A Formal Theory of Undirected Graphs in Higher-Order Logic

Ching-Tsun Chou ⟨chou@cs.ucla.edu⟩

Computer Science Department, University of California at Los Angeles
Los Angeles, CA 90024, U.S.A.

Abstract. This paper describes a formal theory of undirected (labeled) graphs in higher-order logic developed using the mechanical theorem-proving system HOL. It formalizes and proves theorems about such notions as the empty graph, single-node graphs, finite graphs, subgraphs, adjacency relations, walks, paths, cycles, bridges, reachability, connectedness, acyclicity, trees, trees oriented with respect to roots, oriented trees viewed as family trees, top-down and bottom-up inductions in a family tree, distributing associative and commutative operations with identities recursively over subtrees of a family tree, and merging disjoint subgraphs of a graph. The main contribution of this work lies in the precise formalization of these graph-theoretic notions and the rigorous derivation of their properties in higher-order logic. This is significant because there is little tradition of formalization in graph theory due to the concreteness of graphs. A companion paper [2] describes the application of this formal graph theory to the mechanical verification of distributed algorithms.

Last revised: 21 June 1994. Appeared in the 7th International Workshop on Higher Order Logic Theorem Proving and Its Applications, edited by T.F. Melham and J. Camilleri, pp. 144–157, LNCS 859, Springer-Verlag, 1994.

1 Introduction

In view of the many applications in computer science that graph theory has, the scantiness of work on the formalization of graph theory in HOL—the mechanical theorem-proving system for higher-order logic developed by Gordon *et al.* [5, 6]—is somewhat surprising. The only prior work we are aware of is that of Wong [9], who was concerned mainly with formulating and proving properties of paths in *directed* graphs in order to reason about railway signaling schemes. As we are interested in the mechanical verification in HOL of such distributed algorithms as those in [4, 8], we need a formal theory of *undirected* graphs in HOL that includes at least the notion and important properties of *trees*. Consequently we have gradually formalized a considerable amount of graph theory in HOL over the past two or three years. The purpose of this paper is to describe this formal graph theory in some detail; a companion paper [2] describes its application to the mechanical verification of distributed algorithms.

Our graph theory consists of five segments with the following dependency relation:

$$\begin{tikzcd}
\text{BASIC} & \text{TREE} \arrow[l] & \text{ORIENT} \arrow[l] \arrow[d] & \text{CONSTR} \arrow[l] \\
 & & \text{ACI} &
\end{tikzcd}$$

where X $\longrightarrow$ Y means that X depends on Y. The segment BASIC (described in Section 5) defines the basic notions of graph theory, including a somewhat unusual definition of undirected *labeled* graphs, and derives their basic properties. The segment TREE (Section 6) is a theory of paths, cycles, bridges, reachability, connectedness, acyclicity, and trees (a tree is defined to be a connected and acyclic graph). After proving many facts about these notions, it culminates with the theorem stating that there is a unique path between any two nodes in a tree. The uniqueness of paths implies that a tree with a distinguished node—called the root—can be oriented and viewed as a family tree by taking the root as the progenitor. The segment ORIENT (Section 7) is a theory of such oriented trees and contains theorems for doing both top-down and bottom-up inductions in a family tree and for distributing recursively an associative and commutative operation with an identity (an ACI operation, for short) over the subtrees of a family tree. The latter task relies on the segment ACI (Section 4), which defines a constant for generalizing ACI operations to finite sets. Finally, the segment CONSTR (Section 8) introduces several constructors of graphs that preserve properties like finiteness, connectedness and acyclicity (and hence "tree-ness" as well). Almost all theorems in our graph theory hold for both finite and infinite graphs. If a theorem holds for finite graphs only, it will be so stated explicitly.

Our graph theory is strictly definitional, meaning that it is developed from the initial theory of HOL without introducing any axioms other than definitions. The definitional approach has two well-known advantages. The first is *consistency*: definitions never introduce any inconsistencies. Since the initial theory of HOL is consistent [6], our graph theory is consistent as well. The second is *eliminability*: definitions can, in principle, be eliminated. So an auxiliary definition not involved in the statement of a result can safely be forgotten as far as that result is concerned, even if it is used in the proof of that result. To be sure, sticking to the definitional approach is sometimes laborious, since even "obvious" propositions must be honestly proved. But the logical security thus gained is well worth the effort.

The main contribution of this work lies *not* in the mathematical sophistication of the graph theory formalized, which is elementary (and all covered in, say, the first two chapters of Even's book [3]), but in the precise formalization of graph-theoretic notions and the rigorous derivation of their properties in higher-order logic. This is significant for two reasons. Firstly, it is not easy to formalize a piece of mathematics in *any* mechanical theorem-proving system. Most proofs and many definitions in typical mathematical textbooks contain gaps that are easy for a human reader to fill (at a subconscious level, in most cases) but too hard for a mechanical prover to overcome. So the human operator of a mechanical prover has to judiciously formulate and arrange definitions and theorems so that the desired results can be proved with as little effort as possible. This involves a lot of planning and experimentation. Secondly, it is particularly difficult to formalize those branches of mathematics, such as graph theory and combinatorics, in which there is little tradition of formalization because of the very concreteness of the objects being studied. In fact, as de Bruijn [1] pointed out, the more abstract a piece of mathematics is, the smaller the gaps usually are and the easier the formalization is.

The rest of this paper is organized as follows. Sections 2 and 3 are preliminaries about higher-order logic and predicates as sets, respectively. The segments of our graph theory are described in Sections 4 through 8. Due to space limitations, only the most significant theorems are listed and no proof is given. Section 9 is the conclusion.

