Evolving graph-structures and their implicit computational complexity

Daniel Leivant, Jean-Yves Marion

Evolving graph-structures and their implicit computational complexity

Daniel Leivant[1] and Jean-Yves Marion[2]

[1] Indiana University, USA, leivant@indiana.edu
[2] Université de Lorraine, LORIA, France, Jean-Yves.Marion@loria.fr

Abstract. Dynamic data-structures are ubiquitous in programming, and they use extensively underlying directed multi-graph structures, such as labeled trees, DAGs, and objects. This paper adapts well-established static analysis methods, namely data ramification and language-based information flow security, to programs over such graph structures. Our programs support the creation, deletion, and updates of both vertices and edges, and are related to pointer machines. The main result states that a function over graph-structures is computable in polynomial time iff it is computed by a terminating program whose graph manipulation is ramified, provided all edges that are both created and read in a loop have the same label.

1 Introduction

The interplay of algorithms and data-structures has been central to both theoretical and practical facets of programming. A core method of this relation is the organization of data-structures by underlying directed multi-graphs, such as trees, DAGs, and objects, where each vertex points to a record. Such data structures are often thought of as "dynamic", because they are manipulated by algorithms that modify the underlying graph, namely by creating, updating and removing vertices and edges. Our imperative language is inspired by pointer machines [6,10] and by abstract state machines [2].

In this work we propose a simple and effective static analysis method for guaranteeing the feasible time-complexity of programs over many dynamic data-structures. Most static analysis efforts have focused in recent years on program termination and on safety and security. Our work is thus a contribution to another strand of static analysis, namely computational complexity.

Static analysis of computational complexity is based on several methods, classified broadly into descriptive ones (i.e. related to Finite Model Theory), and applicative (i.e. identifying restrictions of programs and proof methods that guarantee upper bounds on the complexity of computation). One of the most fruitful applicative methods has been ramification, also referred to as tiering. Initially this method was used for inductive data, such as words and natural numbers, but lately the method has been applied to more general forms of data. The intuition behind ramification is that programs' execution time depends on the nature of information flow during execution, and that such flow can be constrained naturally and effectively by imposing a precedence relation regulating the information flow,

e.g. from higher tier to lower tier. Here we refer to a ramification that pertains to commands of our imperative language as well as to expressions denoting graph elements.

Our main result states that a function over graph-structures is computable in polynomial time iff it is computed by a terminating program whose graph manipulation is ramified, provided all edges that are both created and read in a loop have the same label. This result considerably extends previous uses of ramification in implicit computational complexity, and this extension touches on some of the most important aspects of programming, which have been disregarded in previous research. A simple modification of the proof gives an analogous characterization of Logspace computation over graph structures.[3] We thus believe that this work is of both theoretical and practical significance. Our results raise interesting questions about relations between data ramification and typed systems for program security [12,9], where the concept of information flow is explicit.

2 Evolving graph-structures

2.1 Sorted partial structures

The framework of sorted structures is natural for the graph-structures we wish to consider, with vertices and data treated as distinct sorts. Data might itself be sorted, but that does not concern us here. Recall that in a sorted structure, if $\mathcal{V}$ and $\mathcal{D}$ are sorts, then a function f is of type $\mathcal{V} \to \mathcal{D}$ if its domain consists of the structure elements of sort $\mathcal{V}$, and its range of elements of sort $\mathcal{D}$.

The graphs we consider are essentially deterministic transition graphs: edges are labeled (to which we refer as "actions"), and every vertex has at most one out-edge with a given label. Such graphs are conveniently represented by partial functions, corresponding to actions. An edge labeled $\mathbf{f}$ from vertex u to vertex v is represented by the equality $\mathbf{f}(u) = v$. When u has no $\mathbf{f}$-labeled out-edges, we leave $\mathbf{f}(u)$ undefined. To represent function partiality in the context of structures, we post a special constant **nil**, assumed to lie outside the sorts (or, equivalently, in a special singleton sort)[4]. We write $f : \mathcal{V} \rightharpoonup \mathcal{D}$, and say that f is a *partial function from sort $\mathcal{V}$ to sort $\mathcal{D}$,* if $f : \mathcal{V} \to (\mathcal{D} \cup \{\mathbf{nil}\})$. We write $\mathcal{V} \rightharpoonup \mathcal{D}$ for the type of such partial functions.

2.2 Graph structures

We consider sorted partial structures with a distinguished sort $\mathcal{V}$ of *vertices.* To account for the creation of new vertices, we also include a sort $\mathcal{R}$ of *reserved-vertices.* For simplicity, and without loss of generality, we assume only one additional sort $\mathcal{D}$, which we dub *data.* A *graph vocabulary* is a sorted vocabulary for these sorts, where we have just five sets of identifiers: $\mathbb{V}$ (the vertex constants), $\mathbb{D}$ (the data

[3] We also present extensions of our language to incorporate recursion.

