The Semantics of Graphical Languages*

Stephan Ellner†
Google, Inc
Mountain View, CA, USA
besan@cs.rice.edu

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

ABSTRACT

Visual notations are pervasive in circuit design, control systems, and increasingly in mainstream programming environments. Yet many of the foundational advances in programming language theory are taking place in the context of textual notations. In order to map such advances to the graphical world, and to take the concerns of the graphical world into account when working with textual formalisms, there is a need for rigorous connections between textual and graphical expressions of computation.

To this end, this paper presents a graphical calculus called Uccello. Our key insight is that Ariola and Blom's work on sharing in the cyclic lambda calculus provides an excellent foundation for formalizing the semantics of graphical languages. As an example of what can be done with this foundation, we use it to extend a graphical language with staging constructs. In doing so, we provide the first formal account of sharing in a multi-stage calculus.

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 of such domains include circuit design and control system design, and examples of mainstream tools include a wide range of hardware CAD design environments, data-flow languages like LabVIEW [12, 16], Simulink [23], and Ptolemy [14], 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.

Recognizing this need, we sought existing accounts of the semantics of graph-based representations of programs, and of formal connections between graph-based representations and visual representations. 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 [3] and spreadsheet-based [1, 10, 13] languages. Citrin et al. give a purely graphical description of an object-oriented language called VIPR [4] and a functional language called VEX [5], but the mapping to and from textual representations is only treated informally. Erwig [9] 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".

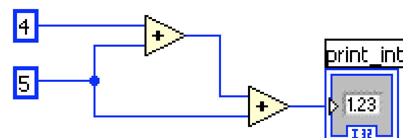
Our key observation is that Ariola and Blom's work on sharing in the cyclic lambda calculus [2] provides a suitable – and possibly *necessary* – starting point. Their work establishes a formal connection between text-based and graph-based representations of programs. The two representations are not one-to-one because of a subtle mismatch between textual and graphical representations in how they express sharing of values. It is not clear how a formal connection between a text-based and a graph-based formulation can be developed without addressing this issue.

To understand the role of sharing, we must first note that it is essential for compact representation. For example, without sharing, the butterfly circuit for computing the FFT would be exponentially larger [6, Figure 32.5].

Next, we note that values are shared in a graph by having more than one edge come out of one output port. In contrast, textual representations use repeated occurrences of a variable to provide the basic mechanism for sharing values. This means that an edge corresponds to a variable use only if that variable is used more than once. If there is only one outgoing edge, there may or may not be a corresponding variable in the textual representation. For example, both of the following C code fragments

```
int x = 4;
int y = 5;              int y = 5;
print_int(x+y+y);       print_int(4+y+y);
```

correspond to a single LabVIEW graph



*Supported by NSF ITR-0113569 "Putting Multi-Stage Annotations to Work", Texas ATP 003604-0032-2003 "Advanced Languages Techniques for Device Drivers", and NSF SOD-0439017 "Synthesizing Device Drivers".

†The work was done while this author was at Rice University.

While the first code fragment assigns a local variable name to the constant 4, the second snippet uses the constant 4 directly. But there is no corresponding distinction in a graph. We do not know of a notion in textual syntax that corresponds exactly to the notion of sharing provided by graphs. This mismatch is a result of the conventions that govern what is accepted as a visual representation of a program and what is accepted as a textual representation. Exploring alternative possibilities for visual representations is beyond the scope of this paper: We take what is in popular use as the starting point. Similarly, we find the possibilities for exploring alternative design options on the textual side relatively limited: Disallowing variable declarations that are used only once, or requiring all subterms to be explicitly named would be unnatural restrictions for the programmer. They are also problematic from the technical point of view. For example, these constraints are not preserved by standard reasoning principles such as substitution. In practice, this would mean that a natural rewrite from one textual representation to another would lead to a program that does not satisfy these additional constraints.

We postulate that this treatment of sharing in both representations is a necessary complication in any connection between a textual representation of a programming language with the richness of the lambda calculus and a graphical representation of the same language.

