The Geometry of Computation-Graph Abstraction

Koko Muroya
University of Birmingham, UK

Steven W. T. Cheung
University of Birmingham, UK

Dan R. Ghica
University of Birmingham, UK

Abstract

The popular library TensorFlow (TF) has familiarised the mainstream of machine-learning community with programming language concepts such as data-flow computing and automatic differentiation. Additionally, it has introduced some genuinely new syntactic and semantic programming concepts. In this paper we study one such new concept, the ability to extract and manipulate the state of a computation graph. This feature allows the convenient specification of parameterised models by freeing the programmer of the bureaucracy of parameter management, while still permitting the use of generic, model-independent, search and optimisation algorithms. We study this new language feature, which we call 'graph abstraction' in the context of the call-by-value lambda calculus, using the recently developed Dynamic Geometry of Interaction formalism. We give a simple type system guaranteeing the safety of graph abstraction, and we also show the safety of critical language properties such as garbage collection and the beta law. The semantic model suggests that the feature could be implemented in a general-purpose functional language reasonably efficiently.

CCS Concepts •**Theory of computation** → *Semantics and reasoning;* •**Software and its engineering** → **Formal language definitions;**

Keywords Geometry of Interaction, semantics of programming languages, TensorFlow

ACM Reference format:
Koko Muroya, Steven W. T. Cheung, and Dan R. Ghica. 2018. The Geometry of Computation-Graph Abstraction. In *Proceedings of LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, Oxford, United Kingdom, July 9–12, 2018 (LICS '18),* 10 pages.
DOI: 10.1145/3209108.3209127

1 TF as a programming language

TensorFlow (TF) is a popular and successful framework for machine learning, based on a data-flow model of computation [1]. It is programmable via an API, available in several mainstream languages, which is presented as a shallowly embedded domain-specific language (DSL). As a programming language, TF has several interesting features. First of all, it is a *data-flow* language, in which the nodes are mathematical operations (including matrix operations), state manipulation, control flow operations, and various low-level management operations. The programmer uses the host language, which can be Python, Java, Haskell etc., to construct a language term ('computation graph'). The graphs are computationally inert until they are activated in a 'session', in which they can perform or be subjected to certain operations. Two such operations are essential, execution and training. The execution is the usual modus operandi of a data-flow graph, mapping inputs to outputs. Training is the wholesale update of the stateful elements of a computation graph so that a programmer-provided error measure ('loss function') is minimised. The optimisation algorithm computing the new state of the computation graph is also user provided, but it may use automatic differentiation.

Many ingredients of TF are not new, in particular data-flow [7] and automatic differentiation [11]. However, the language quietly introduced a striking new semantic idea, in order to support the training mode of operation of a computation graph, the wholesale update of the stateful elements of a data-flow graph. To enable this operation, the state elements of the graph can be collected into a single vector ('tensor'). These parameters are then optimised by a generic algorithm, such as gradient descent, relative to the data-flow graph itself and some loss function.

We are dissecting TF's variable update into two simpler operations. The first one, which is the focus of this paper, is turning a stateful computation graph into a function, parameterised by its former state. We call this 'graph abstraction' (abs). The second step is the actual update, which in the case of TF is imperative. In this paper we will consider a functional update, realised simply by applying the abstracted graph to the optimised parameters.

For the sake of simplicity and generality, we study graph abstraction in the context of a pure higher-order functional language for transparent data-flow computation. In this language 'sessions' are not required because computation graphs are intrinsic in the semantics of the language. A term will be evaluated as a conventional computation or will result in the construction of a data-flow graph, depending on its constituent elements. Consequently, any term of the language can participate in the formation of data-flow graphs, including lambda abstractions and open terms.

The blending of data-flow into a functional language is an idea with roots in functional reactive programming [18], although our

```
x = tf.placeholder("float")
a = tf.Variable(1)
b = tf.Variable(0)
# Construct computation graph for linear model
model = tf.add(tf.multiply(x, a), b)
with tf.Session() as s:
  s.run(init)
  # Train the model
  s.run(optimiser, data, model, loss_function)
  # Compute y using the updated model
  y = s.run(a) * 7 + s.run(b)
```

Figure 1. Linear regression in TF

LICS '18, Oxford, United Kingdom

978-1-4503-5583-4/18/07…$15.00
DOI: 10.1145/3209108.3209127

semantic model is more akin to self-adjusting computation [2]. The new feature is the ability to collect certain elements of the graph ('variables' in TF lingo, 'cells' in our terminology) into a single data-structure, in order to update it as a whole. The way this is handled in our language is by deprecating a data-flow graph into a lambda expression with the collected cell vector as its argument.

We call our calculus 'idealised tensor flow' (ITF). Let us see how a basic example is handled in TF vs. ITF. For readability, we use a simplified form of the Python bindings of TF. The program is a parameterised linear regression model, optimised then used by applying it to some value (7), as in Fig. 1. The corresponding program in ITF is given below:

$$
\begin{aligned}
&\mathit{let}\ a = \{1\}\\
&\mathit{let}\ b = \{0\}\\
&\mathit{let}\ \mathit{model}\ x = a \times x + b\\
&\mathit{let}\ (\mathit{model}', p) = \mathsf{abs}\ \mathit{model}\\
&\mathit{let}\ p' = \mathit{optimiser}\ \mathit{data}\ p\ \mathit{model}'\ \mathit{loss\_function}\\
&\mathit{let}\ \mathit{model}'' = \mathit{model}'\ p'\\
&\mathit{let}\ y = \mathit{model}''\ 7
\end{aligned}
$$

or, more concisely

$$
\begin{aligned}
&\mathit{let}\ (\mathit{model}', p) = \mathsf{abs}\ (\lambda x.\{1\} \times x + \{0\})\\
&\mathit{let}\ y = \mathit{model}'\ (\mathit{optimiser}\ \mathit{data}\ p\ \mathit{model}'\ \mathit{loss\_function})\ 7
\end{aligned}
$$

In both TF and ITF a data-flow network corresponding to the expression $a \times x + b$ is created, where a and b are variables (cells respectively, indicated by $\{-\}$). New values of a and b are computed by an optimiser parameterised by the model, training data, and a loss function. As it is apparent, in TF the computation graph is constructed explicitly by using constructors such as *tf.add* and *tf.multiply* instead of the host language operators $(+, \times)$. In contrast, in ITF a term is turned into a graph whenever cells are involved. Another key difference is that in TF the variables are updated in place by the optimiser, whereas in ITF the $\mathit{let}\ (f, p) = \mathsf{abs}\ t$ construct 'abstracts' a data-flow graph t into a regular function f, while collecting the default values of its cells in a vector p.

2 ITF

Let $\mathbb{F}$ be a (fixed) set and $\mathbb{A}$ be a set of names (or *atoms*). Let $(\mathbb{F}, +, -, \times, /)$ be a field and $\{(V_a, +_a, \times_a, \bullet_a)\}_{a \in \mathbb{A}}$ an $\mathbb{A}$-indexed family of vector spaces over $\mathbb{F}$. The types T of the languages are defined by the grammar $T ::= \mathbb{F} \mid V_a \mid T \to T$. We refer to the field type $\mathbb{F}$ and vector types V_a as ground types. Besides the standard algebraic operations contributed by the field and the vector spaces, we provide a family of fold operations fold_a, which are always over the bases of the vector space indexed by a:

$$
\begin{aligned}
0, 1, p &: \mathbb{F} && \text{(field constants)}\\
+, -, \times, / &: \mathbb{F} \to \mathbb{F} \to \mathbb{F} && \text{(operations of the field } \mathbb{F})\\
+_a &: V_a \to V_a \to V_a && \text{(vector addition)}\\
\times_a &: \mathbb{F} \to V_a \to V_a && \text{(scalar multiplication)}\\
\bullet_a &: V_a \to V_a \to \mathbb{F}, && \text{(dot product)}\\
\mathsf{fold}_a &: (V_a \to V_a \to V_a) \to V_a \to V_a && \text{(fold)}
\end{aligned}
$$

All vector operations are indexed by a name $a \in \mathbb{A}$, and symbols $+$ and $\times$ are overloaded. The role of the name a will be discussed later. Throughout the paper, we use \$ to refer to a ground-type operation (i.e. $\$ \in \{+, -, \times, /, +_a, \times_a, \bullet_a \mid a \in \mathbb{A}\}$), and # to refer to a primitive operation (i.e. $\# \in \{+, -, \times, /, +_a, \times_a, \bullet_a, \mathsf{fold}_a \mid a \in \mathbb{A}\}$).