[4] This is a minor, but deliberate, departure from the usual ontological (i.e. Church-style) typing of sorted structure.

constants), $\mathbb{F}$ (function-identifiers for labeled edges, of type $\mathcal{V} \rightharpoonup \mathcal{V}$), $\mathbb{G}$ (function-identifiers for data, of type $\mathcal{V} \to \mathcal{D}$), and $\mathbb{R}$ (the relation-identifiers), each of some type $\tau \times \cdots \times \tau$, where each τ is $\mathcal{V}$ or $\mathcal{D}$. As syntactic parameters we use $\mathbf{v} \in \mathbb{V}$, $\mathbf{d} \in \mathbb{D}$, $\mathbf{f} \in \mathbb{F}$, $\mathbf{g} \in \mathbb{G}$, and $\mathbf{R} \in \mathbb{R}$.

Given a sorted vocabulary Σ as above, a *Σ-structure* $\mathcal{S}$ consists of a vertex-universe $\mathcal{V}_\mathcal{S}$ which is finite, a potentially infinite reserve-universe $\mathcal{R}_\mathcal{S}$, a data-universe $\mathcal{D}_\mathcal{S}$, a distinct object $\perp$ to interpret **nil**, and a sort-correct interpretation $\mathbf{A}_\mathcal{S}$ for each Σ-identifier $\mathbf{A}$: $\mathbf{v}_\mathcal{S} \in \mathcal{V}_\mathcal{S}$; $\mathbf{d}_\mathcal{S} \in \mathcal{D}_\mathcal{S}$; $\mathbf{g}_\mathcal{S} \in [\mathcal{V}_\mathcal{S} \to \mathcal{D}_\mathcal{S}]$ (a *data-function*); $\mathbf{f}_\mathcal{S} \in [\mathcal{V}_\mathcal{S} \rightharpoonup \mathcal{V}_\mathcal{S}]$ (a *partial* function), and for a relation-identifier $\mathbf{R}$, a relation $\mathbf{R}_\mathcal{S} \subseteq \tau_\mathcal{S} \times \cdots \tau_\mathcal{S}$. Note that we do not refer to functions over data, nor to functions of arity > 1. Also, the fact that our graphs are edge-deterministic is reflected in our reprensetation of edges by functions.

Our graph structures bear similarity to the `struct` construct in the programming language C, and to objects (without behaviors or methods): a vertex identifies an object, and the state of that object is given by fields that are specified by the unary function identifiers. This is why Tarjan, in defining similar structures [11], talks about *records* and *items* rather than vertices and edges. The restriction of a graph-structure $\mathcal{S}$ to the sort $\mathcal{V}$ of vertices can be construed as a labeled directed multi-graph, in which there is an edge labeled by $\mathbf{f}$ from vertex u to vertex v exactly when $\mathbf{f}_\mathcal{S}(u) = v$. Thus the fan-out of each graph is bounded by the number of edge-labels in Σ. Examples of graph structures abound, see examples in Section 5. Linked-lists of data is an obvious one, of which words (represented as linked lists of alphabet-symbols) form a special case.

2.3 Expressions

Expressions are generated from a set $\mathbb{X}$ of vertex-variables, a set $\mathbb{Y}$ of data-variables, and the vocabulary identifiers, as follows. Equality here does not conform strictly to the sort discipline, in that we allow equations of the form $V = \mathbf{nil}$[5].

$$
\begin{array}{l}
V \in \mathrm{VExpr} ::= X \mid \mathbf{nil} \mid \mathbf{v} \mid \mathbf{f}(V) \quad \text{where } X \in \mathbb{X} \\
D \in \mathrm{DExpr} ::= Y \mid \mathbf{d} \mid \mathbf{g}(V) \quad \text{where } Y \in \mathbb{Y} \\
B \in \mathrm{BExpr} ::= V = V \mid D = D \mid \neg(B) \mid \mathbf{R}(E_1 \ldots E_n) \quad \text{where } \mathbf{R} : \tau^n,\ E_i : \tau
\end{array}
$$

3 Imperative programs over graph-structures

3.1 Programs

We refer to a skeletal imperative language, which supports pointers:

$$
\begin{array}{rl}
P \in \mathrm{Prg} ::= & X{:=}V \mid Y{:=}D \mid \mathbf{f}(X){:=}V \mid \mathbf{g}(X){:=}D \mid \mathbf{New}(X) \\
& \mid \mathbf{skip} \mid P; P \mid \mathbf{if}\,(B)\,\{P\}\,\{P\} \mid \mathbf{while}\,(B)\,\{P\}
\end{array}
$$

The boolean expression B of conditional and iterative commands is said to be their *guard*. We posit that each program is given with a finite set $\mathbb{X}_0 \subset \mathbb{X}$ of *input variables.*

[5] Negation is useful, however, in defining commands' semantics and dispensing with truth constants.

