

Kent Academic Repository

Full text document (pdf)

Citation for published version

Luo, Yong and Chitil, Olaf (2007) Algorithmic Debugging for Locally Defined Functions. Technical report. University of Kent, Canterbury, Canterbury

DOI

Link to record in KAR

<https://kar.kent.ac.uk/14558/>

Document Version

UNSPECIFIED

Copyright & reuse

Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research

The version in the Kent Academic Repository may differ from the final published version.

Users are advised to check <http://kar.kent.ac.uk> for the status of the paper. **Users should always cite the published version of record.**

Enquiries

For any further enquiries regarding the licence status of this document, please contact:

researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at <http://kar.kent.ac.uk/contact.html>

Computer Science at Kent

Algorithmic Debugging for Locally Defined Functions

Yong Luo and Olaf Chitil

Technical Report No. 8 - 07
August 2007

Copyright © 2007 University of Kent
Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

Algorithmic Debugging for Locally Defined Functions

Yong Luo and Olaf Chitil

Computing Laboratory, University of Kent

Abstract The purpose of the document is to prove the correctness of Algorithmic Debugging where the traces for local functions are generated in a new way. The processes of generating computation graphs follow exactly what we might do by hand. Therefore, we can be confident that the graphs are correct. We do not need to justify the graphs by comparing λ -lifted programs.

1 Basic Definitions

In this section we give some basic definitions.

Definition 1. (*Nodes, Atoms*)

- A *node* is a sequence of letters r , f and a , i.e. $\{r, f, a\}^*$.
- *Atoms*:
 1. a constructor is an atom;
 2. a function symbol is an atom;
 3. a node combined with a function symbol is an atom. For example, $m.f$ is an atom where m is a node and f is a function symbol.

Notation: In the future, we shall say that g is a function if g is an atom but not a constructor.

Definition 2. (*Terms, Patterns, Rewriting rule and Program*)

- *Terms*:
 1. an atom is a term;
 2. a node is a term;
 3. a variable is a term;
 4. (Application) MN is a term if M and N are terms.
- *Patterns*:
 1. a variable is a pattern;
 2. $cp_1\dots p_n$ is a pattern if c is a constructor and p_1, \dots, p_n are patterns, and the arity of c is n .

- A **simple rewriting rule** is of the form $f p_1 \dots p_n = R$ where f is a function and p_1, \dots, p_n ($n \geq 0$) are patterns and R is a term.
- A **rewriting rule** is in one of two forms:
 1. (top-level functions without local functions) a simple rewriting rule.
 2. (top-level functions with local functions) the form

$$\begin{aligned}
 f p_1 \dots p_n = R \\
 \text{where } g_1 q_{1_1} \dots q_{m_1} = R_1 \\
 \dots \\
 g_k q_{1_k} \dots q_{m_k} = R_k
 \end{aligned}$$

where $f p_1 \dots p_n = R$ and $g_j q_{1_j} \dots q_{m_j} = R_j$ are simple rewriting rules. g_1, \dots, g_k are the local functions of f .

- A **program** is a finite set of rewriting rules.

If a simple rewriting rule is of the form $f = R$ we call it a constant rewriting rule and f is a constant.

Definition 3. (Node expression and Computation graph)

- A **node expression** is either
 - an atom, or
 - a node, or
 - an application of two nodes, which is of the form $m \circ n$.
- A **computation graph** is a set of pairs which are of the form (n, e) , where n is a node and e is a node expression.

Notation: $dom(G)$ denotes the set of nodes in a computation graph G .

2 Pattern matching

The pattern matching algorithm for a graph has two different results, either a set of substitutions or “doesn’t match”.

- Let G be a computation graph, and $m \in dom(G)$. The final node in a sequence of reductions starting at m , $last_G(m)$:

$$last_G(m) = \begin{cases} last_G(mr) & \text{if } mr \in dom(G) \\ last_G(n) & \text{if } (m, n) \in G \text{ and } n \text{ is a node} \\ m & \text{otherwise} \end{cases}$$

The purpose of this function is to find out the most evaluated point for m .