Terms t are defined by the grammar $t ::= x \mid \lambda x^T.t \mid t\ t \mid p \mid t\ \#\ t \mid \{p\} \mid \mathsf{A}^T_a(f, x).t$, where T is a type, f and x are variables, and $p \in \mathbb{F}$ is an element of the field. We identify $t\ \mathsf{fold}_a\ u$ with $\mathsf{fold}_a\ t\ u$. The novel syntactic elements of the language are cells $\{p\}$ and a family of type- and name-indexed graph abstractions $\mathsf{A}^T_a(f, x).t$. Graph abstraction as discussed in the introduction is defined as syntactic sugar $\mathsf{abs}\ t \equiv (\mathsf{A}(f, x).(f, x))\ t$.

Let $A \subset_{\mathrm{fin}} \mathbb{A}$ be a finite set of names, Γ a sequence of typed variables $x_i{:}T_i$, and $\vec{p}$ a sequence of elements of the field $\mathbb{F}$ (i.e. a vector over $\mathbb{F}$). We write $A \vdash \Gamma$ if A is the support of Γ. The type judgements are of shape: $A \mid \Gamma \mid \vec{p} \vdash t : T$, and type derivation rules are given below.

$$
\frac{A \vdash \Gamma, T}{A \mid \Gamma, x : T, \Delta \mid - \vdash x : T}
\qquad
\frac{A \mid \Gamma, x : T' \mid \vec{p} \vdash t : T}{A \mid \Gamma \mid \vec{p} \vdash \lambda x^{T'}.t : T' \to T}
$$

$$
\frac{p \in \mathbb{F}}{A \mid \Gamma \mid - \vdash p : \mathbb{F}}
\qquad
\frac{A \mid \Gamma \mid \vec{p} \vdash t : T' \to T \quad A \mid \Gamma \mid \vec{q} \vdash u : T}{A \mid \Gamma \mid \vec{p}, \vec{q} \vdash t\ u : T}
$$

$$
\frac{A \mid \Gamma \mid \vec{p} \vdash t_1 : T_1 \quad A \mid \Gamma \mid \vec{q} \vdash t_2 : T_2 \quad \# : T_1 \to T_2 \to T}{A \mid \Gamma \mid \vec{p}, \vec{q} \vdash t_1\ \#\ t_2 : T}
$$

$$
\frac{p \in \mathbb{F}}{A \mid \Gamma \mid p \vdash \{p\} : \mathbb{F}}
\qquad
\frac{A, a \mid \Gamma, f : V_a \to T', x : V_a \mid \vec{p} \vdash t : T \quad A \vdash \Gamma, T', T}{A \mid \Gamma \mid \vec{p} \vdash \mathsf{A}^{T'}_a(f, x).t : T' \to T}
$$

Note that the rules are linear with respect to the cells $\vec{p}$. In a derivable judgement $A \mid \Gamma \mid \vec{p} \vdash t : T$, the vector $\vec{p}$ gives the collection of all the cells in the term t.

Graph abstraction $\mathsf{A}^{T'}_a(f, x).t$ serves as a binder of the name a and, therefore, it requires in its typing a unique vector type V_a collecting all the cells. Because of name a, this vector type cannot be used outside of the scope of the graph abstraction. An immediate consequence is that variables f and x used in the abstraction of a graph share the type V_a, so that this type cannot be involved in other graph abstractions. This is a deliberate restriction, because abstracting different graphs results in vectors of parameters of unknown, at compile-time, sizes. Mixing such vectors would be a source of unsafe behaviour.

3 Graph-rewriting semantics

We first present an abstract machine, with roots in the Geometry of Interaction [10], which will be used to interpret the language. The state of the machine is a graph with a selected edge (*token*) annotated with extra information. The token triggers graph rewriting in a deterministic way by selecting redexes, and it also propagates information through the graph. This abstract machine is a variant of the Dynamic GoI (DGoI) machine, which has been used to give uniform, cost-accurate models for call-by-need and call-by-value computation [13, 14]. The graph-rewriting style of the DGoI will prove to be a convenient execution model which matches the data-flow-graph intuitions of TF and ITF. The interpretation is 'operational', in the sense that computational costs of its steps are at most linear in the size of the program. Some proofs and in-depth discussions can be only found in an extended version.

3.1 Graphs and graph states

A graph is defined by a set of nodes and a set of edges. The nodes are partitioned into *proper nodes* and *link nodes*. A distinguished list of link nodes forms the *input interface* and another list of link nodes forms the *output interface*. Edges are directed, with at least

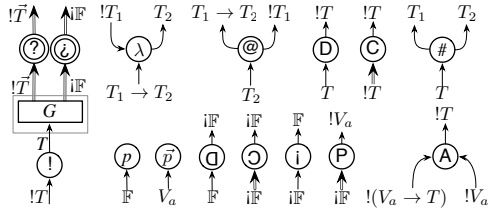


Figure 2. Connection of edges

one endpoint being a link node. An input link (i.e. a link in the input interface) is the source of exactly one edge and the target of no edge. Similarly an output link (i.e. a link in the output interface) is the source of no edge and the target of exactly one edge. Every other link must be the source of one edge and the target of another one edge. A graph may contain adjacent links, but we identify them as a single link, by the notion of 'wire homeomorphism' [12] used in a graphical formalisation of string diagrams. We may write $G(n,m)$ to indicate that a graph G has n links in the input interface and m links in the output interface. From now on we will refer to proper nodes as just 'nodes', and link nodes as 'links'.

Links are labelled by *enriched* types $\tilde{T}$, defined by $\tilde{T} ::= T \mid !T \mid ¡\mathbb{F}$ where T is any type of terms. Adjacent links are labelled with the same enriched types, to be coherent with the wire homeomorphism. If a graph has only one input, we call it 'root', and say the graph has enriched type $\tilde{T}$ if the root has the enriched type $\tilde{T}$. We sometimes refer to enriched types just as 'types', while calling the enriched type ¡$\mathbb{F}$ 'cell type' and an enriched type $!T$ 'argument type'. Note that the types used by the labels are ignored during execution, but they make subsequent proofs easier.

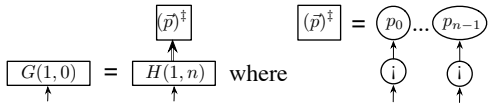
Nodes are labelled, and we call a node labelled with X an 'X-node'. We have several sorts of labels. Some represent the basic syntactic constructs of the lambda calculus: λ (abstraction), @ (application), $p \in \mathbb{F}$ (scalar constants), $\vec{p} \in \mathbb{F}^n$ (vector constants), and # (primitive operations). Node P handles the decomposition of a vector in its elements (coordinates). Node A indicates the graph abstraction. Nodes !, ?, D, and C play the same role as exponential nodes in proof nets [9], handling sharing and copying for argument types. Adaptations of these nodes, namely ¡, ¿, ⊐ and ⊃, are for sharing (but not copying) of cells. Note that we use generalised contractions (C, ⊃) of any input arity, which includes weakening. We sometimes write W (resp. ⋀) to emphasise a contraction C (resp. ⊃) has no inputs and hence is weakening.