2 Higher-Order Logic

The version of higher-order logic used in this paper is the one supported by the HOL theorem-proving system [6]. In general, the following typographic conventions are followed when writing formulas: lowercase Greek letters stand for type variables, *slanted* font for type constants, *italic* font for term variables, and sans serif font for term constants. Free variables in a definition or theorem are implicitly universally quantified unless otherwise stated. The symbol $\triangleq$ means "equals by definition"; "iff" means "if and only if". Lists are enclosed by square brackets ($[\cdots]$); list concatenation is denoted by $\frown$ (written as an infix).

3 Predicates as Sets

A **predicate** P is a function whose type is of the form $\alpha \rightarrow \mathit{bool}$, where α is called the **domain** of P. In this paper a set is *identified* with its characteristic predicate so that the following holds:

$$\{\, x : \alpha \mid P(x) \,\} \;=\; P$$

With this identification, the usual operations on sets, such as $\subseteq$, $\cap$, $\cup$, and $\setminus$ (set difference), are applicable to predicates as well. In particular, $x \in P$ is now synonymous with $P(x)$ and will be used interchangeably with it. The meaning of the $\{\cdots \mid \cdots\}$ notation for sets is given by:

$$\begin{aligned} t \in \{x : \alpha \mid Q[x]\} \;&=\; Q[t] \\ t \in \{x :: P \mid Q[x]\} \;&=\; t \in P \wedge Q[t] \end{aligned}$$

where the $Q[x]$ notation stresses that the variable x may (but does not necessarily!) occur free in the term Q and $Q[t]$ is the term obtained by substituting t for the free occurrences of x in Q (with the usual proviso preventing capture of free variables). It should be pointed out that both $\in$ and the $\{\cdots \mid \cdots\}$ notation are used in this paper solely for the sake of readability: they are never used in the actual implementation, in which $x \in P$ is replaced by $P(x)$ and definitions of the forms $S \triangleq \{x \mid Q[x]\}$ and $S \triangleq \{x :: P \mid Q[x]\}$ by $S(x) \triangleq Q[x]$ and $S(x) \triangleq P(x) \wedge Q[x]$, respectively. (Definitions are the only places where the $\{\cdots \mid \cdots\}$ notation is used.)

Other notions involving sets that are needed in this paper are the following. **Restricted quantifications** over sets are defined by:

$$\begin{aligned}
(\,\forall\, x :: P.\ Q[x]\,) &\triangleq (\,\forall\, x.\ x \in P \Rightarrow Q[x]\,)\\
(\,\exists\, x :: P.\ Q[x]\,) &\triangleq (\,\exists\, x.\ x \in P \wedge Q[x]\,)\\
(\,\varepsilon\, x :: P.\ Q[x]\,) &\triangleq (\,\varepsilon\, x.\ x \in P \wedge Q[x]\,)
\end{aligned}$$

The **union** of an indexed family of sets, $F : \iota \rightarrow \alpha \rightarrow bool$, over a set of indices $I : \iota \rightarrow bool$ is:

$$\mathsf{Union}(F)(I) \triangleq \{\, x : \alpha \mid \exists\, j :: I.\ x \in F(j)\,\}$$

Let $f : \alpha \rightarrow \beta$ and $P : \alpha \rightarrow bool$. The **image** of P under f is:

$$\mathsf{Image}(f)(P) \triangleq \{\, y : \beta \mid \exists\, x :: P.\ y = f(x)\,\}$$

For a binary relation $R : \alpha \rightarrow \alpha \rightarrow bool$, f is **one-one modulo** R over P iff:

$$\mathsf{OneOneMod}(R)(f)(P) \triangleq \forall\, x\ y :: P.\ (f(x) = f(y)) \Rightarrow R(x)(y)$$

See Section 5.2 for how $\mathsf{OneOneMod}$ is used. "P is a finite set" is denoted by $\mathsf{Finite}(P)$; see [7] for how to define Finite in HOL.

4 Generalizing ACI Operations to Finite Sets

Let $\mathsf{ACI}(op, id)$ mean that $op : \beta \rightarrow \beta \rightarrow \beta$ (which will be written as an infix) is an **a**ssociative and **c**ommutative operation and $id : \beta$ is an **i**dentity for it:

$$\begin{aligned}
&\mathsf{ACI}(op, id) \triangleq\\
&\quad (\,\forall\, x\ y\ z.\ x\ op\ (y\ op\ z) \ =\ (x\ op\ y)\ op\ z\,) \ \wedge\\
&\quad (\,\forall\, x\ y.\ x\ op\ y \ =\ y\ op\ x\,) \ \wedge\\
&\quad (\,\forall\, x.\ x\ op\ id \ =\ x\,)
\end{aligned}$$

For example, both $\mathsf{ACI}(+, 0)$ and $\mathsf{ACI}(*, 1)$ are true.

