# 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),* 24 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

## 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 [8] and automatic differentiation [12]. 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 [19], although our 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.

```
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

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 $\mathsf{fold}_a$, which are always over the bases of the vector space indexed by $a$:

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

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, $\Gamma$ 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 $\Gamma$. 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 [11], 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 [14, 15]. 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.

### 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 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' [13]

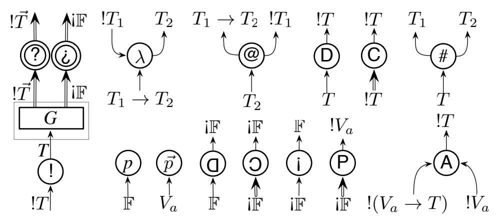


**Figure 2.** Connection of edges

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.

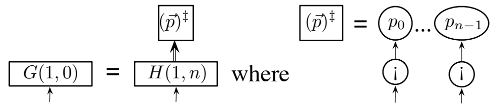
Nodes are labelled, and we call a node labelled with $X$ an '$X$-node'. We have several sorts of labels. Some represent the basic syntactic constructs of the lambda calculus: $\lambda$ (abstraction), @ (application), $p \in \mathbb{F}$ (scalar constants), $\vec{p} \in \mathbb{F}^n$ (vector constants), 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 [10], handling sharing and copying for argument types. Adaptations of these nodes, namely ¡, ¿, ᗡ and Ɔ, are for sharing (but not copying) of cells. Note that we use generalised contractions (C, Ɔ) of any input arity, which includes weakening. We sometimes write W (resp. ʍ) to emphasise a contraction C (resp. Ɔ) 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 $\lambda$, 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, $k = k_1 \$ k_2$, $X \in \{¡, ¿, ᗡ, \mathsf{D}\}$, $Y \in \{¡, ¿, ᗡ\}$ and $Z \in \{\mathsf{C}, Ɔ\}$.

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 ($\lambda$) either returns the token with a value $\lambda$ 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 $\lambda$-@ 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 $\vec{p} \in \mathbb{F}^n$. When passing a $Z$-node (i.e. C or Ɔ) 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

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

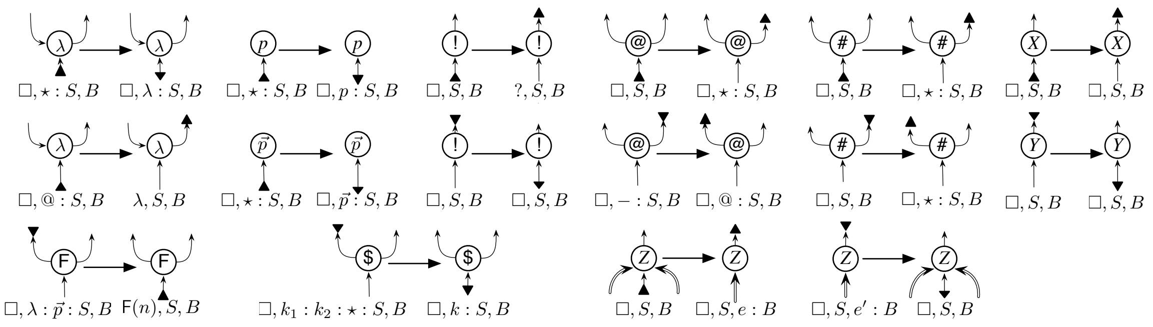


**Figure 3.** Pass transitions

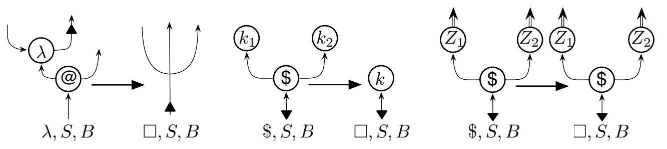


**Figure 4.** Rewrite transitions: computation

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 $\lambda$-@ pair, a key step in beta reduction. Following the rewrite, the incoming output link of $\lambda$ 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.

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

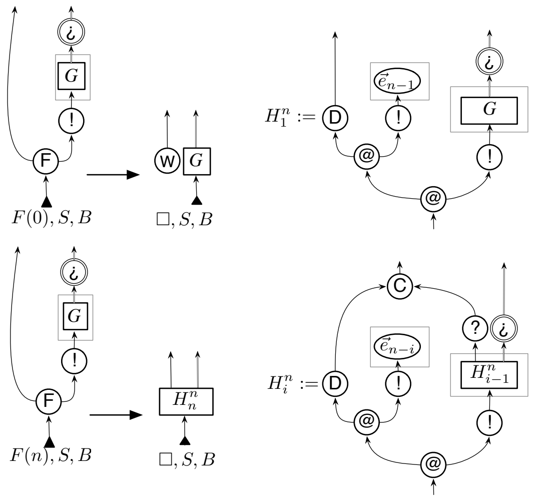


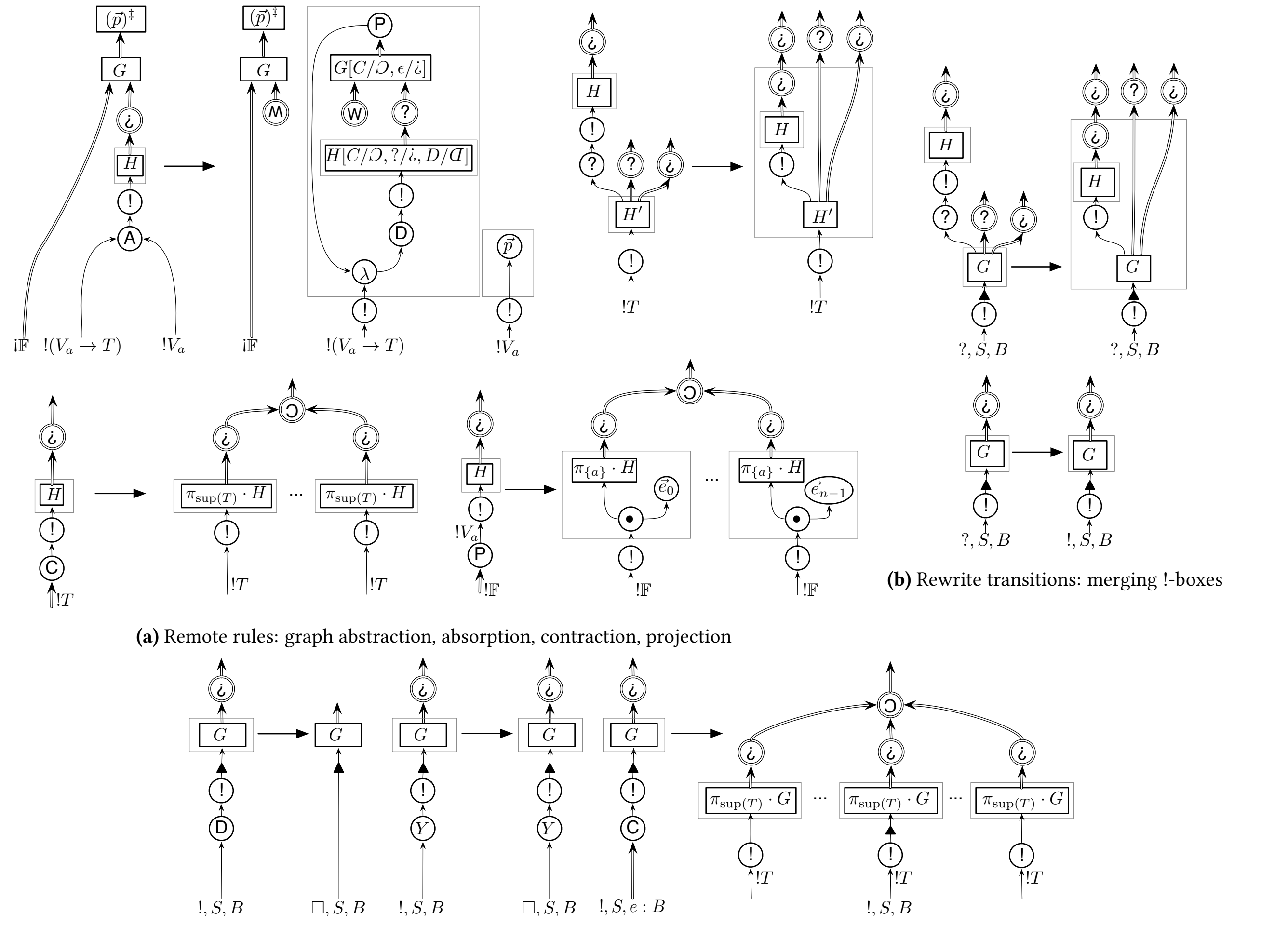
**Figure 5.** Rewrite transition: unfolding over the bases

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 $0 \leq j < i$, 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 $v'$.

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.



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

**(b)** Rewrite transitions: merging !-boxes

**(c)** Rewrite transitions: copying closed !-boxes

**Figure 7.** Rewrite transitions: !-boxes

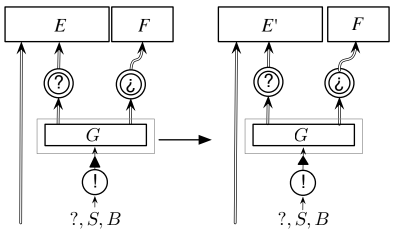


**Figure 6.** Remote rules triggering

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 $\lambda$-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 $\pi_N$, 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. This is discussed in more depth in Appendix B, noting 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 ($Y \notin \{\mathrm{D},\mathrm{C}\}$) change rewrite mode to pass mode, by setting the rewrite flag to $\square$. 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 ¡-nodes. Transitions do not change redex interfaces. $\square$

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 $\kappa$, 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 $\sigma$, the final state $Final(G,\kappa)$ such that $\sigma \to^* Final(G,\kappa)$ is unique up to name permutation, if it exists.*

*Proof.* See Appendix A. $\square$

**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 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 $\Delta$ 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 $\kappa$ such that: $Init((A \mid - \mid \vec{p} \vdash t : T)^{\ddagger}) \to^* Final(G,\kappa)$.*

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 $\kappa$ carried by the token as it 'exits' the graph $G$. The value $\kappa$ will always be either a scalar, or a vector, or the symbol $\lambda$ indicating a function-value result.

The proof is given in Appendix G. It 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.

# 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

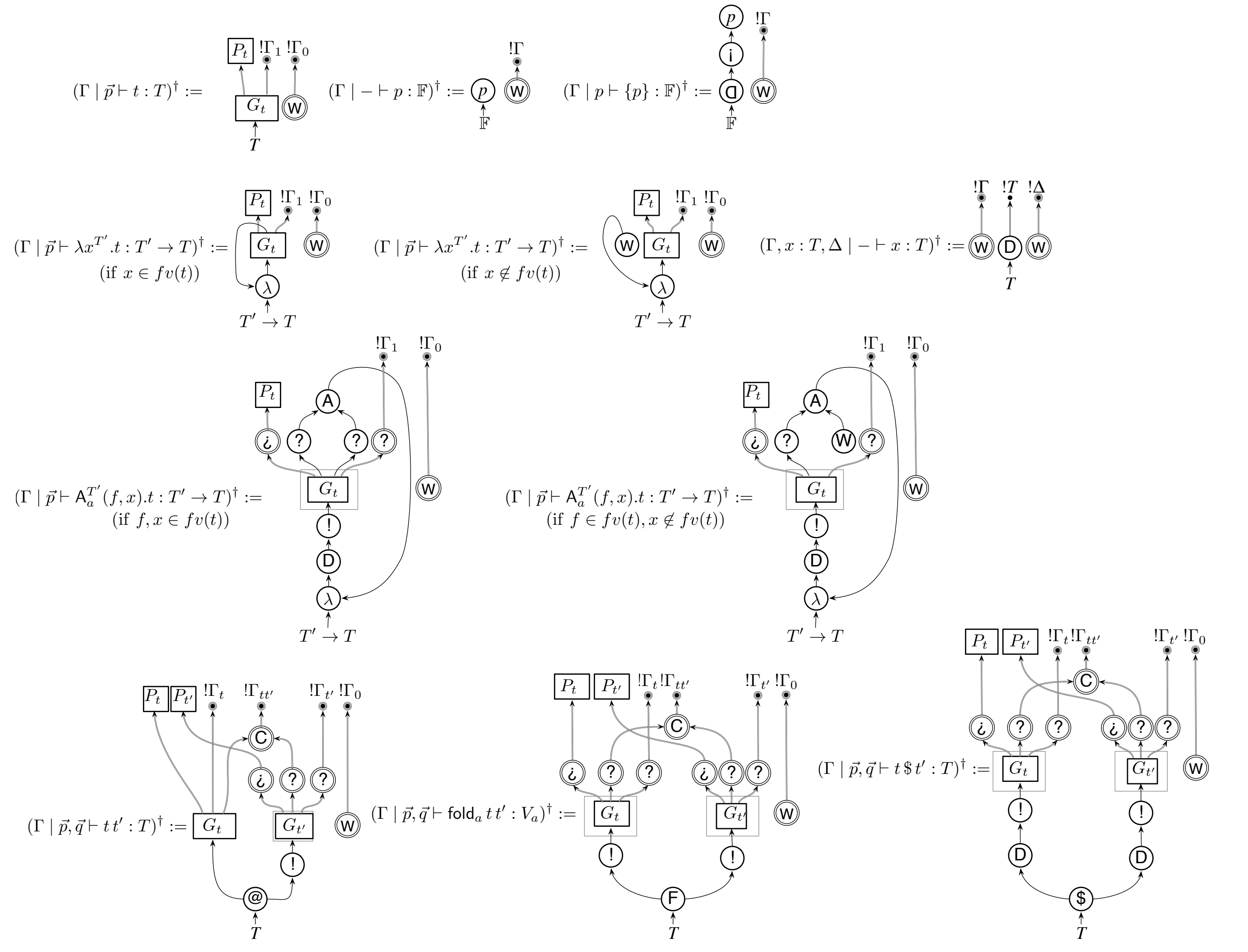

$(\Gamma \mid \vec{p} \vdash t : T)^\dagger :=$
$(\Gamma \mid - \vdash p : \mathbb{F})^\dagger :=$
$(\Gamma \mid p \vdash \{p\} : \mathbb{F})^\dagger :=$
$(\Gamma \mid \vec{p} \vdash \lambda x^{T'}.t : T' \to T)^\dagger :=$
(if $x \in fv(t)$)
$(\Gamma \mid \vec{p} \vdash \lambda x^{T'}.t : T' \to T)^\dagger :=$
(if $x \notin fv(t)$)
$(\Gamma, x : T, \Delta \mid - \vdash x : T)^\dagger :=$
$(\Gamma \mid \vec{p} \vdash \mathsf{A}_a^{T'}(f,x).t : T' \to T)^\dagger :=$
(if $f, x \in fv(t)$)
$(\Gamma \mid \vec{p} \vdash \mathsf{A}_a^{T'}(f,x).t : T' \to T)^\dagger :=$
(if $f \in fv(t), x \notin fv(t)$)
$(\Gamma \mid \vec{p}, \vec{q} \vdash t\,t' : T)^\dagger :=$
$(\Gamma \mid \vec{p}, \vec{q} \vdash \mathsf{fold}_a\,t\,t' : V_a)^\dagger :=$
$(\Gamma \mid \vec{p}, \vec{q} \vdash t \,\$\, t' : T)^\dagger :=$

**Figure 8.** Inductive translation

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* $(pcim, p) = \mathsf{abs}\ ci$ | (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.