We use the following diagrammatic conventions. Link nodes are not represented explicitly, and their labels are only given when they cannot be easily inferred from the rest of the graph. By graphical convention, the link nodes at the bottom of the diagram represent the input interface and they are ordered left to right; the link nodes at the top of the diagram are the output, ordered left to right. A double-stroke edge represents a bunch of edges running in parallel and a double stroke node represents a bunch of nodes.

The connection of edges via nodes must satisfy the rules in Fig. 2, where $!\vec{T}$ denotes a sequence $!T_1, \ldots, !T_m$ of enriched types, and $\# : T_1 \to T_2 \to T$ is a primitive operation. The outline box in Fig. 2 indicates a sub-graph $G(1, n_1 + n_2)$, called a *!-box*. Its input is connected to one !-node ('principal door'), while the outputs are connected to n_1 ?-nodes ('definitive auxiliary doors'), and n_2 ¿-nodes ('provisional auxiliary doors').

A *graph context* is a graph with exactly one distinguished node that has label '□' and any interfaces. We write a graph context as $\mathcal{G}[\square]$ and call the unique extra □-node 'hole'. When a graph G has the same interfaces as the □-node in a graph context $\mathcal{G}[\square]$, we write $\mathcal{G}[G] = \mathcal{G}[\square/G]$ for the substitution of the hole by the graph G. The resulting graph $\mathcal{G}[G]$ indeed satisfies the rules in Fig. 2, thanks to the matching of interfaces.

Finally, we say a graph $G(1,0)$ is *composite*, if its ¡-nodes satisfy the following: (i) they are outside !-boxes; (ii) there is a unique total order on them; and (iii) their outputs are connected to (scalar) constant nodes. Each connected pair of a ¡-node and a constant node is referred to as 'cell'. A composite graph $G(1,0)$ can be uniquely decomposed as below, and written as $G = H \circ (\vec{p})^{\ddagger}$:



where $H(1,n)$ contains no ¡-nodes, $\vec{p} \in \mathbb{F}^n$, and cells are aligned left to right according to the ordering. A graph is said to be *definitive* if it contains no ¡-nodes and all its output links have the cell type ¡$\mathbb{F}$. The graph-rewriting semantics works on composite graphs.

Definition 3.1 (Graph states)**.** A *graph state* $\sigma = ((G,e),\delta)$ consists of a composite graph $G = H \circ (\vec{p})^{\ddagger}$ with a distinguished link e, and *token data* $\delta = (d, f, S, B)$ that consists of a direction $d ::= \uparrow \mid \downarrow$, a rewrite flag $f ::= \square \mid \lambda \mid \$ \mid ? \mid ! \mid \mathsf{F}(n)$, a computation stack $S ::= \square \mid @ : S \mid \star : S \mid \lambda : S \mid p : S \mid \vec{p} : S$, and a box stack $B ::= \square \mid e' : B$, where $p \in \mathbb{F}$, $\vec{p}$ is a vector over $\mathbb{F}$, n is a natural number, and e' is a link of the graph G.

In the definition above we call the link e of (G,e) the 'position' of the token. The rewrite flag determines the applicable graph rewriting. The computation stack tracks intermediate results of program evaluation and the box stack tracks duplications. We call λ, scalar and vector constants 'token values'. Together, the two stacks determine the trajectory of the token, which models the flow of program evaluation.

3.2 Transitions

We define a relation on graph states called *transition* $((G,e),\delta) \to ((G',e'),\delta')$. Transitions are either *pass* or *rewrite*.

Pass transitions occur if and only if the rewrite flag is □. These transitions do not change the overall graph but only the token, as shown in Fig. 3. In particular, the stacks are updated by changing only a constant number of top elements. In the figure, only the node targeted by the token is shown, with token position and direction indicated by a black triangle. The symbol '–' denotes any token value, $k = k_1 \,\$\, k_2$, $X \in \{¡, ¿, ⊐, \mathsf{D}\}$, $Y \in \{¡, ¿, ⊐\}$ and $Z \in \{\mathsf{C}, ⊃\}$.

The order of evaluation is right-to-left. A left-to-right application is possible, but more convoluted for ordinary programs where the argument is often of ground type. An abstraction node (λ) either returns the token with a value λ at the top of the computation stack or triggers a rewrite, if @ is at the top of the computation stack, hence no a downward pass transition for application. The token never exits an application node (@) downward due to rewrite rules which eliminate λ-@ node pairs.

A ground-type operation ($) is applied to top two values of the computation stack, yielding a value $k = k_1 \,\$\, k_2$, in its downward pass transition. The downward pass transition over a fold operation raises the rewrite flag $\mathsf{F}(n)$, using the size of the token value $\vec{p} \in \mathbb{F}^n$.

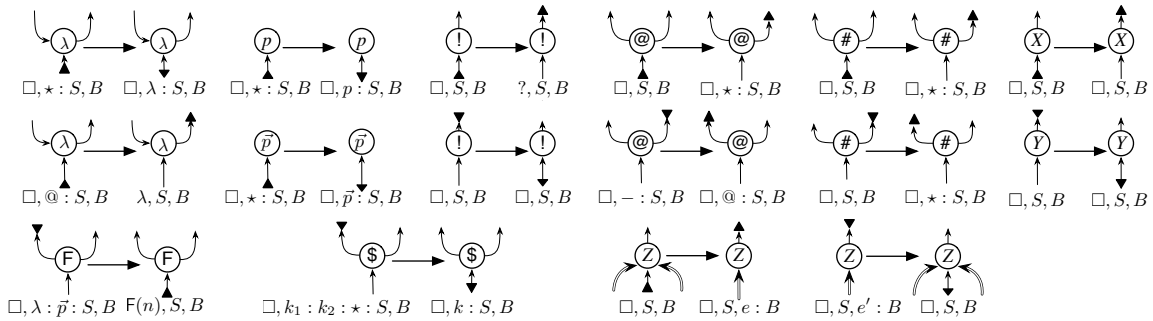


Figure 3. Pass transitions

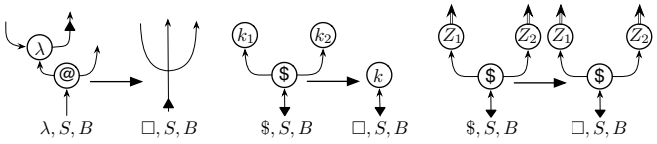


Figure 4. Rewrite transitions: computation

When passing a Z-node (i.e. C or Ↄ) upwards, the token pushes the old position e to the box stack. It uses the top element e' of the box stack as a new position when moving downwards the Z-node, requiring e' to be one of the inputs of the node. The other nodes (?, A and P) only participate in rewrite transitions.

Rewrite transitions are written as

$$((\mathcal{G}[G], e), (d, f, S, B)) \to ((\mathcal{G}[G'], e'), (d, f', S, B'))$$

and they apply to states where the rewrite flag is not □, i.e. to which pass transitions never apply. They replace the (sub-)graph G with G', keeping the interfaces, move the position, and modify the box stack, without changing the direction and the computation stack. We call the sub-graph G 'redex', and a rewrite transition 'f-rewrite transition' if a rewrite flag is f before the transition.

The redex may or may not contain the token position e, but it is always defined relative to it. We call a rewrite transition 'local' if its redex contains the token position, and 'remote' if not. Fig. 4, Fig. 7b and Fig. 7c define local rewrites, showing only the redexes. We explain some rewrite transitions in detail.

The rewrites in Fig. 4 are *computational* in the sense that they are the common rewrites for CBV lambda calculus extended with constants (scalars and vectors) and operations. The first rewrite is the elimination of a λ-@ pair, a key step in beta reduction. Following the rewrite, the incoming output link of λ will connect directly to the argument, and the token will enter the body of the function. Ground-type operations ($) also reduce their arguments, if they are constants k_1 and k_2, replacing them with a single constant $k = k_1 \,\$\, k_2$. If the arguments are not constant-nodes (Z_1 and Z_2 in the figure), then they are not rewritten out, leading to the creation of computation (data-flow) graphs when cells are involved.