Let $f : \alpha \rightarrow \beta$, $P : \alpha \rightarrow bool$, and $x : \alpha$. Extending the method used in [7] to define the cardinalities of finite sets, a constant Sop (for "**s**et **op**eration") can be introduced, via the *constant specification* mechanism in HOL [6], with the following property:

$$\begin{aligned}
&\mathsf{ACI}(op, id) \wedge \mathsf{Finite}(P) \wedge x \notin P \ \Rightarrow\\
&\quad (\,\mathsf{Sop}(op, id)(f)(\emptyset) \ =\ id\,) \ \wedge\\
&\quad (\,\mathsf{Sop}(op, id)(f)(\{x\} \cup P) \ =\ f(x)\ op\ \mathsf{Sop}(op, id)(f)(P)\,)
\end{aligned} \tag{1}$$

Informally, (1) amounts to saying that:

$$\mathsf{Sop}(op, id)(f)(\{x_1, \ldots, x_n\}) \ =\ f(x_1)\ op\ \cdots\ op\ f(x_n)\ op\ id \tag{2}$$

Equation (2) shows clearly why the assumption $\mathsf{ACI}(op, id)$ is necessary: since $\{x_{i_1}, \ldots, x_{i_n}\} = \{x_1, \ldots, x_n\}$ for any permutation $(i_1, \ldots, i_n)$ of $(1, \ldots, n)$, the order in which op is applied to $f(x_i)$'s had better be unimportant; the identity id is needed to take care of the empty set (i.e., when $n = 0$). For example, $\mathsf{Sop}(+, 0)(f)(P)$ and $\mathsf{Sop}(*, 1)(f)(P)$ are what are usually written as $\sum_{x \in P} f(x)$ and $\prod_{x \in P} f(x)$, respectively. In particular, if $f(x) = 1$ for all x, then $\mathsf{Sop}(+, 0)(f)(P)$ is just the cardinality of P.

Let $I : \iota \rightarrow bool$ and $F : \iota \rightarrow \alpha \rightarrow bool$. It can be deduced from (1) that:

$$\begin{aligned}
&\mathsf{ACI}(op, id) \land \mathsf{Finite}(I) \land (\forall j :: I.\ \mathsf{Finite}(F(j))\,) \land \\
&(\forall j\ k :: I.\ (j \neq k) \Rightarrow (F(j) \cap F(k) = \emptyset)\,)\ \Rightarrow \\
&\ (\,\mathsf{Sop}(op, id)(f)(\mathsf{Union}(F)(I))\ =\ \mathsf{Sop}(op, id)(\,\mathsf{Sop}(op, id)(f) \circ F\,)(I)\,)
\end{aligned} \tag{3}$$

where $\circ$ denotes function composition. Informally, (3) says that:

$$\begin{aligned}
&\mathsf{Sop}(op, id)(f)(F_1 \cup \cdots \cup F_m)\ = \\
&\ \mathsf{Sop}(op, id)(f)(F_1)\ op\ \cdots\ op\ \mathsf{Sop}(op, id)(f)(F_m)
\end{aligned} \tag{4}$$

provided that the F_j's are finite and mutually disjoint. For example, if $(op, id) = (+, 0)$, then (4) can be rephrased as:

$$\sum_{x \in F_1 \cup \cdots \cup F_m} f(x)\ =\ \sum_{j \in \{1,\ldots,m\}} \left(\sum_{y \in F_j} f(y) \right)$$

5 Basic Notions of Graph Theory

5.1 Nodes, Edges, and Links

Let α and β be the types of **nodes** and **edges**, respectively. Then $\alpha \times \beta \times \alpha$ is the type of **links**. A link (p, e, q) is essentially a directed edge, where $\mathsf{src}(p, e, q) \triangleq p$ is its **source**, $\mathsf{des}(p, e, q) \triangleq q$ is its **destination**, $\mathsf{via}(p, e, q) \triangleq e$ is its **via**, and $\mathsf{rev}(p, e, q) \triangleq (q, e, p)$ is its **reverse**. Two links l and l' are **almost equal** iff they are either identical or reverse of each other: $(l \approx l')\ \triangleq\ (l = l') \lor (l = \mathsf{rev}(l'))$.

5.2 What Is a Graph?

An (**undirected**, **labeled**) **graph** consists of a set $N : \alpha \rightarrow bool$ of nodes, a set $E : \beta \rightarrow bool$ of edges, and a set $L : \alpha \times \beta \times \alpha \rightarrow bool$ of links representing the incidence relation between nodes and edges, such that:

$$\begin{aligned}
&\mathsf{Graph}(N, E, L)\ \triangleq \\
&\ (\,\mathsf{Image}(\mathsf{src})(L) \subseteq N\,)\ \land\ (\,\mathsf{Image}(\mathsf{des})(L) \subseteq N\,)\ \land \\
&\ (\,\mathsf{Image}(\mathsf{via})(L) = E\,)\ \land\ (\,\mathsf{Image}(\mathsf{rev})(L) = L\,)\ \land \\
&\ \mathsf{OneOneMod}(\approx)(\mathsf{via})(L)
\end{aligned} \tag{5}$$

Definition (5) is convenient for proving closure properties of constructors of graphs (Section 8), because it contains no quantifiers and many theorems about operations on sets (such as Image, $\mathsf{OneOneMod}$, and $\subseteq$) can be proved separately. But the reader may doubt whether it captures faithfully the notion of undirected labeled graphs. The following theorem should lay such doubts to rest:

$$\begin{array}{l} \mathsf{Graph}(N, E, L) \ = \\ \quad (\,\forall\, l :: L.\ \mathsf{src}(l) \in N \wedge \mathsf{des}(l) \in N \wedge \mathsf{via}(l) \in E \wedge \mathsf{rev}(l) \in L\,) \ \wedge \\ \quad (\,\forall\, e :: E.\ \exists\, l :: L.\ e = \mathsf{via}(l)\,) \ \wedge \\ \quad (\,\forall\, l\, l' :: L.\ (\mathsf{via}(l) = \mathsf{via}(l')) \Rightarrow (l \approx l')\,) \end{array} \tag{6}$$

For a graph G, the first clause on the right-hand side of (6) requires that every link in G has its src, des, via, and rev each belonging to the appropriate component of G, the second requires that every edge in G is the via of some link in G, and the third requires that if two links in G share the same via, then they must be almost equal (i.e., either identical or reverse of each other). Consequently, for any edge e in G, there are exactly two links in G with e being their via and these two links are reverse of each other. (Note that these two links may be identical, which is the case iff e is a self-loop.) This shows that (5) captures the notion of undirectedness correctly. Note also that (5) allows infinite graphs, self-loops, and multiple edges. As the reader will see, much graph theory can be developed without ruling out these possibilities.