3.2 Example: Tarjan's Union Algorithm

The following graph-algorithm is due to Tarjan [11, p.21]. It refers to a representation of sets by linked-lists, whose initial vertex also serves as a name for the set. The linked-list is represented by a partial-function `next`, and the function `parent` maps each node to the head of its linked-list. The algorithm generates, given as input two lists r, q representing disjoint sets, a list representing their union. It successively inserts right after r's head the entries of q; thus r is maintained as the name of the union.

```
while (q ≠ nil) {save := next(q);
                 parent(q) := r;  % parent and next are modified
                 next(q) := next(r);
                 next(r) := q;
                 q := save}
```

We shall see that our tiering method admits the program above.

3.3 Evolving structures

In defining the semantics of an "uninterpreted" imperative program one refers to structures for that program's vocabulary, augmented with a store (i.e. environment, valuation). For programs over a Σ-structure $\mathcal{S}$, a *store* consists then of a function $\mu = \mu_X \cup \mu_Y$, where $\mu_X : \mathbb{X} \to \mathcal{V}_\mathcal{S} \cup \{\bot\}$, and $\mu_Y : \mathbb{Y} \to \mathcal{D}_\mathcal{S}$. A *$\Sigma$-configuration* is a pair consisting of a Σ-structure $\mathcal{S}$ and a store μ. We chose the phrase *configuration* to stress their dynamic nature, as computation stages of an evolving structure.

Commands of imperative languages are interpreted semantically as partial-functions that map an initial configuration to a final one. Here we have two commands whose intended semantic interpretation is to modify the structure itself. A command $\mathbf{New}(X)$ modifies the sorts, by moving an element of $\mathcal{R}_\mathcal{S}$ to $\mathcal{V}_\mathcal{S}$, and updating the store to have X point to the new element. We write $(\mathcal{S}, \mu)[\nu X]$ for the resulting configuration. Thus X points to a fresh vertex. More importantly, a command of the form $\mathbf{f}(X) := V$ modifies the semantics of the partial-function $\mathbf{f}_\mathcal{S}$, in that for $a = \mu(X)$ and w the value of V in $(\mathcal{S}, \mu)$ (defined formally below), an **f**-edge $v \rightsquigarrow u$ is replaced by $v \rightsquigarrow w$. We write $(\mathcal{S}, \mu)[\mathbf{f}(X) \leftarrow w]$ for the resulting interpretation.

Our structure updates are obviously related to Gurevich's abstract sequential machines (ASM) [2]. Gurevich divides identifiers into two classes: static identifiers, whose interpretation remains constant, and dynamic identifiers, whose interpretation may evolve during computation. Here the only dynamic identifiers are the edge functions. An ASM computation progresses through *"states"*, where every state is a structure. In contrast, we refer to configurations, because program variables play a central role in our imperative programs. Thus, the execution of our programs progresses through configuration.

Our programming language is related, more broadly, to pointer machines. Tarjan [11] defined a pure reference machine, consisting of an expandable collection of records and a finite number of registers. Pure reference machines are easily simulated by our programs, and vice versa.

Our programs are also related to Schönhage machines [10]. Each such machine consists of a finite control program combined with a dynamic structure (which is essentially the same as our graph-structures). Schönhage machines are an extension of Kolmogorov-Uspensky machines [6]. Another source of inspiration is the work of Jones & als. on blob model of computations [3].

3.4 Semantics of expressions

We give next the evaluation rules for Σ-expressions E in a Σ-configuration $(\mathcal{S}, \mu)$, writing $\mathcal{S}, \mu \models E \stackrel{e}{\Rightarrow} a$ to indicate that E evaluates to element a of $\mathcal{S}$.[6]

$$\frac{\mathbf{b} \in \mathbb{V} \cup \mathbb{D}}{\mathcal{S}, \mu \models \mathbf{b} \stackrel{e}{\Rightarrow} \mathbf{b}_{\mathcal{S}}} \qquad \frac{Z \in \mathbb{X} \cup \mathbb{Y}}{\mathcal{S}, \mu \models Z \stackrel{e}{\Rightarrow} \mu(Z)} \qquad \frac{\mathcal{S}, \mu \models E \stackrel{e}{\Rightarrow} a \qquad \mathbf{h} \in \mathbb{F} \cup \mathbb{G}}{\mathcal{S}, \mu \models \mathbf{h}(a) \stackrel{e}{\Rightarrow} \mathbf{h}_{\mathcal{S}}(a)}$$

$$\frac{\mathcal{S}, \mu \models E_i \stackrel{e}{\Rightarrow} a_i \quad \langle a_1..a_n \rangle \in \mathbf{R}_{\mathcal{S}}}{\mathcal{S}, \mu \models \mathbf{R}(E_1, \ldots, E_n)} \qquad \frac{\mathcal{S}, \mu \models E_i \stackrel{e}{\Rightarrow} a_i \quad \langle a_1..a_n \rangle \notin \mathbf{R}_{\mathcal{S}}}{\mathcal{S}, \mu \models \neg\mathbf{R}(E_1, \ldots, E_n)}$$