- Let G be a computation graph, and $m \in \text{dom}(G)$. The head of the term at m , $\text{head}_G(m)$:

$$\text{head}_G(m) = \begin{cases} \text{head}_G(\text{last}_G(i)) & \text{if } (m, i \circ j) \in G \\ a & \text{if } (m, a) \in G \text{ and } a \text{ is an atom} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Let G be a computation graph, and $m \in \text{dom}(G)$. The arguments of the function at m , $\text{args}_G(m)$, is defined as follows.

$$\text{args}_G(m) = \begin{cases} \langle \text{args}_G(\text{last}_G(i)), j \rangle & \text{if } (m, i \circ j) \in G \\ \langle \rangle & \text{otherwise} \end{cases}$$

Note that the arguments of a function are a sequence of nodes.

Now, we define two functions match_1 and match_2 which are mutually recursive. The arguments of match_1 are a node and a pattern. The arguments of match_2 are a sequence of nodes and a sequence of patterns.

- match_1 :

$$\begin{aligned} \text{match}_{1G}(m, x) &= [m/x] \text{ where } x \text{ is a variable} \\ \text{match}_{1G}(m, cq_1 \dots q_k) &= \begin{cases} \text{match}_{2G}(\text{args}_G(m), \langle q_1, \dots, q_k \rangle) & \text{if } \text{head}_G(m) = c \\ \text{does not match} & \text{otherwise} \end{cases} \end{aligned}$$

where $m' = \text{last}_G(m)$.

- match_2 :

$$\begin{aligned} \text{match}_{2G}(\langle m_1, \dots, m_n \rangle, \langle p_1, \dots, p_n \rangle) &= \text{match}_{1G}(m_1, p_1) \cup \dots \cup \text{match}_{1G}(m_n, p_n) \end{aligned}$$

where \cup is the union operator. Notice that if $n = 0$ then

$$\text{match}_{2G}(\langle \rangle, \langle \rangle) = []$$

If any m_i does not match p_i , $\langle m_1, \dots, m_n \rangle$ does not match $\langle p_1, \dots, p_n \rangle$.

If the length of two sequences are not the same, they do not match.

For example, $\langle m_1, \dots, m_s \rangle$ does not match $\langle p_1, \dots, p_{s'} \rangle$ if $s \neq s'$.

- We say that G at m matches the left-hand side $f p_1 \dots p_n$ of a rewriting rule with $[m_1/x_1, \dots, m_k/x_k]$ if $\text{head}_G(m) = f$ and

$$\text{match}_{2G}(\text{args}_G(m), \langle p_1, \dots, p_n \rangle) = [m_1/x_1, \dots, m_k/x_k]$$

In the substitution form $[m/x]$, m is not a term but a node. The definition of pattern matching and its result substitution sequence will become important for making computation order irrelevant when we generate graphs.

3 Renaming and Program for local functions

Suppose that G at m matches the left-hand side of a rewriting rule with $[m_1/x_1, \dots, m_l/x_l]$, and the rewriting rule has local functions as follows.

$$\begin{aligned} f p_1 \dots p_n &= R \\ \text{where } g_1 q_{1_1} \dots q_{r_1} &= R_1 \\ &\dots \\ g_k q_{1_k} \dots q_{r_k} &= R_k \end{aligned}$$

We generate a set of new simple rewriting rules, L_m , called local functions at m . All the local functions g_1, \dots, g_k in the local rewriting rules are renamed to $m.g_1, \dots, m.g_l$, and all the free variable in R_1, \dots, R_k are substituted by $[m_1/x_1, \dots, m_l/x_l]$. Then, L_m looks like the following:

$$\begin{aligned} m.g_1 q_{1_1} \dots q_{r_1} &= R'_1 \\ &\dots \dots \\ m.g_k q_{1_k} \dots q_{r_k} &= R'_k \end{aligned}$$

4 ART

The function *graph* is defined as follows.