The three components of a graph are accessed by: $\mathsf{Node}(N, E, L) \triangleq N$, $\mathsf{Edge}(N, E, L) \triangleq E$, and $\mathsf{Link}(N, E, L) \triangleq L$.

5.3 Finite Graphs

A graph is **finite** iff its components are all finite:

$$\mathsf{GFinite}(N, E, L) \ \triangleq \ \mathsf{Finite}(N) \wedge \mathsf{Finite}(E) \wedge \mathsf{Finite}(L)$$

Note that, by (5), the finiteness of N and E already implies the finiteness of L.

5.4 Subgraphs

A graph is a **subgraph** of another graph iff the components of the former are subsets of the corresponding components of the latter:

$$(N, E, L) \sqsubseteq (N', E', L') \ \triangleq \ (N \subseteq N') \wedge (E \subseteq E') \wedge (L \subseteq L')$$

Clearly, the subgraph relation is a partial order, i.e., it is reflexive, antisymmetric, and transitive.

5.5 Adjacent Nodes, Edges, and Links of a Node

Let G be a graph (i.e., $\mathsf{Graph}(G)$) and n be a node in G (i.e., $n \in \mathsf{Node}(G)$). The set of **adjacent edges** of n in G is:

$$\mathsf{AdjEdge}(G)(n) \ \triangleq \ \{\, e \mid \exists\, l :: \mathsf{Link}(G).\ (\, n = \mathsf{src}(l)\,) \wedge (\, e = \mathsf{via}(l)\,)\,\} \tag{7}$$

Note that, by (5), substituting "$n = \mathsf{des}(l)$" for "$n = \mathsf{src}(l)$" in (7) would have resulted in an equivalent definition of $\mathsf{AdjEdge}$.

Let e be an adjacent edge of n in G (i.e., $e \in \mathsf{AdjEdge}(G)(n)$). The **outgoing link**, **incoming link**, and **opposite node** of n over e in G are:

$$\begin{aligned} \mathsf{out}(G)(n)(e) &\triangleq \varepsilon\, l :: \mathsf{Link}(G).\ (\, n = \mathsf{src}(l)\,) \wedge (\, e = \mathsf{via}(l)\,) \qquad (8)\\ \mathsf{inc}(G)(n)(e) &\triangleq \mathsf{rev}(\mathsf{out}(G)(n)(e))\\ \mathsf{opp}(G)(n)(e) &\triangleq \mathsf{des}(\mathsf{out}(G)(n)(e)) \end{aligned}$$

To avoid being bogged down in proofs involving ε-terms, a definition like (8) is used solely to prove the *well-definedness* and *uniqueness* theorems for the defined constant, from which all other properties of the defined constant are derived. For example, the well-definedness theorem for out is:

$$\begin{aligned} &\forall\, G :: \mathsf{Graph}.\ \forall\, n :: \mathsf{Node}(G).\ \forall\, e :: \mathsf{AdjEdge}(G)(n).\\ &\quad \mathsf{Link}(G)(\mathsf{out}(G)(n)(e)) \wedge (\, \mathsf{src}(\mathsf{out}(G)(n)(e)) = n\,)\\ &\qquad\qquad\qquad\qquad\qquad \wedge (\, \mathsf{via}(\mathsf{out}(G)(n)(e)) = e\,) \end{aligned}$$

and the uniqueness theorem for out is:

$$\begin{aligned} &\forall\, G :: \mathsf{Graph}.\ \forall\, n :: \mathsf{Node}(G).\ \forall\, e :: \mathsf{AdjEdge}(G)(n).\\ &\quad \forall\, l :: \mathsf{Link}(G).\ (\, \mathsf{src}(l) = n\,) \wedge (\, \mathsf{via}(l) = e\,) \Rightarrow (\, l = \mathsf{out}(G)(n)(e)\,) \end{aligned}$$

Similar well-definedness and uniqueness theorems are proved for all other constants defined using the ε-symbol, but they are not listed due to lack of space.

6 Trees

In this section, G is a graph, l is a link in G, and p and q are nodes in G. Free occurrences of G (respectively, l, p and q) in a theorem are implicitly quantified over by $\forall\, G :: \mathsf{Graph}$ ($\forall\, l :: \mathsf{Link}(G)$, $\forall\, p :: \mathsf{Node}(G)$ and $\forall\, q :: \mathsf{Node}(G)$).

6.1 Walks, Paths, and Cycles

A **walk** from p to q in G is a list of *links* in G such that the first link starts at p, the last link ends at q, and each link except the last link ends at where the next link starts:

$$\begin{aligned} &(\, \mathsf{Walk}(G)(p)(q)([\ ]) \triangleq (p = q)\,) \wedge\\ &(\, \mathsf{Walk}(G)(p)(q)([h]^\frown t) \triangleq h \in \mathsf{Link}(G) \wedge (\, p = \mathsf{src}(h)\,) \wedge \mathsf{Walk}(G)(\mathsf{des}(h))(q)(t)\,) \end{aligned}$$