Rewrite rules for the fold operations are in Fig. 5. Once the rewrite flag $\mathsf{F}(n)$ is raised, the sub-graph G above the fold node (F) is recursively unfolded n times. This yields G itself with a weakening (W) if $n = 0$, and a graph H_n^n otherwise. If $n > 0$, for any $0 < i < n$, the i-th unfolding H_i^n inserts an application to the basis $\vec{e}_{n-i} \in \mathbb{F}^n$, noting that the bases themselves are not syntactically available.

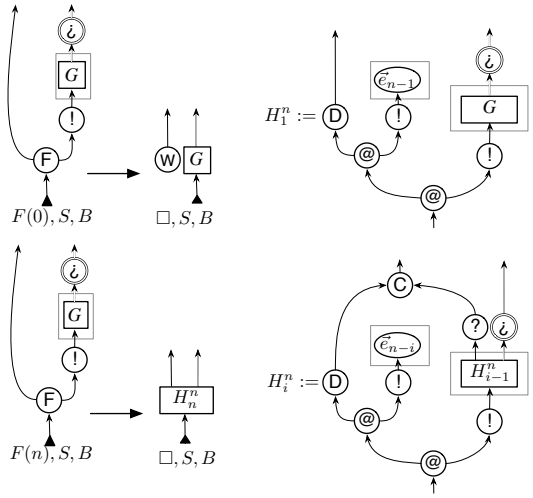


Figure 5. Rewrite transition: unfolding over the bases

The rewrites in Figs. 7a–7c define three classes of rewrites involving !-boxes. They govern duplication of sub-graphs, and the behaviour of graph abstraction, including application of its result function. They are triggered by rewrite flags '?' or '!' whenever the token reaches the principal door of a !-box.

The first class of the !-box rewrites are *remote* rules, in which the rewrites apply to parts of the graphs that have not been reached by the token yet. A redex of a remote rule is determined relative to the token position, namely as a sub-graph of E in Fig. 6 that consists of a !-box H, whose principal door is connected to either a A-node, a P-node with more than one inputs, a C-node, or another !-box. The principal door of the !-box H has to satisfy the following: (i) the node is 'box-reachable' (see Def. 3.2 below) from one of definitive auxiliary doors of the !-box G (in Fig. 6), and (ii) the node is in the same 'level' as one of definitive auxiliary doors of the !-box G, i.e. the node is in a !-box if and only if the door is in the same !-box.

Definition 3.2 (Box-reachability)**.** In a graph, a node/link v is *box-reachable* from a node/link v' if there exists a finite sequence of directed paths $p_0, \ldots, p_i$ such that: (i) if $i > 0$, for any $0 \le j < i$, the path p_j ends with the root of a !-box and the path p_{j+1} begins with an output link of the !-box, and (ii) the path p_0 begins with v and the path p_i ends with v'.

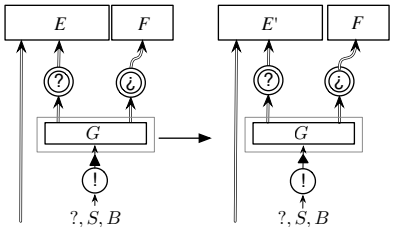


Figure 6. Remote rules triggering

We call the sequence of paths in the above definition 'box-path'. Box-reachability is more general than normal graph reachability, since it may involve a !-box whose doors are not connected.

In order to define the remote rewrite rules let us introduce some notation. We write $G[X/Y]$ for a graph G in which all Y-nodes are replaced with X-nodes of the same signature, and write $G[\epsilon/Y]$ for a graph G in which all Y-nodes (which must have one input and one output) are replaced with links. The remote rewrite rules are given in Fig. 7a.

The top left remote rule is graph abstraction, that takes into account the sub-graph $G \circ (\vec{p})^{\ddagger}$ outside its redex (i.e. the !-box H, its doors and the A-node). The sub-graph $G \circ (\vec{p})^{\ddagger}$ contains exactly all nodes that are graphically reachable, in a directed way, from auxiliary doors (¿) of the !-box H. It is indeed a composite graph, with G containing only Ↄ-nodes or ¿-nodes, due to the typing. The cells $(\vec{p})^{\ddagger}$ may not be all the cells of the whole graph, but a unique and total order on them can be inherited from the whole graph.

Upon applying the graph abstraction rule, the two input edges of the A-node will connect to the result of graph abstraction, a function and arguments. The function is created by replacing the cells $(\vec{p})^{\ddagger}$ with a projection (P), inserting a λ-node and a dereliction (Ɑ). A copy of the cells used by other parts of the graph is left in place, which means the sub-graph $G \circ (\vec{p})^{\ddagger}$ is left unchanged. Another copy is transformed into a single vector node $(\vec{p})$ and linked to the second input of graph abstraction, which now has access to the current cell values. The unique and total ordering of cells $(\vec{p})^{\ddagger}$ is used in introducing the P-node and the $\vec{p}$-node, and makes graph abstraction deterministic.

Note that the graph abstraction rule is the key new rule of the language, and the other remote rules are meant to support and complement this rule. These remote rules can involve nodes only reached by box-reachability, because we want all parameters of a model to be extracted in graph abstraction, including those contributed, potentially, by its free variables. A 'shallow' local version of graph abstraction would be simpler and perhaps easier to implement but not as powerful or interesting.

The bottom left remote rule eliminates a contraction node (C), and replicates the !-box H connected to the contraction. The bottom right rule handles vector projections. Any graph H handling a vector value with n dimensions is replicated n times to handle each coordinate separately. The projected value is computed by applying the dot product using the corresponding standard base. In these two rules, the names in H are refreshed using the name permutation action π_N, where $N \subseteq \mathbb{A}$, defined as follows: all names in N are preserved, all other names are replaced with fresh (globally to the whole graph) names.

Names indexing the vector types must be refreshed, because as a result of copying, any graph abstraction may be executed several times, and each time the resulting computation graphs and cells must be kept distinct from previously abstracted computation graphs and cells. Note that in general types are ignored during execution but including them in the graphs makes proofs easier.

The top right remote rule cause an 'absorption' of the !-box H into the !-box H' it is connected to. Because the ?-nodes of !-boxes arise from the use of global or free variables, this box-absorption process models that of closure-creation in a conventional operational semantics. The !-box H' in Fig. 7a is required not to be the !-box where the token position is.

The local version of absorption, where the lower !-box has the token position in it, belongs to the second class of !-box rewrites shown in Fig. 7b. After this local absorption is exhaustively applied, the rewrite flag changes from '?' to '!', and the last class of rewrites, shown in Fig. 7c are enabled. These rules handle copying of shared closed values, i.e. !-boxes accompanied by no ?-nodes.

The first two rules in Fig. 7c ($Y \notin \{\mathsf{D}, \mathsf{C}\}$) change rewrite mode to pass mode, by setting the rewrite flag to □. The third rewrite copies a !-box. It requires the top element e of the box stack to be one of input links of the contraction node (C). The link e determines the copy of the !-box G that has the new token position in. As in the remote duplication rule, names are refreshed in the new copies.

All transitions presented so far are well-defined.

Proposition 3.3 (Form preservation)**.** *All transitions send a graph state to another graph state, in particular a composite graph $G \circ (\vec{p})^{\ddagger}$ to a composite graph $G' \circ (\vec{p})^{\ddagger}$ of the same type.*

Proof. Transitions make changes only in definitive graphs, keeping the cells $(\vec{p})^{\ddagger}$ which contains only constant nodes and i-nodes. Transitions do not change redex interfaces. □

Recall that we identify adjacent links in a graph as a single link, using wire homeomorphism. All transitions can be made consistent with wire homeomorphism by incorporating the 'identity' pass transition that only changes the token position along a link.