Definition 4. (*graph*) Let G be a computation graph, and $m \in \text{dom}(G)$. The function *graph* takes two arguments. The first argument is a node and the second is a term.

$$\begin{aligned} \text{graph}(n, e) &= \{(n, e)\} \quad \text{where } e \text{ is an atom or a node} \\ \text{graph}(n, MN) &= \begin{cases} \{(n, M \circ N)\} & \text{if } M \text{ and } N \text{ are nodes} \\ \{(n, M \circ na)\} \cup \text{graph}(na, N) & \text{if only } M \text{ is a node} \\ \{(n, nf \circ N)\} \cup \text{graph}(nf, M) & \text{if only } N \text{ is a node} \\ \{(n, nf \circ na)\} \cup \text{graph}(nf, M) & \text{otherwise} \\ \cup \text{graph}(na, N) \end{cases} \end{aligned}$$

Generate an ART

- For a starting term M , the starting ART is $\text{graph}(r, M)$. Note that the start term has no nodes inside.
- (**ART rule 1**) If an ART G at m matches the left-hand side of a simple rewriting rule $f p_1 \dots p_n = R$ in the program L with $[m_1/x_1, \dots, m_l/x_l]$, then we generate a new ART.

$$G \cup \text{graph}(mr, R[m_1/x_1, \dots, m_l/x_l])$$

- (**ART rule 2**) If an ART G at m matches the left-hand side of a rewriting rule in the program L with $[m_1/x_1, \dots, m_l/x_l]$, and the rewriting rule has local functions as follows.

$$\begin{aligned}
 f \ p_1 \dots p_n &= R \\
 \text{where } g_1 \ q_{1_1} \dots q_{r_1} &= R_1 \\
 &\dots\dots \\
 g_k \ q_{1_k} \dots q_{r_k} &= R_k
 \end{aligned}$$

Then we generate a set of new rewriting rules L_m and a new ART.

$$G \cup \text{graph}(mr, R'[m_1/x_1, \dots, m_l/x_l])$$

where R' is obtained from R by renaming all the local functions in R .

- (**ART rule 3**) If an ART G at m matches the left-hand side of a simple rewriting rule $(s.f)p_1 \dots p_n = R$ in the program L_s with $[m_1/x_1, \dots, m_k/x_k]$, then we generate a new ART.

$$G \cup \text{graph}(mr, R[m_1/x_1, \dots, m_k/x_k])$$

- An ART is generated from the starting ART and by applying the *ART rules* repeatedly. Note that the order in which nodes are chosen to compute has no influence in the final graph.

The following simple properties of an ART will be used later.

Lemma 1. *Let G be an ART.*

- If $m \in \text{dom}(G)$ then there is at least one letter r in m .
- If $mr \in \text{dom}(G)$ then $m \in \text{dom}(G)$ or $m = \varepsilon$ where ε is the empty sequence.
- If $mr \in \text{dom}(G)$ then $(m, n) \notin G$ for any node n .

Proof. The first and second are trivial. The third is proved by contradiction. If $(m, n) \in G$ then $\text{head}_G(m)$ is undefined. There cannot be a computation at m , i.e. $mr \notin G$.

5 EDT

Generating an Evaluation Dependency Tree

Definition 5. (*Parent edges*)

$$\begin{aligned}
 \text{parent}(nf) &= \text{parent}(n) \\
 \text{parent}(na) &= \text{parent}(n) \\
 \text{parent}(nr) &= n
 \end{aligned}$$

Note that $parent(r) = \varepsilon$ where ε is the empty sequence.

Definition 6. (*children and tree*) Let G be an ART, and mr a node in G (i.e. $mr \in dom(G)$). *children* and *tree* are defined as follows.

- *children*:

$$children(m) = \{n \mid nr \in dom(G) \text{ and } parent_G(n) = m\}$$

- *tree*:

$$tree(m) = \{(m, n_1), \dots, (m, n_k)\} \cup tree(n_1) \cup \dots \cup tree(n_k)$$

$$\text{where } \{n_1, \dots, n_k\} = children(m)$$

Usually, a single node of a computation graph represents many different terms. We are particularly interested in two kinds of terms of nodes, the most evaluated form and the redex.