1.1 Contributions

The key contribution of this paper is the observation that Ariola and Blom's work on sharing in the lambda calculus can be used as the formal basis to capture the abstract syntax of visual languages in the form of graph-based syntax, and to map new concepts in text-based programming languages to a graph-based setting. Using a visual calculus that we call Uccello, we show how concepts from visual languages map to a graph-based representation naturally, and how to extend the original calculus with staging constructs typical in textual multi-stage languages [22]. The concept of scope for the subgraph representing a lambda abstraction suggests a natural visual abstract syntax for staging constructs. While strictly speaking unnecessary, shading is used to distinguish levels, and makes the abstract syntax more readable. The resulting calculus maintains a one-to-one correspondence between visual programs and a multi-stage extension of the text-based lambda-calculus. We then use this formal connection to lift the semantics of multi-stage languages to the graphical setting. Graph reductions have corresponding reductions at the term level, and similarly, term reductions have corresponding reductions at the graph level.

In addition to exposing the complexity of maintaining a correspondence between visual and textual representations of programs, a technical byproduct of studying this particular extension is that we develop a high-level account of sharing in a multi-stage calculus. Ariola and Blom's infrastructure suggests that copying must be performed in both Escape and Run reductions. Defining the graph-based reductions for multi-stage constructs requires us to specify precisely what must be copied. Our main technical result shows that this graph-based semantics is correct.

Non-contributions: This paper does not develop a particular visual language. The visual abstract syntax used in this paper is minimal, and is designed for analytic clarity rather than for visual programming. With respect to visual languages, our work is a study into how some existing visual languages may be related to textual languages. The focus of the work is addressing problems that arise at the level of abstract syntax in both graphical and textual settings, and how a basic obstacle in the connection between the two must be overcome. We do not claim the expertise to argue for or against visual or textual languages, and we do not. We only acknowledge that both styles are in popular use, and formally connecting them would be useful.

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 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 Uccello. Section 4 defines textual representations for Uccello 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 can be found in Ellner's thesis [7].

2. LABVIEW AND LAMBDA-GRAPHS

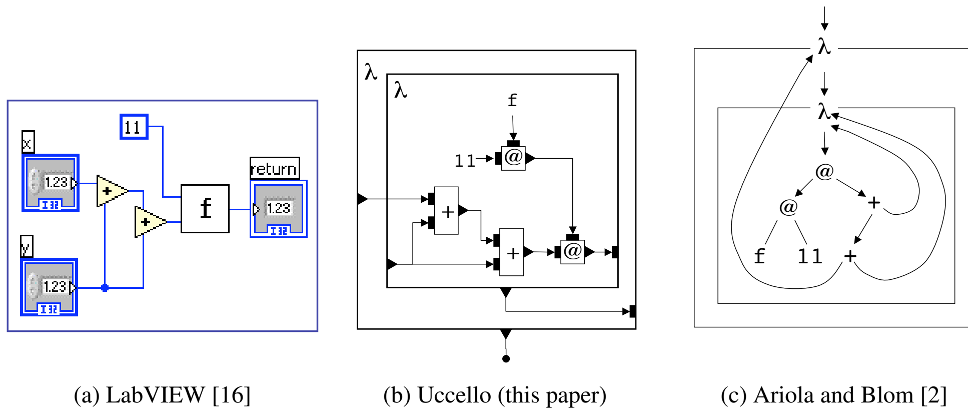
The practical motivation for the calculus studied in the rest of this paper is to extend popular languages such as LabVIEW or Simulink with higher-order functional and staging features. The main abstraction mechanism in LabVIEW is to declare functions; Figure 1 (a) displays the syntax for defining a function with two formal parameters in LabVIEW. Uccello 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 [2] for the same computation. In this representation, a lambda abstraction is drawn as a box describing the scope of the parameter bound by the abstraction. An edge between two nodes in the graph mean that one node corresponds to a subterm of the other in the textual representation. If the direction of the relation is not obvious, an arrow is drawn from the superterm to the subterm.[1] Parameter references are drawn as back-edges to a lambda abstraction. The lambda-graph (c) may appear less closely related to (a) than the Uccello graph (b). But 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 Uccello 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 UCCELLO

The core language features of Uccello are function abstraction and function application as known from the λ-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 [22]. Before defining the syntax of Uccello 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).

[1] In Ariola and Blom's graphs, free variables and constants are not considered nodes by themselves, hence the missing arrows on edges leading to variables and constants in graph (c).



