Article

Not peer-reviewed version

An Imperative Term Graph Programming Language

David A. Plaisted *

Posted Date: 5 November 2025

doi: 10.20944/preprints202511.0214.v1

Keywords: term graphs; programming languages; imperative languages

 doi:10.20944/preprints202511.0214.v1

Article

An Imperative Term Graph Programming Language

David A. Plaisted

Department of Computer Science, UNC Chapel Hill, Chapel Hill, NC 27599-3175; Email: plaisted@cs.unc.edu; Phone: (919) 590-6051

Abstract

In contrast to some other term rewriting languages based on term graphs, the Imperative Term Graph Programming Language (ITGL) is an imperative term graph language, with assignment statements, conditional statements, iterative statements, arrays with destructive operations, procedures, and recursion. States consist of a term graph together with an environment, and statements map states to states. Pure functional languages may need to copy arrays when modifying them, which can lead to inefficiencies, but imperative languages avoid this problem. The syntax and semantics of the language ITGL are presented, followed by proofs of two of its properties called term dependence and isomorphism dependence, and proofs of some other properties as well. In addition, some possibilities for caching in this language are explored. The application of this language as an abstract language for algorithms which can then be translated into other common imperative languages is also mentioned.

Keywords: term graphs; programming languages; imperative languages

---

1. Introduction

We explore the possibility of using term graphs as a basis for an imperative language. We present a language based on this idea. The presented language has a very simple syntax and semantics. A complete description of the syntax and semantics takes about 13 pages, including some features that are not really necessary such as a term rewriting facility. The language also has the efficiency of arrays in imperative programming languages. The language has some interesting mathematical properties which are explored.

1.1. Term Graphs

A term graph is a graphical structure in which the nodes represent terms. Common subterms can be represented only once, with shared structure. Terms are very basic and universal mathematical objects which can also be used to represent data structures in programming languages. If f is a function symbol and the t_i are terms then $f(t_1 \cdots t_n)$ is a term. In the term graph language presented here, terms do double duty. They can be expressions to be evaluated such as $\gcd(x+3,7)$ or they can be data structures such as cons(a,cons(b,NIL)) as in LISP. The former make use of defined symbols and the latter make use of constructors which can appear in data structures. Only constructors may appear in term graphs which implies that terms represented by term graphs are ground (variable-free) constructor terms.

If there are cycles in the term graph then the terms that are represented can be infinite such as $f(a, f(a, f(a, \cdots)))$. This permits term graphs to represent data structures such as doubly linked lists which have cycles in them. Two dimensional arrays can be represented in term graphs using finite terms, as an array whose elements are arrays, or more precisely, as a term whose top-level subterms are also terms. Of course, all terms of arity one or more have top-level subterms which are terms, so two dimensional arrays do not require any extension to the language. One might also implement two dimensional arrays by one dimensional arrays, translating a pair of indices into a single index.

Languages based on term graphs are typically defined by term rewriting [DJ90,BN98,DP01]. The language presented here is unusual in this respect because it is based on term graphs but is an

 doi:10.20944/preprints202511.0214.v1

imperative programming language. Thus we call this language an imperative term graph language (ITGL). However, a facility for specifying and implementing term rewriting in ITGL is also presented. This gives the language a convenient pattern matching facility. For simplicity the language is untyped.

1.2. Imperative Languages and ITGL

Imperative languages map states of the machine to states of the machine while functional languages replace expressions by equivalent expressions. In ITGL the state of the machine is represented by a term graph and an environment, and statements in the language map such states to states. The environment essentially tells the values assigned to program variables. More precisely, the nodes in the term graph correspond roughly to storage locations (addresses) or registers in which the values of program variables are stored. The environment maps program variables to such nodes which tell where the value of the program variable is stored. These storage locations are made explicit in the semantics of ITGL. However, the nodes never appear explicitly in the language itself. Each such node has a constructor function symbol and a list of children, which are nodes. By tracing down these links and reading the function symbols one can construct a term from a node in the term graph. By making the storage locations explicit it is straightforward for example to represent the fact that two program variables are stored in the same location so that changing the value of one variable will also change the value of the other. There is no way to change the function symbol assigned to a node in the term graph but the children of the node can be changed. This corresponds to the fact that it is hard to change the name of an array in a conventional imperative language but one can easily assign values to elements of the array. One can change the value of a program variable by an assignment statement, which can cause the variable to point to a different storage location, but this will not affect the values of other variables stored at the original location. However, changing the children of the node will affect the values of all variables stored at that node.

The constructs one wants to have in imperative languages include assignment statements, conditional statements, iterative statements, procedure definitions, and procedure calls. Arithmetic expressions and other functional expressions are also essential. One also wants to have arrays and to be able to replace an element of an array. In ITGL arrays are terms and are treated like any other terms, and assignments to an element of an array are implemented as replacing a top-level subterm of a term in a term graph.

The formal semantics of ITGL is presented assuming termination, then defining the denotational semantics of this language and proving properties of it by induction on the length of the computation. This paper only considers the terminating case, but it should be possible to extend the semantics to nontermination using least fixpoint semantics and complete partial orderings.

In this language parameters are essentially passed by reference. Also, there are no global variables, so that all variables in a procedure have to be passed in as parameters or locally assigned.

Formalisms such as Hoare logics and abstract interpretations that enable proofs of correctness in other imperative languages should also work for this language. In general, a simple syntax and semantics facilitates the proofs of correctness of programs. In this regard the simplicity of the syntax and semantics of ITGL is an advantage. Simplicity also makes it easier to construct correct translations into other languages. A complicated language will necessitate a complicated translation. The fact that ITGL does not have interrupts or input-output also decreases its complexity. The simplicity of ITGL also means that the language is not likely to change in any essential way, so that a program that runs in ITGL should still run 50 years from now. This simplicity also makes the language easier to learn. Finally, the simplicity, together with an imperative style, should enable efficient implementations of the language. Of course, the simplicity may also restrict the flexibility of the language.

1.3. Imperative Languages Versus Functional Languages

Why was ITGL designed as an imperative language and not a functional language? The main reason was that defining it this way seemed to be simpler than defining it using term rewriting.

 doi:10.20944/preprints202511.0214.v1

Both kinds of languages have their advantages. The typical machine architecture has a state which consists of the contents of memory and the contents of various registers, and instructions change the state of the machine. This architecture is most closely related to the imperative language design which may imply that imperative languages can be more readily mapped to conventional machines and may have an efficiency advantage. Even with architectures with massive parallelism such as GPUs, one basically has a collection of von Neumann machines operating together. But pure functional language have more mathematical elegance because their execution mechanism involves replacing terms by equal terms.

Both kinds of languages use arrays, which are needed for efficiency because arbitrary elements of the array can be accessed quickly. In ITGL arrays are considered as terms and thus inherit the semantics of terms which differs in some respects from the typical semantics of arrays. This is an interesting issue mathematically but we believe it will not have much impact on practical programs. In general, imperative languages with side effects do not need to copy arrays when an array element is changed. Pure functional languages need to copy arrays when an element of the array is changed, unless the array references are single threaded. By single threaded we mean that after an array is modified, the old versions of it will never again be referenced. This use of the term may be non-standard.

If array references are not single threaded, then no formalism can have both efficient array operations and avoid side effects. If array references are not single threaded, then there may be references to old versions of the array, so it will be necessary to copy the array in order to keep all of these versions available. It doesn't matter whether one is using functional programming, monads [Wad95], continuation style parameter passing, or an imperative style, the same constraint still applies. Because pure functional programming avoids side effects, it cannot avoid array copying if the array references are not single threaded.

In ITGL, as in other imperative languages, array assignments are destructive which means that the array is modified in place and all references to the array change. In particular, in ITGL, if an array assignment is done inside a procedure, the term graph is changed and this change persists when the procedure is exited. In a pure functional language such side effects are not allowed because one can only replace expressions by equivalent expressions. This means that when replacing an element of an array in a pure functional language, one must copy the whole array if there are multiple references to it. This can cause considerable inefficiency. A straightforward copying of an array will cost linear time and storage. This copying can be made more efficient by storing the array as a binary tree, but there is still a logarithmic overhead in the worst case for each array assignment compared to imperative languages if the array is copied. If an array is stored as a binary tree, then one can construct a binary tree representing a copy of the array with one element changed by reusing much of the structure of the old array. This takes time proportional to the logarithm of the size of the array.

It is possible to implement arrays efficiently in a pure functional language using monads if references to the arrays are single threaded. This issue is discussed in a paper by Wadler [Wad95]. For example, one can use state monads. A state monad [Mog91] operates on pairs (v, S) where v is a value and S is a state and the state can be an array, but state monads are not as familiar to many programmers as conventional imperative arrays are.

Similarly, functional languages are not as popular for applications or in academia. Perhaps the conventional model of instructions changing the state of a machine is easier for people to understand.

Continuation style programming [SS75] is another programming style, but it is unnatural for many programmers and is not popular in academia and industry. It may also be more prone to error.

1.4. Side Efects

ITGL is an imperative language. Imperative languages have side effects but these are not allowed in pure functional languages. What are the advantages of side effects? Are they necessary?

First, exactly what is a side effect? We will say that it is a change in the value of a global variable (or the state of a file or device), caused by a procedure or function but outside of its scope. In an

 doi:10.20944/preprints202511.0214.v1

imperative language this often happens when modifying an element of an array without choosing a new name for the modified array.

With side effects it is possible for two calls to $f(x)$ in sequence to return different results, for example, if f is a counter that modifies the value of x. In a pure functional language two calls to $f(x)$ in the same context should always return the same value. Also, in an imperative language it is easy to write an iterative loop to assign a value to each element of an array, while this may require recursion in a functional language, and even then there is a side effect unless a new name is chosen for the modified array each time. Sorting algorithms typically move elements around in an array, which is a side effect of the code if the array is not continually renamed. However, continually renaming the array requires passing it around from one procedure call to another repeatedly. Memoization (remembering a value so that it does not have to be recomputed) involves a side effect. This can happen, for example, in the computation of the Fibonacci sequence.

In a typical imperative language, to compute the n^{th} element of the Fibonacci sequence one would have something like $f(n,x) = \{\text{if } x[n] = 0 \text{ then } x[n] := f(n-1,x) + f(n-2,x)\}$; return $x[n]$ where the array x holds the values computed and $f(n,x)$ is the n^{th} Fibonacci number. The calls to f modify the array x without choosing a new name for it, so the change to the array is a side effect. In a pure functional language x could not be modified; the new value of the array x would have to be given a new name and passed back from the recursive calls to f. Then the updated x would have to be passed on to other calls to f. Returning the new value of the array each time and passing it on to the next call would make the code more complicated. This would also require copying the array unless the array were referenced in a single threaded manner. Of course, in a functional language one could compute the Fibonacci sequence efficiently by writing a function to return both $f(n)$ and $f(n-1)$ at the same time, and using a simple recursion.

Updating a database also has a side effect, unless one wants to pass back the entire modified and renamed database when making an update and pass the modified database on to others who may want to use it. In a concurrent system this may not even be possible.

Also, in graph algorithms it is often necessary to mark nodes concerning their status, whether they have been visited or not. The graph is typically stored in an array, so these updates involve assigning to an array element. This is a side effect which is acceptable in an imperative language but can only be avoided in a pure functional language by continually passing around the most recent version of the array. That would make the functions in the graph algorithm return the array along with possibly other information, making them more complicated. Making functions more complicated might impact correctness proofs. One could make the return of the array value somehow implicit in a functional language, but then different functions might operate on different arrays, so what they depend on and modify would have to be specified in some manner. It might also be difficult to interface functions that operate on different arrays. One can make this all uniform in a functional language by making all functions depend on and return the values of all global variables. But this is implicitly what imperative languages do, so then one has essentially made a functional language into an imperative language.

Explicitly passing an array or a state between functions in a functional language need not incur an extra cost, if the array or state is accessed in a single threaded manner; the array or state can simply be modified in place and renamed. However, to maintain correctness this approach requires a method to ensure that the old versions of the array or state are never accessed, increasing the burden on the programmer or compiler.

Pure functional languages have alternate means to perform many of the things that imperative languages do, and to do them efficiently, but many programmers find the imperative style of programming to be more natural. Another possibility is to have functional languages with some imperative features. This section was partially based on some queries to the AI system on the Google search engine in September, 2025.

 doi:10.20944/preprints202511.0214.v1

1.5. What Is an Algorithm?

The ITGL language is related to the question, what is an algorithm? What do we mean by a particular algorithm such as insertion sort? Clearly one can write an insertion sort in many languages, so it's not the same as a particular program. It's also not the same as an algorithm in the sense of something that can be computed by a Turing machine because there are many Turing computations that do not implement insertion sort. It has something to do with a specification of an input-output relation of a program but it is more than that because an algorithm in the conventional sense also specifies how the computation proceeds (such as in Euclid's algorithm for the greatest common divisor). Maybe it has something to do with a particular form of a proof of the final input-output relation using Hoare's axioms. This question has been considered by various authors in the literature [Var12,BFG+12,Hil15,BDG09,Mos01,BG03b,Gur00,BG03a]. Algorithm texts frequently use pseudo-code to specify algorithms but without a precise semantics.

We consider algorithms whose input and output are abstract mathematical objects such as graphs, trees, or integers. This definition includes such algorithms as maximum flow, traveling salesman, and greatest common divisor algorithms which map abstract mathematical objects into other mathematical objects. This definition leaves out many algorithms such as memory management algorithms. We want to specify a language that can be used to give an abstract definition of algorithms on abstract mathematical objects, something like the pseudo-code in algorithm texts, but with a precise semantics. Because of its simple syntax and semantics and imperative style, ITGL is one candidate for such a language to be used to specify algorithms. In algorithm texts, algorithms are typically presented in an imperative style, so it is convenient to adopt an imperative formalism for describing abstract algorithms. We hope that such an abstract definition of an algorithm can be translated into particular programs in typical imperative programming languages. For this it helps to have similar programming constructs in the abstract language and common imperative languages to make the translation simple and natural. Then one can verify the abstract algorithm once and if the translations are correct, one will obtain verified algorithms in many typical programming languages. Translating ITGL into typical imperative languages should be facilitated because both ITGL and the target language are imperative and because of the simplicity of the semantics of ITGL. However, the use of ITGL for abstract definition of algorithms and the translation of such algorithms into typical imperative programming languages are not further discussed in this paper.

An insertion sort program and a greatest common divisor program in ITGL are given as examples of abstract specifications of an algorithm. The ITGL language has some features that are not needed for abstract specification of algorithms and so these would not have to be used in such specifications. For example, the rewrite rule facility and the caching mechanism would not be needed.

The question arises whether ITGL is the only language that can be used for abstractly specifying algorithms. In fact, it is not the only language, and may not even be the best language for this purpose. However, because algorithms are typically written in an imperative style, it is natural that a language for specifying algorithms should be imperative, with program variables, assignment statements, conditional statements, iterative statements, arrays with side effects, procedure definitions, and procedure calls. Also, the language should be possible to implement efficiently and should have a simple syntax and semantics. Because of its characteristics ITGL is a good candidate for such a language.

1.6. Organization of This Paper

This paper is organized as follows. First the syntax of the language is presented, then a definition of the denotational semantics. The language has several different kinds of constructs: Imperative statements, functional expressions, and three kinds of procedures: imperative procedures, rewrite procedures, and compiled procedures. States in the language consist of an environment and a term graph. Functional expressions return a node in the term graph and a state. A garbage collection procedure for the term graph is not specified, and because of this one can prove that the size of

 doi:10.20944/preprints202511.0214.v1

the term graph never decreases. A notion of variable binding is defined and it is shown that all constructs preserve variable binding, namely, the environment maps program variables to nodes in the term graph. If a couple of kinds of statements in the language are not used then the statements and procedures are term dependent in the sense that terms in the resulting state are functions of the terms in the input state. If all kinds of statements in the language are used then the language is not term dependent but it is still isomorphism dependent.

Quite a bit of space is devoted to the issue of caching in the term graph: When can all copies of the same term be stored at the same node and when must they be stored in different nodes? Caching means that if two or more data structures are identical then only one representative needs to be stored, with multiple references to it. This saves space but one might want to have separate copies of a data structure in case one does not want destructive operations on one copy of a structure to affect the other copies. A systematic way of dealing with this issue is presented.

We give a simple facility for input output but this is not the main focus of the language.