3.5 Semantics of programs

The semantics of programs is defined below:

$$\frac{\mathcal{S}, \mu \models E \stackrel{e}{\Rightarrow} a}{\mathcal{S}, \mu \models Z{:=}E \stackrel{s}{\Longrightarrow} \mathcal{S}, \mu[Z \leftarrow a] \models \mathbf{skip}} \qquad \frac{}{\mathcal{S}, \mu \models \mathbf{New}(X) \stackrel{s}{\Longrightarrow} (\mathcal{S}, \mu)[\nu X] \models \mathbf{skip}}$$

$$\frac{\mathcal{S}, \mu \models X \stackrel{e}{\Rightarrow} a \qquad \mathcal{S}, \mu \models V \stackrel{e}{\Rightarrow} b}{\mathcal{S}, \mu \models \mathbf{f}(X){:=}V \stackrel{s}{\Longrightarrow} \mathcal{S}, \mu[\mathbf{f}(a) := b] \models \mathbf{skip}}$$

$$\frac{\mathcal{S}, \mu \models P_1 \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models P_1'}{\mathcal{S}, \mu \models P_1; P_2 \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models P_1'; P_2} \qquad \frac{\mathcal{S}, \mu \models P_1 \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models \mathbf{skip}}{\mathcal{S}, \mu \models P_1; P_2 \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models P_2}$$

$$\frac{\mathcal{S}, \mu \models B}{\mathcal{S}, \mu \models \texttt{if } (B)\{P_0\}\{P_1\} \stackrel{s}{\Longrightarrow} \mathcal{S}, \mu \models P_0} \qquad \frac{\mathcal{S}, \mu \models \neg B}{\mathcal{S}, \mu \models \texttt{if } (B)\{P_0\}\{P_1\} \stackrel{s}{\Longrightarrow} \mathcal{S}, \mu \models P_1}$$

$$\frac{}{\mathcal{S}, \mu \models \mathbf{while}(B)\{P\} \stackrel{s}{\Longrightarrow} \mathcal{S}, \mu \models P; \mathbf{while}(B)\{P\}} \qquad \frac{}{\mathcal{S}, \mu \models \mathbf{while}(B)\{P\} \stackrel{s}{\Longrightarrow} \mathcal{S}, \mu \models \mathbf{skip}}$$

The phrase $\mathcal{S}, \mu \models P \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models P'$ conveys that evaluating a program P starting with configuration $(\mathcal{S}, \mu)$ is reduced to evaluating P' in configuration $(\mathcal{S}', \mu')$; i.e., P reduces to P' while updating $(\mathcal{S}, \mu)$ to $(\mathcal{S}', \mu')$.

An *initial configuration* is a configuration $(\mathcal{S}, \mu)$ where $\mu(X) = \mathbf{nil}$ for every non-input variable X. A program P *computes* the partial function $[\![P]\!]$ with initial configurations as input, defined by: $[\![P]\!](\mathcal{S}, \mu) = (\mathcal{S}', \xi)$ iff $\mathcal{S}, \mu \models P \ (\stackrel{s}{\Longrightarrow})^* \ \mathcal{T}, \xi \models \mathbf{skip}$.

[6] Here we consider equality as just another relation.

3.6 Run-time

We say that a program P *runs in time* t on input $(\mathcal{S},\mu)$, and write $\mathit{Time}_P(\mathcal{S},\mu) = t$, when $\mathcal{S},\mu \vDash P\ (\overset{\mathrm{s}}{\Longrightarrow})^t\ \mathcal{T},\xi \vDash \mathbf{skip}$ for some $(\mathcal{T},\xi)$. We take the size $|\mathcal{S},\mu|$ of a configuration $(\mathcal{S},\mu)$ to be the number n of elements in the vertex-universe V. Since the number of edges is bounded by n^2, we disregard them here. We also disregard the size of the data-universe, because our programs do not modify the data present in records. A program P is *running in polynomial time* if there is a $k > 0$ such that $\mathit{Time}_P(\mathcal{S},\mu) \leqslant k \cdot |\mathcal{S},\mu|^k$ for all configurations $(\mathcal{S},\mu)$,

4 Ramifiable programs

4.1 Tiering

Program tiering, also referred to as *ramification,* has been introduced in [7] and used in restricted form already in [1]. It serves to syntactically control the run-time of programs. Here we adapt tiering to graph-structures. The main challenge here is the evolution of structures in course of computation. To address it, we consider a finite lattice $\mathbb{T} = (T, \preceq, \mathbf{0}, \vee, \wedge)$, and refer to the elements of T as *tiers.* However, in order to simplify soundness proofs, and without loss of generality, we will focus on the boolean lattice $\mathbb{T} = (\{\mathbf{0},\mathbf{1}\}, \leqslant, \mathbf{0}, \vee, \wedge)$. We use lower case Greek letters α, β as discourse parameters for tiers.

