A Graph-Rewriting Perspective of the Beta-Law

Dan R. Ghica[1], Koko Muroya[1,2], and Todd Waugh Ambridge[1]

[1]University of Birmingham
[2]RIMS, Kyoto University

This preliminary report studies a graphical version of Plotkin's call-by-value equational theory, in particular its soundness with respect to operational semantics. Although an equational theory is useful in safe program transformation like compiler optimisation, proving its soundness is not trivial, because it involves analysis of interaction between evaluation flow and a particular sub-program of interest. We observe that soundness can be proved in a direct and generic way in the graphical setting, using small-step semantics given by a graph-rewriting abstract machine previously built for evaluation-cost analysis. This would open up opportunities to think of a cost-sensitive equational theory for compiler optimisation, and to prove contextual equivalence directly in the presence of language extensions.

1 Call-by-Value Equational Theory, Graphically

We first transfer Plotkin's call-by-value equational theory [5] to a graphical setting, assuming the graph representation of terms that is inductively defined in Fig. 1. We aim at the very basic lambda-calculus, whose terms are defined by $t, u ::= x \mid \lambda x.t \mid t\,u$ Both variables and abstractions are referred to as values. A term with no free variables is said to be closed.

A graph G is directed, and may have n open incoming edges ("input") and m open outgoing edges ("output"), in which case we say the graph G has *interface* (n, m) and may write $G(n, m)$. Nodes have specific degrees determined by labels, of which λ and @ are taken from the syntactical constructs (i.e. abstraction and application), and !, ?, D and C_m are taken from the exponential fragment of proof nets [1]. We use bold edges/nodes to represent a bunch of edges/nodes. A C_m-labelled node has m inputs.

A dashed box, called !-box, delimits a graph of interface $G(1, n)$ and comes with one !-labelled node connected to the unique input and n ?-labelled nodes connected to the outputs. This !-box structure, taken from proof nets, is used to manage duplicable sub-graphs. In particular, it is used to represent only and all values, which captures the fact that only a value can be substituted and hence copied in the call-by-value evaluation.

The graph representation $t^\dagger$ of a term t has always one input ("root"), and each of its output corresponds to one occurrence of a free variable of t. In the representation $(\lambda x.t)^\dagger$ of abstraction, where the bound variable m appears m times in t, all m edges corresponds to occurrences of x are connected to a C_m-labelled node.

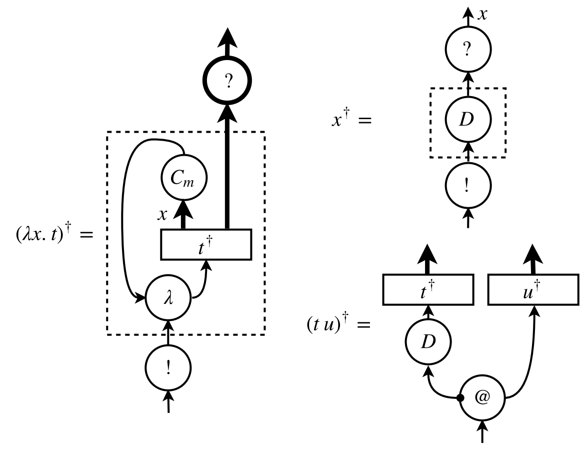


Figure 1: Graph Representation of Terms

A graph-equation formula $G =_g H$ between graphs G and H of the same interface is given by the following rules, as well as the rules that make the graph-equation $=_g$ an equivalence.

$$\frac{G \prec_\chi H}{G =_g H}\ (\chi\text{-rule}) \qquad \frac{G =_g H}{\mathcal{G}[G] =_g \mathcal{G}[H]}\ (\text{Cong})$$

A basic χ-rule is given by a relation $\prec_\chi$ between graphs of the same interface, parameterised by $\chi \in \{\lambda, C, D, ?\}$. All these relations are inspired by cut elimination of proof nets [1]. The λ-rule models elimination of constructs λ (abstraction) and @ (application), and the C-rule models duplication of a value (i.e. a !-box). The other rules, D-rules and ?-rules together models replacement of a single occurrence of a variable with a value. The congruence rule (Cong) involves a *graph-context* $\mathcal{G}$ that is a graph with exactly one special node ("hole") of label $\square$ and arbitrary interface. Replacing the hole with a graph G yields a graph $\mathcal{G}[G]$. We illustrate these rules by an example, in Appendix A.