1.7. Previous Work

As for previous work in this area, the paper "Term Graph Rewriting" by Barendregt et al [BvEG+87] relates graph rewriting to term rewriting and gives a correctness result. Their concept of a term graph is essentially identical to ours. It allows cycles and does not require that all nodes be reachable from a root node. They also have a notion of isomorphism between term graphs but it is not the same as our notion. Their graphs are not restricted to constructors and may contain variables. The paper Towards an Intermediate Language based on Graph Rewriting by Barendregt et al [BEG+87] gives a language Lean for specifying computations in terms of graph rewriting. The graphs they use are similar to ours, in that they contain no variables, may have cycles, and there is no requirement that all nodes are reachable from a root. Their graphs are used in a way similar to our term graph. A Lean program consists of a set of rewrite rules. The language has functions similar to our compiled procedures. The paper Graph rewriting: An algebraic and logic approach [COU90] considers graph rewrite rules and graph grammars and their relation to category theory. The formalism is more general than that of term graphs and term graph rewriting. Graph rewriting: A bibliographical guide by Courcelle [Cou95] gives a survey of various notions of graph rewriting. The paper Jungle evaluation [HKP88] considers jungles, which are acyclic hypergraphs in which it is not necessary for all the nodes to be reachable from the root. Such structures are manipulated by jungle rewrite rules which generalize term rewrite rules. Such rules can handle non-left-linear rewrite rules and permit sharing of structure on the right-hand side of a rule to speed up term rewriting. The paper Term Graph Rewriting by Klop [Klo96] considers many different related systems. He considers term graphs as systems of recursion equations that may essentially involve cycles and also discusses the lambda calculus. An example system such as he considers is $\{X = f(X, Y), Y = g(X)\}$. He considers operations that involve replacing a variable by its definition. The paper "Term Graph Rewriting" by Plump [Plu99] requires the term graphs to be acyclic. This paper is mainly concerned with the relation between term graphs and term rewriting. He formalizes term graphs using hyperedges which include a node and its children. Also, all nodes must be reachable from the root. The graphs are not restricted to constructors and may contain variables. Nodes may have more than one incoming edge. He also discusses isomorphisms. The paper Rewriting on cyclic structures: Equivalence between the operational and the categorical description [CG99] considers only acyclic rewrite rules but possibly cyclic term graphs, and relates their definition of term graph rewriting to category theory. They consider directed graph representations in which there can be multiple references to shared structures. The paper Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics [DLLL05] makes global addresses explicit, and these correspond to memory addresses in an implementation. Their memory addresses are similar to the nodes in ITGL. In Modeling Pointer Redirection as Cyclic Term-graph Rewriting [DEP07] the authors discuss pointers in cyclic term graphs using the double pushout approach. Their term graphs include structures such as circular lists and doubly linked lists. Their proposal focuses on pointer rewriting. Drag Rewriting by Nachum Dershowitz, Jean-Pierre Jouannaud,

 doi:10.20944/preprints202511.0214.v1

and Fernando Orejas [DJO24] considers a very general model of graph rewriting. Drags permit the formalization of non-linear rewrite rules in graph rewriting, which has been a difficult issue to date. Drags are finite directed rooted labeled ordered multi-graphs. From the paper: "Drags, the alternative model considered here, are arbitrary directed graphs equipped with roots and sprouts that facilitate composition. Internal vertices, as in term graphs, are labeled by function symbols having a fixed arity. Sprouts are non-internal vertices without successors, labeled by variables. Our framework is able to faithfully encode term rewriting without restriction, hence finally solving this old question positively." The paper An algebraic definition for control structures [Cou80] considers rational expressions which are expressions denoting regular trees. These are the possibly infinite trees having a finite number of subtrees, and they are typically obtained by the unraveling of possibly cyclic finite graphs.

The paper "Transforming imperative programs into bisimilar logically constrained term rewrite systems via injective functions from configurations to terms" [NKK25] presents a translation from a simple imperative language into a logically constrained term rewriting system (LCTRS) and proves its correctness. An LCTRS is basically a conditional term rewriting system in which the condition is a formula from some underlying logical theory. The condition is interpreted using the underlying theory. This is a pleasant formalism and avoids some of the complexities of conditional term rewriting in which the condition is evaluated by rewriting. A paper by Plaisted [Pla13] gives an abstract discussion of the idea of an imperative language for describing algorithms, such that these algorithms can then be translated into various imperative style programming languages. This paper does not consider or define any specific language. An earlier paper by Plaisted [Pla05] extends this proposal to define an abstract language and then translate programs into various common imperative languages. It considers a system for proving that such programs are correct. This paper also considers nonterminating computations using least fixpoints and denotational semantics. Some papers by Barnett and Plaisted [BP18,BP21,PB20] describe a rewrite-based imperative style language with arrays and side effects that also uses a term graph as the basic data structure. These papers are defined using orthogonal term rewriting systems and make use of term graphs with destructive array operations. The application of this language to abstract descriptions of algorithms is also mentioned. However, this system seems more complicated than the system proposed in the current paper. In his thesis [Tao23], Tao Tao describes a rewriting-based imperative style language with side effects and its implementation on a field-programmable gate array.

2. Syntax

$T(F, X)$ is the set of terms over a set F of symbols and a denumerable set X of program variables. Terms may be infinite.

The letters r s and t denote terms (also called functional expressions).

The letters f g and h denote function symbols.

The letters x y and z are used for program variables and logical variables.

The letters i j k m and n denote program variables that are intended to have integer values and in general denote integers as usual.

Elements of F can be constructors or defined function symbols.

The functions **top**, **arg**, and **arity** are abstract functions on terms and also are function symbols in the language that operate on term graphs. So we write **atop**, **aarg**, and **aarity** for the abstract functions and **top**, **arg**, and **arity** for the function symbols in the language with similar functions.

If t is a term then **atop**(t) is its top level function symbol and **aarg**(i, t) is the i^{th} argument of f. So regarding $f(x, y)$ as a term, **atop**$(f(x, y)) = f$, **aarg**$(1, f(x, y)) = x$, and **aarg**$(2, f(x, y)) = y$. The arity of a function symbol is the number of arguments it takes so **aarity**$(f) = 2$ for this example.

The statements and expressions in this language consist of procedure definitions, functional expressions which are terms, and imperative statements which can be assignment statements, conditional statements, and iterative statements. We give some idea of the semantics of some of the syntactic constructs while presenting the syntax.

 doi:10.20944/preprints202511.0214.v1

2.1. Programs

A program is a term (functional expression) followed by a double semicolon, followed by some number of procedure definitions, each followed by a double semicolon. There is a period at the end of the program. It is intended that the evaluation of the functional expression or of any procedure in the program will only cause execution of procedures in the program.

If the programmer forgets to give a procedure definition for a function symbol, then that symbol will be considered as a constructor.

2.2. Syntax of Functional Expressions (Terms)

A term (functional expression) is a function symbol followed by a list of terms (functional expressions). A program variable (a variable) is also a functional expression.

2.3. Constructors

A constructor is a function symbol that is not defined in an imperative procedure definition or in a rewrite rule procedure definition or defined as a compiled function. If a function symbol is intended to be an imperative procedure but the procedure definition is forgotten, that symbol will be considered as a constructor. Among the constructors, there is the constant symbols $\perp_u$ for undefined variables and $\perp_n$ for nontermination. Though we do not explicitly consider nontermination we reserve the symbol $\perp_n$ for it. This symbol might be useful for defining a least fixed point semantics. There is also another constant symbol $\perp_{err}$ for error handling such as for $x/0$. There are also special symbols that are technically constructors but which may be evaluated in different ways than other constructors.

Among the special symbols there is a function symbol **arg** such that roughly speaking $\mathbf{arg}(i,t)$ evaluates to the i^{th} argument t_i of a constructor f in a term t of the form $f(t_1 \cdots t_n)$. The expression $\mathbf{arg}(i,t)$ can also be written as $t[i]$. (More precisely t evaluates to a variable that is bound to a constructor term of the form $f(t_1 \cdots t_n)$.) There is a function symbol **top** such that roughly speaking $\mathbf{top}(t)$ for a term t of form $f(t_1 \cdots t_n)$ evaluates to a constant c_f where f is a constructor For distinct constructors f and g, c_f and c_g are distinct. Also, roughly speaking, $\mathbf{arity}(c_f)$ evaluates to n for a constructor f of arity n. A more precise semantics for these symbols is given later. There is also an equality function symbol equal.top which on evaluation tests if the top symbols of two constructor terms are the same. There is a function symbol equal.node whose value on evaluation depends on the state and will be explained later. The function symbol "copy" can only appear in a term of the form $copy(x)$ where x is a program variable. So a term $copy(t)$ cannot appear as a proper subterm of another term. Integers and Booleans are considered as constructors and as individual constants.

2.4. Arity Declarations

Arities are declared by how a function symbol is used, so if $f(s,t)$ is in the program then f has arity two. It is an error to use the same function symbol with two different arities. We assume $f[n]$ is a function symbol with arity n distinct from f and from $f[m]$ for any m different from n. So $f[3](x,y,z)$ is a valid term. More precisely $f[t]$ is a permitted notation where t evaluates to an integer. This permits essentially arrays of a dimension that is specified by the input. If t does not evaluate to a nonnegative integer then this is an error. One can write $f[t](s \cdots s)$ to indicate a term where all the arguments of f are the term s evaluated one at a time from left to right. A term $g(t \cdots t)$ is in error if g does not already have an arity specified by previous appearance in a term unless g is of the form $f[s]$ for some function symbol f and term s. We assume there is a way to declare function symbols and their arities when necessary.

2.5. Procedure Definitions

A procedure definition can be an imperative function (procedure) definition, a rewrite rule procedure definition, or a compiled function (procedure) definition.

 doi:10.20944/preprints202511.0214.v1

2.6. Syntax of Imperative Function Definitions

An imperative function definition is a function symbol followed by a list of formal parameters (program variables), followed by an imperative statement, a functional expression, and a double semicolon as follows:

$$\text{procedure } f(x_1 \cdots x_n)\{P\}t;;$$

Here f is a defined function symbol, the x_i are formal parameters, and P is an imperative statement. Formal parameters of procedures are program variables and must be distinct. Formal parameters can be assigned into but their modified values will not be transferred back to the calling point. Program variables that are not formal parameters can appear in imperative statements and functional expressions in a procedure definition.

2.7. Rewrite Rule Procedure Definitions

A rewrite rule procedure definition is of the form

$$\text{rewrite } f\{r_1 \to s_1, \cdots, r_n \to s_n\};;$$

where f is a defined function symbol and the r_i and s_i are terms. The expressions $r_i \to s_i$ are called rewrite rules. The r_i must be left linear (each variable occurs at most once). Also, all variables in s_i must appear also in r_i. The terms r_i must be of the form $f(t_1 \cdots t_m)$ where the t_i are finite constructor terms that may include variables. The terms s_i must be finite (non cyclic). The left linearity restriction makes the implementation more efficient because it is not necessary to test for the equality of two terms when mapping formal to actual paremeters.

2.8. Compiled Function Definitions

A compiled function definition is of the form

$$\text{compiled } f(x_1 \cdots x_m) \to y \text{ where } A[x_1 \cdots x_m, y];;$$

Here f is a defined function symbol, the x_i are program variables, and y is a program variable. Also A is an assertion in some logical theory giving the required relation between the x_i and the output y, for example $y = x_1 + x_2$ and y, x_1, x_2 are integers. It is required that for all finite constructor terms $s_1 \cdots s_m$ there is at most one constructor term t such that $A[s_1 \cdots s_m, t]$ is a theorem of the logical theory.

The instances of such a compiled function definition are expressions of the form $f(s_1 \cdots s_n) \to t$ where the s_i and t are finite ground (variable free) terms composed of constructors and the assertion A holds, that is, $A[s_1 \cdots s_n t]$ is a theorem of the appropriate logical theory. There may be infinitely many such instances.

The intention is that compiled functions would be implemented in some other language. For example, addition could be defined in this language using sequences of zeroes and ones to represent binary numbers, but it would be cumbersome. A compiled function for addition could take two constant symbols representing integers and compute their sum as another constant symbol more efficiently than this.

2.9. Imperative Statements

Imperative statements and programs are indicated by the letters P and Q. An imperative statement can be an assignment statement, a conditional statement, an iterative statement, or a concatenation of two imperative statements. Semicolon indicates concatenation.

An assignment statement can be a simple assignment statement of the form $x \leftarrow t$ where x is a program variable and t is a functional expression (a term), a multiple assignment statement of the form $(x_1 \cdots x_n) \leftarrow t$ where the x_i are program variables, an argument replacement statement of the form $x[i] \leftarrow t$ where x is a program variable, or a copy statement of the form $x \leftarrow copy(y)$ where x

 doi:10.20944/preprints202511.0214.v1

and y are program variables. A copy statement is syntactically a special case of a simple assignment statement. The copy function symbol can only appear in a statement of the form $x \leftarrow copy(y)$ for program variables x and y.

2.10. Iterative Statements

An iterative statement can be a while statement or a for statement. A while statement is of the form

$$\text{while } r \text{ do } \{P\}$$

where r is a functional expression and P is an imperative statement. A "for" statement has the form

$$\text{for } i = j \text{ step } k \text{ until } n \text{ do } \{P\}$$

where P is an imperative statement and i, j, k and n are program variables.

2.11. Conditional Statements

A conditional statement has the form

$$\text{if } t \text{ then } \{P_1\} \text{ else } \{P_2\}$$

where t is a functional expression and P_1 and P_2 are imperative statements.

2.12. Sequence

If P_1 and P_2 are imperative statements then so is $P_1; P_2$.

2.13. Grammar

We now give a grammar in the extended context free style for the language.

<program> ::= <fcnl expr> ;; <proc def>*.

<fcnl expr> ::= <fcn symbol> <fcnl expr>* | <vbl>

<fcn symbol> ::= <defined symbol> | <constructor>

<proc def> ::= <imperative proc def> | <rewr proc def> | <compiled fcn def>

<imper stmt> ::= <assg stmt> | <cond stmt> | <iterative stmt> | <imper stmt> ; <imper stmt>

<assg stmt> ::= <vbl> ← <fcnl expr> | (<vbl>*) ← <fcnl expr> | <vbl>[<int>] ← <fcnl expr> | <vbl> ← copy (<vbl>) %distinct vbls in <vbl>*%

<imperative proc def> ::= procedure <defined symbol> (<vbl>*) { <imper stmt> } <fcnl expr> ;; %distinct vbls in <vbl>*%

<cond stmt> ::= if <fcnl expr> then { <imper stmt> } else { <imper stmt> }

<cond stmt> ::= if <fcnl expr> then { <imper stmt> }

<iterative stmt> ::= while <fcnl expr> do { <imper stmt> } | for <vbl> = <vbl> step <vbl> until <vbl> do { <imper stmt> }

<rewr proc def> ::= rewrite <defined symbol> { <rewrite rule>* } ;;

<rewrite rule> ::= <fcnl expr> → <fcnl expr> %rules are left linear, all variables on rhs appear in lhs, all fcnl exprs have the procedure name at the top level%

<compiled fcn def> ::= compiled <defined symbol>(<vbl>*) → <vbl> where <logical expr> ;;

%vbls in <vbl>* distinct, logical expr may have free variables in <vbl>* but no other free variables%

 doi:10.20944/preprints202511.0214.v1

3. Semantics

In specifying the semantics of the language, it is necessary to distinguish between symbols that are a part of the language, and other auxiliary mathematical functions that are used to define the semantics of the language.

3.1. Term Graphs

The language makes use of term graphs to store data. Term graphs are indicated by the letters G and H. A term graph is a set of triples (node, function symbol, list of children nodes) where there is at most one triple for any node. The notation $v : f(v_1 \cdots v_n)$ indicates the triple $(v, f, (v_1 \cdots v_n))$. If G is a term graph then the set of nodes of G is $\{v : \text{ for some } f \text{ and } (v_1 \cdots v_n), (v, f, (v_1 \cdots v_n)) \in G\}$. Nodes can be indicated by the letters u and v. The symbol f is the label of node v if the triple $(v, f, (v_1 \cdots v_n))$ is in G, and f must be a constructor symbol. The list $(v_1 \cdots v_n)$ is the list of children of v. Now nodes(G) is the set of nodes in G, $l[u, G]$ is the label of node u in G, **arity**$[u, G]$ is the number of children of u and args$[u, G]$ is the list of children. Also arg$[i, u, G]$ is the node which is the i^{th} child of u. These functions are easy to confuse with similar functions on terms such as **atop**(t), **aarity**(t), and **aarg**(i, t).

Given a term graph G and a vertex v in G, define term(v, G) to be the possibly infinite term defined by:

If v is $v : f(v_1 \cdots v_n)$ in G then term(v, G) is $f(\text{term}(v_1, G), \cdots \text{term}(v_n, G))$ so that **atop**(term(v, G)) = f and **aarg**$(i, \text{term}(v, G)) = \text{term}(v_i, G)$. In this formalism term$(v, G)$ does not contain any variables. Therefore, for all v in G, term(v, G) is a ground constructor term, possibly infinite if G has cycles. It is possible that two distinct nodes u and v satisfy term$(u, G) \equiv$ term(v, G) where $\equiv$ means syntactic identity. Because terms $f(t_1 \cdots t_n)$ are used also as arrays, it is possible to have more than one copy of an array with the same function symbol f.

For example, if $G = \{v_1 : f(v_2, v_3), v_2 : a, v_3 : g(v_4, v_5), v_4 : b, v_5 : c\}$ then term$(v_1, G) = f(a, g(b, c))$. If $G = \{v_1 : f(v_2, v_2), v_2 : g(v_3), v_3 : a\}$ then term$(v_1, G) = f(g(a), g(a))$. If $G = \{v_1 : f(v_2, v_1), v_2 : a\}$ then term$(v_1, G) = f(a, f(a, \dots))$.

The symbols $\perp_u$ and $\perp_e$ are constructors so these can be the labels of nodes in a term graph. We assume that all term graphs have a node with label $\perp_u$.

Define Replace$[G, u, i, u']$ as G with $u : f(u_1 \cdots u_i \cdots u_n)$ (for node u in G) replaced by $u : f(u_1 \cdots u' \cdots u_n)$ in G where u' is a node in G. This operation is used to define assignment to an element of an array (argument replacement). If $i \leq 0$ or $i > n$ then Replace$[G, u, I, u'] = G$.

Suppose G' = Replace$[G, u, i, u']$. Then term(u, G') = term(u, G) with the i^{th} argument replaced by term(u', G) if $0 < i \leq n$.

There are also functions make.term, new.term and cache.term that are auxiliary functions used to define the semantics of the language. Define make.term$(f(v_1 \cdots v_n), G)$ to be a pair (v, G') where v is a node and G' is $G \cup \{v : f(v_1 \cdots v_n)\}$. Depending on the implementation, the node v can be a new node, not present in G, or else it can be an existing node v of G if $v : f(v_1 \cdots v_n)$ already is an element of G. Define new.term$(f(v_1 \cdots v_n), G)$ to be a pair (v, G') where v is a new node not present in G and G' is $G \cup \{v : f(v_1 \cdots v_n)\}$. The operation new.term is needed for an explicit copy operation. A related operation cache.term will be described later. The functions make.term and new.term are the only functions that can change the set of nodes in the term graph and they can only do this by adding a node to the set of nodes. This implies that the set of nodes in the term graph never gets smaller and may only continually get larger as a program executes.