Given $\mathbb{T}$, we consider $\mathbb{T}$-*environments* (Γ, Δ). Here Γ assigns a tier to each variable in $\mathbb{V}$, whereas Δ assigns to each function identifier $\mathbf{f} : \mathcal{V} \rightharpoonup \mathcal{V}$ one or several expressions of the form $\alpha \to \beta$, so that either

1. all types in $\Delta(\mathbf{f})$ are of the form $\alpha \to \alpha$, in which case we say that $\mathbf{f}$ is *stable* in the environment; or
2. all types in $\Delta(\mathbf{f})$ are of the form $\alpha \to \beta$, with $\beta \prec \alpha$, and we say that $\mathbf{f}$ is *reducing* in the environment.

A *tiering assertion* is a phrase of the form $\Gamma, \Delta \vdash V : \alpha$, where V is a vertex-expression and (Γ, Δ) a $\mathbb{T}$-environment. The correct tiering assertions are generated by the tiering system in Figure 1.

4.2 Ramifiable programs

Given a lattice $\mathbb{T}$, a program P is $\mathbb{T}$-*ramifiable* if there is a $\mathbb{T}$-environment (Γ, Δ) for which $\Gamma, \Delta \vdash P : \alpha$ for some α, and such that $\Gamma(X) = \mathbf{1}$ for every input variable $X \in \mathbb{X}_0$.[7] Thus, ramifiable programs can be construed as programs decorated with tiering information.

Lemma 1 (Subject reduction).
If $\mathcal{S},\mu \vDash P \ \overset{\mathrm{s}}{\Longrightarrow}\ \mathcal{S}',\mu' \vDash P'$ *and* $\Gamma, \Delta \vdash P : \alpha$ *then* $\Gamma, \Delta \vdash P' : \alpha$.

Lemma 2 (Type inference). *The problem of deciding, given a program* P *and a lattice* $\mathbb{T}$*, whether* P *is* $\mathbb{T}$*-ramifiable, is decidable in polynomial time.*

[7] Recall that each program is assumed given with a set $\mathbb{X}_0$ of input variables.

$$\frac{}{\Gamma, \Delta \vdash \mathbf{c} : \alpha} \qquad \frac{\Gamma(X) = \alpha}{\Gamma, \Delta \vdash X : \alpha} \qquad \frac{\alpha \to \beta \in \Delta(\mathbf{f}) \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash \mathbf{f}(V) : \beta}$$

$$\frac{\Gamma, \Delta \vdash V_i : \alpha}{\Gamma, \Delta \vdash \mathbf{R}(V_1, \ldots, V_n) : \alpha} \qquad \frac{\Gamma, \Delta \vdash V_i : \alpha}{\Gamma, \Delta \vdash V_0 = V_1 : \alpha}$$

Fig. 1. Tiering rules for vertex and boolean expressions

$$\frac{\Gamma, \Delta \vdash X : \alpha \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash X{:=}V : \alpha} \qquad \frac{\Gamma, \Delta \vdash \mathbf{f}(X) : \alpha \qquad \Gamma, \Delta \vdash V : \alpha}{\Gamma, \Delta \vdash \mathbf{f}(X){:=}V \ : \alpha}$$

$$\frac{\Gamma, \Delta \vdash X : \mathbf{0}}{\Gamma, \Delta \vdash \mathbf{New}(X) \ : \mathbf{0}} \qquad \frac{\Gamma, \Delta \vdash B : \alpha \qquad \Gamma, \Delta \vdash P : \alpha}{\Gamma, \Delta \vdash \mathbf{while}(B)\{P\} : \alpha} \, \mathbf{0} \prec \alpha$$

$$\frac{}{\Gamma, \Delta \vdash \mathbf{skip} : \mathbf{0}} \qquad \frac{\Gamma, \Delta \vdash P : \alpha \qquad \Gamma, \Delta \vdash P' : \beta}{\Gamma, \Delta \vdash P' \ ; \ P' : \alpha \vee \beta}$$

$$\frac{\Gamma, \Delta \vdash B : \alpha \quad \Gamma, \Delta \vdash P_i : \alpha}{\Gamma, \Delta \vdash \ \texttt{if} \ (B)\{P_0\}\{P_1\} \ : \alpha} \qquad \frac{\Gamma, \Delta \vdash P : \beta}{\Gamma, \Delta \vdash P : \alpha} \ (\beta \preceq \alpha)$$

Fig. 2. Tiering rules for programs

Proof. We associate with each vertex-variable X a "tier-variable" α_X, and with each function $\mathbf{f} \in \mathbb{F}$ two variables α_f and β_f, with the intent that $\alpha_f \to \beta_f$ is a possible tiering of $\mathbf{f}$. The typing rules for tiers give rise to a set of linear constraints on these tier-variables, a problem which is poly-time decidable.

4.3 Stationary and tightly-modifying loops

We say that a function identifier $\mathbf{f}$ is *probed* in P if it occurs in P either in some assignment $X := V$ or in the guard of a loop or a branching command. For example, $\mathbf{f}$ is probed in $X := \mathbf{f}(V)$, as well as in $\texttt{if} \ (\mathbf{f}(X) \neq \texttt{nil})\{P\}\{P'\}$. The identifier $\mathbf{f}$ is *modified* in P if it occurs in an assignment $\mathbf{f}(X) := V$ in P.

