of nodes for the graph $\gamma(M_1^{n+1})$. From the inductive hypothesis we get $\vdash^{n+1} V_1$. The graph $g = \gamma(\langle M_1^{n+1}\rangle)$ consists of a Bracket node $v$ with the nodes in $V_1$ placed in the scope of $v$, i.e. $V = V_1 \uplus \{v\}$ and $contents(v) = V_1$. By the definition of graph validity, $L(v) = \langle\rangle$ and $\vdash^{n+1} contents(v)$ imply $\vdash^n V$.
   - $M^n \equiv \sim M_1^k$ where $n = k + 1$. Let $V_1$ be the set of nodes for the graph $\gamma(M_1^k)$. From the inductive hypothesis we get $\vdash^k V_1$. The graph $g = \gamma(\sim M_1^k)$ consists of an Escape node $v$ with the nodes in $V_1$ placed in the scope of $v$, i.e. $V = V_1 \uplus \{v\}$ and $contents(v) = V_1$. By the definition of graph validity, $L(v) = \sim$ and $\vdash^k contents(v)$ imply $\vdash^{k+1} V$ and so $\vdash^n V$.
   - $M^n \equiv !\ M_1^n$. Let $V_1$ be the set of nodes for the graph $\gamma(M_1^n)$. The graph $g = \gamma(!\ M_1^n)$ consists of a Run node $v$ with the root of $\gamma(M_1^n)$ connected to $v$'s in port. From the inductive hypothesis we get $\vdash^n V_1$. This means that $\vdash^n u$ for all nodes $u$ in $toplevel(V_1)$. By definition of graph validity, $L(v) =!$ implies $\vdash^n v$. Since $V = V_1 \uplus \{v\}$, this establishes that $\vdash^n u$ for all nodes $u$ in $toplevel(V)$, and therefore $\vdash^n V$.
2. Let $g \in \mathbb{G}$ with $\vdash^n V$. We first show that for any $U \subseteq V$, the set of letrec declarations $decl(U)$ contains only right-hand sides in $\mathbb{M}^n$. It follows that $\tau(g) = mkrec(decl(V), r) \in \mathbb{M}^n$. We proceed by strong induction on the size of $U$ and assume that $decl(U)$ contains only right-hand sides in $\mathbb{M}^n$ for all $U \subset V$. We also assume that $\vdash^n V$, i.e. $\vdash^n v$ for all $v \in toplevel(V)$. The set $decl(v)$ contains a declaration $x_v = term(v)$ for each such $v$. By a case analysis on the label of $v$, we show that $term(v) \in \mathbb{M}^n$ for each such $v$.
   - $L(v) = \bullet$. We have $term(v) = x_v$ and clearly, $x_v \in \mathbb{M}^n$.
   - $L(v) = \lambda$. The well-formedness condition ensures that $g$ contains exactly one incoming edge for $v$ of the form $(s, v.\mathsf{return})$. We then have $term(v) = \lambda y_v.mkrec(decl(contents(v)), name(s))$. We know that $contents(v) \subset V$. By the inductive hypothesis we know that all letrec right-hand sides in $decl(contents(v))$ are in $\mathbb{M}^n$. Together with the fact that $name(s)$ expands to a variable name, this establishes that $\lambda y_v.mkrec(decl(contents(v)), name(s))$ is in $\mathbb{M}^n$.
   - $L(v) = @$. The well-formedness condition ensures that $g$ contains exactly two incoming edges for $v$. These edges have the form $(s_1, v.\mathsf{fun})$ and $(s_2, v.\mathsf{arg})$. We then have $term(v) = name(s_1)\ name(s_2)$. We know that $name(s_1)$ and $name(s_2)$ expand to just variable names, and therefore $(name(s_1)\ name(s_2)) \in \mathbb{M}^n$.
   - $L(v) = \langle\rangle$. The well-formedness condition ensures that $g$ contains exactly one incoming edge for $v$ of the form $(s, v.\mathsf{return})$. We then have $term(v) = \langle mkrec(decl(contents(v)), nam$ We know that $contents(v) \subset V$. By the inductive hypothesis we know that all letrec right-hand sides in $decl(contents(v))$ are in $\mathbb{M}^n$. Together with the fact that $name(s)$ expands to a variable name, this establishes that $\langle mkrec(decl(contents(v)), name$ is in $\mathbb{M}^n$.
   - $L(v) = \sim$. Similar to the previous case.
   - $L(v) = !$. The well-formedness condition ensures that $g$ contains exactly one incoming edge for $v$ of the form

$(s, v.\mathsf{in})$. We then have $term(v) = !\ name(s)$. We know that $name(s)$ expands to just a variable name, and therefore $(!\ name(s)) \in \mathbb{M}^n$.