It would be possible to do garbage collection on the term graph, which could cause the set of nodes in the term graph to get smaller.

3.2. Environments

An environment is a function from program variables to nodes in a term graph G. It is required that undefined variables map to a node in G with label $\perp_u$. Environments map from all of the

 doi:10.20944/preprints202511.0214.v1

denumerably infinite set of program variables to nodes in G. Only finitely many of them will be defined and these are the only ones that need to be explicitly stored.

3.3. States

A state $(e.G)$ consists of a term graph G and an environment e that maps program variables to nodes in G. States are indicated by S and T. Also, $\text{env}(S)$ is the environment of a state S and $g(S)$ is the term graph of S so that $\text{env}((e, G)) = e$ and $g((e, G)) = G$. The function $\text{nodes}(S)$ is extended to states by $\text{nodes}((e, G)) = \text{nodes}(G)$

To show that a pair (e, G) is a state it has to be shown that e maps program variables to nodes in G. A pair (e, G) in which for some variable x, $e(x)$ is not a node of G, is said to be non variable binding; otherwise it is variable binding. All but finitely many variables will be undefined at any time and so they will map to a node in G.

The function "term" is extended to states by $\text{term}(x, (e, G)) = \text{term}(e(x), G)$ for program variables x. However, we often prefer the more complex notation $\text{term}(e(x), G)$ because it makes clear exactly how the term of a program variable is computed.

Sometimes in specifying the semantics it is necessary to modify the environment of a state; this can be done using the function fix.env defined by $\text{fix.env}(e', (e, G)) = (e', G)$. The function fix.env is an auxiliary function and is not in the syntax of the language.

We say that two environments e and e' conflict if for some variable $x, e(x) \neq e'(x)$.

Program variables x have a term value and a node value in a state S. If S is (e, G) then the term value of x in S is $\text{term}(e(x), G)$ which is $\text{term}(x, S)$ and the node value is $e(x)$.

The function $\text{equal.node}(x, y)$ (a node equality test) tests if the nodes $e(x)$ and $e(y)$ are the same. This function is in the syntax of the language.

3.4. Undefined Variables

Now $e(x) = v$ for some v with label $\perp_u$ for undefined variables. Formal parameters have initial values as specified by the call to the procedure. Also the environment $\perp_u$ is defined to be the environment e such that for all program variables x, $\text{term}(e(x), G) = \perp_u$. A more detailed semantics for procedures is given later.

We don't specify how $\perp_u$ and $\perp_{err}$ are handled.

3.5. Denotational Semantics (Assuming Termination)

We define the denotational semantics $[[q]]$ of statements q in the language by induction on the length of a computation. We also do proofs of properties of the language by a similar induction. Note that the notation for the denotational semantics used here differs from that in the book [NK14] that includes Isabelle proofs of its results. However, that language does not have arrays or procedures.

Our semantics uses innermost left to right evaluation for functional expressions (terms).

First we give an informal description of the denotational semantics, then a more formal description.

If t is a functional expression (a term) then semantically it maps from states to (node,state) pairs. During the execution (evaluation) of a functional expression, the state can be modified by evaluating functions in the term and by creating structure in the term graph to store the result computed by the term. Other structure besides this may be added to the graph during the evaluation of the expression. The Replace operation also modifies existing structure in the term graph. In this way the functions (procedures) in the term can modify the term graph.

The original environment is restored after the evaluation of a functional expression, though other environments may be created for other functions during the evaluation.

$\mathcal{S} \to \mathcal{V} \times \mathcal{S}$ (states $\to$ nodes $\times$ states) is the type of the semantics $[[t]]$ of functional expressions t; the environment does not change and if $[[t]](S) = (v, S')$ then v is a node in the graph $g(S')$ of state S'. We often write $[[t]](S)$ as $[[t]]S$.

 doi:10.20944/preprints202511.0214.v1

If P is an imperative statement then $[[P]]$ maps from states to states. The environment and graph may change. $\mathcal{S} \to \mathcal{S}$ (states $\to$ states) is the type of the semantics $[[P]]$ of imperative statements P. We often write $[[P]](S)$ as $[[P]]S$.

3.6. Length of a Computation Sequence

We define the length of a computation sequence and use it to do proofs by induction of properties of the computation, assuming termination. To prove that something is true of a statement P on a state S we assume that it is true for all (P', S') that are smaller than (P, S) in the ordering and show that it is true for (P, S). Assuming termination, this shows that the property is true of all pairs (P, S).

We use the notation $[P, S]$ for the length of the computation sequence for $[[P]](S)$ of a statement or expression P operating on a state S. $[P, S]$ is a nonnegative integer value that decreases with each computation in a terminating computation. We define $[P, S]$ along with the semantics for each kind of statement P. For auxiliary functions used to define the semantics we can define $[E]$ for an expression E as the length of the computation it represents. Hopefully the context will make this clear. Another way to formalize this is not to require $[P, S]$ to have integer values but just abstractly to put a partial ordering on the pairs $[P, S]$ where $[P, S] > [P', S']$ if the computation of $[P', S']$ is a strict subcomputation of the computation for $[P, S]$. Then we can say that the computation for $[[P]](S)$ is terminating if there are no infinite descending sequences starting from the initial pair $[P, S]$. This works by König's Lemma because each $[P, S]$ spawns only finitely other pairs $[P', S']$ such that $[P, S] > [P', S']$. The advantage of the partial ordering formalism is that it makes sense even for infinite computations and possibly can help to define a denotational semantics in that case. However, defining $[P, S]$ as an integer value is simpler for terminating computations.

3.7. Semantics of Imperative Statements

Now we define the semantics of imperative statements.

3.7.1. Assignment Statements (Note This Includes Simple Assignment Statements, Multiple Assignment Statements, Copy Statements, and Argument Replacements)

For an environment e define $e[y \leftarrow w]$ by $e[y \leftarrow w](y) = w$ and $e[y \leftarrow w](x) = e(x)$ for $x \not\equiv y$.
$[[x \leftarrow t]]S$ = fix.env(env$(S)[x \leftarrow v], S')$ where $(v, S') = [[t]]S$
(Simple assignment statement)
$[x \leftarrow t, S] = [t, S] + 1$.
For the partial order version, $[x \leftarrow t, S] > [t, S]$.

At the level of an assignment statement $x \leftarrow t$ the environment of S' is the same as the environment of S by Theorem 4 which will be proved later. However env(S) has to be modified to reflect the new assignment to x.

Now the sequence $x \leftarrow f[n](0 \cdots 0); y \leftarrow f[n](0 \cdots 0)$ makes two different arrays unless the term $f[n](0 \cdots 0)$ is in the set $\mathcal{T}$ to be discussed later. But $x \leftarrow f[n](0 \cdots 0); y \leftarrow x$ just makes one array pointed to by both x and y. In the former case an argument replacement statement $x[i] \leftarrow t$ in state (e, G) (defined below) will not change term$(e(y), G)$ but in the latter case it will. Note that arg$[j, e(x), G]$ can be a term of the form $f[n](0, \cdots, 0)$, too.

3.7.2. Semantics of $(x_1 \cdots x_n) \leftarrow t$ (Multiple Assignment Statement)

$$[[(x_1 \cdots x_n) \leftarrow t]] = [[x \leftarrow t; x_1 \leftarrow \arg(1, x), \cdots, x_n \leftarrow \arg(n, x)]]$$
$$[(x_1 \cdots x_n) \leftarrow t, S] = [x \leftarrow t; x_1 \leftarrow \arg(1, x), \cdots, x_n \leftarrow \arg(n, x), S] + 1$$

For the partial order version,
$[(x_1 \cdots x_n) \leftarrow t, S] > [x \leftarrow t; x_1 \leftarrow \arg(1, x), \cdots, x_n \leftarrow \arg(n, x), S]$.

The semantics of arg will be defined later. There is no need to define a tupling operation $(t_1 \cdots t_n)$ because one can use $f(t_1 \cdots t_n)$ for that where f is a constructor, and then use the arg function to extract the t_i. The arity function can be used to determine the number of arguments of f.

 doi:10.20944/preprints202511.0214.v1

3.7.3. Copy Statement

If $e(y) = v : f(v_1 \cdots v_n)$ then
$[[x \leftarrow \text{copy}(y)]](e, G) = (e[x \leftarrow v'], G'$ for some v' such that (v', G')
= new.term$(f(v_1 \cdots v_n), G)$.
$[x \leftarrow \text{copy}(y), (e, G)] = 1$. No partial order statement is needed here.

In the copy statement, the variable x is assigned a node v' distinct from v but having an identical term.

3.7.4. Argument Replacement

This statement is useful for implementing arrays efficiently.

Let $(v, S') = [[t]](e, G)$ and suppose $S' = (e, G')$. Let $S = (e, H)$ where H = Replace$[G', e(A), I, v]$. Then $[[A[i] \leftarrow t]](e, G) = (e, H)$. Here A is a program variable.

We informally speak of this statement as replacing the i^{th} argument of term$(e(A), G')$ with term(v, G'). We say that the argument replacement operation is of type (term$(e(A), G')$,term(v, G'), term$(e(A), H)$).

$[A[i] \leftarrow t, (e, G)] = [t, (e, G)] + 1$. For the partial order version, $[A[i] \leftarrow t, (e, G)] > [t, (e, G)]$.

3.7.5. Conditional Statements

$[[$if t then $\{P_1\}$ else $\{P_2\}]]S$ =
if term$(v, g(S'))$ = $\perp_u$ then S' else
if term$(v, g(S'))$ = true then $[[P_1]]S'$ else $[[P_2]]S'$
where $(v, S') = [[t]]S$.

[if t then $\{P_1\}$ else $\{P_2\}, S]$ = if term$(v, g(S'))$ = $\perp_u$ then $[t, S] + 1$ else if term$(v, g(S'))$ = true then $[t, S] + [P_1, S'] + 1$ else $[t, S] + [P_2, S'] + 1$ where $(v, S') = [[t]]S$.

For the partial order version, [if t then $\{P_1\}$ else $\{P_2\}, S] > [t, S]$, and also if term$(v, g(S'))$ = true then [if t then $\{P_1\}$ else $\{P_2\}, S] > [P_1, S']$, and if term$(v, g(S')) \not\equiv$ true then [if t then $\{P_1\}$ else $\{P_2\}, S] > [P_2, S']$. From now on the partial order version will be omitted.

In a conditional statement, first t is evaluated. If it returns undefined then the state S' that results from the evaluation of t is returned and neither P_1 nor P_2 is executed. If t returns true then P_1 is executed else P_2 is executed.

If the second part of the statement is omitted the semantics is simpler.
$[[$if t then $\{P_1\}]]S$ =
if term$(v, g(S'))$ = $\perp_u$ then S' else
if term$(v, g(S'))$ = true then $[[P_1]]S'$ else S'
where $(v, S') = [[t]]S$.
[if t then $\{P_1\}, S$]: similar to [if t then $\{P_1\}$ else $\{P_2\}, S$].

3.7.6. Iterative Statements

$[[$while t do $\{P\}]]S$ =
if (term$(v, g(S'))$ = $\perp_u$ then S' else
if term$(v, g(S')) \not\equiv$ true then S' else
$[[$while t do $P]]([[P]]S')$
where $(v, S') = [[t]]S$

[while t do $\{P\}, S]$ = if (term$(v, g(S'))$ = $\perp_u$ then $[t, S] + 1$ else if term$(v, g(S')) \not\equiv$ true then $[t, S] + 1$ else [while t do $\{P\}$,$[[P]]S'] + [t, S] + 1$ where $(v, S') = [[t]]S$.

Now term$(v, g(S'))$ is essentially the value resulting from evaluating t in state S and S' is the state resulting from the execution of t. If t evaluates to undefined or anything other than true, the loop exits in state S'. Otherwise the loop iterates on the state $[[P]](S')$ resulting from the execution of P in state S'.

for $i = j$ step k until n do $\{P\}$: Equivalent to the following:
$i \leftarrow j$;

 doi:10.20944/preprints202511.0214.v1

while $i \leq n$ do $\{P; i \leftarrow i + k\}$
One might define until in a similar style.

3.7.7. Sequence of Statements

For two imperative statements define $[[P_1; P_2]]S$ as $[[P_2]]([[P_1]](S))$.
$[(P_1; P_2), S] = [P_1, S] + [P_2, [[P_1]](S)] + 1$.

3.8. Nontermination

To handle nontermination it would be necessary to make use of the symbol $\perp_n$ and possibly some form of denotational semantics using complete partially ordered sets. Note that no imperative statement changes the set of nodes in the term graph.

3.9. Evaluating Functional Expressions

For functional expressions in general, we assume leftmost innermost evaluation so that when a term $f(t_1 \cdots t_n)$ is evaluated, the top-level subterms t_i will evaluate to nodes v_i of G that represent ground constructor terms. Computing $[[P]](e_0, G)$ where P is a functional expression, e_0 is an environment, and G is a term graph is done as follows.

If P is a variable x then $[[P]](e_0, G) = (e_0(x), (e_0, G))$ and $[P, (e_0, G)] = 1$.

If P is a term $f(t_1, \cdots, t_n)$ where the t_i are terms in $T(F, X)$ then let $S_0 = (e_0, G)$, let $(v_1, S_1) = [[t_1]]S_0$ where v_1 is a node in $g(S_1)$, let $(v_2, S_2) = [[t_2]]S_1$, ..., and let $(v_n, S_n) = [[t_n]]S_{n-1}$. All the S_i have the same environment e_0 and all the v_i are nodes of $g(S_i)$. The environment e_0 is needed to evaluate the t_i because they can contain program variables from the calling procedure.

If any of the t_i fail to terminate then the whole expression does too and the value is $\perp_n$; of course, it may not be possible to compute this value because of nontermination. We do not specify how $\perp_u$ or $\perp_{err}$ are handled if some t_i returns them.

Let S be the state with $\text{env}(S) = e_0$ and $g(S) = g(S_n)$.

Then $[[f(t_1 \cdots t_n)]](e_0, G)$ = finish.function$(f(v_1 \cdots v_n), S)$ where the v_i are as above and finish.function remains to be defined. So finish.function assumes the arguments of f have been evaluated left to right to obtain the nodes $v_1 \cdots v_n$ in the term graph and to obtain the state S. Then finish.function applies f to these nodes $v_1 \cdots v_n$ and to the resulting state S in a way that depends on what kind of a function f is. Finish.function returns a (node, state) pair. The notation $f(v_1 \cdots v_n)$ is convenient but the nodes v_i cannot actually appear as arguments to f in a functional expression in the language. This notation $f(v_1 \cdots v_n)$ could be more precisely written as $(f, v_1, \cdots, v_n)$. The nodes v_i of G may appear as children of a node of the form $v : f(v_1 \cdots v_n)$ eventually in the term graph. Also $[f(t_1 \cdots t_n), (e_0, G)] = [t_1, S_0] + [t_2, S_1] + \cdots + [t_n, Sn - 1]$+[finish.function$(f(v_1 \cdots v_n), S)] + 1$. We want to compute finish.function$(f(v_1 \cdots v_n), S)$ where S is the state with $\text{env}(S) = e_0$ and $g(S) = g(S_n)$ for various kinds of function symbols f. First it is necessary to define some auxiliary functions for rewriting.

None of these functional expressions directly change the term graph.

3.10. Rewriting Semantics

The use of term graphs makes pattern matching convenient. Therefore this language provides for pattern matching in a rewriting facility. The rewriting facility is not needed for Turing equivalence. The rewriting facility does one rewrite at the top level and then does recursive evaluation. The subterms are first evaluated left to right to normal forms. Recall that all normal forms are constructor terms and even if an implicit type error occurs, the normal form is $\perp_e$, which is also a constructor. This language does not have an explicit type system. This paper does not have a formal discussion of term rewriting systems. There is an extensive literature on this topic; for example, see [DJ90,BN98,DP01]. The evaluation of functions defined by rewriting is performed by the function graph.rewrite defined as follows. How this relates to finish.function is explained below.

 doi:10.20944/preprints202511.0214.v1

Suppose the function symbol f is defined by rewriting in a statement of the form

$$\text{rewrite } f\{r_1 \to s_1, \cdots, r_k \to s_k\}$$

in which the syntactic restrictions given earlier are obeyed. In particular the top symbols of the r_i must all be f, the r_i are linear, and all variables in s_i appear also in r_i.

Let E represent the expressioni $f(v_1 \cdots v_n)$. Then graph.rewrite$(E,(e,G))$ returns a (node, state) pair and is defined as

if match.list$(r_1,E,G) \neq \perp_e$ then $[[s_1]]$(match.list$(r_1,E,G),G)$ else if
match.list$(r_2,E,G) \neq \perp_e$ then $[[s_2]]$(match.list$(r_2,E,G),G)$ else … else if
match.list$(r_k,E,G) \neq \perp_e$ then $[[s_k]]$(match.list$(r_k,E,G),G)$ else $[[\perp_e]](e,G)$.
In this last case there is an implicit type error.
Also as for the termination measure, [graph.rewrite$(E,(e,G))$] =
if match.list$(r_1,E,G) \neq \perp_e$ then $1+$[s_1,(match.list$(r_1,E,G),G)$] else
if match.list$(r_2,E,G) \neq \perp_e$ then $1+$[s_2,(match.list$(r_2,E,G),G)$] else … else
if match.list$(r_k,E,G),G) \neq \perp_e$ then $1+$[s_k,(match.list$(r_k,E,G),G)$] else 1.
Match.list$(r, f'(v'_1,\cdots,v'_{n'}),G)$ returns an environment (a substitution) and is defined as follows, assuming r is $g(q_1,\cdots,q_m)$:
if $g \not\equiv f'$ then $\perp_e$ else if $g \equiv f'$ and $n' = m$ then
(if match$(q_1,v'_1,G) = \perp_e$ then $\perp_e$
else if match$(q_2,v'_2,G) = \perp_e$ then $\perp_e$ else $\cdots$
else if match$(q_m,v'_m,G) = \perp_e$ then $\perp_e$
else match$(q_1,v'_1,G) \cup \cdots \cup$ match(q_m,v'_m,G)).
match(r,v,G) is defined as follows:
if r is a variable then $\{r \to v\}$ [i.e. match$(r,v,G)(r) = v$] else match.list$(r, f'(v'_1 \cdots v'_n),G)$ where $v : f'(v'_1 \cdots v'_n)$ is in G.