Fix a lattice $\mathbb{T}$, and a $\mathbb{T}$-environment (Γ, Δ). By the tiering rules, if a loop $\mathbf{while}(B)\{P\}$ is of tier α then $\Gamma, \Delta \vdash B : \alpha$. We say that the loop is *stationary* if no $\mathbf{f} \in \mathbb{F}$ of type $\alpha \to \alpha$ is modified therein. The loop above is *tightly-modifying* if it has modified function-identifiers of type $\alpha \to \alpha$, but at most one of those is also probed. In other words, all edges that are both created and read in a loop have the same label. For instance, in Example 3.2 above, **next** is both modified and probed, but **parent** is modified without being probed. Thus the loop, with its obvious tiering environment, is tightly-modifying.[8]

[8] Note that **next** and **parent** are of type $\mathbf{1} \to \mathbf{1}$, and all variables are of tier $\mathbf{1}$. Set union can be iterated because the result r is of tier $\mathbf{1}$, unlike in most other works.

4.4 Main Characterization Theorem

Given a lattice $\mathbb{T}$ and $\Gamma, \Delta \vdash P : \alpha$, we say that (Γ, Δ) is a *tight* ramification of P if Γ is an initial tiering, and each loop of P is stationary or tightly-modifying. We say that P is *tightly-ramifiable* if it has a tight $\mathbb{T}$-ramification (Γ, Δ), with Γ initial, for some non-trivial $\mathbb{T}$.

Theorem 1. *A function over graph-structures is computable in polynomial time iff it is computed by a terminating and tightly-ramifiable program.*

The Theorem follows from the Soundness Lemma 6 and the Completeness Proposition 1 below.

5 Examples of ramified programs

Tree insertion. The program below inserts the tree T into the binary search tree whose root is pointed-to by x. The input variables are x and T.

```
if (x¹ = nil)
  {x¹:=T:1;} % then clause
  { % else clause
   while ((x¹ ≠ nil) and (key(T¹) ≠ key(x¹)))
     {if (key(T¹) < key(x¹)) {p¹:=x¹; x¹ := left(x¹)¹}
                             {p¹:=x¹; x¹:=right(x¹)¹} }:1;
if (key(T¹) < key(p¹)) {left(p¹) := T¹:1}
                       {right(p¹):= T¹:1}
```

Note that neither **left** nor **right** is modified in the loop, so the loop is stationary.

Copying lists. Here we use **New** to copy a list, where the copy is in reverse order. Note that the source list is of tier **1** while the copy is of tier **0**.

```
y⁰ = nil : 0;
while (x¹ ≠ nil)
       { z⁰:=y⁰:0; New(y⁰); suc(y⁰):=z⁰:0;
         x¹ := suc(x¹):1 }:1
```

The loop is stationary, because the updated occurrence of **suc** is of type $\mathbf{0} \to \mathbf{0}$.

6 Soundness of programs for feasibility: run-time analysis

We show next that every tightly-ramified program computes a PTime function over configurations. The proof is based on the following observations, which we articulate more precisely below. First, if we start with a configuration where no vertex is assigned to variables of different tiers, then all configurations obtained in the course of computation have that property. We are thus assured that vertices can be ramified unambiguously.

The tiering rules imply that a program P of tier **0** cannot have loops, and is therefore evaluated in $\leqslant |P|$ steps. At the same time, the value of a variable

of tier **1** depends only on vertices of tier **1**. This implies, as we shall see, that the number of iterations of a given loop must be bounded by the number of possible configurations that may be generated by its body. Our restriction to tightly-modifying ramification guarantees that the number is polynomial.

6.1 Non-interference

Lemma 3 (Confinement). *Let (Γ, Δ) be an environment. If $\Gamma, \Delta \vdash P : \mathbf{0}$, then $\Gamma(X) = \mathbf{0}$ for every variable X assigned-to in P.*

The proof is a straightforward structural induction. Note also that a program P of tier **0** cannot have a loop, and is thus evaluated within $|P|$ steps.

We say that a vertex-tiering Γ is *compatible* with a store μ if $\Gamma(X) \neq \Gamma(X')$ implies $\mu(X) \neq \mu(X')$ for all $X, X' \in \mathbb{X}$. We say that Γ is an *initial tiering* if $\Gamma(X)$ is **1** for X initial (i.e. $X \in \mathbb{X}_0$), and **0** otherwise. Thus an initial tiering is always compatible with an initial store.

Lemma 4. *Suppose that* $\Gamma, \Delta \vdash P : \alpha$ *and* $\mathcal{S}, \mu \models P \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models P'$. *If μ is compatible with Γ then so is μ'.*

The proof is straightforward by structural induction on P.

We show next that tiering, when compatible with the initial configuration, guarantees the non-interference of lower-tiered values in the run-time of higher-tiered programs. A similar effect of tiering, albeit simpler, was observed already in [7]. This is also similar to the security-related properties considered in [12]. Non-interference can also be rendered algebraically, as in [8].