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* $(pwrm, p) = \mathsf{abs}\ wr$ | (parameterised WR model) |
| *let* $pwr = opt\_wr\ pwrm$ | (learn WR parameters) |
| *let* $wrm = pwrm\ pwr$ | (concrete WR model) |

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

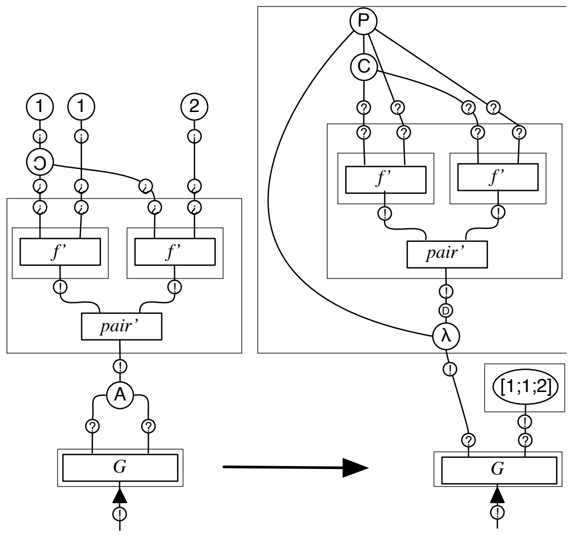


**Figure 9.** Graph-abstracting the CI model

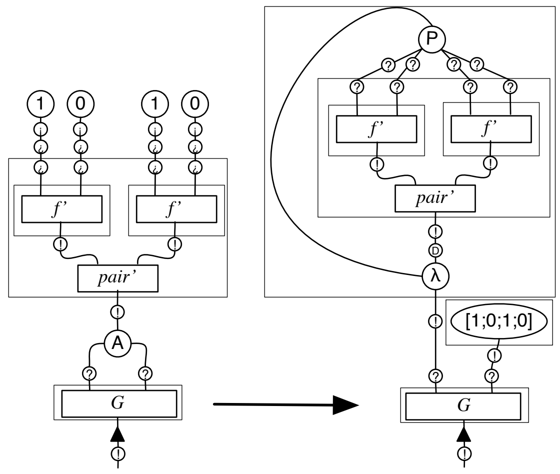


**Figure 10.** Graph-abstracting the WR model

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 ($[k_0; \dots ; k_m]$ is a vector and $p[i]$ is element access), for example:

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

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 $\kappa$ 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 $\mathcal{G}[\square]$ 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 $\sigma_2'$ 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 $\sigma_1$, then there exist composite graphs $G_1$ and $G_2$ and a token value $\kappa$ such that $\sigma_1 = Final(G_1, \kappa)$ and $\sigma_2 = Final(G_2, \kappa)$.

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 $\kappa$ such that the following are equivalent: $\sigma_1 \to^* Final(G_1, \kappa)$ for some composite graph $G_1$, and $\sigma_2 \to^* Final(G_2, \kappa)$ for some composite graph $G_2$.*

*Proof.* See Appendix H. □

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 $\mathcal{G}[\square]$. 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 $\mathcal{G}[\square]$ 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 $G_1 \cong G_2$. □

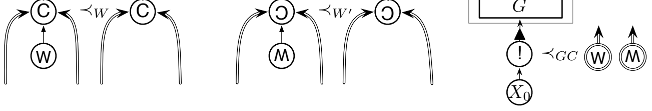
Finally, the notion of contextual equivalence of terms can be defined as a restriction of the graph-contextual equivalence, to graph-contexts that arise as translations of (syntactical) contexts.

**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 $\lambda$-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 $v ::= p \mid \lambda x^T.t \mid \mathsf{A}_a^T(f, x).t$. 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.*

*Sketch of proof.* See Appendix H.1. □

## 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 [18].

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 [17]. 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, 16]. 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) [14, 15], 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 [15], 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 end-points of a path, thus trading off space and time costs. The idea of jumping can be found in the GoI literature [7, 9].

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. 1989. The structure of multiplicatives. *Arch. Math. Log.* 28, 3 (1989), 181–203.

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

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

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

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

[11] 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.

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

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

[14] 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.

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

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

[17] 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.

[18] 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

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

## A Determinism

The only sources of non-determinism are the choice of fresh names in replicating a !-box and the choice of ?-rewrite transitions (Fig. 7a and Fig. 7b). Introduction of fresh names has no impact on execution, as we can prove 'alpha-equivalence' of graph states.

**Proposition A.1** ('Alpha-equivalence' of graph states)**.** *The binary relation $\sim_\alpha$ of two graph states, defined by $((G,e),\delta) \sim_\alpha ((\pi\cdot G,e),\delta)$ for any name permutation $\pi$, is an equivalence relation and a bisimulation.*

*Proof.* Only rewrite transitions that replicate a !-box (in Fig. 7a and Fig. 7c) involve name permutation. Names are irrelevant in all the other transitions. □

We identify graph states modulo name permutation, namely the binary relation $\sim_\alpha$ in the above proposition. Now non-determinism boils down to the choice of ?-rewrites, which however does not yield non-deterministic overall executions.

**Proposition A.2** (Determinism)**.** *If there exists a sequence*

$$((G,e),\delta) \to^* ((G',e'),(d',\Box,S',B')),$$

*any sequence of transitions from the state $((G,e),\delta)$ reaches the state $((G',e'),(d',\Box,S',B'))$, up to name permutation.*

*Proof.* The applicability condition of ?-rewrite rules ensures that possible ?-rewrites at a state do not share any redexes. Therefore ?-rewrites are confluent, satisfying the so-called diamond property: if two different ?-rewrites $((G,e),\delta) \to ((G_1,e_1),\delta_1)$ and $((G,e),\delta) \to ((G_2,e_2),\delta_2)$ and are possible from a single state, both of the data $\delta_1$ and $\delta_2$ has rewrite flag ?, and there exists a state $((G',e'),\delta')$ such that $((G_1,e_1),\delta_1) \to ((G',e'),\delta')$ and $((G_2,e_2),\delta_2) \to ((G',e'),\delta')$. □

**Corollary A.3** (Prop. 3.5)**.** *For any graph state $\sigma$, the final state $Final(G,\kappa)$ such that $\sigma \to^* Final(G,\kappa)$ is unique up to name permutation, if it exists.*

**Corollary A.4** (Cor. 3.6)**.** *For any initial state $Init(G)$, the final state $Final(G,\kappa)$ such that $Init(G) \to^* Final(G,\kappa)$ is unique up to name permutation, if it exists.*

## B Validity

This section investigates a property of graph states, *validity*, which plays a key role in disproving any failure of transitions. It is based on three criteria on graphs.

In the lambda calculus one often assumes that bound variables in a term are distinct, using the alpha-equivalence, so that beta-reduction does not cause unintended variable capturing. We start with an analogous criterion on names.

**Definition B.1** (Bound/free names)**.** A name $a \in \mathbb{A}$ in a graph is said to be:

1. *bound* by an A-node, if the A-node has input types $V_a \to T$) and $!V_a$, for some type $T$.
2. *free*, if a $\vec{p}$-node has input type $V_a$ or a P-node has output type $V_a$.

**Definition B.2** (Bound-name criterion)**.** A graph $G$ meets the *bound-name criterion* if any bound name $a \in \mathbb{A}$ in the graph $G$ satisfies the following.

**Uniqueness.** The name $a$ is not free, and is bound by exactly one A-node.

**Scope.** Bound names do not appear in types of input links of the graph $G$. Moreover, if the A-node that binds the name $a$ is in a !-box, the name $a$ appears only strictly inside the !-box (i.e. in the !-box, but not on its interfaces).

The name permutation action accompanying rewrite transitions (Fig. 7a and Fig. 7c) is an explicit way to preserve the above requirement in transitions.

**Proposition B.3** (Preservation of bound-name criterion)**.** *In any transition, if an old state meets the bound-name criterion, so does a new state.*

*Proof.* In a ?-rewrite transition that eliminates a A-node, the name $a \in \mathbb{A}$ bound by the A-node turns free. As the name $a$ is not bound by any other A-nodes, it does not stay bound after the transition. The transition does not change the status of any other names, and therefore preserves the uniqueness and scope of bound variables.

Duplication of a !-box, in a rewrite transition involving a C-node or a P-node applies name permutation. The scope of bound names is preserved by the transition, because if an A-node is duplicated, all links in which the name bound by the A-node appears are duplicated together. The scope also ensures that, if an A-node is copied, the name permutation makes each copy of the node bind distinct names. Therefore the uniqueness of bound names is not broken by the transition.

Any other transitions do not change the status of names. □

The second criterion is on free names, which ensures each free name indicates a unique vector space $\mathbb{F}^n$.

**Definition B.4** (Free-name criterion)**.** A graph $G$ meets the *free-name criterion* if it comes with a 'validation' map $v\colon \mathrm{FR}_G \to \mathbb{N}$, from the set $\mathrm{FR}_G$ of free names in the graph $G$ to the set $\mathbb{N}$ of natural numbers, that satisfies the following.

- If a $\vec{p}$-node has input type $V_a$, the vector $\vec{p}$ has the size $v(a)$, i.e. $\vec{p} \in \mathbb{F}^{v(a)}$
- If a P-node has output type $!V_a$, it has $v(a)$ input links, i.e. $n = v(a)$.

The validation map is unique by definition. We refer to the combination of the bound-name criterion and the free-name criterion as 'name criteria'.

**Proposition B.5** (Preservation of name criteria)**.** *In any transition, if an old state meets both the bound-name criterion and the free-name criterion, so does a new state.*

*Proof.* With Prop. B.3 at hand, we here show that the new state fulfills the free-name criterion.

A free name is introduced by a ?-rewrite transition that eliminates a A-node. The name was bound by the A-node and not free before the transition, because of the bound-name criterion (namely the uniqueness property). Therefore the validation map can be safely extended.

The name permutation, in rewrite transitions that duplicate a !-box, applies for both bound names and free names. It introduces fresh free names, without changing the status of names, and therefore the validation map can be extended accordingly.

Some computational rewrite rules (Fig. 4) act on links with vector type $V_a$, however they have no impact on the validation map. Any other transitions also do not affect the validation map. □

The last criterion is on the shape of graphs. It is inspired by Danos and Regnier's correctness criterion [6] for proof nets.

**Definition B.6** (Covering links)**.** In a graph $G(1,n)$, a link $e$ is *covered* by another link $e'$, if any box-path (see Def. 3.2) from the root of the graph $G$ to the link $e$ contains the covering link $e'$.

**Definition B.7** (Graph criterion)**.** A graph $G(1,n)$ fulfills the *graph criterion* if it satisfies the following.

**Acyclicity** Any box-path, in which all links have (not necessarily the same) argument types, is acyclic, i.e. nodes or links appear in the box-path at most once. Similarly, any directed path whose all links have the cell type $¡\mathbb{F}$ is acyclic.

**Covering** At any $\lambda$-node, its incoming output link is covered by its input link. Any input link of A-node or P-node is covered by a ?-node.

**Proposition B.8** (Preservation of graph criterion)**.** *In any transition, if an old state meets the graph criterion, so does a new state.*

*Proof.* An @-rewrite transition eliminates a pair of a $\lambda$-node and an @-node, and connects two acyclic box-paths of argument types. The resulting box-path being a cycle means that there existed a box-path from the free (i.e. not connected to the $\lambda$-node) output link of the @-node to the incoming output link of the $\lambda$-node before the transition. This cannot be the case, as the incoming output link must have been covered by the input link of the $\lambda$-node. Therefore the @-rewrite does not break the acyclicity condition. The condition can be easily checked in any other transitions.

The covering condition is also preserved. Only notable case for this condition is the graph abstraction rule that introduces a $\lambda$-node and a P-node. □

Finally the validity of graph states is defined as below. The validation map of a graph is used to check if the token carries appropriate data to make computation happen.

**Definition B.9** (Queries and answers)**.** Let $d\colon A \to \mathbb{N}$ be a map from a finite set $A \subset_{\mathrm{fin}} \mathbb{A}$ of names to the set $\mathbb{N}$ of natural numbers. For each type $\tilde{T}$, two sets $\mathrm{Qry}_{\tilde{T}}$ and $\mathrm{Ans}^d_{\tilde{T}}$ are defined inductively as below.