All the pass transitions are deterministic and so are local rewrites. Remote and copying rewrites are not deterministic but are confluent, as no redexes are shared between rewrites. Therefore, the overall beginning-to-end execution is deterministic.

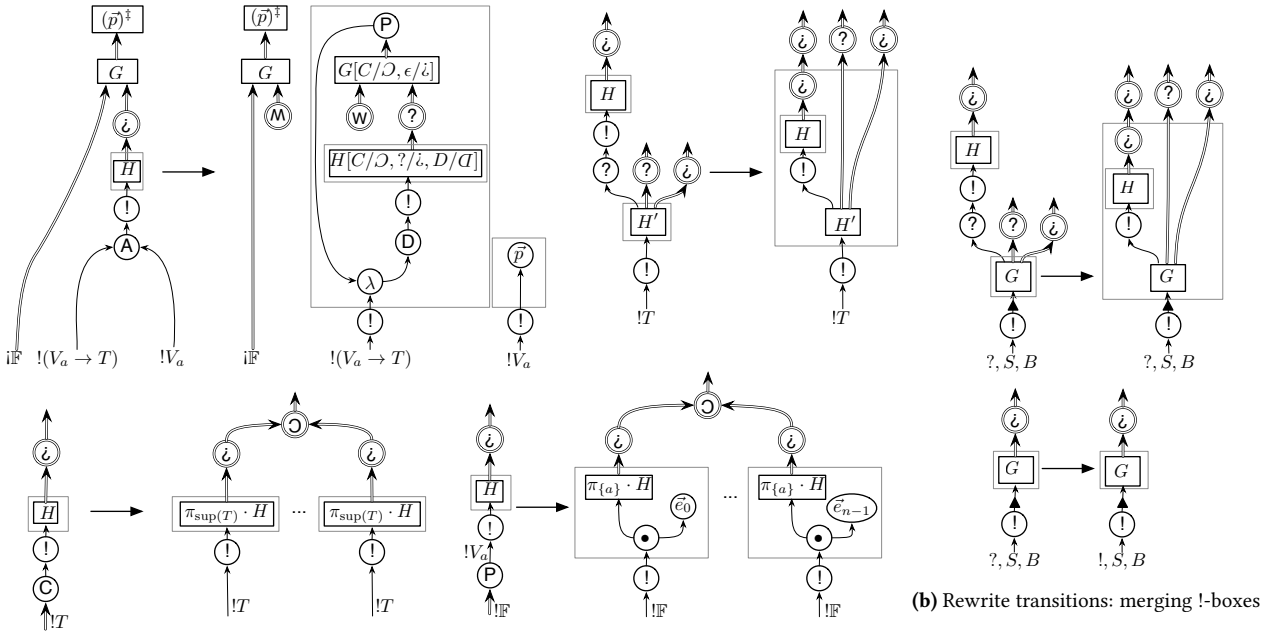
Definition 3.4 (Initial/final states and execution)**.** Let G be a composite graph with root e. An *initial* state $Init(G)$ on the graph G is given by $((G, e), (\uparrow, \square, \star : \square, \square))$. A *final* state $Final(G, \kappa)$ on the graph G, with a token value κ, is given by $((G, e), (\downarrow, \square, \kappa : \square, \square))$. An *execution* on the graph G is any sequence of transitions from the initial state $Init(G)$.

Proposition 3.5 (Determinism of final states)**.** *For any graph state σ, the final state $Final(G, \kappa)$ such that $\sigma \to^* Final(G, \kappa)$ is unique up to name permutation, if it exists.* □

Corollary 3.6 (Determinism of executions)**.** *For any initial state $Init(H)$, the final state $Final(G, \kappa)$ such that $Init(H) \to^* Final(G, \kappa)$ is unique up to name permutation, if it exists.*

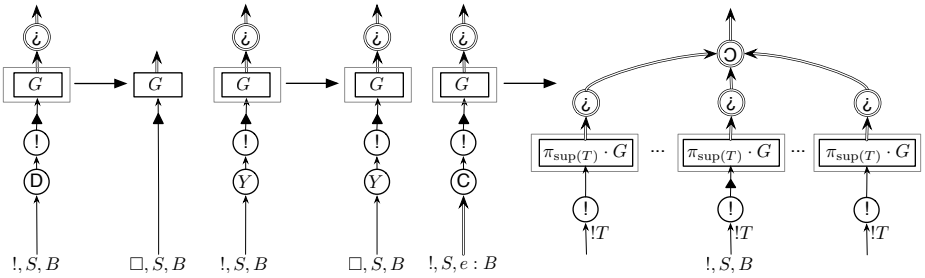
3.3 Translation of terms to graphs

A derivable type judgement $A \mid \Gamma \mid \vec{p} \vdash t : T$ is inductively translated to a composite graph $(A \mid \Gamma \mid \vec{p} \vdash t : T)^{\dagger}$, as shown in Fig. 8, where names in type judgements are omitted. The top left graph



(a) Remote rules: graph abstraction, absorption, contraction, projection

(b) Rewrite transitions: merging !-boxes



(c) Rewrite transitions: copying closed !-boxes

Figure 7. Rewrite transitions: !-boxes

in the figure shows the general pattern of the translation, where $(A \mid \Gamma \mid \vec{p} \vdash t : T)^{\dagger}$ has three components: weakening nodes (W), cells $P_t = (\vec{p})^{\ddagger}$, and the rest G_t. The translation uses variables as additional annotations for links, to determine connection of output links. In the figure, the annotation $!\Gamma$ denotes the sequence $x_0 : !T_0, \ldots, x_{m-1} : !T_{m-1}$ of variables with enriched types, made from $\Gamma = x_0 : T_0, \ldots, x_{m-1} : T_{m-1}$, and $!\Delta$ is made from Δ in the same way. The other annotations are restrictions of $!\Gamma$. Let $\mathrm{FV}(u)$ be the set of free variables of a term u. The annotation $!\Gamma_1$, appearing in inductive translations of typing rules with one premise, is the restriction of $!\Gamma$ to $\mathrm{FV}(t)$, and $!\Gamma_0$ is the residual. The annotations $!\Gamma_t$, $!\Gamma_{tt'}$ and $!\Gamma_{t'}$, in translations of typing rules with two premises, are restrictions of $!\Gamma$ to $\mathrm{FV}(t)\backslash\mathrm{FV}(t')$, $\mathrm{FV}(t) \cap \mathrm{FV}(t')$ and $\mathrm{FV}(t')\backslash\mathrm{FV}(t)$, respectively. Note that the translation is not compositional in the component of weakening nodes (W).

3.4 Soundness

The first technical result of this paper is soundness, which expresses the fact that well typed programs terminate correctly, which means they do not crash or diverge. The challenge is, as expected, dealing with the graph abstraction and related rules.

Theorem 3.7 (Soundness)**.** *For any closed program t such that $A \mid - \mid \vec{p} \vdash t : T$, there exist a graph G and a token value κ such that: $\mathit{Init}((A \mid - \mid \vec{p} \vdash t : T)^{\ddagger}) \to^{*} \mathit{Final}(G, \kappa)$.*

Our semantics produces two kinds of result at the end of the execution. One, intensional result, is the graph G. It will involve the cells of values $\vec{p}$ and computation depending on them, which are not reduced during execution. The other one, extensional result, is the value κ carried by the token as it 'exits' the graph G. The value κ will always be either a scalar, or a vector, or the symbol λ indicating a function-value result.

The proof, omitted here due to the limited space, uses logical predicates on definitive graphs, to characterise safely-terminating graphs inductively on types. The key step is to prove that graph abstraction preserves the termination property of a graph, which involves an analysis of sub-graphs that correspond to data-flow (i.e. ground-type computation only with cells, constants and ground-type operations). Graph abstraction enables more rewrites to be applied to a graph, by turning non-duplicable cells into duplicable function arguments of ground types. Thanks to the call-by-value evaluation, the newly enabled rewrites can only involve the data-flow sub-graphs and hence do not break the termination property.