None of these functions directly change the term graph. However evaluating $[[s_i]]$(match.list$(r_i,E,G),G)$ may change the term graph.

The basic idea for functions defined by a sequence of rewrite rules is to extract the terms term(v_i,G) from $v_1 \cdots v_n$„ do a top level rewrite on the term f(term(v_1,G), $\cdots$ term(v_n,G)) using the first applicable rewrite rule for f from the list $\{r_1 \to s_1,\cdots,r_k \to s_k\}$, and evaluate the resulting term recursively. The v_i are obtained from the terms t_i as indicated above by evaluating the t_i left to right. Here we are not concerned with confluence issues. The function graph.rewrite performs this rewrite and makes use of match.list to get a substitution (an environment) mapping the left hand side r of a rewrite rule to f(term(v_1,G), $\cdots$ term(v_n,G)). In evaluating match.list$(r, f(v_1,\cdots v_n),G)$, r cannot be a variable because the left hand side of a rewrite rule cannot be a variable. The function match.list does not modify G and does not make use of the current environment e. In the routine match.list we make use of the routine match(r,v,G) to match term r to node v in G. Here r is a subterm of the left-hand side of a rewrite rule. This routine match is similar to match.list but also permits the term r to be a variable. These definitions assume that r is left-linear so that the variables in the different q_i are distinct. Because of left linearity there should be no conflicts in the union of environments in match or match.list. Match (and therefore match.list) terminates even if G has cycles because we assume r is a finite cycle-free term (rewrite rules are finite).

3.11. Evaluation of Finish.Function

The evaluation of finish.function depends on what kind of a function is being specified.

3.11.1. Compiled Functions

If the function symbol f appears in a definition of the form

$$\text{compiled } f(x_1 \cdots x_n) \to y \text{ where } A[x_1 \cdots x_n, y];;$$

 doi:10.20944/preprints202511.0214.v1

then as mentioned earlier the instances of such a compiled function are expressions of the form $f(s_1 \cdots s_n) \rightarrow t$ where the s_i and t are finite ground (variable free) constructor terms and the assertion A holds, that is, $A[s_1 \cdots s_n, t]$ is a theorem of the appropriate logical theory. These instances can be considered as ground rewrite rules. Then the semantics are the same as if the compiled function were defined by the possibly infinite list of such rewrite rules. Alternatively, suppose t_i = term$(v_i, g(S))$. If the t_i are all finite constructor terms and there is one and only one constructor term t such that $A[t_1 \cdots t_n, t]$ is a theorem of the relevant logical theory, then finish.function$(f(v_1 \cdots v_n), S) = [[t]](S)$. Otherwise, finish.function$(f(v_1 \cdots v_n), S) = [[\perp_e]](S)$. Also as for the termination measure, [finish.function$(f(v_1 \cdots v_n), S)] = 1$.

For all non-logical function symbols in the underlying theory there is a corresponding compiled function. For example, there is a compiled function

$$\text{compiled } x_1 + x_2 \rightarrow y \text{ where } y = x_1 + x_2;;$$

The occurrence of + on the left is considered as an infix defined symbol in ITGL and the occurrence on the right is considered as a symbol of the underlying theory.

3.11.2. Constructors

If f is a constructor then finish.function$(f(v_1 \cdots v_n), (e, G)) = (v, (e, G'))$ where G' = make.term$(f(v_1 \cdots v_n), G)$ (as defined earlier) and [finish.function$(f(v_1 \cdots v_n), (e, G))] = 1$. Thus env$([[f(t_1 \cdots t_n)]](e, G)) = e$.

This is the place where the set of nodes in the term graph increases. This can happen in the function make.term, new.term, or cache.term.

3.11.3. Rewriting

If f is defined by rewriting then finish.function$(f(v_1 \cdots v_n), S)$ = graph.rewrite$(f(v_1, \cdots, v_n), S)$. Note that the environment e_0 is not needed for graph.rewrite because the v_i all represent ground terms and the variable case was covered above. So here also env$([[E]](e_0, G)) = e_0$ and [finish.function $(f(v_1 \cdots v_n), S))]$ = [graph.rewrite$(f(v_1 \cdots v_n), S)$] as given earlier.

3.11.4. Procedures

Suppose f is an imperative procedure defined by $f(x_1 \cdots x_n)\{B\}t;;$ where the x_i are the formal parameters.

Then finish.function$(f(v_1 \cdots v_n), S)$ is defined as follows where $S = (e_0, G)$: Let e_1 be defined by $e_1(x_i) = v_i$ and $e_1(x) = \perp_u$ for other program variables x. So no other values can be passed in to the procedure except by the formal parameters. Therefore free variables in the procedure do not pass any values in.

Let S_1' be fix.env(e_1, S).

Let S_2' be $[[B]]S_1'$ and note that B may change e_1.

Let (v_1', S_3') be $[[t]]S_2'$ and recall that $[[t]]$ does not change env(S_2').

Then finish.function$(f(v_1 \cdots v_n), (e_0, G)) = (v_1', (e_0, g(S_3')))$.

Later we will show that (e_0, G) is variable binding because e_0 only refers to vertices in $g(S_3')$.

[-25]Thus in this case also env$([[f(t_1 \cdots t_n)]](e_0, G)) = e_0$ so if program variables are changed during the evaluation of $f(t_1 \cdots t_n)$ they will be restored. Also [finish.function$(f(v_1 \cdots v_n), S)] = [B, S_1] + [t, S_2'] + 1$.

None of these functions directly modify G but the evaluation of B and t and the t_i in finish.function may modify G.

3.11.5. Special Symbols

We now define some special defined function symbols. For these the constants true, false, c_f, $\perp_{err}$, and integers i are considered as constructor functions with arity zero. Although top, equal.top,

 doi:10.20944/preprints202511.0214.v1

equal.node, arg, and arity are technically constructors, they will never appear in the term graph because during their evaluation by finish.function they are removed. The function symbol copy also cannot appear in the term graph though it is technically a constructor.

finish.function(**top**$(v), S)$ = finish.function(c_f, S) if label$[v, g(S)] = f$.

If f is an individual constant and label$[v, g(S)] = f$. then finish.function(**top**$(v), S)$ = finish.function (f, S).

finish.function(equal.top$(v_1, v_2), S)$ = if (label$[v_1, g(S)]$ = label$[v_2, g(S)]$) then finish.function (true,S) else finish.function (false, S).

finish.function(equal.node$(v_1, v_2), S)$ = if $(v_1 = v_2)$ then finish.function(true,S) else finish.function (false,S).

finish.function(**arg**$(v_1, v_2), S)$ = (v_3, S) if label$[v_1, g(S)] = i, i > 0$, **arity**$[v_2, g(S)] \geq i$, and arg$[i, v_1, g(S)] = v_3$.

finish.function(**arg**$(v_1, v_2), S)$ = finish.function$(\perp_{err}, S)$ if label$[v_1, g(S)] = i$, **arity**$[v_2, g(S)] < i$ or $i \leq 0$.

finish.function(**arity**$(v), S)$ = finish.function(i, S) if arity$[v, g(S)] = i$. We are assuming that integers are individual constants here.

In all these cases [finish.function$(f(v_1 \cdots v_n), S)$] = 1.

3.11.6. Array Initialization

It turns out that in $f(t \cdots t)$ all the t need not return the same value even though the environment is restored after each functional expression. This is because of the effect of argument replacement.

Consider this sequence:

procedure d $\{x \leftarrow h(1)\}g(f(x), f(x));;$

procedure $f(x)\{y \leftarrow$ arg$(1, x); x[1] \leftarrow$ arg$(1, x) + 1\}y;;$

Suppose g and h are constructors (because they are not defined). Then when d is called it will call g. The first argument of g will return $f(h(1))$ which will be arg$(1, h(1))$ or 1. The second argument of g is $f(x)$ but now x is bound to $f(h(2))$ because of the argument replacement statement in procedure f so the second argument of g will evaluate to 2. If instead $g(f(x), f(x), f(x))$ were called its arguments would evaluate to 1, 2, and 3, respectively.

4. Example Procedures

4.1. Example Imperative Procedures

For a rewrite procedure, consider a procedure concat for concatenating two lists in the LISP cons format so that concat(cons(a,cons(b,nil)),cons(c,cons(d,nil))) = cons(a,cons(b,cons(c,cons(d,nil)))). This can be done by the following rewrite procedure:

$$\text{rewrite concat } \{\text{concat}(\text{cons}(x, y), z) \rightarrow \text{ cons}(x,\text{concat}(y, z)), \text{ concat}(\text{nil},z) \rightarrow z\};;$$

Here cons, a, b, c, d, and nil are constructors. Suppose (e, G) is a state in which term$(e(x), G)$ = cons(a,cons(b,nil)) and term$(e(y), G)$ = cons(c,cons(d,nil)). Suppose $(v, (e', G'))$ = [[concat(x, y)]](e, G). Then $e' = e$ and term(v, G') = cons(a,cons(b,cons(c,cons(d,nil)))).

Here is an imperative procedure for insertion sort. For this we assume that the standard arithmetic operations are pre-defined as compiled functions.

procedure isort(x_1, x_2)
{for $j \leq 2$ step 1 until x_2 do
{key $\leftarrow$ **arg**(j, x_1); $i \leftarrow j - 1$;
while $i > 0$ and **arg**$(i, x_1) >$ key do
$\{x_1[i + 1] \leftarrow$ **arg**$(i, x_1); i \leftarrow i - 1\}$;
$x_1[i + 1] \leftarrow$ key$\}\}$
$x_1;;$

 doi:10.20944/preprints202511.0214.v1

Of course this program is more readable abbreviating **arg**(j, x_1) by x_1[j]. Assuming that (e, G) is a state in which term$(e(x), G) = f(a_1 \cdots a_n)$ for some constructor symbol f of arity n and $(v, (e', G'))$ = [[isort(x, n)]](e, G) then $e' = e$ and term(v, G') will be of the form $f(b_1 \cdots b_n)$ where $b_1 \cdots b_n$ is a sorted permutation of $a_1 \cdots a_n$. It does not seem difficult to automatically translate this program into an insertion sort program in many conventional imperative progamming languages.

Here is a procedure for the greatest common divisor:

procedure gcd(x_1, x_2)
{}
if $\{x_1 > x_2\}$ then $\{\text{gcd}(x_1 - x_2, x_2)\}$ else
$\{$ if $x_2 > x_1\}$ then $\{\text{gcd}(x_1, x_2 - x_1)\}$ else $\{x_1\}$;;

This definition seems close to the pseudo-code definitions given in algorithm texts. The initial {} indicates that this procedure does not have an initial section.

For compiled functions, assume a standard encoding of integers as individual constants that are constructors in the language. If Peano Arithmetic is the underlying theory then one can assume standard names for functions and predicates in the theory. In general one might let theories be taken from Isabelle or some other logical framework and use their names for symbols in the theory. As an example of a compiled function, to define double(x) with the specification $y = x + x$ one might have this procedure:

compiled double$(x) \to y$ where $y = x + x$;;

4.2. Top Level Functional Expressions

Recall that a program is a functional expression followed by a list of procedure definitions. As for input-output we can for simplicity assume that the input can be given by the functional expression at the top level of a program and the value of this expression can be returned to the user in some form. Of course, in an implementation some other means of input output could be specified. The initial state for evaluating the top level functional expression has a term graph with one node v with label $\perp_u$, an environment mapping all program variables to the node v, and any free variables in the functional expression also mapped to v by the environment. The functional expression will create a term graph G and return a node w in G. Then the term term(w, G) can be printed out for the user. If the term is infinite, them some finite representation of it can be printed. For example, the term $h(f(a, f(a, f(a, \cdots))))$ could be printed as $h(X)$ where $X = f(a, X)$. However, the language ITGL is intended to be translatable into other languages and in that case what matters is not the functional expression at the top level but the procedure definitions that can be used in the target language. The discussion of such translations is beyond the scope of this paper.

Here is an example of a program in the language ITGL:

$f(3, 5)$;;
procedure $f(x, y)\{x \leftarrow \text{add1}(x); y \leftarrow \text{add1}(y)\}x + y$;;
compiled add1$(x) \to y$ where $y = x + 1$;;.

This program would compute the value 10.

5. Proofs of Properties

5.1. Term Graph Only Gets Larger

We now show that in the execution of a program the term graph only increases in size. The basic idea is that the only function that directly change the set of nodes in the term graph is new.term$(f(v_1 \cdots v_n), G)$ which is called by finish.function$(f(v_1 \cdots v_n), S)$, and by statements of the form $x \leftarrow copy(y)$.

Theorem 1. *(The Subset Property). If P is a functional expression and $(v, S_2) = [[P]]S_1$ then nodes$(S_1) \subseteq$ nodes(S_2). Also, if P is an imperative statement and $S_2 = [[P]]S_1$ then nodes$(S_1) \subseteq$ nodes(S_2). Recall that nodes$((e, G))$ is defined as nodes(G).*

 doi:10.20944/preprints202511.0214.v1

Proof. We can assume by induction that both parts of the theorem are true on pairs $[P', S_1']$ smaller than $[P, S_1]$ in the ordering and show they are true for the pair $[P, S_1]$. We give a few examples.

Suppose P is the simple assignment statement $x \leftarrow t$ where t is a functional expression. Then $[[x \leftarrow t]]S_1 = \text{fix.env}(\text{env}(S_1)[x \leftarrow v], S_3)$ where $(v, S_3) = [[t]]S_1$. By induction, $\text{nodes}(S_1) \subseteq \text{nodes}(S_3)$. Because fix.env does not change the set of nodes, $\text{nodes}(S_2) = \text{nodes}(S_3)$. Therefore $\text{nodes}(S_1) \subseteq \text{nodes}(S_2)$.

The proof for argument replacement is similar. Suppose $S_2 = [[A[i] \leftarrow t]]S_1$. Let $(v, S_3) = [[t]]S_1$. Then by the definition of argument replacement, if $S_3 = (e, G')$ then $S_2 = (v, (e, \text{Replace}\,[G', (A), I, v]))$. Now $\text{nodes}(S_2) = \text{nodes}(G')$ by definition of Replace, $\text{nodes}(S_3) = \text{nodes}(G')$ by definition of nodes on states, so $\text{nodes}(S_2) = \text{nodes}(S_3)$, and reasoning as above $\text{nodes}(S_1) \subseteq \text{nodes}(S_3)$ so $\text{nodes}(S_1) \subseteq \text{nodes}(S_2)$.

Suppose P is the copy statement $x \leftarrow \text{copy}(y)$. Let $S_1 = (e, G)$ and $S_2 = [[P]](e, G)$, then $S_2 = (e[x \leftarrow v'], G')$ for some v' such that $(v', G') = \text{new.term}(f(v_1 \cdots v_n), G)$. Now $\text{new.term}(f(v_1 \cdots v_n), G)$ is a pair (v, G') where v is a new node not present in G and G' is $G \cup \{v : f(v_1 \cdots v_n)\}$. So $\text{nodes}(G) \subseteq \text{nodes}(G')$. Since $\text{nodes}(G') = \text{nodes}(S_2)$, $\text{nodes}(S_1) \subseteq \text{nodes}(S_2)$.

In general for imperative statements P_1 and P_2, if $S_2 = [[P_1; P_2]](S_2)$ then $S_2 = [[P_2]](S_3)$ where $S_3 = [[P_1]](S_2)$. Assuming by induction that $\text{nodes}(S_1) \subseteq \text{nodes}(S_3)$ and $\text{nodes}(S_3) \subseteq \text{nodes}(S_2)$ one obtains that $\text{nodes}(S_1) \subseteq \text{nodes}(S_2)$. Many imperative statements can be expressed as compositions in this way so that the subset property can be proved by induction.

As for functional statements, let $S_2 = [[f(t_1 \cdots t_n)]](e_0, G) = \text{finish.function}(f(v_1 \cdots v_n), S)$ where the v_i are as in Section 3.9 and obtained by evaluation of t_1 through t_n in order. It follows by induction on properties of the t_i that $\text{nodes}(G) \subseteq \text{nodes}(S)$. We want to show that $\text{nodes}(G) \subseteq \text{nodes}(S_2)$. For this it suffices to show that $\text{nodes}(S) \subseteq \text{nodes}(S_2) = \text{finish.function}(f(v_1 \cdots v_n), S)$.

If f is a constructor then finish.function will call make.term which may add a node to $\text{nodes}(S)$ or leave $\text{nodes}(S)$ unchanged. In either case the desired result follows.

If f is a procedure definition then it can be expressed as a composition of simpler statements for which the subset property can be assumed by induction. By simple reasoning one obtains $\text{nodes}(S) \subseteq \text{nodes}(S_2)$.

If f is a procedure defined by rewriting then in the evaluation a rewrite rule is chosen resulting in a functional expression to evaluate and the result can be assumed by induction.

If f is a compiled function then $\text{finish.function}(f(v_1 \cdots v_n), S) = [[t]]S$ for some functional expression t for which the desired subset result can be assumed by induction.