The difference between our graphical theory and Plotkin's syntactical theory [5] lies in basic rules. Out of the two syntactical basic rules, the α-rule $\lambda x.t =_s \lambda y.(t[y/x])$ (where y is not free in t) becomes trivial in the graphical theory. Difference of variable names does not change the shape of graph representation (recall that variables labelling edges in Fig. 1 are auxiliary). The other syntactical basic rule, the β-rule $(\lambda x.t)\,v =_s t[v/x]$ is now decomposed into four graphical basic rules (λ, C, D and ?). This amounts to extend the lambda-calculus and the syntactical theory with explicit substitutions $t[x \leftarrow u]$. The decomposition in particular discloses duplication of values as the C-rule.

2 Proving Soundness

The key property of an equational theory is *soundness*, consistency with operational semantics. It ensures that two terms equated by the theory are *contextually equivalent*, i.e. indistinguishable in any contexts under the operational semantics.

We turn the soundness theorem of the syntactical theory [5, Thm. 5] graphical, by introducing a graphical version of contextual equivalence. Like a context C is given as a term with one "hole", a *graph-context* $\mathcal{G}$ is defined as a graph with exactly one special node ("hole") of label $\Box$ and arbitrary interface. Replacing the hole $\Box(n,m)$ with a graph $G(n,m)$ of matching interface yields a graph $\mathcal{G}[G]$.

We state soundness below with a black-box operational semantics for now: let $\Downarrow_g$ be an "evaluation" relation between graphs of interface $(1,0)$. Recall that closed terms are represented by graphs of this interface $(1,0)$.

Definition 2.1 (Graph-contextual equivalence)**.** Two graphs $G_1(n,m)$ and $G_2(n,m)$ are *graph-contextually equivalent*, written as $G_1 \cong_g G_2$, if for any graph context $\mathcal{G}[\Box]$ that makes two graphs $\mathcal{G}[G_1]$ and $\mathcal{G}[G_2]$ of interface $(1,0)$, $G_1 \Downarrow_g H_1$ for some graph $H_1(1,0)$ if and only if $G_2 \Downarrow_g H_2$ for some graph $H_2(1,0)$, and moreover $H_1 =_g H_2$.

Theorem 2.2 (Soundness)**.** *For any graphs G and H of interface $(1,0)$, if $G =_g H$ then $G \cong_g H$.*

What matters in proving soundness, in particular contextual equivalence, is formulation of the operational semantics $\Downarrow_g$. In the usual syntactical setting, small-step semantics is not preferred, because it makes control flow of evaluation explicit by decomposing a program into a context and a sub-term under evaluation. It tends to be hard to analyse interaction between a particular sub-term and the control flow. Plotkin indeed uses compositional big-step semantics, which is proved to be equivalent to small-step semantics.

However, our observation is that one can prove contextual equivalence directly using small-step semantics, in a graphical setting. We use in particular the graph-rewriting abstract machine, previously developed for evaluation-cost analysis [3], as small-step semantics.[1] This has a potential benefit of designing a cost-sensitive equational theory, which could be used for compiler optimisation.

The graph-rewriting machine works on graph representation of a whole program, and notably models control of evaluation as a selected edge of the graph. Evaluation is modelled by two kinds of machine steps: searching steps where the evaluation control is moved around the graph to detect a redex, and rewriting steps where a graph-rewriting rule is applied, resembling reduction of a sub-term.

This low-level, less-structured nature of the machine enables us to analyse the interaction between the evaluation control and a particular sub-graph, and hence to prove contextual equivalence, in a simple, step-wise way. Namely, the interaction boils down to the following two situations. The first situation is when the evaluation control physically enters the sub-graph of interest, and the second is when the evaluation control triggers a rewriting that affects the sub-graph.

This proof idea is realised by the following notion of *U-simulation*, a variant of simulation. We give a general definition for a state transition system $\to$ with distinguished final states, where $(\cdot)^+$ denotes the transitive closure. It is adapted from the one we used to prove a version of beta-equivalence in the presence of an exotic language extension [2].

Definition 2.3 (U-simulation)**.** A binary relation R on states is a *U-simulation*, if it satisfies the following two conditions. (I) If $\sigma_1 \ R \ \sigma_2$ and a transition $\sigma_1 \to \sigma_1'$ is possible, then (i) there exists a graph state σ_2' such that $\sigma_2 \to \sigma_2'$ and $\sigma_1' \ R^+ \ \sigma_2'$, or (ii) there exists a sequence $\sigma_1' \to^* \sigma_1''$ of (possibly no) transitions such that $\sigma_1'' \ R \ \sigma_2$. (II) If $\sigma_1 \ R \ \sigma_2$ and no transition is possible from the graph state σ_1, no transition is possible from the graph state σ_2 either. Moreover σ_1 is a final state if and only if σ_2 is a final state.