$$\mathrm{Qry}_{\mathbb{F}} = \mathrm{Qry}_{¡\mathbb{F}} := \{\star\}, \qquad \mathrm{Ans}^d_{\mathbb{F}} = \mathrm{Ans}^d_{¡\mathbb{F}} := \mathbb{F}$$

$$\mathrm{Qry}_{V_a} := \{\star\}, \qquad \mathrm{Ans}^d_{V_a} := \begin{cases} \mathbb{F}^{d(a)} & (\text{if } a \in A) \\ \emptyset & (\text{otherwise}) \end{cases}$$

$$\mathrm{Qry}_{T_1 \to T_2} := \{\star, @\}, \qquad \mathrm{Ans}^d_{T_1 \to T_2} := \{\lambda\}$$

$$\mathrm{Qry}_{!T} := \mathrm{Qry}_T, \qquad \mathrm{Ans}^d_{!T} := \mathrm{Ans}^d_T.$$

**Definition B.10** (Valid states)**.** A state $((G,e),(d,f,S,B))$ is *valid* if the following holds.

1. The graph $G$ fulfills the name criteria and the graph criterion.
2. If $d = \uparrow$ and the position $e$ has type $\rho$, the computation stack $S$ is in the form of $X : S'$ such that $X \in \mathrm{Qry}_\rho$.
3. Let $v$ be the validation map of the graph $G$. If $d = \downarrow$ and the position $e$ has type $\rho$, the set $\mathrm{Ans}^v_\rho$ is not empty, and the computation stack $S$ is in the form of $X : S'$ such that $X \in \mathrm{Ans}^v_\rho$.

**Proposition B.11** (Preservation of validity)**.** *In any transition, if an old state is valid, so is a new state.*

*Proof.* Using Prop. B.5 and Prop. B.8, the proof boils down to check the bottom two conditions of validity. Note that no rewrite transitions change the direction and the computation stack. When the token passes a \$-node downwards, application of the primitive operation \$ preserves the last condition of validity. All the other pass transitions are an easy case. □

In an execution, validity of intermediate states can be reduced to the criteria on its initial graph.

**Proposition B.12** (Validity condition of executions)**.** *For any execution $Init(G_0) \to^* ((G,e),\delta)$, if the initial graph $G_0$ meets the name criteria and the graph criterion, the state $((G,e),\delta)$ is valid.*

*Proof.* The initial state $Init(G_0)$ has the direction $\uparrow$, and its computation stack has the top element $\star$. Since any type $\rho$ satisfies $\star \in \mathrm{Qry}_\rho$, the criteria implies validity at the initial state $Init(G_0)$. Therefore the property is a consequence of Prop. B.11. □

## C Stability

This section studies executions in which the underlying graph is never changed.

**Definition C.1** (Stable executions/states)**.** An execution $Init(G) \to^* ((G,e),\delta)$ is *stable* if the graph $G$ is never changed in the execution. A state is *stable* is there exists a stable execution to the state itself.

A stable execution can include pass transitions, and rewrite transitions that just lower the rewrite flag, as well. Since the only source of non-determinism is rewrite transitions that actually change a graph, a stable state comes with a unique stable execution to the state itself.

The stability property enables us to backtrack an execution in certain ways, as stated below.

**Proposition C.2** (Factorisation of stable executions)**.**

1. *If an execution $Init(G) \to^* ((G,e),\delta)$ is stable, it can be factorised as $Init(G) \to^* ((G,e'),\delta') \to^* ((G,e),\delta)$ where the link $e'$ is any link covering the link $e$.*
2. *If an execution $Init(G) \to^* ((G,e),(\downarrow,\square,X : S,B))$ is stable, it can be factorised as $Init(G) \to^* ((G,e),(\uparrow,\square,\star : S,B)) \to^* ((G,e),(\downarrow,\square,X : S,B))$.*

*Proof of Prop. C.2.1.* The proof is by induction on the length of the stable execution $Init(G) \to^* ((G,e),\delta)$. When the execution has null length, the last position $e$ is the root of the graph $G$, and the only link that can cover it is the root itself.

When the execution has a positive length, we examine each possible transition. Rewrite transitions that only lower the rewrite flag are trivial cases. Cases for pass transitions are the straightforward use of induction hypothesis, because for any link and a node, the following are equivalent: (i) the link covers one of outgoing output links of the node, and (ii) the link covers all input links of the node. □

*Proof of Prop. C.2.2.* The proof is by induction on the length $n$ of the stable execution $Init(G) \to^n ((G,e),(\downarrow,\square,X : S,B))$.

As the first state and the last state cannot be equal, base cases are for single transitions, i.e. when $n = 1$. Only possibilities are pass transitions over a $\lambda$-node, a $p$-node or a $\vec{p}$-node, all of which is in the form of $((G,e),(\uparrow,\square,\star : S,B)) \to ((G,e),(\downarrow,\square,X : S,B))$.

In inductive cases, we will use induction hypothesis for any length that is less than $n$. If the last transition is a pass transition over a $\lambda$-node, a $p$-node or a $\vec{p}$-node, the discussion goes in the same way as in base cases. All the other possible last transitions are: pass transitions over a node labelled with !, ¡, ¿, ᗡ or Ↄ; and rewrite transitions that do not change the underlying graph but discard the rewrite flag \$.

If the last transition is a pass transition over a $Z$-node such that $Z \in \{!, ¡, ¿, ᗡ, Ↄ\}$, the last position (referred to as $in(Z)$) is input to the $Z$-node, and the second last position (referred to as $out(Z)$) is output of the $Z$-node. Induction hypothesis (on $n-1$) implies the factorisation below, where $n = m + l + 1$:

$$\begin{aligned} Init(G) &\to^m ((G,out(Z)),(\uparrow,\square,\star : S,B')) \\ &\to^l ((G,out(Z)),(\downarrow,\square,X : S,B')) \\ &\to ((G,in(Z)),(\downarrow,\square,X : S,B)). \end{aligned}$$

Moreover the state $((G,out(Z)),(\uparrow,\square,\star : S,B'))$ must be the result of a pass transition over the $Z$-node. This means we have the following further factorisation if $Z \neq !$,

$$\begin{aligned} Init(G) &\to^{m-1} ((G,in(Z)),(\uparrow,\square,\star : S,B)) \\ &\to ((G,out(Z)),(\uparrow,\square,\star : S,B')) \\ &\to^l ((G,out(Z)),(\downarrow,\square,X : S,B')) \\ &\to ((G,in(Z)),(\downarrow,\square,X : S,B)) \end{aligned}$$

and the one below if $Z = !$.

$$\begin{aligned} Init(G) &\to^{m-3} ((G,in(Z)),(\uparrow,\square,\star : S,B)) \\ &\to ((G,out(Z)),(\uparrow,?,\star : S,B)) \\ &\to ((G,out(Z)),(\uparrow,!,\star : S,B)) \\ &\to ((G,out(Z)),(\uparrow,\square,\star : S,B')) \\ &\to^l ((G,out(Z)),(\downarrow,\square,X : S,B')) \\ &\to ((G,in(Z)),(\downarrow,\square,X : S,B)) \end{aligned}$$

If the last transition is a rewrite transition that discards the rewrite flag \$, it must follow a pass transition over a \$-node. Let $in$, $out_1$ and $out_2$ denote input, left output and right output, respectively, of the \$-node. We obtain the following factorisation where $n = m + l_2 + l_1 + 3$ and $k = k_1 \$ k_2$, using induction hypothesis twice (on $n-2$ and $n - l_1 - 3$).

$$\begin{aligned} Init(G) &\to^{m-1} ((G,in),(\uparrow,\square,\star : S,B)) \\ &\to ((G,out_2),(\uparrow,\square,\star : \star : S,B)) \\ &\to^{l_2} ((G,out_2),(\downarrow,\square,k_2 : \star : S,B)) \\ &\to ((G,out_1),(\uparrow,\square,\star : k_2 : \star : S,B)) \\ &\to^{l_1} ((G,out_1),(\downarrow,\square,k_1 : k_2 : \star : S,B)) \\ &\to ((G,in),(\downarrow,\$,k : S,B)) \\ &\to ((G,in),(\downarrow,\square,k : S,B)) \end{aligned}$$

□

Inspecting the proof of Prop. C.2.2 gives some intensional characterisation of graphs in stable executions. We say a transition 'involves' a node, if it is a pass transition over the node or it is a rewrite transition whose (main-)redex contains the node.

**Proposition C.3** (Stable executions, intensionally). *Any stable execution of the form* $Init(G) \to^h ((G,e),(\uparrow,\square,\star : S,B)) \to^k ((G,e),(\downarrow,\square,X : S,B))$ *satisfies the following.*

- *If the position $e$ has a ground type or the cell type* $¡\mathbb{F}$, *the last $k$ transitions of the stable execution involve nodes labelled with only* $\{p, \vec{p}, \$, ¡, ᗡ, Ↄ \mid p \in \mathbb{F},\ \vec{p} \in \mathbb{F}^n,\ n \in \mathbb{N}\}$.
- *If the position $e$ has a function type, i.e. $T_1 \to T_2$, it is the input of a $\lambda$-node, and $k = 1$.*

*Proof.* The proof is by looking at how factorisation is given in the proof of Prop. C.2.2. Note that, since we are ruling out argument types, i.e. enriched types of the form of $!T$, the factorisation never encounters !-nodes (hence nor ¿-nodes). □

The fundamental result is that stability of states is preserved by any transitions. This means, in particular, rewrites triggered by the token in an execution can be applied beforehand to the initial graph without changing the end result. Another (rather intuitive) insight is that, in an execution, the token leaves no redexes behind it.

**Proposition C.4** (Preservation of stability). *In any transition, if an old state is stable, so is a new state.*

*Proof.* If the transition does not change the underlying graph, it clearly preserves stability. If not, the preservation is a direct consequence of Lem. C.5 and Lem. C.6 below. □

**Lemma C.5** (Stable executions in graph context). *If all positions in a stable execution* $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G],e),\delta)$ *are in the graph context* $\mathcal{G}$, *there exists a stable execution* $Init(\mathcal{G}[G']) \to^* ((\mathcal{G}[G'],e),\delta)$ *for any graph $G'$ with the same interfaces as the graph $G$.*

*Proof.* The proof is by induction on the length of the stable execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G],e),\delta)$. The base case for null length is trivial. Inductive cases are respectively for all possible last transitions. When the last transition is a pass transition, the single node involved by the transition must be in the graph context $\mathcal{G}[\square]$. Therefore the last transition is still possible when the graph $G$ is replaced, which enables the straightforward use of induction hypothesis.

When the last transition is a 'stable' rewrite transition that simply changes the rewrite flag $f$ to $\square$, we need to inspect its redex. Whereas a part of the redex may not be inside the graph context $\mathcal{G}[\square]$, we confirm below that the same last transition is possible for any substitution of the hole, by case analysis of the rewrite flag $f$. Once this is established, the proof boils down to the straightforward use of induction hypothesis. Possible rewrite flags are ground-type operations \$, and symbols ? and ! for !-box rewrites.

If the rewrite flag is \$, the redex consists of one \$-nodes with two nodes connected to its output. The rewrite flag must have been raised by a pass transition over the \$-node, which means the \$-node is in the graph context $\mathcal{G}[\square]$. Moreover, by Lem. C.2.2, the two other nodes in the redex are also in the graph context $\mathcal{G}[\square]$. Therefore the stable rewrite transition, for the rewrite flag \$, is not affected by substitution of the hole.

If the rewrite flag is ?, the redex is a !-box with all its doors. Since the rewrite flag must have been raised by the pass transition over the principal door, the principal has to be in the graph context $\mathcal{G}[\square]$. All the auxiliary doors of the same !-box are also in the graph context $\mathcal{G}[\square]$, by definition of graphs. The stable rewrite transition for the rewrite flag ? is hence possible, regardless of any substitution of the hole, while the !-box itself may be affected by the substitution. If the rewrite flag is !, the redex is a !-box, all its doors, and a node connected to its principal door. This case is similar to the last case. The connected node, to the principal door, is also in the graph context because the token must have visited the node before passing the principal door. □

**Lemma C.6** (Stabilisation of actual rewrites)**.** *Let*

$$((\mathcal{G}[G], e), \delta) \to ((\mathcal{G}[G'], e'), \delta')$$

*be a rewrite transition, where G is the redex replaced with a different graph G′. If the rewrite transition follows the stable execution* $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e), \delta)$*, there exist an input link* $e_0$ *of the hole* □ *(equivalently of G and G′) and token data* $\delta_0$ *such that:*

- *the stable execution can be factorised as* $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e_0), \delta_0) \to^* ((\mathcal{G}[G], e), \delta)$*, where all positions in the first half sequence are in the graph context* $\mathcal{G}[\square]$
- *there exists a sequence* $((\mathcal{G}[G'], e_0), \delta_0) \to^* ((\mathcal{G}[G'], e'), \delta')$ *in which the graph* $\mathcal{G}[G']$ *is never changed.*

*Proof.* The proof is by case analysis of the rewrite flag of the data $\delta$. Note that we only look at rewrites that actually change the graph.

When the rewrite flag $f$ is $\lambda$, the redex contains a connected pair of an @-node and a $\lambda$-node. We represent the outgoing output of the $\lambda$-node by $out(\lambda)$, one output of the @-node connected to the $\lambda$-node by $in(\lambda)$, the other output of the @-node by $out(@)$, and the input of the @-node by $in(@)$. Lem. C.2.2 implies that the stable execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e), \delta)$ can be factorised as below, for some element $X$ of the computation stack.