(a) LabVIEW [16] (b) Uccello (this paper) (c) Ariola and Blom [2]

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

3.1 Abstract Visual Syntax

Abstract syntax for textual languages captures higher-level features of syntax than what we see in string-based representations of programs. Especially in a core calculus, abstract syntax focuses on the key constructs that are most interesting from the semantic point of view. This allows us to separate the semantic treatment from the issues of parsing and printing programs. Similarly, as this paper does not address parsing and layout of visual programs, we use a graph-based abstract syntax for Uccello.[2]

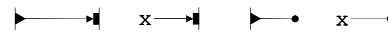
An Uccello program is a graph built from the following components:

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 [2] and represent unresolvable cyclic dependencies that can arise in textual languages with recursion. [3] As shown in Figure 2, nodes are drawn as boxes labeled λ,@,⟨⟩,~, !, and • 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.

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

Ports mark the points in the graph to 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 Uccello. 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 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 Bracket's delaying effect, and return receives the value of that computation.

Edges connect nodes and are drawn as arrows:



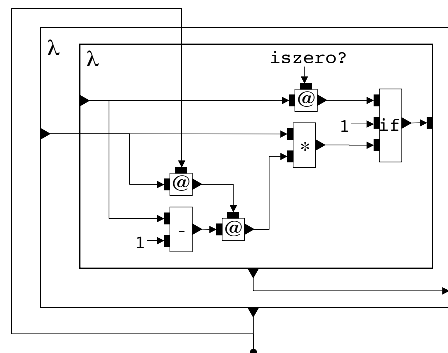
The source of any edge is either the source port of a node or a free variable x. The target of any edge is the target port of some node. The only exception to this is the **root** of the graph. Similar to the root of an abstract syntax tree, it marks the entry-point for evaluating the graph. It is drawn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

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

Example 3.1 (Functional Constructs). *Consider a 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 Uccello, this program is expressed as follows:



[2]An implementation effort beyond the scope of this work shows that the core language can be elegantly extended to allow tailoring the visual notation to different domains [24]. While such syntactic sugar is satisfying from the aesthetic point of view, it makes the formal treatment harder to follow.

[3]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.