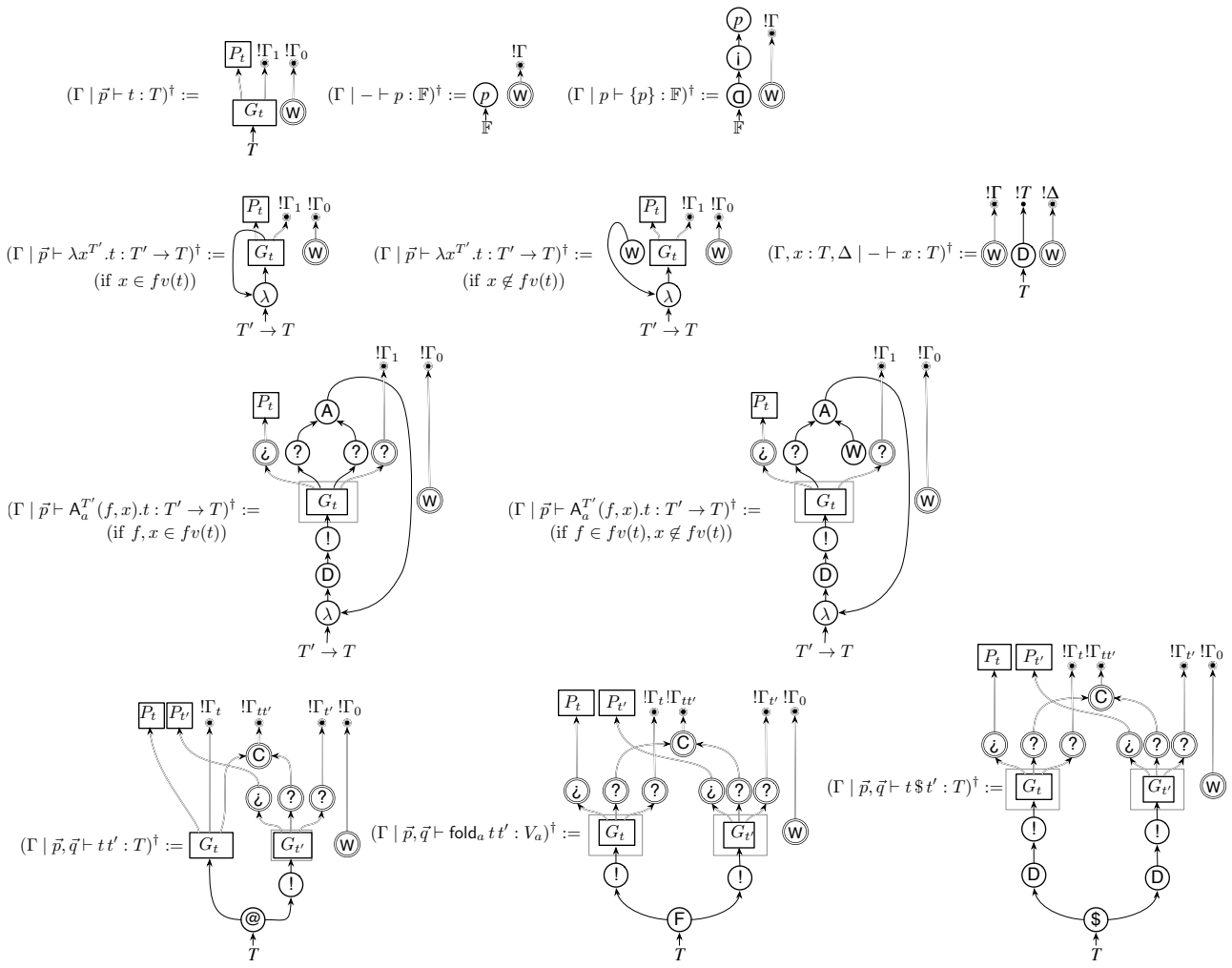


Figure 8. Inductive translation

4 Programming in ITF

Let us consider a more advanced example which will show how the treatment of cells and graph abstraction in ITF reduces syntactic overhead and supports our semantic intuitions. We create a linear model for a set of points in the plane corresponding to (x, y) measurements from some instrument. The model must represent the relationship between y and x not pointwise but as a confidence interval. Concretely, let us look at two (parameterised) such models: linear regression with confidence interval (CI) and weighted regression (WR) [4]. The first model is suitable when training data has measurement errors independent of the value of x, while the second model is suitable when errors vary linearly with x.[1]

Let $pair = \lambda x.\lambda y.\lambda z.z\ x\ y$ be the Church-encoding of pairs and let $f = \lambda a.\lambda b.\lambda x.a \times x + b$ be a generic linear function with unspecified parameters a and b. Let *opt_ci* and *opt_wr* be generic learning functions that can be applied to some model m and seed p, defined elsewhere, suitable for CI and WR, respectively, incorporating the reference data points, suitable loss functions, and optimisation algorithms.

An ITF program for the confidence-interval model is shown below, emphasising each step in the construction.

```
let a = {1}
let ci = pair (f a {1}) (f a {2})          (confidence interval)
let (pcim,p) = abs ci                      (parameterised CI model)
let pci = opt_ci pcim                      (learn CI parameters)
let cim = pcim pci                         (concrete CI model)
```

The model consists of a pair of linear functions which share the same slope (a) but may have different intercepts. The graph abstraction turns the computation graph *ci* into a conventional function *pcim* which will take three parameters. However, the number of parameters of the function is hidden into the vector type of the argument. The generic optimisation function *opt_ci* will compute the best values for the parameters (*pci*) which can be then used to create a concrete model *cim* which can be then used, as a regular function, in the subsequent program.

[1] These examples and more can be explored in the online visualiser: https://cwtsteven.github.io/GoI-TF-Visualiser/

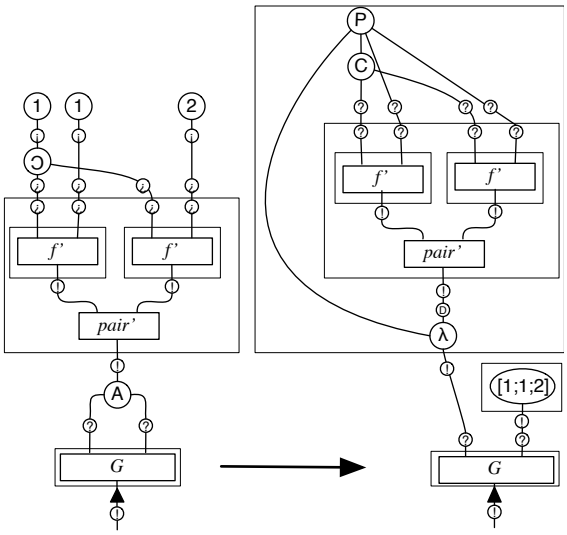


Figure 9. Graph-abstracting the CI model

In contrast, the weighted-regression model is a pair of independent linear functions. The structure of the program is otherwise similar.

$let\ wr = pair\ (f\ \{1\}\ \{0\})\ (f\ \{1\}\ \{0\})$ (weighted regression)
$let\ (pwrm, p) = \text{abs}\ wr$ (parameterised WR model)
$let\ pwr = opt\_wr\ pwrm$ (learn WR parameters)
$let\ wrm = pwrm\ pwr$ (concrete WR model)

These codes can be written more concisely, e.g.

$$let\ ci = (\lambda a.pair\ (f\ a\ \{1\})\ (f\ a\ \{2\}))\ \{1\}$$
$$let\ cim = (\mathsf{A}(pcim, p).pcim\ (opt\_ci\ pcim))\ ci$$
$$let\ wr = pair\ (f\ \{1\}\ \{0\})\ (f\ \{1\}\ \{0\})$$
$$let\ wrm = (\mathsf{A}(pwrm, p).pwrm\ (opt\_wr\ pwrm))\ wr$$

This relatively simple example illustrates several key features of ITF. First, there is no distinction between regular lambda terms and data-flow graphs. A higher-order computation graph is constructed automatically. Second, cells are treated as references rather than as constants, ensuring that the programmer has a grasp on how many parameters can be adjusted by the optimiser. For CI there are three parameters, the (shared) slope and two intercepts, whereas for WR there are four parameters, two slopes and two intercepts. Third, cells are collected into parameters of the graph-abstracted function not just from the term to which abs is applied, but from its free variables as well.