$$\begin{aligned}
Init(\mathcal{G}[G]) &\to^* ((\mathcal{G}[G], in(@)), (\uparrow, \square, S, B))\\
&\to ((\mathcal{G}[G], out(@)), (\uparrow, \square, \star : S, B))\\
&\to^* ((\mathcal{G}[G], out(@)), (\downarrow, \square, X : S, B))\\
&\to ((\mathcal{G}[G], in(\lambda))), (\uparrow, \square, @ : S, B)\\
&\to ((\mathcal{G}[G], out(\lambda)), (\uparrow, \lambda, S, B))
\end{aligned}$$

The four links $out(\lambda)$, $in(\lambda)$, $out(@)$ and $in(@)$ cannot happen in the stable prefix execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], in(@)), (\uparrow, \square, S, B))$, except for the last state, otherwise the rewrite flag $\lambda$ must have been raised in this execution, causing the change of the graph. The other link in the redex, the incoming output of the $\lambda$-node, neither appears in the prefix execution, as no pass transition is possible at the link. Therefore the prefix execution contains only links in the graph context $\mathcal{G}[\square]$, and we can take $e_0$ as $in(@)$ and $(\uparrow, \square, S, B)$ as $\delta_0$. The rewrite yields the state $((\mathcal{G}[G'], e'), (\uparrow, \square, S, B)) = ((\mathcal{G}[G'], e_0), \delta_0)$.

When the rewrite flag $f$ is \$, the redex is a \$-node with two constant nodes ($k_1$ and $k_2$) connected. Let $in(\$)$, $in(k_1)$ and $in(k_2)$ denote the unique input of these three nodes, respectively. The stable execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e), \delta)$ is actually in the following form, where $k = k_1 \, \$ \, k_2$.

$$\begin{aligned}
Init(\mathcal{G}[G]) &\to^* ((\mathcal{G}[G], in(\$)), (\uparrow, \square, \star : S', B'))\\
&\to ((\mathcal{G}[G], in(k_2)), (\uparrow, \square, \star : \star : S', B'))\\
&\to ((\mathcal{G}[G], in(k_2)), (\downarrow, \square, k_2 : \star : S', B'))\\
&\to ((\mathcal{G}[G], in(k_1)), (\uparrow, \square, \star : k_2 : \star : S', B'))\\
&\to ((\mathcal{G}[G], in(k_1)), (\downarrow, \square, k_1 : k_2 : \star : S', B'))\\
&\to ((\mathcal{G}[G], in(\$)), (\downarrow, \$, k : S', B))
\end{aligned}$$

The links $in(\$)$, $in(k_1)$ and $in(k_2)$ cannot appear in the stable prefix execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], in(\$)), (\uparrow, \square, \star : S', B'))$, except for the last state, otherwise the rewrite flag \$ must have been raised and have triggered the change of the graph. As the links $in(k_1)$ and $in(k_2)$ are the only ones outside the graph context $\mathcal{G}[G]$ and the link $in(\$)$ is input of the redex $G$, the prefix execution is entirely in the graph context $\mathcal{G}[\square]$. We can take $in(\$)$ as $e_0$ and $(\uparrow, \square, \star : S', B')$ as $\delta_0$. The rewrite of the redex does not change the position, which means $e_0 = e' = in(\$)$. The resulting graph $G'$ consists of one constant node ($k$), and we have a single transition $((\mathcal{G}[G'], e_0)), (\uparrow, \square, \star : S', B')) \to ((\mathcal{G}[G'], e)), (\downarrow, \square, k : S', B))$ to the result state of the rewrite.

When the rewrite flag is $\mathsf{F}(n)$, the redex is a F-node with one !-box connected to its right output link. By Lem. C.2.2, the stable execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e), \delta)$ is in the form of:

$$\begin{aligned}
Init(\mathcal{G}[G]) &\to^* ((\mathcal{G}[G], in), (\uparrow, \square, \star : S', B'))\\
&\to ((\mathcal{G}[G], out_2), (\uparrow, \square, \star : \star : S', B'))\\
&\to^* ((\mathcal{G}[G], out_2), (\downarrow, \square, \vec{p} : \star : S', B'))\\
&\to ((\mathcal{G}[G], out_1), (\uparrow, \square, \star : \vec{p} : \star : S', B'))\\
&\to^* ((\mathcal{G}[G], out_1), (\downarrow, \square, \lambda : \vec{p} : \star : S', B'))\\
&\to ((\mathcal{G}[G], in), (\uparrow, \mathsf{F}(n), \star : S', B))
\end{aligned}$$

where $in$, $out_2$ and $out_1$ denote the input, the right output and the left output of the F-node, respectively, and $\vec{p} \in \mathbb{F}^n$. Any links in the redex, i.e. the three interface links of the F-node and links in the connected !-box, are covered by the link $in$. Therefore by Lem. C.2.1, these links do not appear in the prefix execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], in), (\uparrow, \square, \star : S', B'))$ except for the last $in$. The last state of the prefix execution has the same token position and token data as the result of the rewrite.

The rewrite flag ? is raised by a pass transition over a !-node, principal door of a !-box. The pass transition is the last one of the stable execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e), \delta)$, i.e.

$$Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], in), \delta) \to ((\mathcal{G}[G], out), \delta)$$

where $in$ and $out$ are respectively the input and output of the !-node. Since any ?-rewrite leaves the !-node in place and keeps the position and data of the token, we have a pass transition $((\mathcal{G}[G'], in), \delta) \to ((\mathcal{G}[G'], out), \delta)$ to the resulting state of the rewrite. It remains to be seen whether the stable prefix execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], in), \delta)$ is entirely in the graph context $\mathcal{G}[\square]$.

First, the links $in$ and $out$ cannot appear in the prefix execution except for the last, otherwise there must have been a non-stable ?-rewrite. When the rewrite flag ? triggers the contraction rule (bottom-right in Fig. 7a) or the absorption rule (left in Fig. 7b), any links in the redex are covered by the link $in$, by definition of graphs. Therefore by Lem. C.2.1, these links neither appear in the prefix execution. When the projection rule (top-right in Fig. 7a) occurs, the interface links of the P-node do not appear in the prefix execution, as there is no pass transition over the P-node. Since the input link of the P-node covers all the other links in the redex, by Lem. C.2.1, no links in the redex have been visited by the token. The case of the graph abstraction rule (left in Fig. 7a) is similar to the projection case. Recall that the redex for the graph abstraction rule excludes the sub-graph ($G \circ (\vec{p})^\ddagger$ in the figure) that stays the same. In graph abstraction case, all links in the redex do not appear in the prefix execution, while the unchanged sub-graph is included by the graph context $\mathcal{G}[\square]$ by assumption.

Finally for the rewrite flag !, the stable execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e), \delta)$ ends with several pass transitions, including one over a !-node, and a rewrite transition that sets the rewrite flag:

$$\begin{aligned}Init(\mathcal{G}[G]) &\to^* ((\mathcal{G}[G], e_0), \delta_0) \to^* ((\mathcal{G}[G], in), (\uparrow, \square, S, B))\\ &\to ((\mathcal{G}[G], out), (\uparrow, ?, S, B)) \to ((\mathcal{G}[G], out), (\uparrow, !, S, B))\end{aligned}$$

where $e_0$ is an input link of the redex $G$, and *in* and *out* are respectively the input and output of the !-node. Inspecting each !-rewrites yields a stable sequence $((\mathcal{G}[G'], e_0), \delta_0) \to^* ((\mathcal{G}[G'], out), (\uparrow, !, S, B))$ to the result state of the rewrite. The inspection also confirms that the stable prefix execution $Init(\mathcal{G}[G]) \to^* ((\mathcal{G}[G], e_0), \delta_0)$ is entirely in the graph context $\mathcal{G}$, as below.

In the first two rewrites of Fig. 7c, the link $e_0$ is the only input of the redex and it covers the whole redex. As the link $e_0$ cannot appear in the stable prefix execution, neither any link in the redex, by Lem. C.2.1. In the last rewrite of Fig. 7c, the input link $e_0$ of the redex covers the whole redex except for the other input links. The uncovered input links, in fact, must have not been visited by the token, otherwise the token has proceeded to a !-node and triggered copying. □

Since stability is trivial for initial states, we can always assume stability at any states in an execution.

**Proposition C.7** (Stability of executions). *In any execution $Init(G_0) \to^* ((G, e), \delta)$, the state $((G, e), \delta)$ is stable.*

*Proof.* This is a consequence of Prop. C.4, since any initial states are trivially stable. □

## D Productivity and safe termination

By assuming both validity and stability, we can prove *productivity*: namely, a transition is always possible at a valid and stable intermediate state.

**Proposition D.1** (Productivity). *If a state is valid, stable and not final, there exists a possible transition from the state.*

*Proof in Sec. D.1.* □

We can obtain a sufficient condition for the safe termination of an execution, which is satisfied by the translation of any program.

**Proposition D.2** (Safe termination). *Let $Init(G_0)$ be an initial state whose graph $G_0$ meets the name criteria and the graph criterion. If an execution $Init(G_0) \to^* ((G, e), \delta)$ can be followed by no transition, the last state $((G, e), \delta)$ is a final state.*

*Proof.* This is a direct consequence of Prop. B.12, Prop. C.7 and Prop. D.1. □

**Proposition D.3** (Safe termination of programs). *For any closed program $t$ such that $- \mid - \mid \vec{p} \vdash t : T$, if an execution on the translation $(- \mid - \mid \vec{p} \vdash t : T)^\ddagger$ can be followed by no transition, the last state of the execution is a final state.*

*Proof.* The translation $(- \mid - \mid \vec{p} \vdash t : T)^\ddagger$ fulfills the name criteria and the graph criterion, which can be checked inductively. Note that all names in the translation are bound. This proposition is hence a consequence of Prop. D.2. □

### D.1 Proof of Prop. D.1

First we assume that the state has rewrite flag $\square$. Failure of pass transitions can be caused by either of the following situations: (i) the position is eligible but the token data is not appropriate, or (ii) the position is not eligible.

The situation (i) is due to the wrong top elements of a computation/box stack. In most cases, it is due to the single top element of the computation stack, or top elements of the box stack, which can be disproved easily by validity, or respectively, stability. The exception is when the token points downwards at the left output of a primitive operation node (#), and the top three elements of the computation stack have to be checked. Let *in*, $out_1$ and $out_2$ denote the input, the left output and the right output of the \$-node, respectively. By stability and Lem. C.2.2, the state is the last state of a stable execution of the following form.

$$Init(G) \to^* ((G, out_1), (\uparrow, \square, \star : S, B)) \to^* ((G, out_1), (\downarrow, \square, X_1 : S, B))$$

The intermediate state has to be the result of a pass transition, i.e.

$$\begin{aligned}Init(G) &\to^* ((G, out_2), (\downarrow, \square, S, B)) \to ((G, out_1), (\uparrow, \square, \star : S, B))\\ &\to^* ((G, out_1), (\downarrow, \square, X_1 : S, B)).\end{aligned}$$

Since the last state is valid, the graph $G$ fulfills the criteria (Def. B.2, Def. B.4 and Def. B.7) and any states in this execution is valid by Prop. B.12. Therefore the computation stack $S$ is in the form of $S = X_2 : S'$, and using Lem C.2.2 again yields:

$$\begin{aligned}Init(G) &\to^* ((G, out_2), (\uparrow, \square, \star : S', B))\\ &\to^* ((G, out_2), (\downarrow, \square, X_2 : S', B))\\ &\to ((G, out_1), (\uparrow, \square, \star : X_2 : S', B))\\ &\to^* ((G, out_1), (\downarrow, \square, X_1 : X_2 : S', B)).\end{aligned}$$

The first intermediate state, again, has to be the result of a pass transition, i.e.

$$\begin{aligned}Init(G) &\to^* ((G, in), (\uparrow, \square, S', B)) \to ((G, out_2), (\uparrow, \square, \star : S', B))\\ &\to^* ((G, out_2), (\downarrow, \square, X_2 : S', B))\\ &\to ((G, out_1), (\uparrow, \square, \star : X_2 : S', B))\\ &\to^* ((G, out_1), (\downarrow, \square, X_1 : X_2 : S', B)).\end{aligned}$$

Since the first intermediate state of the above execution is valid, the computation stack $S'$ is in the form of $S' = \star : S''$, which means $S = X_1 : X_2 : \star : S''$. Moreover validity ensures that the elements $X_1$ and $X_2$ are values and eligible for a pass transition from the last state. In particular, vector operations $+$, $\times$ and $\cdot$ are always given two vectors of the same size.

We move on to the situation (ii), where the token position is not eligible to pass transitions. To disprove this situation, we assume a valid, stable and non-final state from which no pass transition is possible, and derive contradiction.

The first case is the state $((G, e), (\downarrow, \square, S, B))$ where the position $e$ is the incoming output of a $\lambda$-node. By the graph criterion, the position is covered by the outgoing output $out(\lambda)$ of the $\lambda$-node, and Lem. C.2 implies the following stable execution.

$$Init(G) \to^* ((G, out(\lambda)), (\uparrow, f, S', B')) \to^* ((G, e), (\downarrow, \square, S, B))$$

Due to stability, the intermediate state $((G, out(\lambda)), \delta)$ must be the result of a pass transition over the $\lambda$-node. However the transition sets $\lambda$ as the rewrite flag $f$, which triggers elimination of the $\lambda$-node and contradicts stability.

The second case is the state $((G, e), (\downarrow, \square, S, B))$ where the position $e$ is the left output of an @-node. By validity the computation stack $S$ is in the form of $S = \lambda : S'$, and Lem. C.2.2 gives the stable execution

$$Init(G) \to^* ((G, e), (\uparrow, \square, \star : S', B)) \to^* ((G, e), (\downarrow, \square, \lambda : S', B))$$

to the state. The only transitions that can yield the intermediate state, at the left output of the @-node, are rewrite transitions that change the graph, which is contradiction.