This completes the proof sketch. □

5.2. Functional Expressions Do Not Change the Environment

Theorem 2. *If P is a functional expression and $(e', G') = [[T]](e, G)$ then $e' = e$.*

Proof. The only place where the environment is directly changed is in a simple assignment statement, a copy statement, and on entering an imperative procedure. Assignment statements and copy statements only occur in imperative procedures. At the end of an imperative procedure the original environment is restored. □

5.3. Variable Binding

Definition 1. *An imperative statement P* preserves variable binding *if the following is true: For all states S, if S is variable binding and T = [[P]](S) then T is also variable binding. Also a functional expression P* preserves variable binding *if the following is true: If S is variable binding and (v,T) = [[P]](S) then T is also variable binding.*

Lemma 1. *If P and Q are imperative statements that preserve variable binding then P; Q also preserves variable binding.*

 doi:10.20944/preprints202511.0214.v1

Proof. Let $T = [[P;Q]](S)$. Suppose $U = [[P]](S)$ and then $T = [[Q]]U$. Suppose S is variable binding. Because P preserves variable binding, U is also variable binding. Because Q preserves variable binding, T is also variable binding. □

Theorem 3. *If P is an imperative statement then P preserves variable binding. Also, if P is a functional expression, then P preserves variable binding.*

Proof. Using the subset property. The environment may be changed in a simple assignment statement $x \leftarrow t$. However only the binding of x is changed and it is changed to a node v in the term graph where the evaluation of t returns the pair (v, G) with v in G so variable binding is preserved.

The environment may be changed in a copy statement $x \leftarrow \text{copy}(y)$ but then $e(x) = v$ where v is a new node in G such that term(v, G) = term(y, G). So because v is in G variable binding is preserved. The environment is changed on entering a procedure so that for a formal parameter x, $e(x) = v$ for v such that $(v, G) = [[t_i]]$ for some term t_i. Evaluation of succeeding t_j only makes nodes(G) larger so at the end of the evaluation of the actual parameters, $e(x) = v$ and v is still a node of the term graph, so variable binding is preserved. For an imperative procedure definition, the original environment is put back at the end. This preserves variable binding. In particular, we recall the semantics of an imperative procedure definition:

$S = (e_0, G)$ is variable binding at the start of finish.function.

Let S_1' be fix.env(e_1, S).

Let S_2' be $[[B]]S_1'$ and note that B may change e_1.

Let (v_1', S_3') be $[[t]]S_2'$ and recall that $[[t]]$ does not change env(S_2') . Then finish.function$(f(v_1 \cdots v_n), (e_0, G)) = (v_1', (e_0, g(S_3')))$.

Now we want to show that the final state $(e_0, g(S_3'))$ is variable binding. First, fix.env(e_1, S) is variable binding because e_1 maps the formal parameters to nodes of the term graph at the calling point. By induction we can assume S_2' is variable binding and then again by induction S_3' is variable binding and v_1' is in the nodes of $g(S_3')$. Now nodes$(G) \subseteq$ nodes$(g(S_3')$ by repeated applications of the subset property and induction. Because (e_0, G) is variable binding and nodes$(G) \subseteq$ nodes$(g(S_3'))$, $(e_0, g(S_3'))$ is also variable binding and v_1' is in the nodes of $g(S_3')$. □

The fact that ITGL preserves variable binding means that there will be no dangling pointers.

5.4. Semantics Depends Not Only on Terms

Consider this program:

$f(g(a), g(a));;$

procedure $f(x, y)$

$\{x[1] \leftarrow b\}$

$y;;$. [This is part of the procedure definition and indicates the value to be returned]

Now if both $g(a)$ occurrences are at the same node then this program will return $g(b)$ when $f(g(a), g(a))$ is called. If the occurrences are at different nodes of the term graph then it will return $g(a)$. The effect of an argument replacement operation depends not only on the terms but on how they are stored. We next discuss more fully when the result of a program depends only on the terms, and if not, what it does depend on.

5.5. Dependence on Terms if No Argument Replacement or Equal.Node Operations

For each imperative statement or functional expression P we have $[[P]]$ as a function of the state.

Under suitable assumptions including no argument replacement statements and no test for equality of nodes in a program we want to show the following:

Definition 2. *We say that the functional expression P and the function $[[P]]$ are* term dependent, *if for all states (e, G), if $(v', (e', G')) = [[P]](e, G)$ then term(v', G') and term$(e'(x), G')$ for various variables x are functions of term$(e(x), G)$ for various x. More precisely, suppose that for all e_1, G_1, e_2, G_2, v_1', e_1', G_1', v_2',*

 doi:10.20944/preprints202511.0214.v1

e'_2, G'_2, if $\{(v'_1,(e'_1,G'_1)) = [[P]](e_1,G_1)$ and $(v'_2,(e'_2,G'_2)) = [[P]](e_2,G_2)$ and for all x, $term(e_1(x),G_1) = term(e_2(x),G_2)\}$ then $\{term(v'_1,G'_1) = term(v'_2,G'_2)$ and for all x, $term(e'_1(x),G'_1) = term(e'_2(x),G'_2)\}$. This means that the value of the functional expression depends only on the terms of the variables in it. In fact in this case $e'_1 = e_1$ and $e'_2 = e_2$.

Similarly, we say that the imperative statement P and the function $[[P]]$ are term dependent, *if for all states (e,G) and (e',G'), if $(e',G') = [[P]](e,G)$ then the terms $term(e'(x),G')$ for various x are functions of the terms $term(e(x),G)$ for various x.*

We will show that this is true if there are no equal.node tests and no argument replacement statements, that is, statements of the form $x[y] \leftarrow z$ in the program in which P appears. Also one has to be careful about make.term, new.term, and cache.term as will be explained.

Recall that we extend the function term(v,G) to states (e,G) by term$(v.(e.G))$ = term(v,G).

Theorem 4. *Suppose functional expression t is evaluated twice in a program P which is term dependent and contains no argument replacement statements. Suppose that the first evaluation is in a state (e,G_1) and the second is in a state (e,G_2) with the same environment. Then if $(v_1,S_1) = [[t]](e,G_1)$ and $(v_2,S_2) = [[t]](e,G_2)$ then $term(v_1,S_1) = term(v_2,S_2)$.*

Proof. By the subset property, nodes$(G_1) \subseteq$ nodes(G_2). Also, for all nodes v in G_1, the children of v in G_1 will be the same as the children of v in G_2 because the program contains no argument replacement statements. Therefore for all variables x, term(x,G_1) = term(x,G_2). Then by induction on the term structure of t, using the term dependence property, both evaluations of t yield the same result, so term(v_1,S_1) = term(v_2,S_2). □

This implies something about the evaluation of $f[n](t\cdots t)$ if the program is term dependent and contains no argument replacement statements. In particular, all evaluations of t will yield the same result. So suppose the successive evaluations of t yield states $(v_1,G_1),\cdots(v_n,G_n)$ respectively. Then nodes$(G_1) \subseteq$ nodes$(G_2) \subseteq \cdots \subseteq$ nodes(G_n) so by the theorem term(v_1,G_1) = term$(v_2,G_2) = \cdots =$ term(v_n,G_n). This essentially means that all arguments of $f[n]$ will be identical after evaluation of the arguments of $f[n](t\cdots t)$.

5.6. Isomorphism Equivalence Classes

Even if there are argument replacement statements or tests for equality of nodes, we can say something about how the output of a program depends on the input, using the idea of the isomorphism equivalence class of a state.

Definition 3. *An* isomorphism *between term graphs G_1 and G_2 is a one to one onto mapping ϕ from the nodes of G_1 to G_2 such that for all v in G_1, $label[\phi(v),G_2] = label[v,G_1]$, $arity[\phi(v),G_2] = arity[v,G_1]$, and if $(v_1,\cdots,v_n)$ are the children of v in G_1 then $(\phi(v_1),\cdots,\phi(v_n))$ are the children of $\phi(v)$ in G_2. That is, if $args[v,G_1] = (v_1,\cdots,v_n)$ then $args[\phi(v),G_2] = (\phi(v_1),\cdots,\phi(v_n))$. In this case we write that $G_2 = \phi(G_1)$. Two graphs are* isomorphic *and in the same* isomorphism equivalence class *if there is an isomorphism between them.*

Note that the composition of two isomorphisms is an isomorphism, the identity mapping is an isomorphism, and the inverse of an isomorphism is an isomorphism. The idea of isomorphism equivalence is that it specifies which terms and subterms are stored at the same node.

Definition 4. *ϕ is an* isomorphism *from a state $S = (e,G)$ to state $S' = (e',G')$ if ϕ is an isomorphism from S to S' and also for all variables x, $\phi(e(x)) = e'(x)$. We write $S' = \phi(S)$. If there is an isomorphism from S to S' then we say that the states S and S' are* isomorphic. *Also for a state S, $[S]$ is the set of states that are isomorphic to it and this is called the* isomorphism equivalence class *of S.*

 doi:10.20944/preprints202511.0214.v1

Now, at any time all but finitely many program variables will be undefined. So for undefined variables x, if $\phi(e(x)) = e'(x)$, suppose $e(x) = v$ and $e'(x) = v'$. Then $\phi(v) = v'$. Also, as for the labels of v and v', $l[v, G] = l[v', G'] = \perp_u$ in this case.

Note that if $(e', G') = \phi((e, G))$ and ϕ is an isomorphism then for all program variables x, $\text{term}(e'(x), G') = \text{term}(e(x), G)$.

Definition 5. *ϕ is an isomorphism from a pair $(v, (e, G))$ to $(v', (e', G'))$ if $\phi(v) = v'$ and also ϕ is an isomorphism from (e, G) to (e', G'). We say that $(v, (e, G))$ and $(v', (e', G'))$ are isomorphic. This implies that $term(v, G) = term(v', G')$. Also $[(v, (e, G))]$ is the set of node, state pairs that are isomorphic to $(v, (e, G))$. This is called the isomorphism equivalence class of $(v, (e, G))$.*

Definition 6. *ϕ is an isomorphism from a pair (v, G) with v in G to (v', G') with v' in G' if ϕ is an isomorphism from G to G' and also $\phi(v) = v'$.*

We will show that isomorphic states map to isomorphic states (by an imperative statement) and isomorphic states map to isomorphic (value,state) pairs (by a functional statement) even if there are equal.node tests and statements of the form $x[y] \leftarrow z$ in the program.

5.7. Isomorphism (Equivalence Class) Dependence

Definition 7. *A function F from states to states is* equivalence class (e.c.) or isomorphism dependent *if there is a function $F^{[]}$ such that $[F(S)] = F^{[]}([S])$. The same definition works for functions from states to (node, state pairs) so this definition applies both to imperative statements and functional expressions. If $[[P]]$ is e.c. dependent for an imperative statement or functional expression P then we say that P is* e.c. or isomorphism dependent.

Suppose F is such a function $[[P]]$ for some P. If F is isomorphism dependent then the isomorphism class of the result of a program is a function only of the isomorphism class of the input, and doesn't directly depend on the names of the nodes.

Why does this matter? It means that if one is only concerned about the isomorphism class of the output then one need only know the isomorphism class of the input. Also one can write a specification of the program that only specifies how the output class depends on the input class, which may simplify things.

Definition 8. *Define $S_1 \cong_i S_2$ to mean there is an isomorphism from S_1 to S_2 (or S_2 to S_1). Recall that the states S_1 and S_2 are said to be isomorphic. Also $G_1 \cong_i G_2$ if there is an isomorphism from graph G_1 to graph G_2.*

Also $(v_1 \cdots v_n, S) \cong_i (w_1, \cdots, w_n, T)$ if there is an isomorphism from S to T that also maps v_i to w_i for all i. Similarly define $(v_1 \cdots v_n, G) \cong_i (w_1, \cdots, w_n, H)$ for graphs G and H.

Note $S_1 \cong_i S_2$ implies $(e_1(x), S_1) \cong_i (e_2(x), S_2)$ for all program variables x.

Proposition 1. *A function $F = [[P]]$ from states to states is isomorphism dependent if for all states S and T, $S \cong_i T$ implies $F(S) \cong_i F(T)$. A function F from states to (node,state) pairs is isomorphism dependent if for all (node, state) pairs (v, S) and (w, T), $S \cong_i T$ implies $F(S) \cong_i F(T)$.*

In fact we will show that if ϕ is an isomorphism from S to T then there is an isomorphism ϕ' from $F(S)$ to $F(T)$ that extends ϕ, in the sense that if $\phi(v) = w$ for a node v in the term graph of S then v is also in the term graph of $F(S)$ and $\phi'(v) = w$ also. We know that v will also be in the term graph of $F(S)$ by Theorem 1.

Definition 9. *Such a function $F = [[P]]$* satisfies the extension property *if F is isomorphism dependent and in addition if for all states S and T, if ϕ is an isomorphism from S to T then there is an isomorphism ϕ^* from $F(S)$ to $F(T)$ which extends ϕ.*

 doi:10.20944/preprints202511.0214.v1

5.8. Term Dependence

Definition 10. *We can define the same notation $\cong_t$ for terms. In the sense of term equivalence $(e_1, G_1) \cong_t (e_2, G_2)$ means that for all variables x $term(e_1(x), G_1) = term(e_2(x), G_2)$. Then (e_1, G_1) and (e_2, G_2) are said to be* term equivalent. *Also $(v_1 \cdots v_n, S) \cong_t (w_1, \cdots, w_n, T)$ in the sense of term dependence means that $S \cong_t T$ and that $term(v_i, S) = term(w_i, T)$ for all i. This applies in particular to (node,state) pairs that result from the execution of a functional expression. We can also define the* term equivalence class $[S]$ *of a state as the set of states that are term equivalent to it, and define term dependence if there is a function $F^{[]}$ such that $[F(S)] = F^{[]}([S])$ in the sense of term equivalence. This definition of term dependence also extends to (node, state) pairs as before.*

Proposition 2. *A function F from states to states is term dependent if for all states S and T, $S \cong_t T$ implies $F(S) \cong_t F(T)$. A function F from states to (node, state) pairs is term dependent if for all states S, $S \cong_t T$ implies $F(S) \cong_t F(T)$.*

As before, if $[[P]]$ is term dependent then we also say that P is term dependent. For two states S_1 and S_2, if $S_1 \cong S_2$ (isomorphism) then $S_1 \cong S_2$ (term). Thus if two states are isomorphic then they are term equivalent. The converse is not true. However, term dependence does not imply isomorphism dependence. The functions make.term and cache.term are term dependent but depending on the implementation they may not be isomorphism dependent. For example, if their implementation depends on the names of the nodes, then they may not be isomorphism dependent even if they are deterministic. Isomorphism dependence does not imply term dependence. The function equal.node and statements of the form $x[y] \leftarrow z$ are isomorphism dependent but not term dependent. We can distinguish the two senses of $\cong$ as $\cong_i$ and $\cong_t$ when necessary.

The proofs for isomorphism dependence and term dependence are generally so similar that we do both together in most cases. Of course the proof for term dependence assumes the lack of argument replacement statements and node equality tests. Both proofs are done by induction on the measure $[F, S]$ that defines the length of a terminating computation. That is, we assume that the theorems are true for all $[F', S']$ such that $[F', S'] < [F, S]$ and prove that the theorem is true for F and S. For terminating computations $[[F]](S)$, this shows that the theorem is true for all F and S.

5.9. Functional Expressions: Term and Isomorphism Dependence

All functions make use of finish.function so to show that functional expressions are term dependent or isomorphism dependent we first show that the prelude to finish.function is term dependent or isomorphism dependent, and then it only remains to show that finish.function itself is term or isomorphism dependent.

5.10. Finish.Function: Term and Isomorphism Dependence

Here are the details of the proof of term dependence for finish.function: For this we define finish.function in a different but equivalent manner to that given in Section 3.9. The basic idea is that pre.function computes and saves the nodes v_i that represent the values of the terms $t_1 \cdots t_n$ which are arguments to a function symbol f. This is done in a left to right manner. Also the state S resulting from the computation of these terms is saved.

Definition 11. *Define the* prelude pre.function *to finish.function as follows. If $t_1 \cdots t_n$ are terms in $T(F, X)$ then let $S_0 = (e_0, G)$, let $(v_1, S_1) = [[t_1]]S_0$ where v_1 is a node in $g(S_1)$, let $(v_2, S_2) = [[t_2]]S_1, \cdots$, and let $(v_n, S_n) = [[t_n]]S_{n-1}$. Let S be the state with $env(S) = e_0$ and $g(S) = g(S_n)$. All the S_i have the same environment e_0 and all the v_i are nodes of $g(S_i)$. Then the value of pre.function$(t_1 \cdots t_n, (e_0, G_0))$ is the tuple $(v_1, \cdots, v_n, S)$.*

 doi:10.20944/preprints202511.0214.v1

Proposition 3. *If* $(v_1, v_2, \cdots, v_n, S) = pre.function(t_1, \cdots, t_n, (e_0, G))$ *then* $[[f(t_1 \cdots t_n)]](e_0, G) = finish.function(f(v_1 \cdots v_n), S)$ *where the definition of finish.function depends on* f. *The term* $f(v_1 \cdots v_n)$ *could more precisely be written as* $(f, v_1, \cdots, v_n)$.

Proof. By the discussion of finish.function in Section 3.9. □

Lemma 2. *For now letting* $\cong_t$ *refer to terms, if* $(e_0, G) \cong_t (e'_0, G')$ *and* $(v_1, v_2, \cdots, v_n, S) = pre.function(t_1, \cdots, t_n, (e_0, G))$ *and* $(v'_1, v'_2, \cdots v'_n, S') = pre.function(t_1, \cdots, t_n, (e'_0, G'))$ *then* $(v_1, \cdots, v_n, S) \cong_t (v'_1, \cdots, v'_n, S')$. *Also if* $\cong_i$ *for isomorphisms is written instead of* $\cong_t$ *then the isomorphism for* $(v_1, \cdots, v_n, S) \cong_i (v'_1, \cdots, v'_n, S')$ *is an extension of the isomorphism for* $(e_0, G) \cong_i (e'_0, G')$.