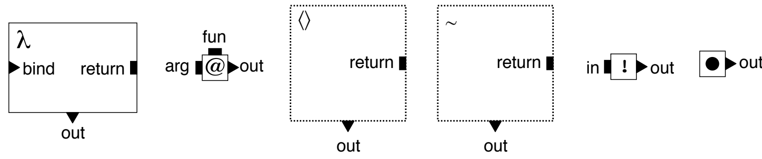


Figure 2: Uccello nodes

Closely following the textual definition, we can visualize the power function as two nested lambda nodes. Consequently, two cascaded application nodes are necessary for `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). *A staged version of the power function can be expressed as follows in MetaOCaml [15]:*[4]

```
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 Uccello as shown to the left of Figure 3. As in the text-based program, in Uccello 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 Uccello 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 Uccello 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.

[4]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.

An Uccello **graph** is then defined as a tuple $g = (V, L, E, S, r)$ where V is a finite set of **nodes**, $L : V \rightarrow \{\lambda, @, \langle\rangle, \sim, !\ , \bullet\}$ is a **labeling function** that associates each node with a label, E is a finite set of **edges**, $S : \{v \in V \mid L(v) \in \{\lambda, \langle\rangle, \sim\}\} \rightarrow \mathcal{P}(V)$ 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.

3.3 Auxiliary Definitions

For any Uccello 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 Uccello 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 Uccello. 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:

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)
λ	{return}	{bind, out}
@	{fun, arg}	{out}
$\langle\rangle, \sim$	{return}	{out}
!	{in}	{out}
$\bullet$	$\emptyset$	{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,

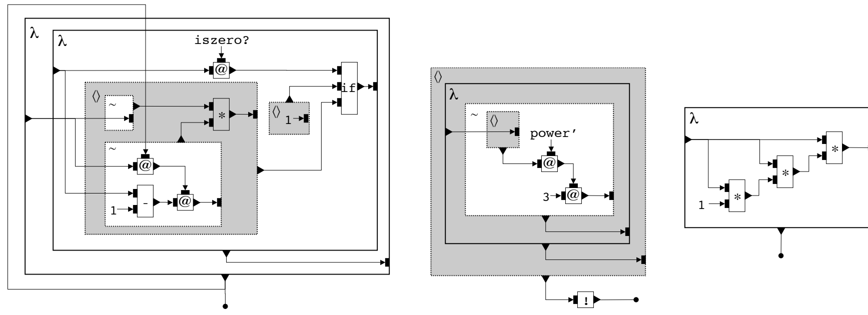
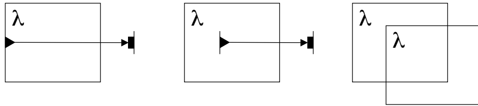


Figure 3: Generating power functions in Uccello

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 $pred(v) = \{(s, v.i) \mid i \in inports(v)\}$.

Scoping - Intuitively, source ports in Uccello 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 $S(v_2) \subseteq S(v_1) \setminus \{v_1\}$.

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 Uccello 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 Uccello programs textually, and define them as follows:

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

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 α-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.

The translation τ 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 τ is applied recursively to the subgraph, and x_v is bound to the term that represents the subgraph. The following definition formalizes this process:

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

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

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

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

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:

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

To avoid the construction of empty letrec terms (letrec D in M) *where D is an empty sequence of declarations, we use the following helper function:*

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

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

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

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

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

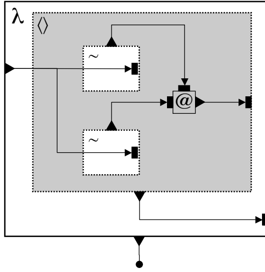
$$\frac{L(v) = \langle\rangle \quad pred(v) = \{(s, v.\mathsf{return})\}}{term(v) = \langle mkrec(decl(contents(v)), name(s))\rangle}$$

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

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

The constraint $v \in toplevel(W)$ in the definition of $decl$ ensures that exactly one equation is generated for each node: if $v \notin toplevel(W)$, then v is in the scope of a different node $w \in W$, and an equation for v is instead included in $term(w)$.

Example 4.1 (Term Construction). *The function τ 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:

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

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 τ 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 τ 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 Uccello 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 [19], 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. *1) $\mathbb{M}_{norm} \subseteq \mathbb{M}$. 2) If $g \in \mathbb{G}$ then $\tau(g) \in \mathbb{M}_{norm}$.*

As we will show, τ 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. [5] 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:

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

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 4.2. *If $M \in \mathbb{M}$ then $\nu(M) \in \mathbb{M}_{norm}$.*