Definition 7. (*Most Evaluated Form*) Let G be an ART. The *most evaluated form* of a node m is a term and is defined as follows.

$$mef(m) = \begin{cases} mef(mr) & \text{if } mr \in dom(G) \\ meft(m) & \text{otherwise} \end{cases}$$

where

$$meft(m) = \begin{cases} a & (m, a) \in G \text{ and } a \text{ is an atom} \\ mef(n) & (m, n) \in G \text{ and } n \text{ is a node} \\ mef(i) mef(j) & (m, i \circ j) \in G \end{cases}$$

One may also use the definition of $last_G(m)$ to define the most evaluated form.

Definition 8. (*redex*) Let G be an ART, and mr a node in G (i.e. $mr \in dom(G)$). *redex* is defined as follows.

- $redex(\varepsilon) = main$
- $redex(m) = \begin{cases} mef(i) mef(j) & \text{if } (m, i \circ j) \in G \\ a & \text{if } (m, a) \in G \text{ and } a \text{ is an atom} \end{cases}$

Generate an EDT

Now, we define the evaluation dependency tree of a graph.

Definition 9. (Evaluation Dependency Tree) Let G be an ART. The evaluation dependency tree (EDT) of G consists of the following two parts.

1. The set $tree(\varepsilon)$;
2. The set of equations; for any node m in $tree(\varepsilon)$ there is a corresponding equation. If $head_G(m)$ is a top-level function, the the equation at m is of the form

$$redex(m) = mef(m)$$

If $head_G(m)$ is a local function of the form $n.f$, then the equation at m is of the form

$$\begin{aligned} redex(m) &= mef(m) \\ &\text{within } redex(n) \end{aligned}$$

6 Proofs

Some of the definitions and proofs are as the same before but some are new. The old proofs still need to be checked again because the basic definitions such as rewriting rules and EDT are changed.

The following theorems suggest that the EDT of an ART covers all the computation in the ART. Although two evaluations may rely on the same evaluation in an ART, every evaluation for algorithmic debugging only needs to be examined once.

Lemma 2. Let G be an ART, and T its EDT. If there is a sequence of nodes m_1, m_2, \dots, m_k such that

$$\begin{aligned} m &\in children(m_1), m_1 \in children(m_2), \dots, \\ m_{k-1} &\in children(m_k), m_k \in children(\varepsilon) \end{aligned}$$

then $m \in dom(T)$.

Proof. By the definition of $tree(\varepsilon)$.

Lemma 3. Let G be an ART. If $mx \in dom(G)$, then $m \equiv \varepsilon$ or there is a sequence of nodes m_1, m_2, \dots, m_k such that

$$\begin{aligned} m &\in children(m_1), m_1 \in children(m_2), \dots, \\ m_{k-1} &\in children(m_k), m_k \in children(\varepsilon) \end{aligned}$$

Proof. By induction on the size of m , and by Lemma 1.

Since $mr \in \text{dom}(G)$, by Lemma 1, we only need to consider the following two cases.

- If $m = \varepsilon$, the statement is obviously true.
- If $m \in \text{dom}(G)$, by Lemma 1, there is at least one letter r in m . We consider the following two sub-cases.
 - $m = rn$, where there is no r in n . Since $mr \in \text{dom}(G)$ and $\text{parent}(rn) = \varepsilon$, we have $rn \in \text{children}(\varepsilon)$.
 - $m \equiv m_1rn$, where there is no r in n . Since $mr \in \text{dom}(G)$ and $\text{parent}(m) = m_1$, we have $m \in \text{children}(m_1)$. Now, because m_1 is a sub-sequence of m , by induction hypothesis, there is a sequence of index numbers m_2, \dots, m_k such that

$$m_1 \in \text{children}(m_2), \dots, m_{k-1} \in \text{children}(m_k), m_k \in \text{children}(\varepsilon)$$

So, there is a sequence of index numbers m_1, m_2, \dots, m_k such that

$$m \in \text{children}(m_1), m_1 \in \text{children}(m_2), \dots, m_k \in \text{children}(\varepsilon)$$

Theorem 1. *Let G be an ART, and T its EDT.*

If $mr \in \text{dom}(G)$, then $m \in \text{dom}(T)$. In other word, T covers all the computations in G .