Walks can also be defined as lists of *edges*, but we have found it more convenient to work with links. A walk is **edge-simple** iff it does not repeat any edge:

$$\begin{aligned} &(\, \mathsf{EdgeSimple}([\ ]) \triangleq \mathsf{T}\,) \wedge\\ &(\, \mathsf{EdgeSimple}([h]^\frown t) \triangleq \mathsf{Every}(\,\lambda\, l.\ \mathsf{via}(l) \neq \mathsf{via}(h)\,)(t) \wedge \mathsf{EdgeSimple}(t)\,) \end{aligned}$$

where $\mathsf{Every}(P)(t)$ asserts that P holds for every element of t:

$$(\,\mathsf{Every}(P)([\ ]) \triangleq \mathsf{T}\,) \wedge$$
$$(\,\mathsf{Every}(P)([h]^\frown t) \triangleq P(h) \wedge \mathsf{Every}(P)(t)\,)$$

A **path** is an edge-simple walk:

$$\mathsf{Path}(G)(p)(q)(w) \triangleq \mathsf{Walk}(G)(p)(q)(w) \wedge \mathsf{EdgeSimple}(w)$$

A **cycle** is a *non-empty* path that starts and ends at the same node:

$$\mathsf{Cycle}(G)(p)(w) \triangleq \mathsf{Path}(G)(p)(p)(w) \wedge (w \neq [\ ])$$

It is important that cycles are non-empty, for otherwise no graph (except the empty graph) would be acyclic (Section 6.3).

6.2 Reachability and Connectedness

Two nodes are **reachable** from each other iff there exists a walk between them:

$$\mathsf{Reachable}(G)(p)(q) \triangleq \exists\, w.\ \mathsf{Walk}(G)(p)(q)(w)$$

As a relation on $\mathsf{Node}(G)$, $\mathsf{Reachable}(G)$ is reflexive, symmetric, and transitive. In other words, it is an equivalence relation; the equivalence class containing p is just $\mathsf{Reachable}(G)(p)$ (viewed as a set). Although *not* every walk is a path, the existence of a walk does imply the existence of a path:

$$\mathsf{Reachable}(G)(p)(q) = \exists\, w.\ \mathsf{Path}(G)(p)(q)(w) \tag{9}$$

Theorem (9) is a good example of those facts about graphs that are obvious intuitively but messy to prove formally. Its proof is an inductive argument that, so to speak, repeatedly removes cycles from a walk until it becomes a path. A graph is **connected** iff any two of its nodes are reachable from each other:

$$\mathsf{Connected}(G) \triangleq \forall\, p\ q :: \mathsf{Node}(G).\ \mathsf{Reachable}(G)(p)(q)$$

It follows from (9) that a graph is connected iff there is a path between any two of its nodes:

$$\mathsf{Connected}(G) = \forall\, p\ q :: \mathsf{Node}(G).\ \exists\, w.\ \mathsf{Path}(G)(p)(q)(w) \tag{10}$$

6.3 Bridges and Acyclicity

A **bridge** is an edge (given as the via of a link) whose removal disconnects its two end points:

$$\mathsf{Bridge}(G)(l) \triangleq \neg\mathsf{Reachable}(\mathsf{DeleteEdge}(l)(G))(\mathsf{src}(l))(\mathsf{des}(l)) \tag{11}$$

where $\mathsf{DeleteEdge}(l)(G)$ is the graph obtained from G by deleting l's via but keeping its src and des:

$$\mathsf{DeleteEdge}(l)(N, E, L) \triangleq (N, E \setminus \{\mathsf{via}(l)\}, L \setminus \{l, \mathsf{rev}(l)\})$$

If l is a bridge in G, then it can be shown by (list) induction on walks that any walk from $\mathsf{src}(l)$ to $\mathsf{des}(l)$ in G must contain l. Conversely, if l is not a bridge, then (11) implies that there is a walk from $\mathsf{src}(l)$ to $\mathsf{des}(l)$ in $\mathsf{DeleteEdge}(l)(G)$, i.e., a walk that does not contain l. In other words, an edge is a bridge iff any walk from one end of it to the other must pass through it:

$$\begin{array}{l}\mathsf{Bridge}(G)(l) \;=\\ \quad \forall\, w :: \mathsf{Walk}(G)(\mathsf{src}(l))(\mathsf{des}(l)).\ \exists\, w'\ w''.\ (w = w' {}^\frown [l] {}^\frown w'')\end{array} \tag{12}$$

If l is a bridge in G, then (12) implies that a cycle w in G containing l must cross l at least twice. But then w is not a path, which is a contradiction. So w cannot contain l and hence must be a cycle in $\mathsf{DeleteEdge}(l)(G)$. Conversely, if l is not a bridge, then (11) implies that there is a path from $\mathsf{src}(l)$ to $\mathsf{des}(l)$ in $\mathsf{DeleteEdge}(l)(G)$, which, together with $\mathsf{rev}(l)$, forms a cycle that is in G but not in $\mathsf{DeleteEdge}(l)(G)$. In other words, an edge is a bridge iff no cycle passes through it:

$$\begin{array}{l}\mathsf{Bridge}(G)(l) \;=\\ \quad \forall\, p :: \mathsf{Node}(G).\ \forall\, w.\ \mathsf{Cycle}(G)(p)(w) \Rightarrow \mathsf{Cycle}(\mathsf{DeleteEdge}(l)(G))(p)(w)\end{array} \tag{13}$$

A graph is **acyclic** iff it contains no cycle:

$$\mathsf{Acyclic}(G) \;\triangleq\; \neg\exists\, p :: \mathsf{Node}(G).\ \exists\, w.\ \mathsf{Cycle}(G)(p)(w)$$

It follows from (13) that a graph is acyclic iff each of its edges is a bridge:

$$\mathsf{Acyclic}(G) \;=\; \forall\, l :: \mathsf{Link}(G).\ \mathsf{Bridge}(G)(l) \tag{14}$$