The (Γ, Δ)-*collapse* of a configuration $(\mathcal{S}, \mu)$ is the configuration $(\mathcal{S}_\Delta, \mu_\Gamma)$, where $\mu_\Gamma(X) = \mu(X)$ if $\Gamma(X) = \mathbf{1}$, and $\mu_\Gamma(X)$ is undefined otherwise; whereas $\mathcal{S}_\Delta$ is the structure identical to $\mathcal{S}$ except that each $\mathbf{f}$ for which $(\mathbf{1} \to \mathbf{1}) \notin \Delta(\mathbf{f})$ is interpreted as $\emptyset$. Thus $(\mathcal{S}_\Delta, \mu_\Gamma)$ disregards vertices that are not not reachable from some variable of tier **1** using edges of type $(\mathbf{1} \to \mathbf{1})$.

The next lemma states that a program's output vertices in tier **1** do not depend on vertices in tier **0**, nor on edges that do not have tier $\mathbf{1} \to \mathbf{1}$.

Lemma 5. *Suppose $\Gamma, \Delta \vdash P : \alpha$, and $\mathcal{S}, \mu \models P \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models P'$. There is a configuration $(\mathcal{S}'', \mu'')$ such that $\mathcal{S}_\Delta, \mu_\Gamma \models P \stackrel{s}{\Longrightarrow} \mathcal{S}'', \mu'' \models P'$, and $(\mathcal{S}''_\Delta, \mu''_\Gamma) = (\mathcal{S}'_\Delta, \mu'_\Gamma)$.*

The proof is straightforward by structural induction on programs.

6.2 Polynomial bounds

(Soundness)

Lemma 6. *Assume that $\Gamma, \Delta \vdash P : \alpha$, where P is tightly-modifying. There is a $k > 0$ such that for every graph-structure $\mathcal{S}$ and every store μ compatible with Γ, if $\mathcal{S}, \mu \models P \stackrel{s}{\Longrightarrow} \mathcal{S}', \mu' \models P'$, then $\mathcal{S}, \mu \models P$ $(\stackrel{s}{\Longrightarrow})^t$ $\mathcal{S}', \mu' \models P'$ for some $t < k + |\mathcal{S}|^k$.*

In proving Lemma 6 we will use the following combinatorial observation. Let $\mathcal{G}$ be a digraph of out-degree 1. We say that a set of vertices C *generates* $\mathcal{G}$ if every vertex in $\mathcal{G}$ is reachable by a path starting at C. The following lemma provides a polynomial, albeit crude, upper bound on the number of digraphs with k generators.

Lemma 7. *The number (up to isomorphism) of digraphs with n vertices, and a generator of size k, is $\leqslant n^{2k^2}$.*

Proof. A *connected* digraph of out-degree 1 must consist of a loop of vertices, with incoming linear spikes. If there are just k generators, then there are at most k such spikes. There are at most k entry points on the loop to choose for these spikes, and each spike is of size $\leqslant n$. So there are at most $n^k \times n^k$ non-isomorphic connected graphs with a generator of size k.

Also, with only k generating vertices we can have at most k connected components. In sum there are at most $(n^{2k})^k = n^{2k^2}$ non-isomorphic graphs of size n with k generators.

Proof of Lemma 6. We proceed by structural induction on P. The only non-trivial observations are as follows. For program composition, we use the Compatibility Lemma 4.

The crucial case is, of course, where P is of the form **while**$(B)Q$. Say $X_1, \ldots, X_m$ are the vertex-variables in B. The tiering rules require that B, and therefore $X_1, \ldots, X_m$, are all of tier **1**.

If Q updates only edges that are not probed in P (including the guard B), then neither the execution of Q nor the evaluation of B is affected, that is all configurations in the computation have the same vertices of tier **1**, with no change in edges that affect the execution of P, by Lemma 5. Thus the truth of B in each invocation of Q is determined by the combinations of values assigned to $X_1, \ldots, X_m$, while the structural changes caused by Q do not affect the execution of Q in subsequent invocations. Since we assume that P terminates, it follows that the combinations of values for $X_1, \ldots, X_m$ must be all different. If n is the number of vertices of tier **1**, then $n \leqslant |\mathcal{S}|$, and there are n^m such combinations. By IH Q terminates in PTime, and therefore so does P.