Proof. By Lemma 3 and 2.

Lemma 4. *Let G be an ART, and T its EDT.*

If $(m, n) \in T$, then $n \in \text{children}(m)$ and $\text{parent}(n) \equiv m$.

Proof. By the definition of *tree*.

Theorem 2. *Let G be an ART, and T its EDT.*

If $(m, n) \in T$ and $m \neq k$, then $(k, n) \notin T$.

Proof. By Lemma 4.

The above theorem suggests that every evaluation for algorithmic debugging only needs to be examined once although two evaluations may rely on the same evaluation. For example, g is defined as $g\ x = (\text{not } x, \text{not } x, \text{not } x)$. When we compute g (*not True*), the equation *not True = False* only appears once in the EDT.

6.1 Semantical Equality

Notations: $M \simeq_I N$ means that M is equal to N with respect to the semantics of the programmer's intention. If the evaluation $M = N$ of a node in an EDT is in the programmer's intended semantics, then $M \simeq_I N$. Otherwise, $M \not\simeq_I N$ i.e. the node is erroneous.

Remark 1. For a local function, the evaluation of a node in an EDT is of the form

$$(m.g)b_1, \dots, b_n = N \\ \text{within } fe_1, \dots, e_k$$

The “within” part helps the programmer to decide whether the evaluation is intended or not, but it will not be used in proofs. We keep the prefix m in order to make semantics of local functions clear. If both m and “within” are removed, we might have $gb_1, \dots, b_n = N$ and $gb_1, \dots, b_n = N'$ where N and N' are different. In practice, one may chose different ways to help the programmer to answer such questions.

6.2 Equivalent rewriting rules

Two kind of rewriting rules, top-level and local-level, are used during the processes of building trace. For a top level rewriting rule, there is no node in the right-hand side. However, for a local rewriting rule of the form $(m.f)p_1 \dots p_n = R$, it is possible that there are nodes in R . When the computation stops and we start to analysis the properties, we regard that the rewriting rule is equivalent to $(m.f)p_1 \dots p_n = R'$ where R' is obtained from R by replacing all the nodes by their most evaluated forms. For example, the nodes in R are m_1, \dots, m_k , then

$$R' \equiv R[mef(m_1)/m_1, \dots, mef(m_k)/m_k]$$

For a top-level rewriting rule $fp_1 \dots p_n = R$, if it used at node m and there are local functions in R , we regard that the rewriting rule is equivalent to $fp_1 \dots p_n = R'$ where R' is obtained from R by renaming all the local functions in R .

6.3 Program faulty

Definition 10. (Program faulty)

- For a simple rewriting rule $fp_1 \dots p_n = R$ without local functions, if there exists a substitution σ such that $(fp_1 \dots p_n)\sigma \not\simeq_I R\sigma$, then we say that the definition of the function f in the program is faulty.

- For a rewriting rule without local functions of the following form

$$\begin{aligned}
f p_1 \dots p_n &= R \\
\text{where } g_1 q_{1_1} \dots q_{m_1} &= R_1 \\
&\dots\dots \\
g_k q_{1_k} \dots q_{m_k} &= R_k
\end{aligned}$$

If there exists a substitution σ such that $(fp_1 \dots p_n)\sigma \not\approx_I (R \text{ within } fp_1 \dots p_n)\sigma$, then we say that the definition of the function f in the program is faulty. If there exists σ and σ' such that $\{(g_i q_{1_i} \dots q_{m_i})\sigma' \not\approx_I R_i \sigma'\}$ within $(fp_1 \dots p_n)\sigma$, then we say that the definition of the local function g_i within f is faulty.

6.4 Correctness of Algorithmic Debugging

The proofs are the same as before, but we should check them again because the definition of program faulty is changed.

Definition 11. *If the following statement is true, then we say that algorithmic debugging is correct.*

- If the equation of a faulty node is $fb_1 \dots b_n = M$, then the definition of the function f in the program is faulty.