Proof. Assuming the $[[t_i]]$ are term dependent, then for $1 \leq i \leq n$, because $(v_i, S_i) = [[t_i]]S_{i-1}$, and $(v'_i, S'i) = [[t_i]]S'_{i-1}$, $(v_1, \cdots v_{i-1}, S_{i-1}) \cong_t (v'_1, \cdots, v'_{i-1}, S'_{i-1})$ implies $(v_1, \cdots v_i, S_i) \cong_t (v'_1, \cdots, v'_i, S'_i)$ so putting all these together and noting that $(e_0, G) \cong_t (e'_0, G')$, $(v_1, \cdots, v_n, S_n) \cong_t (v'_1, \cdots, v'_n, S'_n)$. A similar argument works for isomorphism dependence using the fact that the t_i are isomorphism dependent, which can be assumed by induction. Also by induction on the terms t_i and assuming that the t_i satisfy the extension property we can assume that each isomorphism is an extension of the last one so the second part of the lemma holds as well. □

Corollary 1. *Suppose that* $(v_1, \cdots, v_n, S) \cong_t (v'_1, \cdots, v'_n, S')$ *implies* $finish.function(f(v_1 \cdots v_n), S) \cong_t finish.function(f(v'_1, \cdots, v'_n), S')$. *Then* $[[f(t_1 \cdots t_n)]]$ *is term dependent. Similarly for isomorphism dependence.*

Proof. Suppose $(e_0, G) \cong_t (e'_0, G')$, pre.function$(t_1 \cdots t_n, (e_0, G)) = (v_1 \cdots v_n, S)$ and pre.function $(t_1, \cdots t_n, (e'_0, G')) = (v'_1, \cdots, v'_n, S')$. By the lemma, $(v_1, \cdots, v_n, S) \cong_t (v'_1, \cdots, v'_n, S')$. We are assuming that $(v_1, \cdots, v_n, S) \cong_t (v'_1, \cdots, v'_n, S')$ implies finish.function$(f(v_1 \cdots v_n), S) \cong_t$ finish.function$(f(v'_1, \cdots, v'_n), S')$. Putting these together, finish.function$(f(v_1 \cdots v_n), S) \cong_t$ finish.function$(f(v'_1, \cdots, v'_n), S')$. By Proposition 3, $[[f(t_1 \cdots t_n)]](e0, G) \cong_t [[f(t_1 \cdots t_n)]]\ (e'_0, G')$. Thus $[[f(t_1 \cdots t_n)]]$ is term dependent. Similarly for isomorphism dependence. □

This corollary simplifies proofs of properties of functions in the language. This approach works for both term dependence and isomorphism dependence. These properties need to be shown separately for f being a constructor, a defined procedure, a compiled procedure, and a procedure defined by rewriting. Also it is necessary to look at special functions.

Corollary 2. *Suppose that if* ϕ *is an isomorphism from* $(v_1, \cdots, v_n, S)$ *to* $(v'_1, \cdots, v'_n, S')$ *then there is an isomorphism* ϕ^* *from* $finish.function(f(v_1 \cdots v_n), S)$ *to* $finish.function(f(v'_1, \cdots, v'_n), S')$ *that extends* ϕ. *Then* $[[f(t_1 \cdots t_n)]]$ *satisfies the extension property.*

Proof. By the lemma and using the hypothesis about finish.function. □

6. Constructors: Isomorphism Dependence: Make.Term, New.Term Cache.Term

6.1. Problems

Now if f is a constructor then to compute $f(t_1, \cdots, t_n)$, finish.function makes use of new.term, make.term, or cache.term. In particular, in this case finish.function$(f(v_1 \cdots v_n), (e, G))$ is defined as $(v, (e, \{v\} \cup G))$ where $(v, G \cup \{v\})$ is make.term$(f(v_1 \cdots v_n), G)$ and make.term may call new.term or cache.term. Recall that the functions new.term, make.term, and cache.term are not in the language but are auxiliary functions used to define the semantics of the language. These functions do not cause problems for term dependence assuming no argument replacement or node equality tests, but finish.function may not be isomorphism dependent, depending on their implementation. So the problem here is the possible nondeterminism of these functions; different implementations starting

 doi:10.20944/preprints202511.0214.v1

from the same state may have different outcomes and result in states that are not isomorphism equivalent.

Recall the definition of new.term$(f(v_1 \cdots v_n), G)$ from Section 3.1 where the v_i are nodes, G is a term graph, and f is a constructor. If X is $f(v_1 \cdots v_n)$ for v_i in G and a constructor f then new.term(X, G) is isomorphism dependent, but this is not always true for make.term and cache.term.

A problem with always calling new.term is that if all calls from finish.function for constructors are to new.term instead of make.term and cache.term then this implementation can be highly storage inefficient because each call creates a new node in the term graph, a node with n pointers if the label of the node has n children. We want to find a way to preserve isomorphism dependence with a more storage economical strategy. Garbage collection on the term graph would also help to reduce storage usage. Also for a loop if the index goes from 1 to 10,000 then all the integers from 1 to 10,000 will be stored as terms in the term graph. For each such integer i there will be a node v_i in the term graph having label i. So some kind of garbage collection will be helpful. We do not consider this aspect of storage economization in this paper.

Now make.term$(f(v_1 \cdots v_n), G)$ is a pair (v, G') where v is a node and G' is $G \cup \{v : f(v_1 \ldots v_n)\}$. Here f is a constructor. Depending on the implementation, the node v can be a new node, not present in G, or else it can be an existing node v of G if there is a node $v : f(v_1 \cdots v_n)$ that already is an element of G. Also new.term$(f(v_1 \cdots v_n), G)$ is a pair (v, G') where v is a new node not present in G and G' is $G \cup \{v : f(v_1 \cdots v_n)\}$. Cache.term chooses v to be an existing node of G if a node $v : f(v_1 \cdots v_n)$ in G already exists. If more than one such node exists it chooses one of them arbitrarily.

If c is a zero-ary individual constant, then starting from an empty graph, two calls (v, G) = make.term$(c, \{\})$ and (v', G') = make.term(c, G) in succession can produce either a graph $G' = \{v : c, v' : c\}$ or a graph $G' = \{v : c\}$ depending on whether make.term is implemented as new.term or cache.term. These graphs are not isomorphic. Suppose S is $(e, G) = (\{x \to v_1, y \to v_2\}, \{v_1 : c, v_2 : c\})$ where x and y are distinct variables. Then a call to cache.term(c, G) can produce either (v_1, S) or (v_2, S). These pairs are not isomorphic because of the condition that $\phi(e(z)) = e'(z)$ for variables z. An isomorphism from (v_1, S) to (v_2, S) would map v_1 to v_2, which would thereby map $e(x)$ to $e(y)$, not permitted in an isomorphism. However the states S' that can be produced by a call to new.term on any state S are all isomorphic.

If cache.term is modified so that all nodes with the same label and same children are made identical then cache.term will map isomorphic graphs to isomorphic graphs. It also will be storage efficient and will produce the same set $\{\text{term}(v, G) : v \text{ in } G\}$ as if new.term were called even though the resulting graph will not always be isomorphic to one produced by new.term. However if cache.term$(f(v'), G)$ is implemented this way and called on $(e, G) = (\{x \to v_1, y \to v_2\}, \{v1 : f(v'), v2 : f(v'), v' : c\})$ it will produce the pair $(v, (\{x \to v, y \to v\}, \{v : f(v'), v' : c\}))$ which identifies v_1 and v_2. This may have unintended consequences because now replacing an argument of $e(x)$ using a statement $x[i] \leftarrow z$ will do the same to y. For function symbols with large arity a call to cache.term can be expensive because it will take time proportional to the arity of the function.

6.2. Solutions

Now make.term can call either cache.term or new.term. We want to find a systematic way of deciding when to do cache.term and when to do new.term, and it should be storage efficient. We want also to guarantee isomorphism dependence preservation. To this end we define term$(f(v_1 \cdots v_n), G)$ for v_i in G to be the term $f(\text{term}(v_1, G), \cdots, \text{term}(v_n, G))$, that is, the term t such that **atop**$(t) = f$ and **aarg**(i, t) = term(v_i, G). Then we specify a set $\mathcal{T}$ of finite ground constructor terms that is closed under the subterm relation, and when finish.function$(f(v_1 \cdots v_n), G)$ is called for a constructor f, if term$(f(v_1 \cdots v_n), G)$ is in $\mathcal{T}$ then cache.term is called, otherwise new.term is called. We also specify that $\mathcal{T}$ contains all constant (zero arity) constructor symbols. $\mathcal{T}$ would typically contain small terms or terms such that **atop**(t) has small arity. This saves some storage but still is term dependent and isomorphism dependent. If $\mathcal{T}$ is properly defined, in a manner to be described, it avoids the need to copy a lot of structure when an argument replacement is done on a variable x where term$(e(x), G)$ is

 doi:10.20944/preprints202511.0214.v1

not in $\mathcal{T}$. A possible way to define $\mathcal{T}$ is to have a set of function symbols that can appear in terms in $\mathcal{T}$, and a term is only in $\mathcal{T}$ if it is finite and all function symbols in it are in this set.

It helps to hash terms in $\mathcal{T}$ to detect duplicates. If two terms have the same hash value then a more precise test can be used to test if they are identical. Finite terms can be hashed in a variety of ways. As an example of a hash function on finite terms that depends without much extra work on the hash values of the children, we can define a hash function $H(t)$ on terms t as follows:

Choose a prime p and choose $H(f)$ for function symbol f to be an integer between 1 and $p-1$. Let $H(f(t_1 \cdots t_n))$ be $(H(f) + H(t_1) + \ldots + H(t_n))$ mod p where the $H(t_i)$ are defined similarly in a recursive manner. This is not necessarily the best hash function but it works. Because of the subterm condition on $\mathcal{T}$ we can assume that the $H(t_i)$ have already been computed when $f(t_1 \cdots t_n)$ is evaluated. Suppose cache.term$(f(v_1 \cdots v_n), G)$ is called during the evaluation of finish.function for $f(t_1 \cdots t_n)$. Caching requires one to know if a node w with term(w, G) identical to term$(f(v_1 \cdots v_n), G)$ already exists in G. We can assume that there are no other nodes $w_i \not\equiv v_i$ in G with term(v_i, G) = term(w_i, G) because subterms are evaluated and cached before terms. If there is a node w with term(w, G) syntactically equal to term$(f(v_1 \cdots v_n), G)$ then the children of w will be $(v_1 \cdots v_n)$. To find such a node w if one exists one only has to look at the nodes w' such that $H(w') = H(t)$ where t is term$(f(v_1 \cdots v_n), G)$ and if these are equal then one checks that the function symbol of w' is f and that the children of w' are $(v_1 \cdots v_n)$. It is not necessary to examine the entire term t. And clearly if the function symbol of w' is f and the children of w' are $(v_1 \cdots v_n)$ then term(w', G) = term$(f(v_1 \cdots v_n), G)$. The point is that there should not be two nodes v_1 and v_2 in G such that term(v_1, G) = term(v_2, G) and term(v_1, G) is in $\mathcal{T}$. Instead there should be only one such node. This is for the purpose of economizing the use of storage as much as possible. This kind of a hash function requires time linear in the number of children of a node because each child at least has to be examined, and the test for w' having the same children $(v_1 \cdots v_n)$ may also take time proportional to the number of children, which has to be tested for each such node w'. Therefore it is a good idea to restrict $\mathcal{T}$ so that terms in $\mathcal{T}$ are either of small arity or do not need to have hash values recomputed often, which can happen when argument replacement is done.

This approach works for finite terms but not for infinite terms. When argument replacement is done, a finite term may become infinite and an infinite term may become finite. It is also possible for a finite term to remain finite but its hash value may change. If term(v, G) is infinite this does not necessarily lead to nontermination because such terms are not evaluated; recall this statement about how variables are evaluated:

If P is a variable x then $[[P]](e0, G) = (e0(x), (e0, G))$ and $[P, (e0, G)] = 1$.

The only terms that are evaluated, leading to a call to finish.function, are terms that appear in a program. These terms may have constructors in them, but these terms are all finite. If the evaluation of a finite term leads to an infinite constructor term in the prelude to finish.function, this term is not further evaluated. Although they are not evaluated, such infinite constructor terms term(x, G) can be accessed by finding their top level function symbol **top**(x) or their arguments **arg**(i, x) and by the function equal.top and they can be modified by the argument replacement operation.

First some terminology that is helpful when considering infinite terms.

Definition 12. *If $v : f(v_1 \cdots v_n)$ is in G then the v_i are called* children *of v and v is called a* parent *of the v_i in G. Note that a node can have more than one parent. The ancestor relation is defined in this way: A node is an* ancestor *of itself and if v is a parent of w in G then v is an ancestor of w in G. Also the ancestor relation is transitive. A* cycle *in G is defined as a sequence $v_1 \cdots v_n$ of nodes where $n \geq 1$, each v_i is a child of v_{i+1} in G and $v_1 = v_n$. It is possible for a node to be a parent of itself; this is also a cycle in G.*

If there is a cycle in G containing a node v of G then term(v, G) is infinite. It's fine to create infinite terms, they just can't be in $\mathcal{T}$ because it would entail a lot of effort to hash them. We say a node v of a graph G or a state (e, G) is in $\mathcal{T}$ if term(v, G) is in $\mathcal{T}$. The following information (the status of v) will

 doi:10.20944/preprints202511.0214.v1

be stored with nodes v in G: Whether term(v, G) is in $\mathcal{T}$, the hash value of term(v, G), and the set of parents of v in G. The hash value and the membership in $\mathcal{T}$ can either be given or set to UNKNOWN.

In an argument replacement operation $[[A[i] \leftarrow t]](e, G)$ in state (e, G), suppose that $[[t]](e, G) = (v, (e, G'))$. Recall that we refer to this statement $A[i] \leftarrow t$ as the replacement of the i^{th} argument of term $(e(A), G')$ with term(v, G') and say that the statement is of type (term(A, G'),term(v, G'), term $(e(A), G''))$ where $(e, G'') = [[A[i] \leftarrow t]](e, G)$. Suppose such an argument replacement is of type (s, t', s'). In all cases if t' is infinite or not in $\mathcal{T}$ then s' will not be in $\mathcal{T}$ and none of the ancestors of $e(A)$ will be in $\mathcal{T}$ after this operation. Also, in all cases if v is an ancestor of $e(A)$ in G' then s' will be infinite so that none of the ancestors of $e(A)$ will be in $\mathcal{T}$ after this operation. Therefore the status of all the ancestors will have to be updated. Whenever an argument replacement operation of type (s, t', s') is done and t' is in $\mathcal{T}$ and v is not an ancestor of $e(A)$ then the membership of A in $\mathcal{T}$ needs to be recomputed in some manner, depending on how $\mathcal{T}$ is defined. This recomputation can often be done without searching through the entire term structure of s'. In such cases all four possibilities of s being or not being in $\mathcal{T}$ and s' being or not being in $\mathcal{T}$ are theoretically possible. The decision about membership in $\mathcal{T}$ in these cases depends on how $\mathcal{T}$ is defined.

A possible way to define $\mathcal{T}$ is to specify a set of constructor function symbols and to say that a ground term t is in $\mathcal{T}$ if it is finite and if all function symbols in t are in the set. Another possibility is to define $\mathcal{T}$ as the set of all ground constructor terms of bounded depth, where the depth of a constant symbol is zero and if $f(t_1 \cdots t_n)$ is a term, the depth of it is one plus the maximum depth of the t_i. In this case the depth of term(v, G) should be added to the status of v and updated as necessary.

If terms in $\mathcal{T}$ are defined to be of bounded depth and v is not an ancestor of $e(A)$ in G' it is possible that s and t' are in $\mathcal{T}$ but s' is not if t' has depth greater than the depth of the subterm it is replacing. It is possible also that s is not in $\mathcal{T}$ but s' is in $\mathcal{T}$ if t' has depth smaller than the depth of the subterm it is replacing. Even if both s and s' are in $\mathcal{T}$ it is possible that the hash values of term$(e(A), G'')$ and of the ancestors of $e(A)$ in G'' will need to be recomputed. If $\mathcal{T}$ is defined as the set of all finite terms over a specified set of function symbols, and v is not an ancestor of $e(A)$ in G', then s' is in $\mathcal{T}$ if both s and t' are. However, the hash values of ancestors of $e(A)$ may need to be recomputed.

One may say that with all of this discussion, the semantics of the language is not very simple. But the complexities arise from mathematical properties of the language and also from constraints imposed by the nature of mathematics itself, not from arbitrary decisions about the form of the language.

6.3. Isomorphism Dependence

Now, after considering efficiency issues for make.term, new.term, and cache.term, we turn to isomorphism dependence assuming caching is done using a set $\mathcal{T}$ as mentioned above. As mentioned earlier, if f is a constructor then to compute $f(t_1, \cdots, t_n)$, finish.function makes use of new.term, make.term, or cache.term. In particular, in this case finish.function $(f(v_1 \cdots v_n), (e, G))$ is defined as $(v, (e, \{v\} \cup G))$ where $(v, G \cup \{v\})$ is make.term $(f(v_1 \cdots v_n), G)$ and make.term may call new.term or cache.term.

Now we show that if $(v_1 \cdots v_n, G) \cong_i (v'_1 \cdots v'_n, G')$ then make.term$(f(v_1 \cdots v_n), G) \cong_i$ make.term $(f(v'_1, \cdots v'_n), G')$ if f is a constructor. This will be helpful in showing isomorphism dependence of finish.function for constructors.