The key step in both examples is the graph abstraction. Figs. 9-10 show how the two models differ. The !-box G represents the programming context when graph abstraction is triggered. Pre-abstraction the computation graphs of CI share a cell, resulting post-abstraction in a function with a shared argument. In contrast, the WR computation graph and resultant function involve no sharing.

In the absence of graph abstraction, the obvious alternatives in a functional setting, such as explicitly parameterising models with vectors involves error-prone index manipulation to control sharing ($[k_0; \ldots; k_m]$ is a vector and $p[i]$ is element access), for example:

$$let\ f\ p\ x\ = p[0] \times x + p[1]$$

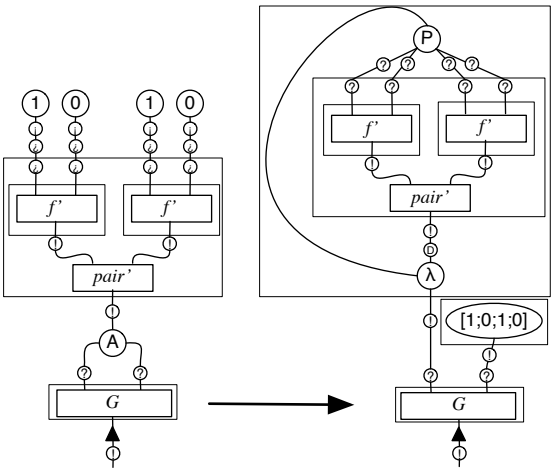


Figure 10. Graph-abstracting the WR model

$$let\ ci\ p = pair\ (f\ [p[0]; p[1]])\ (f\ [p[0]; p[2]])$$
$$let\ cim = ci\ (opt\_ci\ ci)$$
$$let\ wr\ p = pair\ (f\ [p[0]; p[1]])\ (f\ [p[2]; p[3]])$$
$$let\ wrm = wr\ (opt\_wr\ wr)$$

The alternatives are comparably awkward.

5 Contextual equivalence

Usually programs (closed ground-type terms) are equated if and only if they produce the same values. However in the presence of cells, this is not enough. For example, evaluating programs $\{1\} + 2$, $1 + 2$ and $1 + \{2\}$ yields the same token value (3) but different final graphs, which can be made observable by graph abstraction.

Definition 5.1 (Token-value equivalence)**.** Two composite graphs $G_1(0,1)$ and $G_2(0,1)$ are *token-value equivalent*, written as $G_1 \doteq G_2$, if there exists a token value κ such that the following are equivalent: $Init(G_1) \to^* Final(G_1', \kappa)$ for some composite graph G_1', and $Init(G_2) \to^* Final(G_2', \kappa)$ for some composite graph G_2'.

We lift token-value equivalence to a congruence by definition, just like the usual program equivalence is lifted to open terms.

Definition 5.2 (Graph-contextual equivalence)**.** Two graphs $G_1(n,m)$ and $G_2(n,m)$ are *graph-contextually equivalent*, written as $G_1 \cong G_2$, if for any graph context $\mathcal{G}[\Box]$ that makes two composite graphs $\mathcal{G}[G_1]$ and $\mathcal{G}[G_2]$ of ground type, the token-value equivalence $\mathcal{G}[G_1] \doteq \mathcal{G}[G_2]$ holds.

The graph-contextual equivalence $\cong$ is indeed an equivalence relation, and also a congruence with respect to graph contexts. We say a binary relation R on graphs *implies* graph-contextual equivalence, if $R \subseteq \cong$.

In the DGoI machine, the token always moves along a node, and a redex can always be determined as a sub-graph relative to the token position. This locality of the machine behaviour enables us to give some instances of the graph-contextual equivalence by means of the following variant of simulation, 'U-simulation'. Let $(\cdot)^+$ stand for the transitive closure of a binary relation.

Definition 5.3 (U-simulation)**.** A binary relation R on graph 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 $\sigma_1' \to^* \sigma_2$ of (possibly no) transitions. (II) If $\sigma_1\ R\ \sigma_2$ and no transition is possible from the graph state σ_1, then there exist composite graphs G_1 and G_2 and a token value κ such that $\sigma_1 = \mathit{Final}(G_1, \kappa)$ and $\sigma_2 = \mathit{Final}(G_2, \kappa)$.

Intuitively, a U-simulation is the ordinary simulation between two transition systems (the condition (I-i) in the above definition), 'Until' the left sequence of transitions is reduced to the right sequence (the condition (I-ii)). The reduction may not happen, which resembles the weak until operator of linear temporal logic. The condition (I-i) involves the transitive closure R^+, in case the reduction steps are multiplied.

Proposition 5.4. *Let R be a U-simulation. If $\sigma_1\ R\ \sigma_2$, then there exists a token value κ such that the following are equivalent: $\sigma_1 \to^* \mathit{Final}(G_1, \kappa)$ for some composite graph G_1, and $\sigma_2 \to^* \mathit{Final}(G_2, \kappa)$ for some composite graph G_2.* □

We will use U-simulations to see if some rewrites on graphs, which may or may not be triggered by the token, imply the graph-contextual equivalence.

Proposition 5.5. *Let $\prec$ be a binary relation on graphs with the same interface, and its lifting $\overline{\prec}$ on graph states defined as follows: $((\mathcal{G}[G_1], e), \delta)\ \overline{\prec}\ ((\mathcal{G}[G_2], e), \delta)$ iff $G_1 \prec G_2$ and the position e 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 $\cong$.*

Proof. We assume $G_1 \prec G_2$, and take an arbitrary graph context $\mathcal{G}[\Box]$ that makes two composite graphs $\mathcal{G}[G_1]$ and $\mathcal{G}[G_2]$. The lifting $\overline{\prec}$ relates initial states on these composite graphs, i.e. $\mathcal{G}[G_1]\ \overline{\prec}\ \mathcal{G}[G_2]$. Therefore, if it is a U-simulation, these two graphs are token-value equivalent $\mathcal{G}[G_1] \doteq \mathcal{G}[G_2]$, by Prop. 5.4. We can conclude the graph-contextual equivalence $G_1 \cong G_2$. □

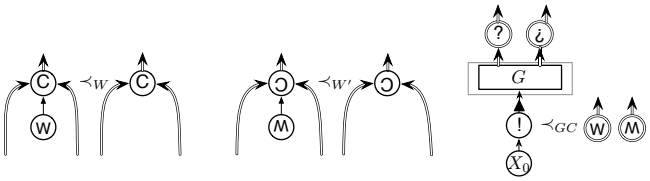
Finally, the notion of contextual equivalence of terms can be defined as a restriction of the graph-contextual equivalence, to graph-contexts that arise as translations of (syntactical) contexts.

Definition 5.6 (Contextual equivalence)**.** Two terms $A \mid \Gamma \mid \vec{p} \vdash t_i : T'$ $(i = 1, 2)$ in the same derivable type judgement are *contextually equivalent*, written as $A \mid \Gamma \mid \vec{p} \vdash t_1 \approx t_2 : T'$, if for any context $C\langle\cdot\rangle^T$ such that the two type judgements $A \mid \Gamma \mid \vec{q} \vdash C\langle t_i\rangle : T$ $(i = 1, 2)$ are derivable for some vector $\vec{q}$ and some ground type T, the token-value equivalence $(A \mid \Gamma \mid \vec{q} \vdash C\langle t_1\rangle : T)^\dagger \doteq (A \mid \Gamma \mid \vec{q} \vdash C\langle t_2\rangle : T)^\dagger$ holds.