**Theorem 5.1 (Correctness of Graphical Reductions)** *Let $g \in \mathbb{G}$, $\delta \in \{\beta\circ, esc, run\}$, $M^0 \in \mathbb{M}^0$ and $g = \gamma(M^0)$.*

1. Graph reductions preserve well-formedness:

$$g \to_\delta h \text{ implies } h \in \mathbb{G}$$

2. Graph reductions are sound:

$g \to_\delta h$ implies $M_1^0 \to^* M_2^0 \to_\delta M_3^0$ for some $M_2^0, M_3^0 \in \mathbb{M}^0$ such that $h = \gamma(M_3^0)$

3. Graph reductions are complete:

$M_1^0 \to_\delta M_2^0$ implies $g \to_\delta h$ for some $h \in \mathbb{G}$ such that $h = \gamma(M_2^0)$

**Proof** We prove the three claims separately.

1. According to the definition of well-formedness for graphs, we need to consider three aspects:
   - Connectivity: The graph reduction rules avoid dangling edges by first changing the nodes to be removed into indirection nodes. In the case of $\beta\circ$, the difference in arity between application and indirection nodes is reflected in the removal of the edge from the lambda node to the fun port of the application node. The graph reductions *esc* and run maintain correct node arity in a similar fashion.
   - Scoping: The first step of all graph reductions (potentially) moves an entire scope into another scope. Assuming a well-formed original graph, it is clear that this operation cannot create any edges leaving a scope, which is the key requirement for correct scoping. The second step of any graph reduction only *removes* scopes, and the rewiring of edges is limited to edges that did not leave the original scope of the respective redex. Again, it is easy to see that these operations do not give rise to edges that leave a scope in the graph.
   - Root condition: This condition requires that the root of the graph not be nested in any scope in the graph. While nodes may be moved into different scopes in all three graph reduction rules, it is not possible for the root of the graph to be moved in this way: if the lambda node of a $\beta\circ$-redex is also the root of the graph, then the edge from the lambda node to the application node is not unique. Therefore, the lambda node will be copied, and *only the copy* will be moved in the scope of the application node, thereby leaving the root of the graph unchanged. Similar reasoning applies to the graph reductions *esc* and *run*.
2. We consider each case of $\delta$ separately.
   - $\delta = \beta\circ$. Let $g'$ be the result of the first step of the $\beta\circ$-reduction on $g$. We can show that $M_1^0$ can be reduced to a term $M_2^0$ with $\gamma(M_2^0) = g'$ such that the $\beta\circ$-redex in $M_2^0$ is of the form $(\lambda x.M_4^0)\ M_5^0$. From the third part of this theorem it follows that reducing the subterm $(\lambda x.M_4^0)\ M_5^0$ to $(\mathsf{letrec}\ x = M_5^0\ \mathsf{in}\ M_4^0)$ yields a term $M_3^0$ with $h = \gamma(M_3^0)$. To this end, we need to consider two cases for the term $M_1^0$.
     – The redex in $g$ corresponds to a subterm of $M_1^0$ of the form

$$(\mathsf{letrec}\ D_m^n\ \mathsf{in}\ (...(\mathsf{letrec}\ D_1^n\ \mathsf{in}\ \lambda x.M_4^0)))\ M_5^0.$$

     This subterm can be reduced by $m$ *lift*-steps to

$$(\mathsf{letrec}\ D_m^n\ \mathsf{in}\ (...(\mathsf{letrec}\ D_1^n\ \mathsf{in}\ (\lambda x.M_4^0)\ M_5^0))).$$

     This subterm of $M_2^0$ now contains the exposed redex $(\lambda x.M_4^0)\ M_5^0$. Therefore, $M_2^0$ corresponds to $g'$, where the reference to the lambda node being applied is unique, and the lambda and application nodes are in the same scope.
     – The redex in $g$ corresponds to a subterm of $M_1^0$ of the form

$$(\mathsf{letrec}\ D_{m_1}^n\ \mathsf{in}\ (...(\mathsf{letrec}\ D_1^n\ \mathsf{in}\ x_1)))\ M_5^0.$$

     Furthermore, $M_1^0$ contains the declarations

$$x_1 = \mathsf{letrec}\ E_{m_2}^n\ \mathsf{in}\ (...(\mathsf{letrec}\ E_1^n\ \mathsf{in}\ x_2))$$

     through

$$x_m = \mathsf{letrec}\ F_{m_m}^n\ \mathsf{in}\ (...(\mathsf{letrec}\ F_1^n\ \mathsf{in}\ \lambda x.M_4^0)).$$