Intuitively, a U-simulation R is the ordinary simulation between two states (the condition (I-i) in the above definition), "Until" the left sequence of transitions is reduced to the right sequence (the condition (I-ii)) up to the relation R itself. This is typically when the evaluation control visits a sub-graph of interest. The sub-graph may not be visited, which resembles the weak until operator of linear temporal logic. The sub-graph may be duplicated by a rewriting step, which is captured by the transitive closure R^+ in the condition (I-i).

Our graph-rewriting abstract machine [3] is given as a deterministic state transition system $\to$, whose state $\sigma = ((G,e),\delta)$ consists of a graph $G(1,0)$, its selected "control" edge e and additional data δ that is a few stacks used to guide the control flow. A graph $G(1,0)$ induces a unique "initial" state $\mathit{Init}(G)$ and a unique "final" state $\mathit{Final}(G)$. The evaluation relation $\Downarrow_g$ is defined by $G \Downarrow_g H$ iff there exists a sequence $\mathit{Init}(G) \to^* \mathit{Final}(H)$.

Thanks to the following property, the soundness proof boils down to a routine work to show that each basic relation $\prec_\chi$, which gives a basic χ-rule of the graph-equational theory, induces a U-simulation $\overline{\prec_\chi}$.

Proposition 2.4. *Let $\prec$ be a binary relation on graphs with the same interface, and its lifting $\overline{\prec}$ on graph states be defined as follows: $((\mathcal{G}[G_1],\ell),\delta) \ \overline{\prec} \ ((\mathcal{G}[G_2],\ell),\delta)$ iff $G_1 \prec G_2$ and the position ℓ is in the graph-context $\mathcal{G}[\Box]$. If the lifting $\overline{\prec}$ is a U-simulation, the binary relation $\prec$ implies the graph-contextual equivalence $\cong$, i.e. $G \prec H \Rightarrow G \cong H$ for any graphs G and H.*

Using the graph-rewriting abstract machine, contextual equivalence can be proved not only directly but also in a generic way using U-simulations. We expect the graphical perspective would also be useful in the presence of language extensions like ground-type operations, conditional statements and effects. In situations with various effects, contextual equivalence is commonly proved via a sound and complete method, e.g. logical relation [4] and environmental bisimulation [6]. We have seen our method applies to a version of the beta-law for a particular instance of language extension [2].

[1] The graph-rewriting abstract machine can be executed on arbitrary closed terms, using our on-line visualiser: `https://koko-m.github.io/GoI-Visualiser/` .

References

[1] Jean-Yves Girard. Linear logic. *Theor. Comp. Sci.*, 50:1–102, 1987.

[2] Koko Muroya, Steven Cheung, and Dan R. Ghica. The geometry of computation-graph abstraction. In *LICS 2018*, 2018. To appear.

[3] Koko Muroya and Dan R. Ghica. Efficient implementation of evaluation strategies via token-guided graph rewriting. In *WPTE 2017*, 2017.

[4] Andrew M. Pitts. Reasoning about local variables with operationally-based logical relations. In *LICS 1996*, pages 152–163. IEEE Computer Society, 1996.

[5] Gordon Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comp. Sci.*, 1(2):125–259, 1975.

[6] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. In *LICS 2007*, pages 293–302. IEEE Computer Society, 2007.

A Example of Graphical Equation

To illustrate the four kinds of basic rules (λ, C, D and ?), we take as an example two terms T_1 and T_2 below,

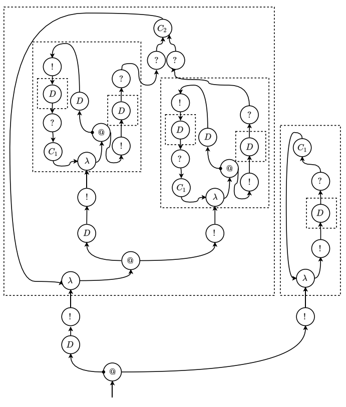
$$\begin{aligned} T_1 &= (\lambda x.(\lambda u.u\,x)\,(\lambda u.u\,x))\,(\lambda y.y) \\ T_2 &= (\lambda u.u\,(\lambda y.y))\,(\lambda u.u\,(\lambda y.y)) \end{aligned}$$

which are beta-equivalent in Plotkin's call-by-value equational theory. Given a basic relation $\prec_\chi$ ($\chi \in \{\lambda, C, D, ?\}$), let a relation $\preccurlyeq_\chi$ between graphs of the same interface be defined by $\mathcal{G}[G_1] \preccurlyeq_\chi \mathcal{G}[G_2]$ iff $G_1 \prec_\chi G_2$. There exists the following chain of relations between graph representations $(T_1)^\dagger$ and $(T_2)^\dagger$ of the two terms:

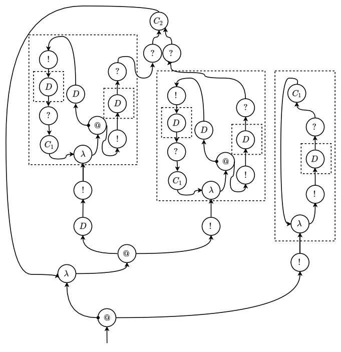
$$\begin{aligned} (T_1)^\dagger = G_0 \preccurlyeq_D G_1 \preccurlyeq_\lambda G_2 \preccurlyeq_C G_3 \\ (\preccurlyeq_?)^2\, G_4\, (\preccurlyeq_?)^2\, G_5\, (\preccurlyeq_D)^2\, G_6 = (T_2)^\dagger, \end{aligned}$$

as shown in Fig. 2. First, the basic relation $\prec_D$ eliminates a !-box structure with a D-labelled node connected to its input (i.e. to a !-labelled node), which amounts to detect an abstraction in the function part of an application. Second, the basic relation $\prec_\lambda$ eliminates a connected pair of a λ-labelled node and an @-labelled node. The rest of the chain models substitution. The basic relation $\prec_C$ duplicates a !-box, i.e. representation of a value. The basic relation $\prec_?$ lets one !-box absorb another !-box connected to one of its output (i.e. to a ?-labelled node), which models replacement of a variable with a value, together with the basic relation $\prec_D$. The formula $(T_1)^\dagger =_g (T_2)^\dagger$ can be proved using the basic rules and the congruence rule accordingly, together with transitivity.

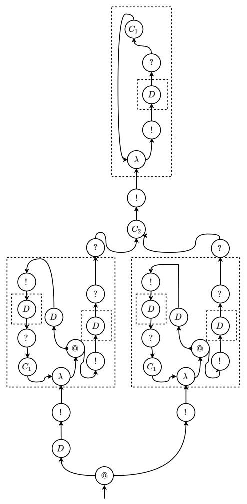
$(T_1)^\dagger =$



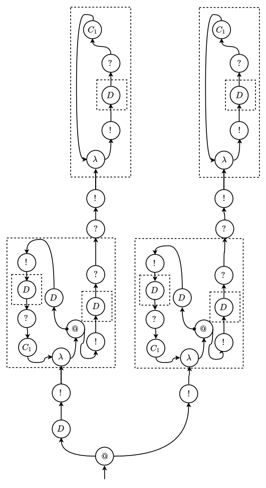
$\preccurlyeq_D$



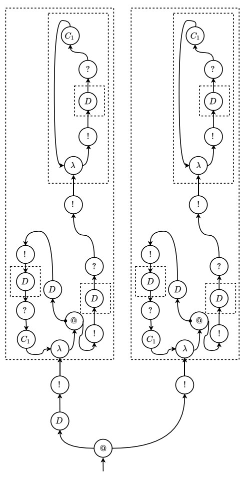
$\preccurlyeq_\lambda$



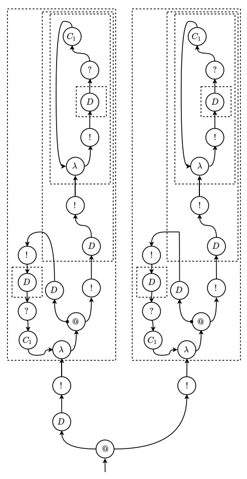
$\preccurlyeq_C$



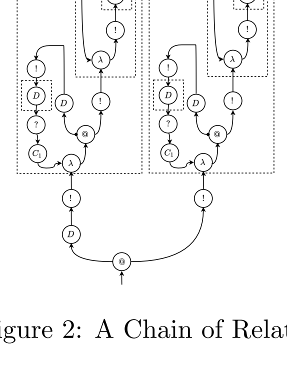
$(\preccurlyeq_?)^2$



$(\preccurlyeq_?)^2$



$(\preccurlyeq_D)^2$



$= (T_2)^\dagger$

Figure 2: A Chain of Relations that Witnesses $(T_1)^\dagger =_g (T_2)^\dagger$