From (12) and (14), it can be shown by (list) induction on paths that a graph is acyclic iff there is at most one path between any two of its nodes:

$$\mathsf{Acyclic}(G) \;=\; \forall\, p\, q :: \mathsf{Node}(G).\ \forall\, w\, w' :: \mathsf{Path}(G)(p)(q).\ (w = w') \tag{15}$$

6.4 What Is a Tree?

A **tree** is a connected and acyclic graph:[1]

$$\mathsf{Tree}(G) \;\triangleq\; \mathsf{Graph}(G) \wedge \mathsf{Connected}(G) \wedge \mathsf{Acyclic}(G) \tag{16}$$

It follows immediately from (10) and (15) that a graph is a tree iff there is a unique path between any two of its nodes:

$$\mathsf{Tree}(G) \;=\; \forall\, p\, q :: \mathsf{Node}(G).\ \exists!\, w.\ \mathsf{Path}(G)(p)(q)(w) \tag{17}$$

That unique path is picked out by $\mathsf{ThePath}$:

$$\mathsf{ThePath}(G)(p)(q) \;\triangleq\; \varepsilon\, w.\ \mathsf{Path}(G)(p)(q)(w) \tag{18}$$

Most theorems about trees below are proved from (17) and (18).

[1] In the actual implementation we define $\mathsf{Tree}(G) \triangleq \mathsf{Connected}(G) \wedge \mathsf{Acyclic}(G)$ in keeping with the convention that a predicate on graphs (except Graph of course) presupposes that its argument is a graph. In this paper we use the definition (16) above in order to simplify notations.

7 Trees with Roots

In this section, T is a tree, l is a link in T, and r (the **root**) and n are nodes in T. Free occurrences of T (respectively, l, r and n) in a theorem are implicitly quantified over by $\forall T :: \mathsf{Tree}$ ($\forall l :: \mathsf{Link}(T)$, $\forall r :: \mathsf{Node}(T)$ and $\forall n :: \mathsf{Node}(T)$).

7.1 Orienting a Tree with Respect to a Root

A link is **rootward** iff it is the first link on the path from its src to the root:

$$\mathsf{Rootward}(T)(r)(l) \;\triangleq\; \mathsf{Path}(T)(\mathsf{src}(l))(r)(\,[l]^\frown\mathsf{ThePath}(T)(\mathsf{des}(l))(r)\,)$$

Every link in T is oriented by $\mathsf{Rootward}$ in a unique way, viz., either it or its rev, but never both, is rootward:

$$\begin{aligned}&(\,\mathsf{Rootward}(T)(r)(l) \lor \mathsf{Rootward}(T)(r)(\mathsf{rev}(l))\,) \land\\ \neg&(\,\mathsf{Rootward}(T)(r)(l) \land \mathsf{Rootward}(T)(r)(\mathsf{rev}(l))\,)\end{aligned} \tag{19}$$

7.2 Viewing a Tree with a Root as a Family Tree

By taking the root r as the progenitor, the tree T can be viewed as a family tree. The **parent edge** of a *non-root* node is the (unique!) adjacent edge whose corresponding outgoing link is rootward:

$$\mathsf{ParEdge}(T)(r)(n) \;\triangleq\; \varepsilon\, e :: \mathsf{AdjEdge}(T)(n).\ \mathsf{Rootward}(T)(r)(\mathsf{out}(T)(n)(e))$$

Note that $\mathsf{ParEdge}(T)(r)(r)$ is unspecified, as the root has no parent. The **kid edges** of a node are those adjacent edges whose corresponding incoming links are rootward:

$$\mathsf{KidEdge}(T)(r)(n) \;\triangleq\; \{\, e :: \mathsf{AdjEdge}(T)(n) \mid \mathsf{Rootward}(T)(r)(\mathsf{inc}(T)(n)(e))\,\}$$

It follows from (19) that every adjacent edge of the root is a kid edge and that an adjacent edge of a non-root node is either the parent edge or a kid edge, but never both:

$$\begin{aligned}&\mathsf{KidEdge}(T)(r)(n) \;=\\ &\quad \textbf{if}\,(n = r)\ \textbf{then}\ \mathsf{AdjEdge}(T)(n)\\ &\qquad\qquad\qquad \textbf{else}\ \ \mathsf{AdjEdge}(T)(n) \setminus \{\mathsf{ParEdge}(T)(r)(n)\}\end{aligned}$$

If e is the parent edge (respectively, a kid edge) of a node n, then $\mathsf{opp}(T)(n)(e)$ is called the **parent node** (a **kid node**) of n:

$$\begin{aligned}\mathsf{ParNode}(T)(r)(n) \;&\triangleq\; \mathsf{opp}(T)(n)(\mathsf{ParEdge}(T)(r)(n))\\ \mathsf{KidNode}(T)(r)(n) \;&\triangleq\; \mathsf{Image}(\mathsf{opp}(T)(n))(\mathsf{KidEdge}(T)(r)(n))\end{aligned}$$

The **descendent nodes** (or simply **descendents**) of a node n are those nodes p such that the path from p to the root must pass through n:

$$\begin{aligned}&\mathsf{DescNode}(T)(r)(n) \;\triangleq\\ &\quad \{\, p :: \mathsf{Node}(T) \mid \mathsf{Path}(T)(p)(r)(\,\mathsf{ThePath}(T)(p)(n)^\frown\mathsf{ThePath}(T)(n)(r)\,)\,\}\end{aligned}$$