     These declarations are placed in $M_1^0$ so that the variables $x_1, ..., x_m$ are visible to the redex. By $m_1$ applications of the *lift* reduction rules, we can reduce the redex to $(\mathsf{letrec}\ D_{m_1}^n\ \mathsf{in}\ (...(\mathsf{letrec}\ D_1^n\ \mathsf{in}\ x_1\ M_5^0)))$. Furthermore, the declarations for $x_1, ..., x_m$ can be reduced by *merge* to

$$x_1 = x_2, E_1^n, ..., E_{m_1}^n, ..., x_m = \lambda x.M_4^0, F_1^n, ..., F_{m_m}^n.$$

     By definition of the $\beta\circ$-redex in the graph $g$, the level of the lambda node is no greater than the level of the application node. This means that in $M_1^0$, the occurrence of $x_1$ in the redex is at a level no less than the level of the definition of the variables $x_1, ..., x_m$. Therefore, we can substitute variables using *sub* until we have the subterm $x_m\ M_5^0$. The term obtained in this way still represents $g$. We can then perform one more substitution using *sub*, yielding a term with the subterm $(\lambda x.M_4^0)\ M_5^0$. This reduction corresponds to moving the lambda node in the same scope as the application node. If going from $g$ to $g'$ did require copying the lambda node, then the term with the subterm $(\lambda x.M_4^0)\ M_5^0$ precisely represents $g'$, and we are done. If the lambda node did not need to be copied, then the nodes corresponding to the declaration $x_m = \lambda x.M_4^0$ are now garbage in the corresponding graph. We therefore need to use *gc* to garbage collect all unreferenced declarations in the set

$$x_1 = x_2, E_1^n, ..., E_{m_1}^n, ..., x_m = \lambda x.M_4^0, F_1^n, ..., F_{m_m}^n.$$

     The resulting term is $M_2^0$ representing $g'$.
   - $\delta = esc$. We proceed similarly to the case for $\beta\circ$. The main difference is the use of the *lift* rule for Brackets instead of the *lift* rule for application to expose the Escape redex in the term $M_1^0$.
   - $\delta = run$. We proceed similarly to the case for $\beta\circ$. The main difference is the use of the *lift* rule for Brackets instead of the *lift* rule for application to expose the Run redex in the term $M_1^0$.
3. We consider each case of $\delta$ separately.
   - $\delta = \beta\circ$. If $M_1^0 \to_{\beta\circ} M_2^0$, then we know that $M_1^0$ contains a subterm of the form $(\lambda x.M_3^0)\ M_4^0$. In the graph $g = \gamma(M_1^0)$, this corresponds to a lambda node and an application node that are in the same scope, and with

the lambda node's out port connected to the application node's fun port. From Lemma 5.1 we know that the contents of the lambda node and the set of nodes in $g$ that reach the application node and are in the same scope are valid at level 0. This establishes the fact that $g$ contains a $\beta\circ$-redex. Since the lambda and application nodes are already in the same scope, reducing this redex in $g$ only involves the second step of the beta-reduction for graphs: we remove the lambda and application nodes, and replace edges from the lambda's bind port by edges originating from the application's argument. This corresponds directly to the subterm $\mathsf{letrec}\ x = M_2^0\ \mathsf{in}\ M_1^0$, which is the result of reducing $(\lambda x.M_1^0)\ M_2^0$.

- $\delta = esc$. We proceed similarly to the case for $\beta\circ$. The term reduction involves a redex of the form $\sim\langle M^0\rangle$, which corresponds at the graph level to a Bracket node nested inside an Escape node. From Lemma 5.1 we know that the contents of the Bracket node are valid at level 0, and therefore that the graph contains an *esc*-redex. The reduction of the redex in the graph simply removes the Bracket and Escape nodes, only keeping the contents of the Bracket node. This corresponds to the subterm $M^0$, which is the result of the term reduction.
- $\delta = run$. We proceed similarly to the case for $\beta\circ$. The term reduction involves a redex of the form $!\langle M^0\rangle$, which corresponds at the graph level to a Bracket node connected to the in port of a Run node. Furthermore, the Bracket and Run nodes are in the same scope. From Lemma 5.1 we know that the contents of the Bracket node are valid at level 0, and therefore that the graph contains a *run*-redex. The reduction of the redex in the graph simply removes the Bracket and Run nodes, only keeping the contents of the Bracket node. This corresponds to the subterm $M^0$, which is the result of the term reduction.