The third case is the state $((G, e), (\uparrow, \square, S, B))$ where the position $e$ is the input of a ?-node, an auxiliary door of a !-box. Since the link $e$ is covered by the root of the !-box, by Lem. C.2.1, the token has visited its principal door, i.e. !-node. This visit must have raised rewrite flag ?. Because of the presence of the ?-node, the rewrite flag must have triggered a rewrite that eliminates the ?-node, which is contradiction.

The fourth case is when the position $e$ is one of the interface (i.e. either input or output) links of an A-node or a P-node. By the covering condition of the graph criterion, this case reduces to the previous case.

The last case is the state $((G, out(Z)), (\downarrow, \square, S, B))$ where the position $out(Z)$ is the output of an $Z$-node, for $Z \in \{\mathsf{D}, \mathsf{C}, ?\}$. If $Z = ?$, i.e. the node is an auxiliary door of a !-box, the position is covered by the root of the !-box. This reduces to the previous case. If not, i.e. $Z \in \{\mathsf{D}, \mathsf{C}\}$, Lem. C.2.2 gives the stable execution

$$Init(G) \to^* ((G, out(Z)), (\uparrow, \square, S, B)) \to^* ((G, out(Z)), (\downarrow, \square, S, B))$$

to the state. The graph criterion (Def. B.7) implies the $Z$-node belongs to an acyclic box-path of argument types. By typing, any maximal acyclic box-path ends with either a $\lambda$-node or a !-node, and the token must visit this node in the second half of the stable execution. This implies contradiction as follows. If the last node of the maximal box-path is a $\lambda$-node, the token must visit the incoming output link of the $\lambda$-node, from which the token cannot been proceeded. If the last node is a !-node, the token must pass the !-node and trigger rewrites that eliminate the $Z$-node.

This completes the first half of the proof, where we assume the state has rewrite flag $\square$. In the second half, we assume that the state has a rewrite flag which is not $\square$, and show the graph of the state contains an appropriate redex for the rewrite flag.

When the rewrite flag is $\lambda$, by stability, the token must be at the outgoing output of a $\lambda$-node which is connected to the left output of an @-node. Therefore the $\lambda$-rewrite is possible. For the rewrite flag $\mathsf{F}(n)$, stability implies that a !-box with no definitive auxiliary doors (?) must be connected to the right output of a F-node. Rewrite transitions for rewrite flags \$ and $\mathsf{F}(n)$ are exhaustive.

When the rewrite flag is !, by stability, the state is the result of the ?-rewrite that only changes the rewrite flag. This means the token is at the root of a !-box with no definitive auxiliary doors (?-nodes). Transitions for the rewrite flag ! are exhaustive for the closed !-box.

When the rewrite flag is ?, the token is at the root of a !-box, which we here call 'inhabited !-box'. By typing, output links of definitive auxiliary doors of the inhabited !-box can be connected to C-nodes, P-nodes, A-nodes, !-nodes, ?-nodes or $\lambda$-nodes. However ?-nodes and $\lambda$-nodes are not the case, as we see below in two steps.

First, we assume that a definitive auxiliary door of the inhabited !-box is connected to another ?-node. This means that the inhabited !-box is inside another !-box, and therefore the token position is covered by the root of the outer !-box. By Lem. C.2.1, the token must have visited the principal door of the outer !-box and triggered the change of the graph, which contradicts stability.

Second, we assume that a definitive auxiliary door of the inhabited !-box is connected to the incoming output link of a $\lambda$-node. This $\lambda$-node cannot be inside the inhabited !-box, since no !-box has incoming output. Clearly there exists a box-path from the token position to the incoming output of the $\lambda$-node. Therefore the token position, the unique input of the !-box, is covered by the input of the $\lambda$-node; otherwise the graph criterion is violated. This covering implies that the token must have passed the $\lambda$-node upwards and triggered its elimination, by Lem. C.2.1, which contradicts stability.

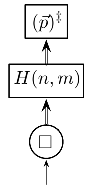
Now, if a !-node is connected to a definitive auxiliary door of the inhabited !-box, the !-node is a principal door of another !-box, and the !-box may have definitive auxiliary doors. Output links of these definitive auxiliary doors, again, can be only connected to C-nodes, P-nodes, A-nodes, or !-nodes, which are principal doors of other !-boxes. This means there can be chains of !-boxes starting from the inhabited !-box, where a principle door of a !-box is connected to a definitive auxiliary door of another !-box via C-nodes and P-nodes. Since the graph of any valid graph state has no output and satisfies the acyclicity property (in Def. B.7), these chains must be acyclic, and hence they must end with !-boxes without definitive auxiliary doors. Therefore, when the rewrite flag ? is raised, there is always a possible remote rule.

The last remark for the rewrite flag ? is about the replacement of nodes in the graph abstraction rule. Typing of links ensures that the replacement never fails and produces a correct graph. In particular the sub-graph $G$ in Fig. 7a only consists of Ↄ-nodes and ¿-nodes. Finally, in conclusion, the state with rewrite flag ? is always eligible for at least one of the rules in Fig. 7a and Fig. 7b. $\square$

## E Provisional contexts and congruence of execution

To deal with shared cells that arise in an execution, we introduce another perspective on composite graphs which takes Ↄ-nodes into account.

**Definition E.1** (Provisional contexts). A graph context of the form



denoted by $\mathcal{P}[\square]$, is a *provisional context* if it satisfies the following.

- The graph $H(n, m)$ consists solely of Ↄ-nodes, and $\vec{p} \in \mathbb{F}^m$.
- For any graph $G(1, n)$ that fulfills the graph criterion, the graph $\mathcal{P}[G]$ also fulfills the graph criterion.

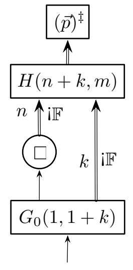
In the above definition, the second condition implies that the graph $H$ contains no loops. We sometimes write $\mathcal{P}[\square]^n_T$ to make explicit the input type and the number of output links of the hole. Note that a graph $\mathcal{P}[G]$, where $\mathcal{P}[\square]$ is a provisional context and $G$ is a definitive graph, is a composite graph. As an extension of Prop. 3.3, we can see a provisional context is preserved by transitions.

**Proposition E.2** (Provisional context preservation)**.** *When a transition sends a graph $G$ to a graph $G'$, if the old graph $G$ can be decomposed as $\mathcal{P}[H]$ where $\mathcal{P}[\square]$ is a provisional context and $H(1,n)$ is a definitive graph, the new graph $G'$ can be also decomposed as $\mathcal{P}[H']$ for some definitive graph $H'(1,n)$, using the same provisional context.*

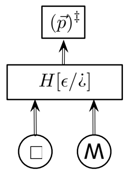
*Proof.* In addition to Prop. 3.3, no transition changes existing Ɔ-nodes. Therefore a provisional context is preserved in any transition. □

Since our operational semantics is based on low-level graphical representation and local token moves, rather than structured syntactical representation, any structural reasoning requires extra care. For example, evaluation of a term of function type is not exactly the same, depending on whether the term appears in the argument position or the function position of function application $t\,u$. The token distinguishes the evaluation using elements $\star$ and @ of a computation stack, which is why we explicitly require termination in definition of $\mathbf{P}_{T_1 \to T_2}$. Moreover congruence of execution is not trivial.

To prove a specific form of congruence, we begin with 'extracting' a provisional context out of a graph context. Let $\mathcal{G}[\square]$ be a graph context, such that for any definitive graph $G(1,n)$ of type $T$, the graph $\mathcal{G}[G]$ is a composite graph. We can decompose the graph context $\mathcal{G}[\square]$ as

$\mathcal{G}[\square] =$ 

where the graph $H(n+k,m)$ consists of all reachable nodes from output links of the hole □. By the assumption on the graph context $\mathcal{G}[\square]$, all the output links of the hole □ have the cell type ¡𝔽. Therefore typing ensures the graph $H$ in fact consists of only Ɔ-nodes and ¿-nodes. We can turn the graph $H \circ (\vec{p})^{\ddagger}$ to a provisional context $\mathcal{P}[\square]$, by dropping all ¿-nodes and adding $k$ weakening nodes (⩕), as below.

$\mathcal{P}[\square] =$ 

We say the provisional context $\mathcal{P}[\square]$ is 'induced' by the graph context $\mathcal{G}[\square]$.

**Proposition E.3** (Congruence of execution)**.** *Let $\mathcal{G}[G]$ be a composite graph where $G(1,n)$ is a definitive graph of type $T$, and $e$ be the root of the hole of the graph context $\mathcal{G}[\square]$. Assume an execution $Init(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'],e'),(d,f,S',B'))$, where the provisional context $\mathcal{P}[\square]$ is induced by the graph context $\mathcal{G}[\square]$. Then, for any stacks $S$ and $B$, there exists a sequence*

$$((\mathcal{G}[G],e),(\uparrow,\square,\star : S,B)) \to^* ((\mathcal{G}[G'],e'),(d,f,S'',B'')) \qquad (1)$$

*of transitions, where $S'' = S'{::}S$ and $B'' = B'{::}B$. The decomposition :: replaces the bottom element □ of the first stack with the second stack. Moreover, if $T$ is a function type, there also exists a sequence*

$$((\mathcal{G}[G],e),(\uparrow,\square,@ : S,B)) \to^* ((\mathcal{G}[G'],e'),\delta') \qquad (2)$$

*of transitions, for some token data $\delta'$.*

*Proof.* By the way the provisional context $\mathcal{P}[\square]$ is induced by the graph context $\mathcal{G}[\square]$, any link of the graph $\mathcal{P}[G]$ has at least one (canonically) corresponding link in the graphs $G$, $H$ and $(\vec{p})^{\ddagger}$. A link may have several corresponding links, because of ¿-nodes dropped in the induced provisional context $\mathcal{P}[\square]$. The sequence (1) is given as a consequence of the following weak simulation result, where weakness is due to ¿-nodes.

***Weak simulation.*** Let

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

be a single transition of the assumed execution $Init(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'],e'),(d,f,S',B'))$. For any computation stack $S_0$ and any box stack $B_0$, there exists a sequence $((\mathcal{G}[H],e),(d,f,S :: S_0,B :: B_0)) \to^* ((\mathcal{G}[H'],e''),(d',f',S' :: S_0,B' :: B_0))$ of transitions from a stable state, where the position $e''$ is one of those corresponding to $e'$.

The proof of the weak simulation follows from the fact that the presence of the graph $G_0$ below the graph $G$ does not raise any extra rewriting, so long as the token data is taken from the execution on the graph $\mathcal{P}[G]$.

Finally, if $T$ is a function type, replacing the element $\star$ with the element @ in the sequence (1) only affects a pass transition over a $\lambda$-node, which sends the computation stack $\star : S$ to $\lambda : S$.

The execution $Init(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'],e'),(d,f,S',B'))$, in fact, can contain at most one pass transition over a $\lambda$-node which changes the computation stack $\star : \square$ to $\lambda : \square$. To make the second such transition happen, some other transition has to remove the top element of the computation stack $\lambda : \square$, however by stability (Prop. C.7), no transition can do this. Moreover, such pass transition can be only the last transition of the execution. Any transitions that can possibly follow the pass transition, which sets direction $\downarrow$ and computation stack $\lambda : \square$, are pass transitions over !-nodes, ¡-nodes, ¿-nodes or Ↄ-nodes; the existence of these nodes contradicts with the type $T = T_1 \to T_2$ of the underlying graph.

Since the sequence (1) weakly simulates the execution, where the weakness comes from only ¿-nodes, we can conclude that there is no occurrence, or exactly one occurrence at the last, in the sequence (1), of the pass transition which is affected by the replacement of the element $\star$ with the element @. Therefore if the sequence (1) contains no such pass transition, the sequence (2) can be directly obtained by replacing the element $\star$ with the element @. Otherwise, cutting the last transition of the sequence (1) just does the job, as the transition does not change the underlying graph and the token position. □

## F Data-flow graphs

This section looks at graphs consisting of specific nodes. The restriction on nodes rules out some rewrites, especially @-rewrites for function application and the graph abstraction rule.

**Definition F.1** (Data-flow graphs)**.** A *data-flow* graph is a graph with no ¡-nodes, that satisfies the following.

- All its input links have ground types.
- Any reachable (in the normal graphical sense) nodes from input links are labelled with $\{p, \vec{p}, \$, !, ?, ¿, \mathrm{D}, \mathrm{C}, ᗡ, Ↄ \mid p \in \mathbb{F},\ \vec{p} \in \mathbb{F}^n,\ n \in \mathbb{N}\}$.

In particular, a data-flow graph is called *pure* if these reachable nodes are not labelled with $\{!, ?, ¿, \mathrm{D}, \mathrm{C}\}$.

Data-flow graphs intensionally characterises graphs of final states. Graphs of final states play the role of 'values', since our semantics implements (right-to-left) call-by-value evaluation.

**Proposition F.2** (Final graphs intensionally). *Let $G \circ (\vec{p})^{\ddagger}$ be a composite graph of (non-enriched) type $T$. If a final state $Final(G \circ (\vec{p})^{\ddagger}, \kappa)$ is stable, the definite graph $G$ satisfies the following.*

1. *When $T$ is a ground type, the graph $G$ is a pure data-flow graph.*
2. *When $T$ is a function type, i.e. $T = T_1 \to T_2$, the root of the graph $G$ is the input of a $\lambda$-node.*

*Proof.* The second case, where $T = T_1 \to T_2$, is a direct consequence of Prop. C.3. For the first case, where $T$ is a ground type, Prop. C.3 tells us that the stable execution $Init(G \circ (\vec{p})^{\ddagger}) \to^* Final(G \circ (\vec{p})^{\ddagger}, \kappa)$ only involves nodes labelled with $\{p, \vec{p}, \$, ¡, ᗡ, Ↄ \mid p \in \mathbb{F}, \vec{p} \in \mathbb{F}^n, n \in \mathbb{N}\}$. It boils down to show that any reachable node from the root of the graph $G$ is involved by the stable execution. We can show this by induction on the maximum length of paths from the root to a reachable node. The base case is trivial, as the root of the graph $G$ coincides with an input link of the reachable node. In the inductive case, induction hypothesis implies that any reachable node is connected to a reachable node which is involved by the stable execution. By Prop. C.2.2 and labelling of the involved node, the stable execution contains a transition that passes the token upwards over the involved node, and hence makes the reachable node of interest involved by the following transition. □

We can directly prove soundness of data-flow graphs.