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

$$\frac{}{\llbracket x\rrbracket_{pre} = \mathsf{letrec}\ _\ \mathsf{in}\ x} \qquad \frac{\llbracket M\rrbracket_{pre} = N' \quad x_1 \text{ fresh}}{\llbracket \lambda x.M\rrbracket_{pre} = (\mathsf{letrec}\ x_1 = \lambda x.N'\ \mathsf{in}\ x_1)}$$

$$\frac{\llbracket M_1\rrbracket_{pre} = \mathsf{letrec}\ Q_1\ \mathsf{in}\ x_1 \quad \llbracket M_2\rrbracket_{pre} = \mathsf{letrec}\ Q_2\ \mathsf{in}\ x_2 \quad x_3 \text{ fresh}}{\llbracket M_1 M_2\rrbracket_{pre} = (\mathsf{letrec}\ Q_1, Q_2, x_3 = x_1 x_2\ \mathsf{in}\ x_3)}$$

$$\frac{\llbracket M\rrbracket_{pre} = \mathsf{letrec}\ Q\ \mathsf{in}\ y \quad \overrightarrow{\llbracket M_j\rrbracket_{pre} = \mathsf{letrec}\ Q_j\ \mathsf{in}\ y_j}}{\llbracket \mathsf{letrec}\ \overrightarrow{x_j = M_j}\ \mathsf{in}\ M\rrbracket_{pre} = (\mathsf{letrec}\ Q, \overrightarrow{Q_j, x_j = y_j}\ \mathsf{in}\ y)}$$

$$\frac{\llbracket M\rrbracket_{pre} = N' \quad x_1 \text{ fresh}}{\llbracket \langle M\rangle\rrbracket_{pre} = (\mathsf{letrec}\ x_1 = \langle N'\rangle\ \mathsf{in}\ x_1)} \qquad \frac{\llbracket M\rrbracket_{pre} = N' \quad x_1 \text{ fresh}}{\llbracket \sim M\rrbracket_{pre} = (\mathsf{letrec}\ x_1 = \sim N'\ \mathsf{in}\ x_1)}$$

$$\frac{\llbracket M\rrbracket_{pre} = \mathsf{letrec}\ Q\ \mathsf{in}\ y \quad x_1 \text{ fresh}}{\llbracket !\ M\rrbracket_{pre} = (\mathsf{letrec}\ Q, x_1 = !\ y\ \mathsf{in}\ x_1)}$$

$$\frac{}{\llbracket N\rrbracket_{norm} = N} \qquad \frac{}{\llbracket \mathsf{letrec}\ _\ \mathsf{in}\ x\rrbracket_{norm} = x}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad \llbracket N'\rrbracket_{norm} = N_1 \quad \llbracket \mathsf{letrec}\ y = \lambda z.N_1, Q\ \mathsf{in}\ x\rrbracket_{norm} = N_2}{\llbracket \mathsf{letrec}\ y = \lambda z.N', Q\ \mathsf{in}\ x\rrbracket_{norm} = N_2}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad \llbracket N'\rrbracket_{norm} = N_1 \quad \llbracket \mathsf{letrec}\ y = \langle N_1\rangle, Q\ \mathsf{in}\ x\rrbracket_{norm} = N_2}{\llbracket \mathsf{letrec}\ y = \langle N'\rangle, Q\ \mathsf{in}\ x\rrbracket_{norm} = N_2}$$

$$\frac{N' \notin \mathbb{M}_{norm} \quad \llbracket N'\rrbracket_{norm} = N_1 \quad \llbracket \mathsf{letrec}\ y = \sim N_1, Q\ \mathsf{in}\ x\rrbracket_{norm} = N_2}{\llbracket \mathsf{letrec}\ y = \sim N', Q\ \mathsf{in}\ x\rrbracket_{norm} = N_2}$$

$$\frac{\llbracket (\mathsf{letrec}\ Q\ \mathsf{in}\ x)[y := z]\rrbracket_{norm} = N \quad y \not\equiv z}{\llbracket \mathsf{letrec}\ y = z, Q\ \mathsf{in}\ x\rrbracket_{norm} = N}$$

Figure 4: The translation functions $\llbracket_\rrbracket_{pre} : \mathbb{M} \to \mathbb{M}_{pre}$ and $\llbracket_\rrbracket_{norm} : \mathbb{M}_{pre} \to \mathbb{M}_{norm}$

$$\frac{v \text{ fresh}}{\gamma_{pre}(x) = (\{v\}, \{v \mapsto \bullet\}, \{(x, v.\mathsf{in})\}, \emptyset, v.\mathsf{out})}$$

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

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

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

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

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

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

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

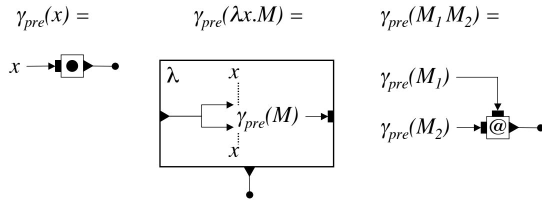
$$\frac{s \neq v.\mathsf{out} \quad (v.\mathsf{out}, t) \notin E \quad \sigma(V, L, E \uplus \{\overrightarrow{(s, t_j)}\}, S \setminus v, r[v.\mathsf{out} := s]) = g}{\sigma(V \uplus \{v\}, L \uplus \{v \mapsto \bullet\}, E \uplus \{(s, v.\mathsf{in})\} \uplus \{\overrightarrow{(v.\mathsf{out}, t_j)}\},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

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

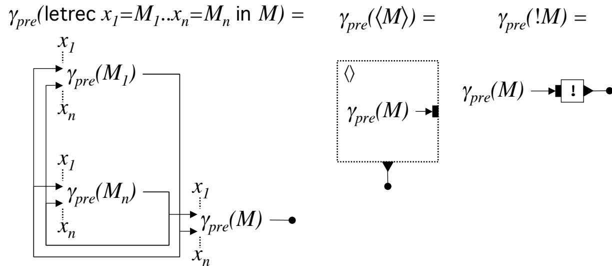
4.3 From Terms to Graphs

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 $\bullet$ 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 one predecessor, we call it an **indirection node**. Since free variables are not represented as nodes in Uccello, 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 Uccello graphs in two steps: A function γ_{pre} maps terms to intermediate graphs, and a simplification function σ maps intermediate graphs to proper Uccello graphs. Before defining these translations formally, we give visual descriptions of γ_{pre} and σ.

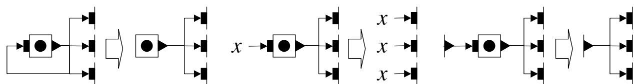


A free variable x is mapped by γ_{pre} to an indirection node with x connected to its in port. A lambda term $\lambda x.M$ maps to a lambda node v, where the pre-graph for M becomes the subgraph of v and all free variables x in the subgraph are replaced by edges originating at the lambda node's bind 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 ($\mathsf{letrec}\ x_1 = M_1, .., x_n = M_n\ \mathsf{in}\ M$), γ_{pre} translates 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 variables 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 Uccello.