For a faulty node m , we have $redex(m) \not\approx_I mef(m)$. We shall find a term N and prove $redex(m) \rightarrow_P N \simeq_I mef(m)$. In order to define N , we need other definitions.

Definition 12. *Let G be an ART and m a node in G . $reduct(m)$ is defined as follows.*

$$reduct(m) = \begin{cases} a & \text{if } (m, a) \in G \text{ and } a \text{ is an atom} \\ mef(n) & \text{if } (m, n) \in G \text{ and } n \text{ is a node} \\ reduct(mf) \ reduct(ma) & \text{if } (m, mf \circ ma) \in G \\ reduct(mf) \ mef(j) & \text{if } (m, mf \circ j) \in G \text{ and } j \neq ma \\ mef(i) \ reduct(ma) & \text{if } (m, i \circ ma) \in G \text{ and } i \neq mf \\ mef(i) \ mef(j) & \text{if } (m, i \circ j) \in G \text{ and } i \neq mf \text{ and } j \neq ma \end{cases}$$

$reduct$ represents the result of a single-step computation. And we shall prove $redex(m) \rightarrow_P reduct(mr) \simeq_I mef(m)$ for a faulty node m . Note that $mef(m) = mef(mr)$ and so we want to prove $reduct(mr) \simeq_I mef(mr)$. In order to prove this, we prove a more general result $reduct(m) \simeq_I mef(m)$ for all $m \in dom(G)$ (see Lemma 6 for the conditions).

We define *branch* and the reduction principle *depth* in order to prove this general result.

Definition 13. (*branch and branch'*) We say that n is a branch node of m , denoted as $\text{branch}(n, m)$, if one of the following holds.

- $\text{branch}(m, m)$;
- $\text{branch}(nf, m)$ if $\text{branch}(n, m)$;
- $\text{branch}(na, m)$ if $\text{branch}(n, m)$.

Let G be an ART.

$$\text{branch}'(m) = \{n \mid nr \in \text{dom}(G) \text{ and } \text{branch}(n, m)\}$$

Note that $\text{branch}'(m)$ is the set of all evaluated branch nodes of m .

Lemma 5. Let G be an ART.

- If $n \in \text{branch}'(mf)$ or $n \in \text{branch}'(ma)$ then $n \in \text{branch}'(m)$.
- If $mr \in \text{dom}(G)$ then $\text{children}(m) = \text{branch}'(mr)$.

Proof. By the definitions of children and branch' .

Definition 14. (*depth*) Let m be a node in an ART G .

$$\text{depth}(m) = \begin{cases} 1 + \max\{\text{depth}(mf), \text{depth}(ma)\} & \text{if } (m, mf \circ ma) \in G \\ 1 + \text{depth}(mf) & \text{if } (m, mf \circ j) \in G \text{ and } j \neq ma \\ 1 + \text{depth}(ma) & \text{if } (m, i \circ ma) \in G \text{ and } i \neq mf \\ 1 & \text{if } (m, i \circ j) \in G \text{ and } i \neq mf \text{ and } j \neq ma \\ 0 & \text{otherwise} \end{cases}$$

Lemma 6. Let G be an ART and m a node in G . If $\text{redex}(n) \simeq_I \text{mef}(n)$ for all $n \in \text{branch}'(m)$, then $\text{reduct}(m) \simeq_I \text{mef}(m)$.

Proof. By induction on $\text{depth}(m)$.

When $\text{depth}(m) = 0$, we have $(m, e) \in G$ where e is a node or an atom.

- If e is a node, then $mr \in G$ by Lemma 1. Then by the definitions of reduct and mef , we have $\text{reduct}(m) = \text{mef}(e)$ and $\text{mef}(m) = \text{meft}(m) = \text{mef}(e)$.
- If e is an atom, we have $\text{reduct}(m) = e$. Now, we consider the following two cases. If $m \in \text{branch}'(m)$, then we have $mr \in \text{dom}(G)$ and $\text{mef}(m) \simeq_I \text{redex}(m) = e$. If $m \notin \text{branch}'(m)$, then we have $mr \notin \text{dom}(G)$ and $\text{mef}(m) = \text{meft}(m) = e$.