**Proposition F.3.** *Let $G(1, n)$ be a data-flow graph, with a link $e$ of ground type which is reachable from the root of $G$. For any vector $\vec{p} \in \mathbb{F}^n$, if a state $((G \circ (\vec{p})^{\ddagger}, e), (\uparrow, \square, S, B))$ is stable and valid, there exists a data-flow graph $G'(1, n)$ that agrees with $G$ on the link $e$, and a computation stack $S'$, such that $((G \circ (\vec{p})^{\ddagger}, e), (\uparrow, \square, S, B)) \to^* ((G' \circ (\vec{p})^{\ddagger}, e), (\downarrow, \square, S', B))$.*

*Proof.* The first observation is that any transition sends a data-flow graph, composed with a graph $(\vec{p})^{\ddagger}$, to a data-flow graph with the same graph $(\vec{p})^{\ddagger}$.

Given a composite graph $G \circ (\vec{p})^{\ddagger}$ where $G$ is a data-flow graph, we define a partial *ranking* map $\rho$ which assigns natural numbers to some links of $G$. The ranking is only defined on links which are reachable from the root of $G$ and labelled with either a ground type or an argument type, as follows: $\rho(e) := 0$ if $e$ is input of a $p$-node ($p \in \mathbb{F}$), a $\vec{q}$-node ($\vec{q} \in \mathbb{F}^k$) or a ᗡ-node; $\rho(e) := max(\rho(e_1), \rho(e_2)) + 1$ if $e$ is input of a \$-node, and $e_1$ and $e_2$ are output links of the \$-node; and $\rho(e) := \rho(e') + 1$ if $e$ is input of a !-node, a ?-node, a D-node or a C-node, and $e'$ is the corresponding output link. This ranking on reachable links is well-defined, as the composite graph $G \circ (\vec{p})^{\ddagger}$ meets the graph criterion.

Since the state $((G \circ (\vec{p})^{\ddagger}, e), (\uparrow, \square, S, B))$ is stable and the position $e$ has ground type, the ranking $\rho$ of the composite graph $G \circ (\vec{p})^{\ddagger}$ is defined on the position $e$. The proof is by induction on the rank $\rho(e)$.

Base cases are when $\rho(e) = 0$. If the position $e$ is input of a ᗡ-node, the graph criterion implies an acyclic directed path from the ᗡ-node to a ¡-node. Intermediate nodes of this path are only Ↄ-nodes, and the proof is by induction on the number of these Ↄ-nodes. Otherwise, the position $e$ is input of a constant node ($p$ or $\vec{q}$), and the proof is by one pass transition over the node.

In inductive cases, induction hypothesis is for any natural number that is less than $\rho(e)$. When the position $e$ is input of a D-node, the graph criterion implies an acyclic directed path from the D-node to a !-node, with only C-nodes as intermediate nodes. This means, from the state $((G \circ (\vec{p})^{\ddagger}, e), (\uparrow, \square, S, B))$, the token goes along the directed path, reaches the !-node, and trigger rewrites. These rewrites eliminate all the nodes in the path, and possibly include remote rules that eliminate other ?-nodes and C-nodes. When these rewrites are completed, the position $e$ and its type are unchanged, but its rank $\rho(e)$ is strictly decreased. Therefore we can use induction hypothesis to prove this case. The last case, when the position $e$ is input of a \$-node, boils down to repeated but straightforward use of induction hypothesis, which may be followed by a \$-rewrite. □

**Corollary F.4** (Soundness of data-flow graphs). *If a data-flow graph $G(1, n)$ meets the name criteria and the graph criterion, for any vector $\vec{p} \in \mathbb{F}^n$, there exist a data-flow graph $G'(1, n)$ and a token value $\kappa$ such that $Init(G \circ (\vec{p})^{\ddagger}) \to^* Final(G' \circ (\vec{p})^{\ddagger}, \kappa)$.*

Graph abstraction enables us to replace a computation graphs with a regular function, parameterised explicitly by what used to be the cells of the computation graph. This replacement is not at all simple; in an execution, it can happen inside a !-box, or happen outside a !-box before the resulting graph gets absorbed by the !-box. Moreover, it can change the number of cells extracted by graph abstraction. Our starting point to formalise this idea of replacement is the notion of 'data-flow chain'. It is a sequence of sub-graphs, which are partitioned by auxiliary doors and essentially representing data flow.

**Definition F.5** (Data-flow chains). In a graph $G$, a *data-flow chain* $\mathbf{D}$ is given by a sequence $D_0(n_0, n_1), \ldots, D_k(n_k, n_{k+1})$ of $k+1$ sub-graphs, where $k$ is a natural number, that satisfies the following.

- The first sub-graph $D_0(n_0, n_1)$ is a data-flow graph.
- If $k > 0$, there exists a unique number $h$ such that $0 < h \leq k$.

  For each $i = 1, \ldots, h-1$, the sub-graph $D_i(n_i, n_{i+1})$ can contain only C-nodes, P-nodes, Ↄ-nodes or !-boxes with their doors, where these !-boxes are data-flow graphs. Input links of the sub-graph $D_i(n_i, n_{i+1})$ are connected to output links of the previous sub-graph $D_{i-1}(n_{i-1}, n_i)$, via $n_i$ parallel ?-nodes. These delimiting parallel doors (?) belong to the same !-box, whose principal door (!) is not included in the whole sequence of sub-graphs.

  For each $j = h, \ldots, k$, the sub-graph $D_j(n_j, n_{j+1})$ solely consists of Ↄ-nodes. Input links of the sub-graph $D_j(n_j, n_{j+1})$ are connected to output links of the previous sub-graph $D_{j-1}(n_{j-1}, n_j)$, via $n_j$ parallel ¿-nodes. These delimiting parallel doors (¿) belong to the same !-box, whose principal door (!) is not included in the whole sequence of sub-graphs.
- The final sub-graph $D_k(n_k, n_{k+1})$ satisfies either one of the following: (i) all its output links have the cell type $¡\mathbb{F}$ and connected to ¡-nodes, or (ii) all its output links are input

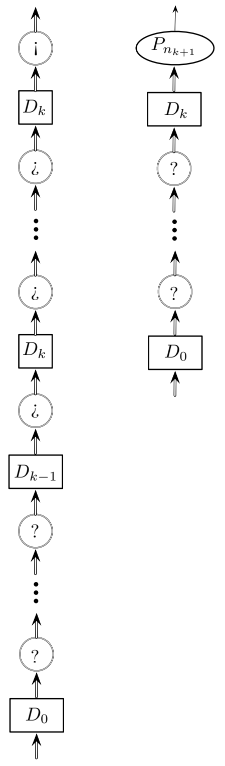


**Figure 11**

links of a single P-node with $n_{k+1}$ inputs, whose output link is connected to a $\lambda$-node.

- If a node of the graph $G$ is box-reachable from an input link of the first sub-graph $G_0$, it is either (i) in the sub-graphs $D_0, \ldots, D_k$, (ii) in auxiliary doors partitioning them, or (iii) box-reachable from an output link of the last sub-graph $G_k$.

We refer to input of the first sub-graph $D_0$ as input of the data-flow chain $\mathbf{D}$, and output of the last sub-graph $D_k$ as output of the data-flow chain $\mathbf{D}$. Fig. 11 illustrates some forms of data-flow chains.

We define a binary relation $\propto$ between definitive graphs that applies single replacement of a data-flow chain. It is lifted to a binary relation $\overline{\propto}$ on some states.

**Definition F.6** (Data-flow replacement of graphs). Let $G(1,n)$ and $H(1,m)$ be two definitive graphs, that contain data-flow chains $\mathbf{D}_G$ and $\mathbf{D}_H$, respectively. Two definitive graphs $G(1,n)$ and $H(1,m)$ satisfies $G \propto H$ if the following holds.

- Two data-flow chains have the same number of input. The data-flow chain $\mathbf{D}_H$ has no more length than the data-flow chain $\mathbf{D}_G$. The data-flow chain $\mathbf{D}_H$ can have an arbitrary number of output, whereas the data-flow chain $\mathbf{D}_G$ has at least one output.
- Exactly the same set of names appears in both graphs $G$ and $H$.
- The graphs $G$ and $H$ only differ in the data-flow chains $\mathbf{D}_G$ and $\mathbf{D}_H$, and their partitioning auxiliary doors.

**Definition F.7** (Data-flow replacements of states). Two stable and valid states $((G \circ (\vec{p})^{\ddagger}, e), (d, f_1, S_1, B_1))$ and $((H \circ (\vec{q})^{\ddagger}, e), (d, f_2, S_2, B_2))$ are related by a binary relation $\overline{\propto}$ if the following holds.

- The definitive graphs satisfy $G \propto H$.
- The position $e$ is either an input link of the data-flow chain replaced by $\propto$, or (strictly) outside the data-flow chain.
- Two stable executions to these states give the exactly same sequence of positions.
- The rewrite flags $f_1$ and $f_2$ may only differ in numbers $n$ of $\mathsf{F}(n)$.
- The validation maps $v_G$ and $v_H$ of these states, respectively, satisfy that: $v_G(a) = 0$ implies $v_H(a) = 0$, for any $a \in \mathbb{A}$ on which they both are defined.

As usual, we use $\overline{\propto}^*$ to denote the reflexive and transitive closure of the relation $\overline{\propto}$.

**Proposition F.8.** *If*

$$((G \circ (\vec{p})^{\ddagger}, e), (\uparrow, f_1, S_1, B_1)) \overline{\propto}^* ((H \circ (\vec{q})^{\ddagger}, e), (\uparrow, f_2, S_2, B_2))$$

*holds, a sequence*

$$((G \circ (\vec{p})^{\ddagger}, e), (\uparrow, f_1, S_1, B_1)) \\ \to^* ((G' \circ (\vec{p}^{\ddagger}), e), (\downarrow, \square, S_1', B_1')) \quad (3)$$

*implies a sequence*

$$((H \circ (\vec{q})^{\ddagger}, e), (\uparrow, f_2, S_2, B_2)) \\ \to^* ((H' \circ (\vec{q})^{\ddagger}, e), (\downarrow, \square, S_2', B_2')) \quad (4)$$

*such that the resulting states are again related by* $\overline{\propto}^*$.

*Proof.* The proof is by induction on the length of the sequence (3). Base cases are when the sequence (3) consists of one pass transition over a constant node (scalar or vector) or a $\lambda$-node, and hence $f = \square$. If the transition is over a $\lambda$-node, the same transition is possible at the state $((H \circ (\vec{q})^{\ddagger}, e), (\uparrow, f, S_2, B_2))$. If the transition is over a constant node, the constant node may be a part of a data-flow chain replaced by a data-flow chain $\mathbf{D}$ in the graph $H$. By stability of the state $((H \circ (\vec{q})^{\ddagger}, e), (\uparrow, f, S_2, B_2))$, we can conclude that any partitioning auxiliary doors of the data-flow chain $\mathbf{D}$ are ¿-nodes, if they are box-reachable from the position $e$. This can be confirmed by contradiction as follows: otherwise the position $e$ must be in a !-box with definitive auxiliary doors (i.e. ?-nodes), which contradicts with stability and the fact $f = \square$. This concludes the proof for base cases.

First inductive case is when the position $e$ is an input link of a data-flow chain $\mathbf{D}_G$, which is replaced with a data-flow chain $\mathbf{D}_H$ in the graph $H$. If the rewrite flag is $f = \square$, similar to base cases, we can see that the data-flow chain $\mathbf{D}_H$ is in fact not partitioned by ?-nodes (but possibly ¿-nodes). Moreover by stability and the graph criterion, output links of the data-flow chain $\mathbf{D}_H$ are not connected to a P-node. This implies that the data-flow chain $\mathbf{D}_H$ with partitioning auxiliary doors altogether gives a data-flow graph. Therefore the sequence (4) can be obtained by Prop. F.4 and Prop. E.3.

If the rewrite flag $f$ is not equal to $\square$, possibilities are $f = \lambda, ?, !$. The $\lambda$-rewrite in the graph $G$ sets the rewrite flag $\square$ and does not change the position $e$. The same $\lambda$-rewrite is also possible in the graph $H$, and we can use induction hypothesis. If $f = ?$, there will be at least one rewrite transitions, in both graphs $G$ and $H$, until the rewrite flag is changed to !. These ?-rewrites may affect the data-flow chains $\mathbf{D}_G$ and $\mathbf{D}_H$. Since the position $e$ is an input of the data-flow chains, these ?-rewrites can only eliminate C-nodes, P-nodes, or ?-nodes that partition the data-flow chains. Elimination of C-nodes and P-nodes is where the transitive closure $\propto^*$ plays a role. It does not change the partitioning structure of the data-flow chains $\mathbf{D}_G$ and $\mathbf{D}_H$. Elimination of ?-nodes in the graph $G$ must introduce ¿-nodes, because the replacement $\propto$ requires the data-flow chain $\mathbf{D}_G$ to have at least one output. Therefore the ?-rewrites changes the data-flow chain $\mathbf{D}_G$ to a new one while keeping its length. On the other hand, elimination of ?-nodes may not happen in the graph $H$, or may decrease the length of the data-flow chain $\mathbf{D}_H$. As a result, after the maximal number of ?-rewrites until the rewrite flag is changed to !, resulting graphs are still related by $\propto^*$ and the position $e$ is not changed. Finally if $f = !$, until the rewrite flag is changed to □, the same nodes (D-nodes and C-nodes) are eliminated in both graphs $G$ and $H$, and both the data-flow chains decrease their length by one, if possible. Once rewrites are done and the rewrite flag □ is set, the position $e$ is still unchanged, and we use induction hypothesis.