It can be shown that every node is a descendent of the root:

$$\mathsf{DescNode}(T)(r)(r) \;=\; \mathsf{Node}(T)$$

In general, the descendents of a node consist of the node itself and the descendents of its kid nodes:

$$\begin{aligned}\mathsf{DescNode}(T)(r)(n) \;=\; & \{n\} \cup \\ & \mathsf{Union}(\mathsf{DescNode}(T)(r))(\mathsf{KidNode}(T)(r)(n))\end{aligned} \tag{20}$$

Furthermore, the right-hand side of (20) is a *partition* in the sense that a node is never a descendent of any of its kid nodes:

$$n \notin \mathsf{Union}(\mathsf{DescNode}(T)(r))(\mathsf{KidNode}(T)(r)(n)) \tag{21}$$

and its kid nodes have mutually disjoint sets of descendents:

$$\begin{aligned}&\forall\, k\; k' :: \mathsf{KidNode}(T)(r)(n).\\ &\;(k \neq k') \;\Rightarrow\; (\,\mathsf{DescNode}(T)(r)(k) \cap \mathsf{DescNode}(T)(r)(k') \;=\; \emptyset\,)\end{aligned} \tag{22}$$

Theorems (20), (21), and (22) ensure that (1) and (3) can be applied to obtain:

$$\begin{aligned}&\mathsf{ACI}(op, id) \wedge \mathsf{GFinite}(T) \;\Rightarrow\\ &\;\textbf{let}\ S \;\triangleq\; \mathsf{Sop}(op, id)(f) \circ \mathsf{DescNode}(T)(r)\ \textbf{in}\\ &\;(\,S(n) \;=\; f(n)\ op\ \mathsf{Sop}(op, id)(S)(\mathsf{KidNode}(T)(r)(n))\,)\end{aligned} \tag{23}$$

Informally, (20) says that:

$$\mathsf{DescNode}(T)(r)(n) \;=\; \{n\} \cup \bigcup_{k \in \mathrm{KidNode}(T)(r)(n)} \mathsf{DescNode}(T)(r)(k)$$

and, if $(op, id) = (+, 0)$, (23) says that:

$$\sum_{p \in \mathrm{DescNode}(T)(r)(n)} f(p) \;=\; f(n) + \sum_{k \in \mathrm{KidNode}(T)(r)(n)} \left(\sum_{q \in \mathrm{DescNode}(T)(r)(k)} f(q) \right)$$

7.3 Top-Down and Bottom-Up Inductions in a Family Tree

Let P be a predicate on nodes. If P holds at the root and, whenever P holds at a node, P holds at each of its kid nodes, then clearly P holds at every node in the tree:

$$\begin{aligned}&P(r) \wedge (\,\forall\, n :: \mathsf{Node}(T).\; P(n) \Rightarrow (\,\forall\, k :: \mathsf{KidNode}(T)(r)(n).\; P(k)\,)\,)\\ &\;\Rightarrow (\,\forall\, n :: \mathsf{Node}(T).\; P(n)\,)\end{aligned}$$

This is called **top-down induction** and proved by list induction on paths emanating from the root. The opposite direction, called **bottom-up induction**, says that if P holds at a node whenever P holds at each of its kid nodes, then P holds at every node in the tree, provided that the tree is finite:

$$\mathsf{GFinite}(T) \wedge (\forall n :: \mathsf{Node}(T).\ (\forall k :: \mathsf{KidNode}(T)(r)(n).\ P(k)) \Rightarrow P(n)) \\ \Rightarrow (\forall n :: \mathsf{Node}(T).\ P(n)) \qquad (24)$$

Bottom-up induction is proved by complete induction on the sizes of subtrees of T. Note that (24) has no special "base case" because its antecedent already implies that P holds at every leaf (a **leaf** is a node with no kid). Also, the finiteness assumption $\mathsf{GFinite}(T)$ is stronger than absolutely necessary: it is sufficient to assume that there is no infinite path in T. But then the proof would require transfinite induction, since a node can still have infinitely many kid nodes. Theorem (24) seems good enough for most applications.

8 Constructors of Graphs

In this section we introduce several constructors of graphs. We have chosen them *not* because of any theoretical consideration, but with an eye to the practical application of reasoning about such distributed algorithms as those in [4, 8].

8.1 Empty Graph

The **empty graph** is the graph that contains no node, no edge, and no link:

$$\mathsf{GEmpty} \ \triangleq\ (\emptyset, \emptyset, \emptyset)$$

Clearly, it is a finite tree:

$$\mathsf{Tree}(\mathsf{GEmpty}) \wedge \mathsf{GFinite}(\mathsf{GEmpty})$$

and a subgraph of any graph:

$$\forall G :: \mathsf{Graph}.\ \mathsf{GEmpty} \sqsubseteq G$$

8.2 Single-Node Graphs

For any node n, the **single-node graph** $\mathsf{SingNode}(n)$ is the graph that contains n as its sole node and no edge and no link:

$$\mathsf{SingNode}(n) \ \triangleq\ (\{n\}, \emptyset, \emptyset)$$

Clearly, it is a finite tree:

$$\forall n.\ \mathsf{Tree}(\mathsf{SingNode}(n)) \wedge \mathsf{GFinite}(\mathsf{SingNode}(n))$$

and a subgraph of any graph containing n:

$$\forall G :: \mathsf{Graph}.\ \forall n :: \mathsf{Node}(G).\ \mathsf{SingNode}(n) \sqsubseteq G$$