Suppose Q does update edges that are probed in P. Since P is assumed tightly-modifying, all such updates are for the same $\mathbf{f} \in \mathbb{F}$. Let C be the set of initial values of variables occurring in P (including $X_1 \ldots X_m$ and possibly others). Let U be the set of vertices reachable from C by some path of tier **1** (i.e. using edges labels by $\mathbf{h} \in \mathbb{F}$ assigned $\mathbf{1} \to \mathbf{1}$ by $\Delta$from C vertices). By Lemma 5, the execution of P, including all iterated invocations of B and Q, has only vertices in U as value of tier **1**. Moreover, U is generated by C, whose size is fixed by the syntax of P. It follows, by Lemma 7, that the number of such configurations is polynomial in the size of U, which in turns in bounded by the size $|\mathcal{S}|$ of the vertex-universe of the structure $\mathcal{S}$. □

7 Completeness of tightly-ramifiable programs for feasibility

Proposition 1. *Every polynomial-time function on graph-structures is computable by a terminating and tightly-ramifiable program.*

Proof. Suppose f is a unary function on graph-structures, which is computed by a Turing machine M over alphabet Σ, modulo some canonical coding of graph structures by strings. Assume that M operates in time $k \cdot n^k$. We posit that M uses a read-only input tape, and a work-tape. We simulate M by a tightly-ramifiable program $\Gamma, \Delta \vdash P : \mathbf{1}$ over graph-structures, as follows. The structures considered simulate each of the two tapes by a double-linked list of records, with the three fields **left**, **right**, and **val**, returning two pointers and an alphabet letter, respectively. We take our data constants to include each $\sigma \in \Sigma$. The input-tape is assigned tier **1**, and the work-tape tier **0**. Initially the work-tape is empty, and new cells are progressively created by using the command **New** at tier **0**. The machine's yield-relation between configurations is simulated at tier **0** by nested conditionals. Finally, we include in our simulation a clock, consisting of k nested loops, as in the following nesting of two loops:

```
u¹ := head¹:1 // head is a pointer to the input tape;
while (u¹ ≠ nil)
              { v¹:=head¹:1;
                 while (v¹ ≠ nil)
                  { v¹ := suc(v¹):1;
                    Transition function is here:0 }:1
                    u¹ := suc(u¹):1 } :1
```

8 Characterization of log-space languages

Hofmann and Schöpp [4] introduced pure pointer programs based on a uniform iterator **forall**, and related them to computation in logarithmic space. Our programs differ from these in supporting modification of the structure, in the guises of vertex creation and edge displacement and deletion. Moreover, our programs are based on a looping construct that treats individually each vertex of the structure.

These differences notwithstanding, our characterization of PTime can be modified to a characterization of log-space, by restricting the syntax of ramifiable-programs. Say that a program P over graph-structures is a *jumping-program* if it uses no edge update. Jones [5] showed that a simple cons-free imperative programming language WHILE$^{\setminus Ro}$ accepts precisely the languages decidable by Turing machines in logarithmic space. Since our jumping-programs can be rephrased in WHILE$^{\setminus Ro}$ they too accept only log-space languages (where input strings are represented as linked lists). Conversely, the store used by a jumping-program is essentially of size $k \cdot log(n) + log(Q)$, where n is the size of the vertex-universe, Q the size of the data-universe, and k the number of variables. Consequently, we have:

Theorem 2. *A language is accepted by a jumping-program iff it is decidable in* LOGSPACE.

9 Adding recursion

It is not hard to augment our programing language with recursion. Here is a procedure that recursively searches for a path from vertex v to w.[9]

```
Proc search(v¹,w¹))
        {if (v=w)¹ return true:1;
        visited(v) = true:1;
        forall t¹ in AdjList(v) % List of adjacency nodes of t
         {if (visited(t)¹=false)
            if (search(t,w)¹=true) return true:1;}
        return false:1;}
```

A restricted form of recursion is *linear recursion,* where at most one recursive call is allowed in the definition of a recursive procedure. Moreover, we suppose that each function body is stationary or tightly-modifying.

Theorem 3. *On its domain of computation, a tightly-ramifiable program with linear recursive calls is computable in polynomial time.*

References

1. S. Bellantoni and S. A. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.
2. Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000.
3. L. Hartmann, N. D. Jones, J. G. Simonsen, and S. B. Vrist. Programming in biomolecular computation: Programs, self-interpretation and visualisation. *Sci. Ann. Comp. Sci.*, 21(1):73–106, 2011.
4. M. Hofmann and U. Schöpp. Pure pointer programs with iteration. *ACM Trans. Comput. Log.*, 11(4), 2010.
5. N. D. Jones. Logspace and ptime characterized by programming languages. *Theor. Comput. Sci.*, 228(1-2):151–174, 1999.
6. A.N. Kolmogorov and V. Uspensky. On the definition of an algorithm. *Uspekhi Mat. Naut*, 13(4), 1958.
7. D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*. Birkhauser-Boston, 1994.
8. J.-Y. Marion. A type system for complexity flow analysis. In *LICS*, 2011.
9. A. Sabelfeld and D. Sands. Declassification: dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.
10. A. Schönhage. Storage modification machines. *SIAM J. Comp.*, 9(3):490–508, 1980.
11. R. E. Tarjan. Reference machines require non-linear time to maintain disjoint sets. In *STOC'77*, pages 18–29. ACM, 1977.
12. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[9] The construct `forall` X `in` $R(u)$, which is "blind," in the sense that it does not depend on node ordering. As a result, no function identifier is probed except **visited**.