Second inductive case is when the position $e$ is the input of a F-node, with rewrite flags $f_1 = \mathsf{F}(n_1)$ and $f_2 = \mathsf{F}(n_2)$ are raised. If $n_1 = 0$, by definition of the relation $\overline{\propto}$, it holds that $n_2 = 0$, and the proof follows from stability. If not, the sequence (3) begins with non-trivial unfolding of the F-node. The sequence (4) can be proved by induction on $n_2$, which is an arbitrary natural number, with $n_2 \leq n_1$ being the base case.

Finally, the last inductive case is when the position $e$ is not in data-flow graphs replaced by $\overline{\propto}^*$. If $f = □$ and the sequence (3) begins with a pass transition, the same transition is possible in the graph $H \circ (\vec{q})^\ddagger$, and we can use induction hypothesis. We use it more than once, when the pass transition is over a \$-node. Possibly the sequence (3) ends with a \$-rewrite, which may not be possible on the other side. However, this is when the position $e$ becomes an input of a data-flow chain in the resulting graph $G'$, and the difference of \$-rewrites is dealt with by the replacement $\propto$. If $f \neq □$, discussion in the first inductive case is valid, except for any ?-rewrites being possible, namely the graph abstraction rule. We use induction hypothesis once consecutive rewrites are done. The key fact is that, when the graph abstraction rule applies to graphs related by the replacement $\propto$, the resulting graphs are again related by $\propto$. The resulting graphs may differ in the size of extracted vectors and in the number of input links of the introduced P-nodes. This is dealt with by the replacement $\propto$ of data-flow chains, in particular, a single constant node itself is a data-flow chain. Note that, if a P-node with no inputs is introduced on the side of graph $G$, it is also introduced on the other side, because any data-flow chain of null output is not replaced by $\propto$. The graph abstraction rule is essentially the only transition that is relevant to the condition of validation maps for the relation $\overline{\propto}$, and it does not violate the condition. This concludes the whole proof. □

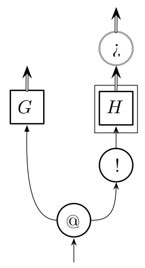
**Corollary F.9** (Safety of dara-flow replacement). *Let $G \circ (\vec{p})^\ddagger$ and $H \circ (\vec{q})^\ddagger$ be composite graphs, meeting the name criteria and the graph criterion, such that $G \propto^* H$. If an execution on the graph $G \circ (\vec{p})^\ddagger$ reaches a final state, an execution on the graph $H \circ (\vec{q})^\ddagger$ also reaches a final state.*

## G Soundness

Our soundness proof uses logical predicates on definitive graphs. These logical predicates are analogous to known ones on typed lambda-terms.

**Definition G.1** (Logical predicates). Given a term $T$, a logical predicate $\mathbf{P}_T$ is on definitive graphs, that meet the name criteria and the graph criterion, of type $T$. It is inductively defined as below.

- When $T$ is a ground type, $G(1,n) \in \mathbf{P}_T$ holds if: for any provisional context $\mathcal{P}[□]^n_T$, there exist a composite graph $H$ and a token value $\kappa$ such that $\mathit{Init}(\mathcal{P}[G]) \to^* \mathit{Final}(H,\kappa)$.
- When $T = T_1 \to T_2$, $G(1,n) \in \mathbf{P}_T$ holds if:
  1. for any provisional context $\mathcal{P}[□]^n_T$, there exists a composite graph $H$ such that $\mathit{Init}(\mathcal{P}[G]) \to^* \mathit{Final}(H,\lambda)$
  2. for any $H(1,m) \in \mathbf{P}_{T_1}$, the following graph, denoted by $G@!H$, satisfies $G@!H \in \mathbf{P}_{T_2}$.



**Proposition G.2** (Deterministic termination). *If $G(1,n) \in \mathbf{P}_T$, for any provisional context $\mathcal{P}[□]^n_T$, there exist a unique definitive graph $G'(1,n)$ of type $T$ and a unique token value $\kappa$ such that $\mathit{Init}(\mathcal{P}[G]) \to^* \mathit{Final}(\mathcal{P}[G'],\kappa)$.*

*Proof.* This is a direct consequence of Prop. 3.6 and Prop. E.2. □

**Proposition G.3** (Congruence of termination). *Let $\mathcal{G}[G]$ be a composite graph where $G(1,n)$ is a definitive graph of type $T$, and $e$ be the root of the hole of the graph context $\mathcal{G}[□]$. If $G(1,n) \in \mathbf{P}_T$ holds, then for any stacks $S$ and $B$, there exist an element $X$ of a computation stack and a sequence*

$$((\mathcal{G}[G],e),(\uparrow,□,\star : S,B)) \to^* ((\mathcal{G}[G'],e),(\downarrow,□,X : S,B))$$

*of transitions. Moreover, if $T$ is a function type, there also exists the following sequence.*

$$((\mathcal{G}[G],e),(\uparrow,□,@ : S,B)) \to^* ((\mathcal{G}[G'],e),(\uparrow,□,@ : S,B))$$

*Proof.* This is a corollary of Prop. E.3. □

The following properties relate logical predicates to transitions, in both forward and backward ways.

**Proposition G.4** (Forward/backward reasoning).

**Forward reasoning** *Let $G(1,n)$ be a definitive graph such that $G \in \mathbf{P}_T$, and $\mathcal{P}[□]^n_T$ be a provisional context. For any execution $\mathit{Init}(\mathcal{P}[G]) \to^* ((H',e),\delta)$ on the graph $\mathcal{P}[G]$, there exists a definitive graph $G'(1,n)$ such that $H' = \mathcal{P}[G']$ and $G' \in \mathbf{P}_T$.*

**Backward reasoning** *A definitive graph $G(1,n)$ satisfies $G \in \mathbf{P}_T$, if: (i) it meets the name criteria and the graph criterion, and (ii) for any provisional context $\mathcal{P}[□]^n_T$, there exist a definitive graph $G'(1,n) \in \mathbf{P}_T$ and a state $((\mathcal{P}[G'],e),\delta)$ such that $\mathit{Init}(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'],e),\delta)$.*

*Proof.* First of all, Prop. E.2 ensures the decomposition $H' = \mathcal{P}[G']$, where $G'(1,n)$ is a definitive graph, in the forward reasoning. We prove the both reasoning simultaneously by induction on the type $T$. Base cases of both reasoning, where $T$ is a ground type, relies on determinism and stability, as we see below.

We begin with the base case of the forward reasoning, where $T$ is a ground type. Given any execution $\mathit{Init}(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'], e), \delta)$ where $G(1,n) \in \mathbf{P}_T$ and $\mathcal{P}[\square]^n_T$ is a provisional context, by Prop. G.2, there exists a unique final state $\mathit{Final}(H'', \kappa)$ such that $\mathit{Init}(\mathcal{P}[G]) \to^* \mathit{Final}(H'', \kappa)$. The uniqueness implies the factorisation $\mathit{Init}(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'], e), \delta) \to^* \mathit{Final}(H'', \kappa)$. Since the state $((\mathcal{P}[G'], e), \delta)$ is stable by Prop. C.7, we have a stable execution $\mathit{Init}(\mathcal{P}[G']) \to^* ((\mathcal{P}[G'], e), \delta)$. As a result we have an execution $\mathit{Init}(\mathcal{P}[G']) \to^* \mathit{Final}(H'', \kappa)$, which proves $G'' \in \mathbf{P}_T$.

In the base case of the backward reasoning, where $T$ is a ground type, $G' \in \mathbf{P}_T$ implies an execution $\mathit{Init}(\mathcal{P}[G']) \to^* \mathit{Final}(H, \kappa)$ to a unique final state (Prop. G.2). Since the last state of the execution $\mathit{Init}(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'], e), \delta)$ is stable by Prop. C.7, there is a stable execution $\mathit{Init}(\mathcal{P}[G']) \to^* ((\mathcal{P}[G'], e), \delta)$. The uniqueness of the final state $\mathit{Final}(H, \kappa)$ gives the sequence $((\mathcal{P}[G'], e), \delta) \to^* \mathit{Final}(H, \kappa)$ of transitions, which yields an execution $\mathit{Init}(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'], e), \delta) \to^* \mathit{Final}(H, \kappa)$ and proves $G \in \mathbf{P}_T$.

In inductive cases of both reasoning, where $T = T_1 \to t_2$, we need to check two conditions of the logical predicate $\mathbf{P}_T$. The first condition, i.e. termination, is as the same as base cases. The other inductive condition can be proved using induction hypotheses of both properties, together with Prop. E.3 and Cor. G.3, as below.

In the inductive case of the forward reasoning, our goal is to prove $G'@!H \in \mathbf{P}_{T_2}$ for any $H(1,m) \in \mathbf{P}_{T_1}$, under the assumption of the execution $\mathit{Init}(\mathcal{P}[G]) \to^* ((\mathcal{P}[G'], e), \delta)$ where $G(1,n) \in \mathbf{P}_{T_1 \to T_2}$. Let $\mathcal{P}'[\square]^{n+m}_{T_2}$ be any provisional context. Since $H \in \mathbf{P}_{T_1}$, Prop. G.3 implies two executions, where the position $e'$ is the right output of the @-node,

$$\mathit{Init}(\mathcal{P}'[G@!H]) \to^* ((\mathcal{P}'[G@!H'], e'), (\downarrow, \square, \kappa : \star : \square, \square)) \quad (5)$$

$$\mathit{Init}(\mathcal{P}'[G'@!H]) \to^* ((\mathcal{P}'[G'@!H'], e'), (\downarrow, \square, \kappa : \star : \square, \square)) \quad (6)$$

such that $\mathit{Init}(\mathcal{P}''[H]) \to^* \mathit{Final}(\mathcal{P}''[H'], \kappa)$ for some provisional context $\mathcal{P}''[\square]^m_{T_1}$ and some token value $\kappa$. By the assumption of $G \in \mathbf{P}_{T_1 \to T_2}$ and Prop. E.3, we can continue the execution (5) as:

$$\begin{aligned} \mathit{Init}(\mathcal{P}'[G@!H]) &\to^* ((\mathcal{P}'[G@!H'], e'), (\downarrow, \square, \kappa : \star : \square, \square)) \\ &\to^* ((\mathcal{P}'[G'@!H'], e), \delta) \end{aligned}$$

for some token data $\delta$. Since $G \in \mathbf{P}_{T_1 \to T_2}$ and $H \in \mathbf{P}_{T_2}$ by the assumption, we can use induction hypothesis of the forward reasoning and obtain $G'@!H' \in \mathbf{P}_{T_2}$. Using induction hypothesis of the backward reasoning along the execution (6), we conclude $G'@!H \in \mathbf{P}_{T_2}$.

In the inductive case of the backward reasoning, we aim to prove $G@!H \in \mathbf{P}_{T_2}$ for any $H(1,m) \in \mathbf{P}_{T_1}$. Let $\mathcal{P}'[\square]^{n+m}_{T_2}$ be any provisional context. Since $H \in \mathbf{P}_{T_1}$, Cor. G.3 gives an execution, where the position $e'$ is the right output of the @-node,

$$\mathit{Init}(\mathcal{P}'[G@!H]) \to^* ((\mathcal{P}'[G@!H'], e'), (\downarrow, \square, \kappa : \star : \square, \square))$$

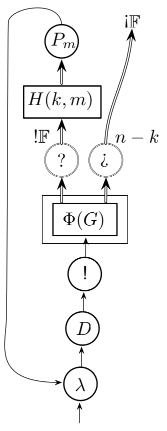
such that there exists a provisional context $\mathcal{P}''[\square]^m_{T_1}$ and an execution $\mathit{Init}(\mathcal{P}''[H]) \to^* \mathit{Final}(\mathcal{P}''[H'], \kappa)$. Using the assumption on the graph $G$, with Prop. E.3, yields its expansion

$$\begin{aligned} \mathit{Init}(\mathcal{P}'[G@!H]) &\to^* ((\mathcal{P}'[G@!H'], e'), (\downarrow, \square, \kappa : \star : \square, \square)) \\ &\to^* ((\mathcal{P}'[G'@!H'], e), \delta) \end{aligned}$$

such that $G' \in \mathbf{P}_{T_1 \to T_2}$, arising in an execution $\mathit{Init}(\mathcal{P}'''[G]) \to^* ((\mathcal{P}'''[G'], e), \delta)$, for some provisional context $\mathcal{P}'''[\square]^n_{T_1 \to T_2}$. Induction hypothesis of the forward reasoning implies $H' \in \mathbf{P}_{T_1}$, and therefore $G'@!H' \in \mathbf{P}_{T_2}$. We conclude $G@!H \in \mathbf{P}_{T_2}$ by induction hypothesis of the backward reasoning. □

The key ingredient of the soundness proof is 'safety' of graph abstraction. This is where we appreciate call-by-value evaluation.

**Proposition G.5** (Safety of graph abstraction). *If $G(1,n) \in \mathbf{P}_T$ holds, the graph $\hat{G}$ given as below*

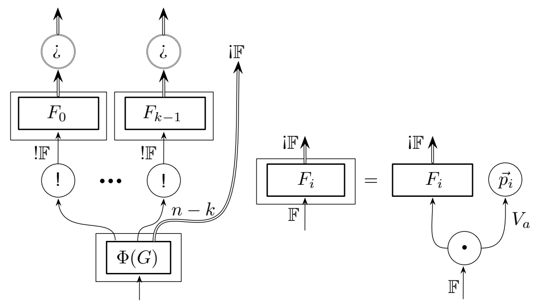
$\hat{G} =$ 

*belongs to $\mathbf{P}_{V_a \to T}$, where:*

- *the name $a \in \mathbb{A}$ is any name that does not appear in the graph $G'$*
- *the graph $H(k,m)$, where $k \leq n$ and $m$ is arbitrary, consists solely of* C*-nodes connected to each other in an arbitrary way, and fulfills the graph criterion*
- *the graph $\phi(G)$ is obtained by: (i) choosing $k$ output links of the graph $G$ arbitrarily, and (ii) replacing any* Ↄ*-nodes,* ¿*-nodes and* ᗡ*-nodes with* C*-nodes,* ?*-nodes and* D*-nodes, respectively, if one of the chosen output links can be reachable from these nodes via links of only the cell type* ¡$\mathbb{F}$.