8.3 Merging Two Disjoint Subgraphs via a Link

In this subsection, H is a graph, G_1 and G_2 are two disjoint subgraphs of H, l is a link in H whose src is in G_1 and whose des is in G_2, and all theorems have the following implicit assumptions:

$$\begin{aligned}&\forall\, H\; G_1\; G_2 :: \mathsf{Graph}.\; G_1 \sqsubseteq H \wedge G_2 \sqsubseteq H \wedge (\mathsf{Node}(G_1) \cap \mathsf{Node}(G_2) = \emptyset) \;\Rightarrow\\ &\forall\, l :: \mathsf{Link}(H).\; \mathsf{src}(l) \in \mathsf{Node}(G_1) \wedge \mathsf{des}(l) \in \mathsf{Node}(G_2) \;\Rightarrow\end{aligned}$$

The **merge** of $G_1 = (N_1, E_1, L_1)$ and $G_2 = (N_2, E_2, L_2)$ via l is defined by:

$$\begin{aligned}&\mathsf{MergeGLG}(G_1)(l)(G_2) \;\triangleq\\ &\;(N_1 \cup N_2, E_1 \cup E_2 \cup \{\mathsf{via}(l)\}, L_1 \cup L_2 \cup \{l, \mathsf{rev}(l)\})\end{aligned}$$

Then $\mathsf{MergeGLG}(G_1)(l)(G_2)$ is not only a graph but a subgraph of H:

$$\mathsf{Graph}(\mathsf{MergeGLG}(G_1)(l)(G_2)) \wedge \mathsf{MergeGLG}(G_1)(l)(G_2) \sqsubseteq H \tag{25}$$

If both G_1 and G_2 are finite, then $\mathsf{MergeGLG}(G_1)(l)(G_2)$ is too:

$$\mathsf{GFinite}(G_1) \wedge \mathsf{GFinite}(G_2) \Rightarrow \mathsf{GFinite}(\mathsf{MergeGLG}(G_1)(l)(G_2)) \tag{26}$$

If both G_1 and G_2 are trees, then $\mathsf{MergeGLG}(G_1)(l)(G_2)$ is too:

$$\mathsf{Tree}(G_1) \wedge \mathsf{Tree}(G_2) \Rightarrow \mathsf{Tree}(\mathsf{MergeGLG}(G_1)(l)(G_2)) \tag{27}$$

8.4 Merging a Link and a Subgraph

In this subsection, H is a graph, G is a subgraph of H, l is a link in H whose src is not in G but whose des is in G, and all theorems have the following implicit assumptions:

$$\begin{aligned}&\forall\, H\; G :: \mathsf{Graph}.\; G \sqsubseteq H \;\Rightarrow\\ &\forall\, l :: \mathsf{Link}(H).\; \mathsf{src}(l) \notin \mathsf{Node}(G) \wedge \mathsf{des}(l) \in \mathsf{Node}(G) \;\Rightarrow\end{aligned}$$

The **merge** of l and G is defined by:

$$\mathsf{MergeLG}(l)(G) \;\triangleq\; \mathsf{MergeGLG}(\mathsf{SingNode}(\mathsf{src}(l)))(l)(G)$$

Theorems (25)–(27) can be specialized to obtain respectively (28)–(30) below:

$$\mathsf{Graph}(\mathsf{MergeLG}(l)(G)) \wedge \mathsf{MergeLG}(l)(G) \sqsubseteq H \tag{28}$$

$$\mathsf{GFinite}(G) \Rightarrow \mathsf{GFinite}(\mathsf{MergeLG}(l)(G)) \tag{29}$$

$$\mathsf{Tree}(G) \Rightarrow \mathsf{Tree}(\mathsf{MergeLG}(l)(G)) \tag{30}$$

9 Conclusion

In this paper we describe a formal theory of undirected labeled graphs in higher-order logic developed using the theorem-proving system HOL. It formalizes many graph-theoretic notions and rigorously proves many theorems about them. It has been successfully applied to the mechanical verification of simple distributed algorithms [2]. In the future we hope to extend it to include a theory of minimum spanning trees in order to reason about the well-known distributed algorithm of Gallager, Humblet, and Spira [4] for computing minimum spanning trees.

Acknowledgements. The author is grateful to Professors Eli Gafni and David Martin for their guidance, to Peter Homeier and the anonymous referees for their helpful comments, and to HOL hackers around the world for sustaining a lively and friendly user community.

References

1. N.G. de Bruijn, "Checking Mathematics with Computer Assistance", in *Notices of the American Mathematical Society*, Vol. 38, No. 1, pp. 8–15, Jan. 1991.
2. Ching-Tsun Chou, "Mechanical Verification of Distributed Algorithms in Higher-Order Logic", in this Proceedings.
3. Shimon Even, *Graph Algorithms*, Computer Science Press, 1979.
4. R.G. Gallager, P.A. Humblet, and P.M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees", in *ACM Trans. on Programming Languages and Systems*, Vol. 5, No. 1, pp. 66–77, Jan. 1983.
5. Michael J.C. Gordon, "HOL: A Proof Generating System for Higher-Order Logic", pp. 73–128 of G. Birtwistle and P.A. Subrahmanyam (ed.), *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, 1988.
6. Michael J.C. Gordon and Tom F. Melham (ed.), *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*, Cambridge University Press, 1993.
7. Tom F. Melham, *The HOL* `pred_set` *Library*, University of Cambridge Computer Laboratory, Feb. 1992.
8. Adrian Segall, "Distributed Network Protocols", in *IEEE Trans. on Information Theory*, Vol. 29, No. 1, pp. 23–35, Jan. 1983.
9. Wai Wong, "A Simple Graph Theory and Its Application in Railway Signalling", pp. 395–409 of M. Archer *et al.* (ed.), *Proc. of 1991 Workshop on the HOL Theorem Proving System and Its Applications*, IEEE Computer Society Press, 1992.

This article was processed using the LaTeX macro package with LLNCS style