Suppose G and G' are isomorphic graphs via isomorphism ϕ. To show that make.term is isomorphism dependent, we want to show how to extend this isomorphism to one from make.term$(f(v_1 \cdots v_n$ $), G)$ to make.term$(f(v'_1, \cdots v'_n), G')$ if $\phi(v_i) = v'_i$ for all i. This implies that term$(f(v_1 \cdots v_n), G)$ = term$(f(v'_1, \cdots, v'_n), G')$. Thus both of these terms or neither of these will be in $\mathcal{T}$. Thus both will be implemented by cache.term (if the terms are in $\mathcal{T}$) or both by new.term (if the terms are not in $\mathcal{T}$). Suppose make.term$(f(v_1 \cdots v_n), G) = (v, H)$ and make.term$(f(v'_1, \cdots v'_n), G') = (v', H')$. Then term$(v, H)$ = term$(f(v_1 \cdots v_n), G)$ and term(v', H') = term$(f(v'_1, \cdots, v'_n), G')$ and term(v, H) = term(v', H'). In either case, whether new.term (for terms not in $\mathcal{T}$) or cache.term (for terms in $\mathcal{T}$) is used, ϕ can be extended to map v to v' and the resulting graphs will still be isomorphic. If new.term is used then both v and v' will be new nodes in G and G', respectively, so ϕ can be extended to an isomorphism

 doi:10.20944/preprints202511.0214.v1

ϕ^* so that $\phi^*(v) = v'$. If cache.term is used then recall that $\mathcal{T}$ is closed under subterms and that the subterms will have been generated and cached already. The top level subterms of term(v, G) will be term(v_i, G). The top level subterms of term(v', G') will be term(v'_i, G'). Thus term(v_i, G) = term(v'_i, G') for all i because of the assumption that $(v_1 \cdots v_n, G) \cong_i (v'_1 \cdots v'_n, G')$ and both terms will be in $\mathcal{T}$ for all i. The children of v will be $(v_1, \cdots, v_n)$ and the children of v' will be $(v'_1, \cdots, v'_n)$. Suppose there are existing nodes $w : f(v_1 \cdots v_n)$ in G and $w' : f(v'_1 \cdots v'_n)$ in G'. A little thought shows that such a w exists iff such a w' exists because G and G' are isomorphic. Then, because caching is done and all nodes with identical terms cache to the same thing, v will be chosen to be w and v' will be chosen to be w'. (Because term(v, G) = term(w, G) and term(v', G') = term(w', G')). So $v = w$ and $v' = w'$. Therefore the existing isomorphism ϕ does not need to be modified and make.term$(f(v_1 \cdots v_n), G)$ and make.term$(f(v'_1, \cdots v'_n), G')$ are isomorphic. Otherwise cache.term operates the same as new.term so the comments for new.term apply here and make.term$(f(v_1 \cdots v_n), G)$ and make.term$(f(v'_1, \cdots v'_n), G')$ are isomorphic.

The homomorphism ϕ^* for make.term$(f(v_1 \cdots v_n), G) \cong_i$ make.term$(f(v'_1, \cdots v'_n), G')$ is an extension of the homomorphism ϕ for $(v_1 \cdots v_n, G) \cong_i (v'_1 \cdots v'_n, G')$. That is, for nodes v and w of G and G', if $\phi(v) = w$ then $\phi^*(v) = w$. As an example of something that is not isomorphism or term dependent, suppose there were a function symbol node.symbol(x) which when evaluated would find the node v of $e(x)$ and then construct a constant symbol c_v that depends on (the name of) v and then for example, a make.term(c_v) might be done. This would not be isomorphism or term dependent. Another example that is not isomorphism or term dependent is a function that would tell if v was less than w in some alphabetic ordering on node names.

6.4. Term and Isomorphism Dependence

Now that we have done the proof for pre.function and have shown how to handle new.term, cache.term, and make.term, we turn to the rest of the proof concerning term and isomorphism dependence. Recall Propositions 2 and 1:

Proposition 4. *A function F is isomorphism dependent if for all states S and T, $S \cong_i T$ implies $F(S) \cong_i F(T)$.*

Proposition 5. *A function F is term dependent if for all states S and T, $S \cong_t T$ implies $F(S) \cong_t F(T)$.*

In more detail, we have the following:

If for an imperative statement P, for all states $S_1, S_2, S_1 \cong_i S_2$ implies $[[P]]S_1 \cong_i [[P]]S_2$ then P is isomorphism dependent.

Similarly, if for a functional expression P, for all states $S_1, S_2, S_1 \cong_i S_2$ implies $[[P]]S_1 \cong_i [[P]]S_2$ then P is isomorphism dependent, where the $[[P]]S_i$ are (node, state) pairs.

The same definitions hold for term dependence with $\cong_t$ referring to terms and not isomorphism classes.

6.5. Cases to Consider: Functional Expressions: Term and Isomorphism Dependence

We want to prove the following:

Theorem 5. *If F is a functional expression in ITGL then $[[F]]$ is term dependent if there are no argument replacement or node equality statements in the program in which F occurs. Also, F is isomorphism dependent even if there are argument replacement statements or node equality statements in the program.*

The proof is done separately for each kind of statement.

For functional expressions it is necessary to look at alternate possibilities depending on the kind of function we are dealing with. Note that all such use finish.function. First we consider rewrite procedures.

 doi:10.20944/preprints202511.0214.v1

6.5.1. Rewrite Procedures

Recall that graph.rewrite$(f(v_1 \cdots v_n),(e,G))$ is defined as
if match.list$(r_1, f(\cdots),G) \neq \perp_e$ then $[[s_1]]$(match.list$(r_1, f(\cdots),G),G)$ else
if match.list$(r_2, f\cdots),G) \neq \perp_e$ then $[[s_2]]$(match.list$(r_2, f(\cdots),G)$ else $\cdots$ else
if match.list$(r_n, f(\cdots),G) \neq \perp_e$ then $[[s_n]]$(match.list$(r_n, f(\cdots),G),G)$ else
$(v,(e,G'))$ where (v,G') = make.term$(f(v_1 \cdots v_n),G)$.

Suppose T = graph.rewrite$(f(v_1 \cdots v_n),(e,G))$ and T'=graph.rewrite$(f(v'_1 \cdots v'_n),(e',G'))$ with $S = (e,G)$ and $S' = (e',G')$. By Corollary 1 concerning pre.function we know that if $(v_1 \cdots v_n, S) \cong_i (v'_1, \cdots, v'_n, S')$ implies $T \cong_i T'$ then graph.rewrite is term (isomorphism) dependent. Now match.list does not change G (or G') and only depends on the terms of the v_i (or v'_i) and the nodes in G (or G') so it operates identically on S and S'. These terms are identical for v_i and v'_i regardless of whether we are dealing with isomorphism or term dependence. So if $T = [[s_k]]$(match.list$(r_k, f(v_1 \cdots),G),G)$ then $T' = [[s_k]]$(match.list$(r_k, f(v'_1 \cdots),G'),G')$ and match.list$(r_k, f(v_1 \cdots),G)$=match.list$(r_k, f(v'_1 \cdots),G')$. Let us call match.list$(r_k, f(v_1 \cdots),G)$ E. Then if $T = [[sk]](E,G)$, $T' = [[s_k]](E,G')$. We can assume by induction that $[[s_k]]$ is isomorphism (term) dependent, thus $T \cong_i T'$. The same argument works for term dependence.

6.5.2. Compiled Functions

Consider a compiled function definition $f(x_1 \cdots x_m) \to y$ where $A[x_1 \cdots x_n, y];;$. For each collection of ground terms instanting the x_i there is a unique ground term instanting y in the execution of a compiled function. So y depends only on the terms instantiating the x_i. Arguing as for rewrite functions, compiled functions are both isomorphism and term dependent.

6.5.3. Constructors

By Lemma 2 about pre.function it is only necessarily to consider the evaluation of finish.function. If f is a constructor then finish.function$(f(v_1 \cdots v_n),(e,G)) = (v,(e,H))$ where (v,H) = make.term$(f(v_1 \cdots v_n),G) = (v, G \cup \{v : f(v_1 \cdots v_n)\})$ for a node v as specified in the discussion of make.term and $\mathcal{T}$. Thus finish.function$(f(v_1 \cdots v_n),(e,G)) = (v,(e, G \cup \{v : f(v_1 \cdots v_n)\})$.

Suppose S and S' are two states and $S \cong_i S'$. By the lemma (Lemma 2) we can assume that $(v_1 \cdots v_n, S) \cong_i (v'_1, \cdots, v_n, S')$ for evaluating finish.function$(f(v_1 \cdots v_n), S)$ and finish.function$(f(v'_1, \cdots, v'_n), S')$. Then finish.function$(f(v_1 \cdots v_n),(e,G)) = (v,(e, G \cup \{v : f(v_1 \cdots v_n)\})$ and finish.function $(f(v'_1 \cdots v'_n), (e',G'))$=$(v',(e',G' \cup \{v' : f(v'_1 \cdots v'_n)\})$. Suppose ϕ is an isomorphism for $(v_1 \cdots v_n, S) \cong_i (v'_1, \cdots, v'_n, S')$. Then let ϕ^* be ϕ extended so that $\phi^*(v) = v'$. Such a ϕ^* exists by the discussion of make.term, cache.term, and new.term. Then ϕ^* demonstrates that finish.function$(f(v_1 \cdots v_n),(e,G)) \cong_i$ finish.function $(f(v'_1 \cdots v'_n),(e',G'))$. Thus for a constructor f, $[[f(t_1 \cdots t_n)]]$ is isomorphism dependent. The argument for term dependence is essentially the same, noting that term(v,G) = term(v',G'). Note again that if $\phi(v) = w$ then $\phi^*(v) = w$. This is a general property of all of these proofs of isomorphism dependence.

6.5.4. Variables

For a variable x, $[[x]](e,G) = (e(x),(e,G))$ which is easily shown to be term and isomorphism dependent.

6.5.5. Special Functions

The function **arg** is easily shown to be isomorphism and term dependent. If two terms are equal their corresponding arguments are equal. If an isomorphism maps node v to w then it also maps corresponding arguments of v to those of w. Similary the defined function **top** is isomorphism and term dependent assuming that constant symbols are in the set $\mathcal{T}$. The function equal.top(x,y) which tests if the nodes $e(x)$ and $e(y)$ have the same top symbol is easily isomorphism and term dependent. The function equal.node(x,y) which tests if $e(x)$ and $e(y)$ are equal is isomorphism dependent but not term dependent unless the terms of x and y are both in $\mathcal{T}$.

 doi:10.20944/preprints202511.0214.v1

6.5.6. Imperative Procedures

The only remaining case to consider is that of an imperative procedure definition. For this the following theorem is helpful:

Theorem 6. *Suppose $S_1 \cong_i S_2$ via isomorphism ϕ. Then there is an isomorphism ϕ^* that extends ϕ such that $[[P]](S_1) \cong_i [[P]](S_2)$ via isomorphism ϕ^*. This means that if $\phi(v) = w$ then $\phi^*(v) = w$ also. This holds for both imperative statements and functional expressions.*

Proof. This can be verified by looking at each kind of statement and verifying that the property is preserved under composition. The only place where the isomorphism directly changes is when make.term is executed, that is, during the evaluation of finish.function for a constructor. In this case a new node may be added to the germ graph and the isomorphism may be extended to this new node. Alternatively, the term graph and isomorphism may be unchanged. Therefore, during the execution of any statement, the only changes to the isomorphism and term graph will be during the evaluations of make.term, so a sequence of such operations will result in an isomorphism ϕ^* that is either identical to ϕ or that extends it. □

Now we consider the case of an imperative procedure definition. Suppose f is a procedure defined by $f(x_1 \cdots x_n)\{B\}t;;$ where x_i are the formal parameters. Let E be the statement $f(t_1 \cdots t_n)$. Then with notation as in Proposition 3 finish.function$(f(v_1 \cdots v_n), S) = [[E]](e_0, G)$. Let (v, U) be finish.function$(f(v_1 \cdots v_n), S$. Recall that v and U are defined as follows:

Let e_1 be defined by $e_1(x_i) = v_i$ and $e_1(x) = \perp_u$ for other program variables x. Let S_1' be such that env$(S_1') = e_1$ and $g(S_1') = g(S)$. Let S_2' be $[[B]]S_1'$ and note that B may change e_1. Let (v_1', S_3') be $[[t]]S_2'$ and recall that $[[t]]$ does not change env(S_2'). Finally, let v be v_1' and let U in this case be $(e_0, g(S_3'))$.

Now we can assume inductively that B and t are term and isomorphism dependent. By the way S_1' is defined, it is obtained from (e_0, G) by a function which is isomorphism and term dependent. By comments about B and t, S_3' is also obtained by a function which is isomorphism and term dependent. Putting the environment e_0 back in does not change this. This follows by Theorem 6 concerning extensions of the isomorphism and Theorem 1 concerning nodes of the term graph. Therefore $[[E]]$ is term and isomorphism dependent under suitable assumptions. (Term dependence requires that argument replacement and node equality statements not be used.)

6.6. Imperative Statements: Term and Isomorphism Dependence

We want to prove the following result:

Theorem 7. *If F is an imperative statement then $[[F]]$ is term dependent if there are no argument replacement statements or node equality statements in the program P in which F occurs. If F is an imperative statement then F is isomorphism dependent even if there are argument replacement statements or node equality statements in the program P.*

Again, the proof is done separately for each kind of statement. Recall that $\cong_t$ is term equivalence and $\cong_i$ is isomorphism equivalence.

6.6.1. Simple Assignment Statements

For an environment e recall that $e[y \leftarrow v]$ is defined by $e[y \leftarrow v](y) = v$ and $e[y \leftarrow v](x) = e(x)$ for $x \not\equiv v$. Also, $[[x \leftarrow t]]S$ = fix.env(env$(S)[x \leftarrow v], S')$ where $(v, S') = [[t]]S$.

By Theorem 2, the environment of S' is the same as that of S. The nodes mentioned in this original environment are still in the term graph because the set of nodes in the term graph never becomes smaller, by Theorem 1. Further, for isomorphism dependence, the isomorphism condition for environments is still satisfied by the current isomorphism by the extension property of Theorem 6.

Note that the value v returned by finish.function is a node in the term graph which if necessary is added to the graph before returning.

 doi:10.20944/preprints202511.0214.v1

Consider two states S_1 and S_2, (e_1, G_1) and (e_2, G_2) respectively. Suppose $(e_1, G_1) \cong (e_2, G_2)$. Let (e_1', G_1') be $[[x \leftarrow t]]S_1$ and (e_2', G_2') be $[[x \leftarrow t]]S_2$. We want to show that $S_1' = (e_1', G_1') \cong S_2' = (e_2', G_2')$ in both senses. Let $(v_1, S_1'') = (v_1, (e_1'', G_1''))$ be $[[t]]S_1$ and $(v_2, S_2'') = (v_2, (e_2'', G_2''))$ be $[[t]]S_2$. Now $e_1'' = e_1$ and $e_2'' = e_2$. Also the graph is not further modified after computing $[[t]]S_1$ and $[[t]]S_2$ so $G_1'' = G_1'$ and $G_2'' = G_2'$. Thus $S_1'' = (e_1, G_1')$ and $S_2'' = (e_2, G_2')$.

First consider term equivalence. Suppose S_1 and S_2 are term equivalent. So $\text{term}(e_1(z), G_1)$ = $\text{term}(e_2(z), G_2)$ for all z. Then $[[t]]S_1$ and $[[t]]S_2$ will be term equivalent by induction. So $(v_1, S_1'') \cong_t (v_2, S_2'')$. Thus $\text{term}(v_1, G_1')$ = $\text{term}(v_2, G_2')$. Then $e_1'(x) = v_1$ and $e_2'(x) = v_2$. So $\text{term}(e_1'(x), G_1')$ = $\text{term}(v_1, G_1')$ = $\text{term}(v_2, G_2')$ = $\text{term}(e_2'(x), G_2')$ and for other variables y because of fix.env, $\text{term}(e_1'(y), G_1')$ = $\text{term}(e_1(y), G_1)$ = $\text{term}(e_2(y), G_2)$ = $\text{term}(e_2'(y), G_2')$ so $(e_1', G_1') \cong_t (e_2', G_2')$.

Now consider isomorphism equivalence. Suppose S_1 and S_2 are isomorphic. Then we can assume for the induction proof that $[[t]]S_1$ and $[[t]]S_2$ are isomorphic. So $(v_1, S_1') \cong_i (v_2, S_2')$, say by isomorphism ϕ. Thus $G_1' \cong_i G_2'$, with v_1 in G_1', v_2 in G_2', and $\phi(v_1) = v_2$. We show that $(e_1', G_1') \cong_i (e_2', G_2')$ by isomorphism ϕ. For this we need that for all program variables y, $\phi(e_1'(y)) = e_2'(y)$. We know that $\phi(v_1) = v_2$ and ϕ maps S_1' to S_2'. For x, $e_1'(x) = v_1$ and $e_2'(x) = v_2$ and $\phi(v_1) = v_2$ so $\phi(e_1'(x)) = e_2'(x)$. For $y \not\equiv x$, $\phi(e_1'(y)) = e_2'(y)$ because $S_1' \cong_i S_2'$ by ϕ. So $(e_1', G_1') \cong_i (e_2', G_2\text{‘})$.

6.6.2. Argument Replacement Statements

Now consider the argument replacement statement $x[y] \leftarrow t$ for isomorphism dependence, not term dependence. Recall the semantics:

$$[[A[i] \leftarrow t]](e, G) = (e, \text{ Replace}[G, e(A), i, t]).$$

Recall that $\text{Replace}[G, u, i, u']$ is G with $u : f(u_1 \cdots u_i \cdots u_n)$ (for node u in G) replaced by $u : f(u_1 \cdots u' \cdots u_n)$.