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:

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

$S \setminus u$ stands for the result of removing node u from any scope in the graph: $(S \setminus u)(v) = S(v) \setminus \{u\}$. 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 4.3. *For any $M \in \mathbb{M}$, $\gamma(M)$ is defined and is a unique, well-formed graph.*

Using the mappings ν, γ, and τ, 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 γ. The function ν 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 4.4 (Soundness of Normalization). *If $M \in \mathbb{M}$. then $\gamma(M) = \gamma(\nu(M))$.*

Lemma 4.5 (Recovery of normal forms). *If $N \in \mathbb{M}_{norm}$ then $N \equiv_\alpha \tau(\gamma(N))$.*

Lemma 4.6 (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 4.4, they translate to the same graph. This graph is shown in Example 4.1. By Lemma 4.6, the terms M_1, M_2, and M_3 must have the same normal form since they map to the same graph by γ.*

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 UCCELLO

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 λ-U [21]. In the context of λ-U, expression families restrict beta-redexes 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 λ-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:

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

In order to combine Ariola and Blom's reduction semantics for cyclic lambda-terms with the reduction semantics for λ-U, we need to account for the difference in beta-reduction between the two formalisms: While λ-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 λ-U is restricted (implicitly by the β rule) to M^0-terms. We make this restriction explicit by defining which contexts are valid at different levels:

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

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 [2] 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 $\rightarrow$ for the compatible extension of the rules in $\mathcal{R}$, and we write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. The idea behind the *sub* rules is to perform substitution on demand after a function application has been performed. In this sense, the *sub* rules and the $\beta\circ$ rule together mimic the behavior of beta-reduction in λ-U.

5.2 Staged Graphs

To define a reduction semantics for Uccello, 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:

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

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

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

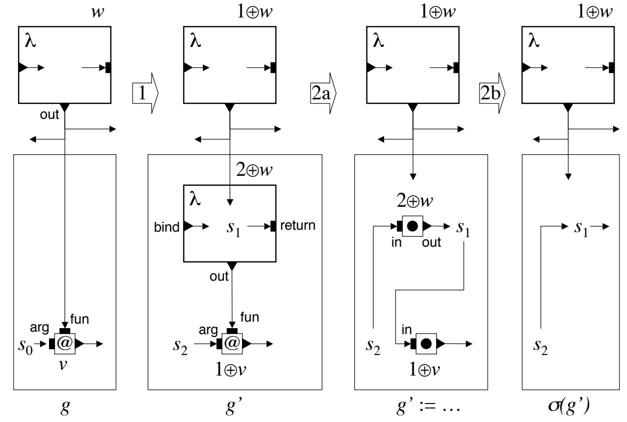


Figure 7: Beta-reduction for Uccello graphs

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 Uccello 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 *sub* rules or the *merge*, *lift*, and *gc* rules (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 an Uccello 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):

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

Figure 6: Term Reduction Rules

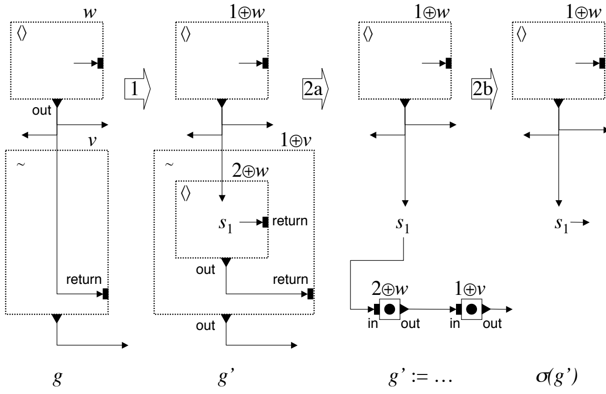


Figure 8: Escape-reduction for Uccello graphs

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 σ (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), $\vdash^0 \{u \mid u \in S(surround(v)) \wedge u \rightsquigarrow v\}$, and $level(w) \leq level(v)$ Then the contraction of the $\beta\circ$-redex v, written $g \rightarrow_{\beta\circ} h$, 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':*

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

Let s_0 be the origin of the unique edge in E with target v.arg. The components of g' are constructed as follows:

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

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

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

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

$$
r' = f_1(r)
$$

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

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

Furthermore, any occurrence of port $2 \oplus w$.bind in E' is replaced by $2 \oplus w$.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):

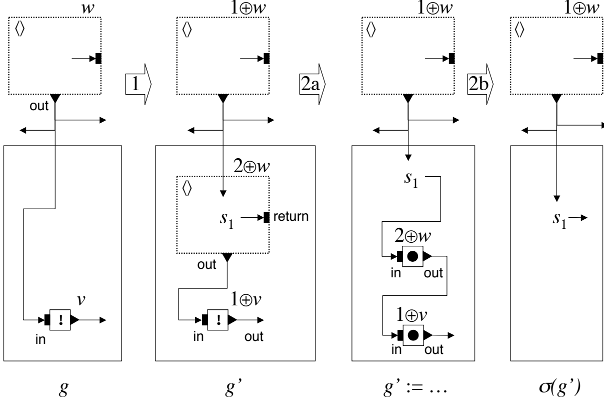


Figure 9: Run-reduction for Uccello graphs

1. Check that the edge (w.out, v.return) is the only edge originating at w.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 σ. Redirect edges so that after simplification, edges that originated at the Escape node's output port ($1 \oplus v$.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.out, v.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. [6] *The set of edges E' is constructed as follows:*