5.1 Garbage collection

Large programs generate sub-graphs which are unreachable and unobservable during execution (*garbage*). In the presence of graph abstraction the precise definition is subtle, and the rules for garbage collection are not obvious. We show safety of some forms of garbage collection, as below.

Proposition 5.7 (Garbage collection)**.** *Let $\prec_W$, $\prec_{W'}$ and $\prec_{GC}$ be binary relations on graphs, defined by*



where the X-node is either a W-node, or a P-node with no input. They altogether imply the graph-contextual equivalence, i.e. $\prec_W \cup \prec_{W'} \cup \prec_{GC}$ implies the graph-contextual equivalence.

Sketch of proof. The relation $\prec_W \cup \prec_{W'} \cup \prec_{GC}$ lifts to a U-simulation, where the condition (I-ii) in Def. 5.3 is not relevant. We then use Prop. 5.5. □

5.2 Beta equivalence

We can prove a form of beta equivalence, where the function argument is a closed value without cells. The substitution $t[u/x]$ is defined as usual. The proof is by making U-simulations out of special cases of λ-rewrites and !-rewrites, and is also by the garbage collection shown above.

Proposition 5.8 (Beta equivalence)**.** *Let v be a value defined by the grammar $v ::= p \mid \lambda x^T.t \mid \mathsf{A}^T_a(f, x).t$. If the type judgement $A' \mid - \mid - \vdash v : T'$ is derivable, the contextual equivalence $A \mid \Gamma \mid \vec{p} \vdash (\lambda x^{T'}.t)\, v \approx t[v/x] : T$ holds.* □

6 Conclusion and related work

Machine learning can take advantage of a novel programming idiom, which allows functions to be parameterised in such a way that a general purpose optimiser can adjust the values of parameters embedded inside the code. The nature of the programming language design challenge is an ergonomic one, making the bureaucracy of parameter management as simple as possible while preserving soundness and equational properties. In this paper we do not aim to assess whether the solution proposed by TF reflects the best design decisions, but we merely note that automating parameter management requires certain semantic enhancements which are surprisingly complex.

The new feature is the extraction of the variable-dependencies of a computation graph (the parameters) into a single vector, which can be then processed using generic functions. Moreover, we place this feature in an otherwise pure, and quite simple, programming language in order to study it semantically (ITF). Our contribution is to provide evidence that this rather exotic feature is a reasonable addition to a programming language: typing guarantees safety of execution (soundness), garbage collection is safe, and a version of the beta law holds. Moreover, the operational semantics does not involve inefficient (worse than linear) operations, indicating a good potential for implementability. Reaching a language comparable in sophistication and efficiency with TF is a long path, but we are making the first steps in that direction [3]. The advantages of using a stand-alone language, especially when there is evidence that it has a reasonably well behaved semantics, are significant, as EDSLs suffer from well known pitfalls [17].

Other than TF, we only know of one other language which supports the ability to abstract on state ('*wormholing*'), with a similar motivation but with a different application domain, data science [16]. For keeping the soundness argument concise the language lacks recursion, but sound extensions of GoI-style machines

with this feature have been studied in several contexts and we do not think it presents insurmountable difficulties [5, 15]. Further extensions of the language, in particular effects, pose serious challenges however.

We chose to give a semantics to ITF using the Dynamic Geometry of Interaction (DGoI) [13, 14], a novel graph-rewriting semantics initially used to give cost-accurate models for various reduction strategies of the lambda calculus. The graph model of DGoI is already, in a broad sense, a data-flow graph with higher-order features, which is a natural fit for the language we aim to model. The semantics of call-by-value lambda calculus is based on the one in [14], where it is shown to be efficient, in a formal sense. In this paper we do not formally analyse the cost model of ITF but we can see, at least informally, that the operations involved in handling language extensions such as cells, computation graphs, and graph abstraction are not computationally onerous. Some of the more expensive operations, such as box-reachability, could be implemented in constant time using 'jump links' between the endpoints of a path, thus trading off space and time costs. The idea of jumping can be found in the GoI literature [6, 8].

Pragmatically speaking, even though the infrastructure required to support computation graphs and graph abstraction involves a non-negligible overhead, the impact of this overhead on the running cost of a typical machine-learning program as a whole is negligible. This is because the running cost of machine-learning programs is dominated by the learning phase, realised by the optimisers. This phase involves only 'conventional' functions, the result of graph abstraction, in which all the overhead can be simply discarded as superfluous. This overhead is only required in the model creation phase, which is not computationally intensive.

This paper represents a first step in the study of ITF, focussing on what we believe to be the most challenging semantic feature of the language. In the future we plan to study the execution mode of the graphs, by propagating automatically changes to the cells through the graph, much like in incremental or self-adjusting computation, and the way such features interact with graph abstraction. Finally, in the longer term, to develop a usable functional counterpart of TF we also aim to incorporate a safe version of automatic differentiation, as well as probabilistic execution.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Umut A. Acar, Matthias Blume, and Jacob Donham. 2013. A consistent semantics of self-adjusting computation. *J. Funct. Program.* 23, 3 (2013), 249–292.

[3] Steven Cheung, Victor Darvariu, Dan R. Ghica, Koko Muroya, and Reuben N. S. Rowe. 2018. A functional perspective on machine learning via programmable induction and abduction. In *FLOPS 2018.* (forthcoming).

[4] William S Cleveland. 1979. Robust locally weighted regression and smoothing scatterplots. *Journal of the American statistical association* 74, 368 (1979), 829–836.

[5] Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. 2015. Parallelism and synchronization in an infinitary context. In *LICS 2015.* IEEE, 559–572.

[6] Vincent Danos and Laurent Regnier. 1996. Reversible, irreversible and optimal lambda-machines. *Elect. Notes in Theor. Comp. Sci.* 3 (1996), 40–60.

[7] Jack B Dennis. 1974. First version of a data flow procedure language. In *Programming Symposium.* Springer, 362–376.

[8] Maribel Fernández and Ian Mackie. 2002. Call-by-value lambda-graph rewriting without rewriting. In *ICGT 2002 (LNCS)*, Vol. 2505. Springer, 75–89.

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

[10] Jean-Yves Girard. 1989. Geometry of Interaction I: interpretation of system F. In *Logic Colloquium 1988 (Studies in Logic & Found. Math.)*, Vol. 127. Elsevier, 221–260.

[11] Andreas Griewank et al. 1989. On automatic differentiation. *Mathematical Programming: recent developments and applications* 6, 6 (1989), 83–107.

[12] Aleks Kissinger. 2012. Pictures of processes: automated graph rewriting for monoidal categories and applications to quantum computing. *arXiv preprint arXiv:1203.0202* (2012).

[13] Koko Muroya and Dan R. Ghica. 2017. The dynamic Geometry of Interaction machine: a call-by-need graph rewriter. In *CSL 2017 (LIPIcs)*, Vol. 82.

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

[15] Koko Muroya, Naohiko Hoshino, and Ichiro Hasuo. 2016. Memoryful Geometry of Interaction II: recursion and adequacy. In *POPL 2016.* ACM, 748–760.

[16] Tomas Petricek. Retrieved 2017. Design and implementation of a live coding environment for data science. (Retrieved 2017). http://tomasp.net/academic/drafts/live/live.pdf.

[17] Josef Svenningsson and Emil Axelsson. 2015. Combining deep and shallow embedding of domain-specific languages. *Computer Languages, Systems & Structures* 44 (2015), 143–165. https://doi.org/10.1016/j.cl.2015.07.003

[18] Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. *ACM sigplan notices* 35, 5 (2000), 242–252.