For the step cases, we proceed as follows.

- If $m \in \text{branch}'(m)$, then we have $mr \in \text{dom}(G)$ and $\text{redex}(m) \simeq_I \text{mef}(m)$. And we need to prove $\text{redex}(m) \simeq_I \text{reduct}(m)$.

Let us consider only one case here. The other cases are similar. Suppose $(m, mf \circ j) \in G$ and $j \neq ma$, then by the definitions we have

$$\begin{aligned}\text{redex}(m) &= \text{mef}(mf) \text{mef}(j) \\ \text{reduct}(m) &= \text{reduct}(mf) \text{mef}(j)\end{aligned}$$

Since for any $n \in \text{branch}'(mf)$, by Lemma 5, we have $n \in \text{branch}'(m)$ and hence $\text{redex}(n) \simeq_I \text{mef}(n)$. By the definition of depth , we also have $\text{depth}(mf) < \text{depth}(m)$. Now, by induction hypothesis, we have $\text{reduct}(mf) \simeq_I \text{mef}(mf)$. And hence we have $\text{redex}(m) \simeq_I \text{reduct}(m)$.

- If $m \notin \text{branch}'(m)$, then $mr \notin \text{dom}(G)$.

Let us also consider only one case. The other cases are similar. Suppose $(m, mf \circ j) \in G$ and $j \neq ma$, then by the definitions we have

$$\begin{aligned}\text{mef}(m) &= \text{mef}(mf) \text{mef}(j) \\ \text{reduct}(m) &= \text{reduct}(mf) \text{mef}(j)\end{aligned}$$

The same arguments as above suffice.

Corollary 1. *Let G be an ART and mr a node in G (i.e. $mr \in \text{dom}(G)$). If $\text{redex}(n) \simeq_I \text{mef}(n)$ for all $n \in \text{children}(m)$, then $\text{reduct}(mr) \simeq_I \text{mef}(m)$.*

Proof. By Lemma 5 and 6.

The condition, $\text{redex}(n) \simeq_I \text{mef}(n)$ for all $n \in \text{children}(m)$, basically means that m does not have any erroneous child nodes.

Lemma 7. *Let G be an ART and mr a node in G (i.e. $mr \in \text{dom}(G)$). Then $\text{redex}(m) \rightarrow_P \text{reduct}(mr)$.*

Proof. Since there is a computation at the node m , we suppose G at node m matches the left-hand side of the rewriting rule $fp_1 \dots p_n = R$ with $[m_1/x_1, \dots, m_k/x_k]$. We need to prove that there exists a substitution σ such that $\text{redex}(m) = (fp_1 \dots p_n)\sigma$ and $\text{reduct}(mr) = R\sigma$. In fact $\sigma = [\text{mef}(m_1)/x_1, \dots, \text{mef}(m_k)/x_k]$.

Now, we need to prove that $\text{redex}(m) = (fp_1 \dots p_n)\sigma$ and $\text{reduct}(mr) = R\sigma$. For the first, we proceed by the definition of redex and pattern matching. For the second, we proceed by the definition of reduct and graph .

Now, we come to the most important theorem, the correctness of algorithmic debugging.

Theorem 3. (Correctness of Algorithmic Debugging) *Let G be an ART, T its EDT and m a faulty node in T . If the equation for the faulty node m is $fb_1\dots b_n = M$, then the definition of f in the program is faulty.*

Proof. By Lemma 7 and Corollary 1, we have $redex(m) \rightarrow_P reduct(mr)$ and $reduct(mr) \simeq_I mef(m)$. Since $fb_1\dots b_n \equiv redex(m) \not\equiv_I mef(m) \equiv M$, we have $fb_1\dots b_n \rightarrow_P reduct(mr)$ and $fb_1\dots b_n \not\equiv_I reduct(mr)$. The computation from $fb_1\dots b_n$ to $reduct(mr)$ is a single step computation, but $fb_1\dots b_n$ is not semantically equal to $reduct(mr)$. So the definition of f in the program must be faulty.