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

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

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

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.out, v.in) is the only edge originating at w.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 σ. Redirect edges so that after simplification, edges that originated at the Run node's output port ($1 \oplus v$.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.out, v.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:

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

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

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

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

5.3 Correctness

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:

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

2. Graph reductions are sound:

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

3. Graph reductions are complete:

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

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 are needed to make their results a suitable basis for formalizing the abstract syntax of visual languages.

[6] 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.

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 the Uccello core calculus. We are already working to develop 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 [18] and reserved graph grammars [25].

Visual and textual languages are often suitable for different tasks. It will be interesting to see if the framework developed here can be used as a basis for Erwig and Meyer's proposal for integrating visual and textual programming [11].

Another important direction 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.

Acknowledgments: We would like to thank Ravi Marwar of National Instruments for presenting Stephan Ellner with a copy of [12]. It was shortly thereafter that Ellner began this work. Kedar Swadi, Samah Abu Mahmeed, Roumen Kaiabachev and Edward Pizzi read and commented on early drafts, and we are grateful for their insightful suggestions. We thank Keith Cooper, Moshe Vardi, Robert "Corky" Cartwright, and Peter Druschel for serving on Ellner's thesis committee. Finally, we thank the participants of DCC'06 for their valuable comments on an earlier draft of this paper [8], as well as the PEPM'07 reviewers for their excellent and detailed feedback.

7. REFERENCES

[1] 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.

[2] Zena M. Ariola and Stefan Blom. Cyclic lambda calculi. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan*, volume 1281 of *Lecture Notes in Computer Science*, pages 77–106. Springer, 1997.

[3] 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.

[4] 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.

[5] 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.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 14th edition, 1994.

[7] Stephan Ellner. Preview: An untyped graphical calculus for resource-aware programming. Master's thesis, Rice University, 2004.

[8] Stephan Ellner and Walid Taha. The semantics of graphical languages. 2006. Available in Informal Proceedings of DCC [20].

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

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

[11] Martin Erwig and Bernd Meyer. Heterogeneous visual languages-integrating visual and textual programming. In *Visual Languages, VL*, pages 318–325, 1995.

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

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

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

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

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

[17] 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`.

[18] 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.

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

[20] Mary Sheeran and Tom Melham, editors. *Sixth International Workshop on Designing Correct Circuits: Vienna, 25–26 March 2006: Participants' Proceedings*. ETAPS 2006, March 2006. A Satellite Event of the ETAPS 2006 group of conferences.

[21] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Available from [17].

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

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

[24] Uccello: A model for supporting higher-order, generative programming in visual languages. Available online from `http://www.resource-aware.org/twiki/pub/RAP/WebHome/final-small.png`, 2006.

[25] 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.