*Proof.* It is easy to see the graph $\hat{G}$ meets the name criteria and the graph criterion, given $G \in \mathbf{P}_T$. Since the graph $\hat{G}$ has a $\lambda$-node at the bottom, the termination condition of the logical predicate $\mathbf{P}_{V_a \to T}$ is trivially satisfied. For any graph $E \in \mathbf{P}_{V_a}$, we prove $\hat{G}@!E \in \mathbf{P}_T$.

Let $\mathcal{P}'[\square]_T$ be any provisional context. By Cor. G.3, an execution on the graph $\mathcal{P}'[\hat{G}@!E]$ first yields the graph $\mathcal{P}'[\hat{G}@!E']$, where the graph $E'$ comes from some execution $\mathit{Init}(\mathcal{P}_0[E]) \to^* \mathit{Final}(\mathcal{P}_0[E'], \kappa)$ to a final state. Then the token eliminates the pair of the $\lambda$-node and the @-node at the bottom of the graph, and triggers the rewrite involving the graph $H$ (C-nodes) and the P-node of the graph $\phi(G)$. This rewrite duplicates the graph $E'$ in a !-box, introducing dot-product nodes and vector nodes. Finally the token eliminates the D-node and the !-box around the graph $\phi(G)$. In the resulting graph, we shall write as $\mathcal{P}'[R]$, the graph $R$ consists of the graph $\phi(G)$, whose output links of type $!\mathbb{F}$ are connected to !-boxes, and further, Ↄ-nodes. The following illustrates the graph $R$, where Ↄ-nodes are omitted.



Let $\vec{F}$ be these !-boxes. They all have type $\mathbb{F}$, and each of them contains the graph $E'$, with a dot-product node and a vector node. Since the provisional context $\mathcal{P}'$ is arbitrary, we can reduce the problem to $R \in \mathbf{P}_T$, using the backward reasoning (Prop. G.4).

If $n = 0$, the replacement $\phi$ actually changes nothing on the graph $G$, and hence $\phi(G) = G$. Therefore, in this case, $R \in \mathbf{P}_T$ follows from $G \in \mathbf{P}_T$, by Prop. G.3. We deal with the case where $n > 0$ below.

First, as a consequence of Cor. F.9, the execution on the graph $\mathcal{P}'[R]$ reaches a final state, given the assumption $G \in \mathbf{P}_T$. This is because, for any provisional context $\mathcal{P}[\square]^n_T$, we have $\mathcal{P}[G] \propto^* \mathcal{P}'[R]$. Since any name appears in the graph $\mathcal{P}[G]$ also appears in the graph $\mathcal{P}'[R]$, we can infer $\mathit{Init}(\mathcal{P}[G]) \overline{\propto}^* \mathit{Init}(\mathcal{P}'[R])$. This means $R \in \mathbf{P}_T$ when $T$ is a ground type, and the termination condition of $R \in \mathbf{P}_T$ when $T$ is a function type.

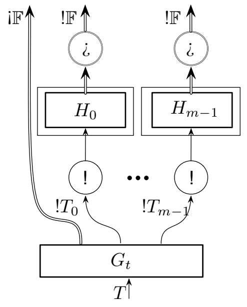
To check the inductive condition of $R \in \mathbf{P}_T$ where $T = T_1 \to T_2$, we need to show $R@!F \in \mathbf{P}_{T_2}$ for any $F \in \mathbf{P}_{T_1}$. By the assumption $G \in \mathbf{P}_{T_1 \to T_2}$, we have $G@!F \in \mathbf{P}_{T_2}$. Using induction hypothesis on this graph yields the graph $\widehat{G@!F} \in \mathbf{P}_{V_b \to T_2}$. Let $\tilde{E} \in \mathbf{P}_{V_b}$ is a graph obtained by renaming $E$. We can take a provisional context $\mathcal{P}''[\square]_{T_2}$ such that an execution on the graph $\mathcal{P}''[(\widehat{G@!F})@!\tilde{E}]$ yields a graph $\mathcal{P}''[(\widehat{G@!F})@!\tilde{E}']$ where the graph $\tilde{E}'$ is a renaming of the graph $E'$. By proceeding the execution, we obtain the graph $R'$, which consists of the graph $\phi(G)@!F$ and !-boxes connected to some outputs of $\phi(G)$, each of which contains the graph $\tilde{E}'$, a dot-product node and a vector node. Since $\widehat{G@!F} \in \mathbf{P}_{V_b \to T_2}$, the forward reasoning (Prop. G.4) ensures $R' \in \mathbf{P}_{T_2}$. Moreover, the graph $R'$ is in fact a renaming of the graph $R@!F$, therefore we have $R@!F \in \mathbf{P}_{T_2}$. □

Finally the soundness theorem, stated below, is obtained as a consequence of the so-called fundamental lemma of logical predicates.

**Theorem G.6** (Thm. 3.7)**.** *For any closed program $t$ such that $A \mid - \mid \vec{p} \vdash t : T$, there exist a graph $G$ and a token value $\kappa$ such that: $\mathit{Init}((A \mid - \mid \vec{p} \vdash t : T)^\ddagger) \to^* \mathit{Final}(G, \kappa)$.*

*Proof.* This is a corollary of Prop G.7 below. □

**Proposition G.7** (Fundamental lemma)**.** *For any derivable judgement $A \mid \Gamma \mid \vec{p} \vdash t : T$, where $\Gamma = x_0 : T_0, \ldots, x_{m-1} : T_{m-1}$, and any graphs $\vec{H} = H_0, \ldots, H_{m-1}$ such that $H_i \in \mathbf{P}_{T_i}$, if the following graph $G$ meets the name criteria and the graph criterion, it belongs to $\mathbf{P}_T$.*

$G =$ 

*Sketch of proof.* The first observation is that the translation $(A \mid \Gamma \mid \vec{p} \vdash t : T)^\dagger$ itself meets the name criterion and the graph criteria. Since $\vec{H} \in \mathbf{P}_\Gamma$, the whole graph again meets the graph criteria. We can always make the whole graph meet the name criteria as well, by renaming the graphs $\vec{H}$. Note that some names in the translation $(A \mid \Gamma \mid \vec{p} \vdash t : T)^\dagger$ are not bound or free, and turns free once we connect the graphs $\vec{H}$.

The proof is by induction on a type derivation, that goes in a similar way to a usual proof for the lambda calculus. To prove $G \in \mathbf{P}_T$, we look at an execution on the graph $G$ using the backward reasoning (Prop. G.4) and the congruence property (Cor. G.3). The only unconventional cases are: the fold operations fold $t\,u$, whose proof is by induction on the number of bases introduced in unfolding the operation; and graph abstraction $\mathsf{A}^{T'}_a(f, x).t$, whose proof relies on Prop. G.5. □

## H Graph-contextual equivalence

**Proposition H.1** (Prop. 5.4)**.** *Let $R$ be a U-simulation. If $\sigma_1 R \sigma_2$, then there exists a token value $\kappa$ 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 $G_2$.*

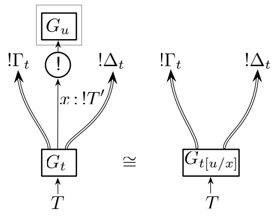
*Proof.* We first prove (i) implies (ii), by induction on the length of the sequence $\sigma_1 \to^* \mathit{Final}(G_1, \kappa)$. In the base case, where $\sigma_1 = \mathit{Final}(G_1, \kappa)$, the condition (II) in Def. 5.3 implies $\sigma_2 = \mathit{Final}(G_2, \kappa)$ for some composite graph $G_2$. The inductive case, where $\sigma_1 \to \sigma_1' \to^* \mathit{Final}(G_1, \kappa)$ for some state $\sigma_1'$, has two situations. The first situation is when the condition (I-i) in Def. 5.3 holds. We can use the induction hypothesis along the transitive closure $R^+$. The second situation is when the condition (I-ii) holds and there exists a sequence $\sigma_1' \to^+ \sigma_2$. Because of the determinism of final states (Prop. 3.5), there exists a sequence from the state $\sigma_2$ to the same final state $\mathit{Final}(G_1, \kappa)$.

Second, we prove (ii) implies (i), also by induction on the length of the sequence $\sigma_2 \to^* \mathit{Final}(G_2, \kappa)$. In the base case, where $\sigma_2 = \mathit{Final}(G_2, \kappa)$, the state $\sigma_1$ either reduces to the same final state (the condition (I-ii)) or is the final state itself (the condition (II)). The inductive case is where $\sigma_2 \to \sigma_2' \to^* \mathit{Final}(G_2, \kappa)$ for some state $\sigma_2'$. This implies the state $\sigma_2$ is not final, and therefore neither is the state $\sigma_1$ (by the condition (II)). Given a possible transition $\sigma_1 \to \sigma_1'$, the first situation is when there exists a state $\sigma_2''$ such that $\sigma_2 \to \sigma_2''$ and $\sigma'1\ R^+\ \sigma_2''$. The states $\sigma_2'$ and $\sigma_2''$ may not be the same, but thanks to the determinism (Prop. 3.5), they must result in the same final state $\mathit{Final}(G_2, \kappa)$. We can then use the induction hypothesis along the transitive closure $R^+$. The second situation is when $\sigma_1' \to^+ \sigma_2$ holds, and hence $\sigma_1 \to^* \mathit{Final}(G_2, \kappa)$ holds. □

## H.1 Beta equivalence

The proof of beta equivalence is via the so-called substitution lemma, stated below. Unlike normal substitution lemmas used for call-by-value evaluation, our version does not require the substitute to be a value. Instead, we require that the substitute is closed and without cells. Whether the current requirement can be relaxed is an open question.

**Lemma H.2** (Substitution lemma)**.** *For derivable judgements* $A \mid \Gamma, x : T', \Delta \mid \vec{p} \vdash t : T$ *and* $A' \mid - \mid - \vdash u : T'$*, if* $x \in \mathrm{FV}(t)$*, a judgement* $A \cup A' \mid \Gamma, \Delta \mid \vec{p} \vdash t[u/x] : T$ *is derivable, and the following graph-contextual equivalence holds:*



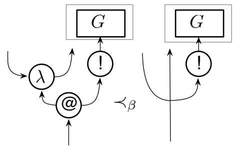
*where graphs* $G_t$*,* $G_u$ *and* $G_{t[u/x]}$ *are components of translations, and* $!\Gamma_t$ *and* $!\Delta_t$ *are restrictions of* $!\Gamma$ *and* $!\Delta$ *to* $\mathrm{FV}(t)$ *(see Fig. 8).*

*Sketch of proof.* Let three relations $\prec_D$, $\prec_C$ and $\prec_?$ on graphs be defined as in Fig. 12 where the graph $G(1,0)$ does not contain any links of cell type $\mathbb{iF}$. The first step is to show each of these three relations lifts to a U-simulation, and hence implies graph-contextual equivalence (Prop. 5.5).

The second step is by induction on the derivation $A \mid \Gamma, x : T', \Delta \mid \vec{p} \vdash t : T$. Because we assume $x \in \mathrm{FV}(t)$, the only base case is where $t$ is the variable $x$. This case can be proved solely by the relation $\prec_D$. Inductive cases can be shown by combination of the other relations, i.e. $\prec_C$ and $\prec_?$. □

**Proposition H.3** (Prop. 5.8)**.** *Let* $v$ *be a value defined by the grammar* $v ::= p \mid \lambda x^T.t \mid \mathsf{A}_a^T(f,x).t$*. 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.*

*Sketch of proof.* Let $\prec_\beta$ be a relation on graphs, defined by



such that: (i) the graph $G(1,0)$ does not contain any links of cell type $\mathbb{iF}$, and (ii) there exists a token value $\kappa$ such that the final state $\mathit{Final}(G,\kappa)$ on the graph $G(1,0)$ is stable (see Def. C.1). It is easy to see that the relation $\prec_\beta$ lifts to a U-simulation, with the help of the congruence property (Cor. E.3). Therefore the relation $\prec_\beta$ implies graph-contextual equivalence, by Prop. 5.5.

If the bound variable $x$ is not free in $t$, the proof of the contextual equivalence $A \mid \Gamma \mid \vec{p} \vdash (\lambda x^{T'}.t)\, v \approx t[v/x] : T$ boils down to the relation $\prec_\beta$ and the garbage collection shown in Prop. 5.7. Otherwise the proof is combination of the relation $\prec_\beta$ and the substitution lemma (Lem. H.2). Note that the component $G_v$ of the translation of the value $v$ indeed satisfies the condition (ii) of the relation $\prec_\beta$. □

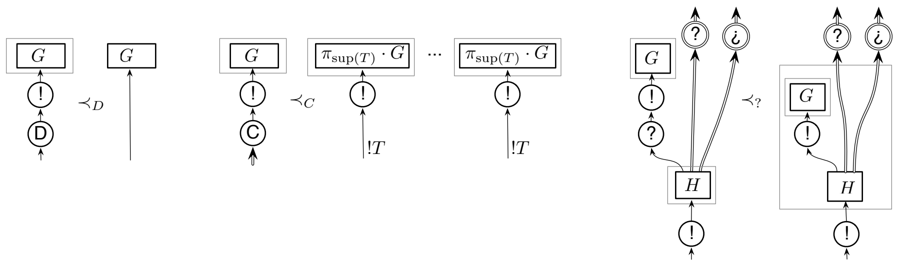


**Figure 12**