Suppose $(e_1', G_1') = [[x[y] \leftarrow t]](e_1, G_1)$ and $(e_2', G_2') = [[x[y] \leftarrow t]](e_2, G_2)$.

Suppose $(e_1, G_1) \cong_i (e_2, G_2)$. We want to show that $(e_1', G_1') \cong_i (e_2', G_2')$. Suppose $(z_1, (e_1, G_1'')) = [[t]](e_1, G_1)$ and $(z_2, (e_2, G_2'')) = [[t]](e_2, G_2)$. We assume by induction that $[[t]]$ is isomorphism dependent so there is an isomorphism ϕ that maps $(z_1, (e_1, G_1''))$ to $(z_2, (e_2, G_2''))$. Thus $\phi(z_1) = z_2$. Suppose $e_1(x)$ is v_1 and $e_2(x)$ is v_2. Then $\phi(v_1) = v_2$. So the label of v_1 in G_1' is the same as the label of v_1 in G_1'' and the label of v_2 in G_2' is the same as the label of v_2 in G_2''. Also G_2' is G_2'' with the y child of v_2 replaced by z_2 and G_1' is G_1'' with the y child of v_1 replaced by z_1. Suppose $v_1 : f(u_1 \cdots u_n)$ is in G_1'' and $v_2 : f(w_1 \cdots w_n)$ is in G_2''. Because $\phi(v_1) = v_2$, $\phi(u_i) = v_i$ for all i. Also $\phi(z_1) = z_2$. Then $v_1 : f(u_1 \cdots z_1 \cdots u_n)$ is in G_1' and $v_2 : f(w_1 \cdots z_2 \cdots w_n)$ is in G_2'. So ϕ maps each child of v_1 in G_1' to the corresponding child of v_2 in G_2' so the conditions for $\phi(e_1(x)) = e_2(x)$ are satisfied and ϕ is an isomorphism from (e_1', G_1') to (e_2', G_2'). The only nodes that have changed from G_1'' to G_1' are $e_1(x)$ and $e_2(x)$. This completes the proof. Note that the same ϕ maps $(z_1, (e_1, G_1''))$ to $(z_2, (e_2, G_2''))$ and also maps (e_1', G_1') to (e_2', G_2') because the names of the nodes have not changed, just one of the child pointers of v_1 and v_2.

6.6.3. New.Term Cache.Term and Make.Term

Finish.function$(f(v_1 \cdots v_n), S)$ has been handled in Section 5.10. Using Corollary 1 about pre.function, we can show that evaluating a functional expression $f(t_1 \cdots t_n)$ where f is a constructor preserves term dependence and equivalence dependence.

If there are no statements $x[y] \leftarrow z$ or equality tests of nodes then all operations are term dependent. For this, new.term cache.term and make.term are not a problem.

6.6.4. Conditional Statements

Recall that
[[if t then $\{P_1\}$ else $\{P_2\}$]]S =
if $\text{term}(v, g(S')) = \perp_u$ then S' else

 doi:10.20944/preprints202511.0214.v1

if term$(v, g(S'))$ = true then $[[P_1]]S'$ else
$[[P_2]]S'$ where $(v, S') = [[t]]S$.
Suppose $S_1 \cong S_2$, we want to show that $[[\text{if } \cdots]]S_1 \cong [[\text{if } \cdots]]S_2$.

Let S_1'' be $[[\text{if } \cdots]]S_1$ and S_2'' be $[[\text{if } \cdots]]S_2$. We know by the lemma about pre.function that if $(v_1, S_1') = [[t]]S_1$ and $(v_2, S_2') = [[t]]S_2$ then $(v_1, S_1') \cong (v_2, S_2')$ assuming that t is term(isomorphism) dependent. This implies that term$(v_1, g(S_1'))$ = term$(v_2, g(S_2'))$ (in both senses). So if term$(v_1, g(S_1'))$ = $\perp_u$ then $S_1'' = S_1'$ and $S_2'' = S_2'$ so $S_1'' \cong S_2''$. If term$(v_1, g(S_1'))$=true then $S_1'' = [[P_1]]S_1'$ and $S_2'' = [[P_1]]S_2'$. Assuming P_1 is term (isomorphism) dependent then $S_1'' \cong S_2''$. If term$(v_1, g(S_1')) \neq$ true then $S_1'' = [[P_2]]S_1'$ and $S_2'' = [[P_2]]S_2'$ so assuming P_2 is term(isomorphism) dependent then $S_1'' \cong S_2''$. In all cases $S_1'' \cong S_2''$.

6.6.5. Iterative Statements

Recall that $[[\text{while } t \text{ do } \{P\}]]S$ = if (term$(v, g(S')) = \perp_u$) then S' else if term$(v, g(S')) \not\equiv$ true then S' else $[[\text{while } t \text{ do } P]]([[P]]S')$ where $(v, S') = [[t]]S$.

Let S_1'' = $[[\text{while } \cdots]]S_1$ and S_2''= $[[\text{while } \cdots]]S_2$. Let $(v_1, S_1') = [[t]]S_1$ and $(v_2, S_2') = [[t]]S_2$. By the lemma then $S_1' \cong S_2'$ if $[[t]]$ is term (isomorphism) dependent. Now either way term(v_1, S_1') = term(v_2, S_2'). If term$(v_1, S_1') = \perp_u$ then $S_1'' = S_1'$ and $S_2'' = S_2'$ so $S_1'' \cong S_2''$. If term$(v_1, g(S_1')) \not\equiv$ true then $S_1'' = S_1'$ and $S_2'' = S_2'$ so $S_1'' \cong S_2''$. Otherwise S_1''=$[[\text{while } \cdots]]([[P]]S_1')$ and S_2''=$[[\text{while } \ldots \cdots]]([[P]]S_2')$. By induction we can assume $[[P]]S_1' \cong [[P]]S_2'$ and that $[[\text{while } \cdots]]$ is term (isomorphism) dependent on $[[P]]S_1$ and $[[P]]S_2$. Thus $S_1'' \cong S_2''$ in this case also.

6.6.6. Composition of Statements

If two statements are term(isomorphism) dependent then so is their composition.

6.6.7. Multiple Assignment Statements

As for $(x_1 \cdots x_n) \leftarrow t$ this can be expressed in terms of other statements so the proof reduces to proofs for them and also for compositions of imperative statements.

6.6.8. Copy Statements

If term(y, S) is in $\mathcal{T}$ then $[[x \leftarrow \text{ copy}(y)]](S)$ is the same as $[[x \leftarrow y]](S)$ because a new node will be created with the same term as y and then it will be cached into $e(y)$ again. Otherwise suppose $S_1 \cong S_2$ and let $S_1' = [[x \leftarrow \text{ copy}(y)]](S_1)$ and let $S_2' = [[x \leftarrow \text{ copy}(y)]](S_2)$ with $S_1' = (e_1', G_1')$ and $S_2' = (e_2', G_2')$. Then $G_1' = G_1 \cup \{v_1\}$ and $G_2' = G_2 \cup \{v_2\}$ and $e_1' = e_1 \cup \{x \leftarrow v_1\}$ and $e_2' = e_2 \cup \{x \leftarrow v_2\}$. As for term dependence $S_1' \cong S_2'$ because term(x, G_1') = term(y, G_1) = term(y, G_2) = term(x, G_2'). As for isomorphism dependence, let ϕ be an isomorphism from G_1 to G_2 with $\phi(e_1(y)) = e_2(y)$. Then extend ϕ to an isomorphism from G_1' to G_2' with $\phi'(e_1(x)) = e_2(x)$).

7. Conclusions

ITGL has been presented as an imperative language that makes use of term graphs. Here we have explored this approach to language design and have noted some of its advantages. Term graphs make use of terms which are a general formalism for expressing data structures. Terms in term graphs can be infinite, but because the term graph is finite, such infinite terms always have a finite representation. The states in ITGL consist of a term graph and an environment, which maps program variables to nodes in the term graph. Languages that make use of term graphs are typically defined using term rewriting systems. The language ITGL is unusual in that it uses term graphs but is an imperative language. This means that all statements map states to states. The use of term graphs makes the language powerful and expressive, but with a simple syntax and semantics. The simplicity of the language. together with its imperative style, should make the language easy to learn and facilitate efficient implementations. A rewriting facility is defined for the language, which gives it a pattern matching facility, but is not essential for the language.

 doi:10.20944/preprints202511.0214.v1

Garbage collection could be performed on term graphs to reduce the storage needed, but this is not explicitly considered here. A mechanism for caching to reduce memory requirements is discussed. Parameters are passed essentially by reference. The language has assignment statements, conditional statements, iterative statements, and procedures. Arrays in ITGL are represented as terms and are treated like any other term. Assignments to array elements are implemented in ITGL as the replacement of top level subterms of a term by another term. If there are multiple references to an array, replacing an element of the array will affect all of these references. Imperative languages including ITGL have such side effects but in return for this one obtains efficient array operations and also a model that is close to the operation of computer hardware. Imperative languages are also popular in industry and academia. However, such languages lack the formal elegance of functional languages. Functional languages can also implement arrays efficiently if references to the array are single threaded, but are not as popular in industry and academia. If array references are not single threaded then arrays have to be copied whenever an array element is changed. This can impose a logarithmic slowdown on array operations. Monads can help to code imperative operations in a functional language but are also harder to understand than imperative languages and might also be harder to verify. Continuation style parameter passing is another formalism that can be considered but does not permit efficient array operations without side effects unless array accesses are single threaded.

ITGL may be useful as a language in itself. However, ITGL as presented here does not have a general input output facility and does not have interrupts. An approach for input and output is specified but the main focus of the language is on defining procedures which can be considered as abstract algorithms. The algorithms intended to be specified in this way are those which map abstract mathematical structures into other mathematical structures. For example, the input may be a graph and a pair of nodes and the output may be an integer giving the length of a shortest path. This kind of algorithm includes many algorithms discussed in the literature but does not include algorithms such as memory management. ITGL is easy to understand and similar to the pseudo code used in algorithm texts to define algorithms, but it has a precise syntax and semantics which could enable proofs of correctness of such abstract algorithms. This may make it useful for algorithm texts. The question of what is an algorithm is discussed in the introduction.

The syntax and semantics of ITGL are simple which should facilitate proofs of correctness, though proofs of correctness are not discussed in the paper. The simplicity of the language may also make ITGL a suitable formalism for the description of abstract algorithms which could then be translated into other widely used imperative languages. ITGL has some features such as term rewriting and caching that would not need to be used for describing algorithms, increasing its simplicity for this application. The semantics of ITGL is defined assuming termination. It may be possible to extend the semantics to nontermination using complete partial orderings and least fixpoints A number of mathematical properties of ITGL are proved, including term dependence and isomorphism dependence. The latter essentially means that the names assigned to nodes in the term graph do not influence the result of a computation. Also, an extended discussion of methods of caching is given. Caching means that only one copy is stored for each cached term. This reduces the storage required while increasing the side effects that occur when a term is modified.

7.1. Further Topics

The language could be extended in various ways, including by adding a more extensive input-output facility and by adding sorts. Also, nothing was mentioned about floating-point numbers although this is essentially implicit in the presentation. The application of logical systems such as Hoare logics for proofs of correctness in IGTL could be studied, as well as techniques for translating ITGL programs and procedures into other common imperative languages. If such translators were implemented they could be used on a trial basis even if they had not yet been verified. Methods for showing that a translation to another language is correct could be developed. This could involve translating a program and its specification into another language and proving that then the translated program satisfies the translated specification. The simplicity of the language ITGL should help in this

 doi:10.20944/preprints202511.0214.v1

regard. It might also be worthwhile to implement the language ITGL. Because of the simplicity of the language, an interpreter or compiler for the language might be easy to write and might not be very large. However, a method for interfacing ITGL code with compiled code for compiled functions would need to be developed. Also, the interaction of the underlying logical theory with error values would need to be specified precisely.

References

BDG09. Andreas Blass, Nachum Dershowitz, and Yuri Gurevich. When are two algorithms the same? *Bulletin of Symbolic Logic*, 15(2):145–168, 2009.

BEG+87. H P Barendregt, M C J D Eekelen, J R W Glauert, J R Kennaway, M J Plasmeijer, and M R Sleep. Towards an intermediate language based on graph rewriting. In *Volume II: Parallel Languages on PARLE: Parallel Architectures and Languages Europe*, page 159–175, Berlin, Heidelberg, 1987. Springer-Verlag.

BFG+12. Mária Bieliková, Gerhard Friedrich, Georg Gottlob, Stefan Katzenbeisser, and György Turán, editors. *SOFSEM 2012: Theory and Practice of Computer Science - 38th Conference on Current Trends in Theory and Practice of Computer Science, Špindlerův Mlýn, Czech Republic, January 21-27, 2012. Proceedings*, volume 7147 of *Lecture Notes in Computer Science*. Springer, 2012.

BG03a. Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 4:578–651, 2003.

BG03b. Andreas Blass and Yuri Gurevich. Algorithms: A quest for absolute definitions. *Bulletin of the EATCS*, 81:195–225, 2003.

BN98. Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, Cambridge, England, 1998.

BP18. L. Barnett and D. Plaisted. Programming by term rewriting. In *10th International Workshop on Computing with Terms and Graphs (TERMGRAPH)*, 2018.

BP21. L. Barnett and D. Plaisted. A term rewriting semantics for imperative style programming: Summary. In Temur Kutsia, editor, Proceedings of the 9th International Symposium on *Symbolic Computation in Software Science*, Hagenberg, Austria, September 8-10, 2021, 2021.

BvEG+87. Hendrik Pieter Barendregt, Marko C. J. D. van Eekelen, John R. W. Glauert, Richard Kennaway, Marinus J. Plasmeijer, and M. Ronan Sleep. Term graph rewriting. In J. W. de Bakker, A. J. Nijman, and Philip C. Treleaven, editors, *PARLE, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, The Netherlands, June 15-19, 1987, Proceedings*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 1987.

CG99. Andrea Corradini and Fabio Gadducci. Rewriting on cyclic structures : equivalence between the operational and the categorical description. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 33(4-5):467–493, 1999.

Cou80. Guy Cousineau. An algebraic definition for control structures. *Theoretical Computer Science*, 12(2):175–192, 1980.

COU90. Bruno COURCELLE. Chapter 5 - graph rewriting: An algebraic and logic approach. In JAN VAN LEEUWEN, editor, *Formal Models and Semantics*, Handbook of Theoretical Computer Science, pages 193–242. Elsevier, Amsterdam, 1990.

Cou95. Bruno Courcelle. Graph rewriting: A bibliographical guide. In Hubert Comon and Jean-Pierre Jounnaud, editors, *Term Rewriting*, pages 74–74, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

DEP07. Dominique Duval, Rachid Echahed, and Frédéric Prost. Modeling pointer redirection as cyclic term-graph rewriting. *Electronic Notes in Theoretical Computer Science*, 176(1):65–84, 2007. Proceedings of the Third International Workshop on Term Graph Rewriting (TERMGRAPH 2006).

DJ90. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, 1990.

DJO24. Nachum Dershowitz, Jean-Pierre Jouannaud, and Fernando Orejas. Drag Rewriting. *arXiv e-prints*, page arXiv:2406.16046, June 2024.

DLLL05. Dan Dougherty, Pierre Lescanne, Luigi Liquori, and Frédéric Lang. Addressed term rewriting systems: Syntax, semantics, and pragmatics: Extended abstract. *Electronic Notes in Theoretical Computer Science*, 127(5):57–82, 2005. Proceedings of the 2nd International Workshop on Term Graph Rewriting (TERMGRAPH 2004).

 doi:10.20944/preprints202511.0214.v1

DP01. Nachum Dershowitz and David A. Plaisted. Rewriting. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, pages 535–610. Elsevier and MIT Press, 2001.

Gur00. Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, July 2000.

Hil15. Robin Hill. What an algorithm is. *Philosophy & Technology*, 56, 01 2015.

HKP88. Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. In D. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification*, pages 92–112, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

Klo96. Jan Willem Klop. Term graph rewriting. In Gilles Dowek, Jan Heering, Karl Meinke, and Bernhard Möller, editors, *Higher-Order Algebra, Logic, and Term Rewriting*, pages 1–16, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

Mog91. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

Mos01. Yiannis N. Moschovakis. *What Is an Algorithm?*, pages 919–936. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

NK14. Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014.

NKK25. Naoki Nishida, Misaki Kojima, and Takumi Kato. Transforming imperative programs into bisimilar logically constrained term rewrite systems via injective functions from configurations to terms. *Journal of Logical and Algebraic Methods in Programming*, 145:101056, 2025.

PB20. David Plaisted and Lee Barnett. A term-rewriting semantics for imperative style programming. *arXiv e-prints*, 2020.

Pla05. D. Plaisted. *An Abstract Programming System*, pages 85–129. Nova Science Publishers, New York, 2005.

Pla13. David A. Plaisted. Source-to-source translation and software engineering. *Journal of Software Engineering and Applications*, Vol.6 No.4A:30–40, 2013.

Plu99. D. Plump. *Term graph rewriting*, page 3–61. World Scientific Publishing Co., Inc., USA, 1999.

SS75. Gerald Jay Sussman and Jr. Steele, Guy L. Scheme: An interpreter for extended lambda calculus. Technical Report AIM-349, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, December 1975.

Tao23. Tao Tao. *RE-LANG: A Parallel-by-default Programming Language*. Dissertation, University of North Carolina at Chapel Hill, Chapel Hill, NC, 2023.

Var12. Moshe Y. Vardi. What is an algorithm? *Commun. ACM*, 55(3):5–5, March 2012.

Wad95. Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer Berlin Heidelberg, 1995.