Graphs and Trees in Isabelle/HOL

Tom Ridge

March 9, 2005

Contents

1 Introduction

We mechanise some results concerning graphs and trees. These results follow [Bol79] closely, but most definitions are standard and to be found in almost any book on elementary graph theory. Our formalisation is very close to that found in [Cho94], although we became aware of this work only after we had finished our mechanisation. Certainly there are many differences. For instance, our edges are sets of vertices, whereas the author of [Cho94] takes edges as atomic objects, and uses an incidence relation to describe when an edge connects two vertices. Like him we aim to be able to handle infinite graphs and trees (one of the primary motivations for avoiding a datatype definition): we use infinite trees, via. König's lemma, to prove properties of backwards simulations.

We start by defining the type of a graph to be an extensible record type consisting of a set of vertices and a set of edges. We introduce the concept of a path as a list of vertices, which are assumed to be adjacent in the graph. We introduce the concepts of a graph being acyclic, and of a graph being connected. A tree is defined to be an acylic and connected graph. The main theorem proved is that there is a unique path between any two nodes in a tree. We use Hilbert's Choice function to define a function from two nodes in a tree, to the unique path between them. We introduce the notion of a rooted tree by extending the graph type to contain an extra field giving the root node of the tree. This orients the graph, since a vertex $v1$ is above another vertex $v2$ if it lies in the unique path from $v2$ to the root. This allows us to introduce the notion of the parent of a node, and the children of a node, and to define *ch-up* t n, the channel leading from a node towards the root, and *ch-down* t n, the set of channels leading from a node away from the root. A leaf is defined to be a node with no children. Then we can define the flattening (from [EM03]) of a tree T as a tree $\underline{T}$ whose root is the same as T, whose vertices consist of the root and leaves of T, and whose edges are all of the form $\{r, l\}$, where r is the root, and l is a leaf.

Pattern 1.1. *Extensible Records*

This is an anti-pattern. Extensible records in Isabelle give a very intuitive way to formalise simple mathematical structures, and to enable predicates used on less defined structures (records with less components- say, a graph) to be used on more defined structures (records with more components- say, a rooted tree) which extend them. They are described in [NPW03], where their connection with the notion of a pair is outlined. In retrospect, we feel that the use of extensible records is actually a bad choice: their equivalent in terms of pairs would have been more suitable. There are several reasons:

1. *Records are non standard, and so are not supported by other HOLs. This makes porting of theorems and definitions more difficult.*
2. *Records are implemented in a way that hides many details. This can make it hard to understand how to express certain statements about records. Further, if we discover that records do not meet our needs, we must recode much of the development. Pairs are more flexible.*
3. *Records are non standard, so that someone reading a HOL proof must understand the implementation of records in order to believe in the proof. Pairs are simpler and place less demands on the reader.*

We now present the above generalities in more detail.

2 Paths, Edges, Edge list, Loop free

Paths are a subtype of lists, meeting certain conditions. Edges are undirected. We represent them as sets of vertices. Thus an edge between x and y is represented as $\{x, y\}$. We do not exclude the possibility that edges may consist of more or less than two vertices (for instance, an edge from a vertex to itself would be represented as $\{x, x\} = \{x\}$).

types *'a pre-path* = *'a list*

types *'a edge* = *'a set*

A path gives rise to a list of edges.

consts *edge-list* :: *'a list* ⇒ *'a edge list*
primrec
edge-list [] = []
edge-list (*x*#*xs*) = (*case xs of* [] ⇒ []
| (*y*#*ys*) ⇒ ({*x*,*y*}#(*edge-list xs*)))

[Bol79], in common with other books on elementary graph theory, excludes the possibility of an edge from a vertex to itself. Such an edge is usually called a loop. [Cho94] claims that most theorems can be proved without this restriction. We prefer to keep it. We define what it means for a path to be loop free (although loop freedom is properly a property of sets of edges, it is always applied in a situation where the set of edges is derived from a path).

constdefs *is-loop-free* :: *'a pre-path* ⇒ *bool*
is-loop-free p ≡ ∀ *x*. ¬ {*x*,*x*} ∈ *set* (*edge-list p*)

3 Graphs, Subgraphs

We introduce an extensible record type to model graphs. Extensible records in Isabelle/HOL are described in [NPW03]. They should be thought of as pairs, with an extra component that may be further refined, together with some meta-logical machinery to handle field selection. Not all inhabitants of this type should be considered graphs, so we introduce a predicate to capture well-formedness. We explicitly disallow loops (edges from a vertex to itself).

record *'a pre-graph* =
Verts :: *'a set*
Edges :: *'a edge set*

constdefs *is-graph* :: ('*a*,'*b*) *pre-graph-scheme* ⇒ *bool*
is-graph g ≡ *Edges g* ⊆ *Pow* (*Verts g*) ∧ (∀ *x*. {*x*,*x*} ∉ (*Edges g*))

We then introduce the notion of subgraph.

constdefs
is-subgraph :: ('*a*,'*b*) *pre-graph-scheme* ⇒ ('*a*,'*c*) *pre-graph-scheme* ⇒ *bool* (- ≤ -)
G ≤ *H* ≡ *Verts G* ⊆ *Verts H* ∧ *Edges G* ⊆ *Edges H*

The notion of subgraph is defined on a *pre-graph-scheme*, so is applicable to any extension of a graph. Unfortunately, this has the side affect that the relation *is-subgraph* is not antisymmetric. On reflection, it is probably best to define functions such as *is-subgraph* on *pre-graph*s rather than *pre-graph-scheme*s.

4 Graph of, Walks, Trails, Circuits, Cycles, Paths

So far, we have not constrained the vertices that appear in a path to be drawn from a given graph. This allows us to reason about paths without the baggage of the underlying graph. However, paths are usually considered to lie in a given graph. We connect paths to graphs in the following way. A path gives rise to a set of vertices (those appearing in the path), and a set of edges (the *edge-list* of the path). Thus, a path represents a graph. We define this graph via. the *graph-of* function.

constdefs *graph-of* :: '*a pre-path* ⇒ '*a pre-graph*
graph-of p ≡ (| *Verts* = *set p*, *Edges* = *set* (*edge-list p*) |)

A path lies in a graph G if its graph is a subgraph of G. We do not define a predicate to handle this definition, but note that frequently lemmas contain conditions asserting that the *graph-of* a path is a subgraph of another graph.

We have only introduced the type of paths as an abbreviation for lists of vertices. Does the empty list represent a path? Can paths repeat vertices? We take the following definitions from [Bol79].

We consider that the empty list does not represent a valid path. Although we could allow the empty list to represent a path (which would aid simplification, since many lemmas would not be conditional), we found that this was confusing, and in the end resulted in more work, since the cases in lemmas that dealt with the empty path were not uniform (that is, they did not resemble the proofs for the other cases, and so had to be handled separately each time). The other condition we place on a path is that it should be loop free. This respects the restriction we place on graphs. This most general form of a path is called a walk.

constdefs *is-walk* :: '*a pre-path* ⇒ *bool*

is-walk p ≡ *p* ≠ [] ∧ *is-loop-free p*

A trail is a walk, all of whose edges are distinct. Note that a trail may contain repeated vertices.

constdefs *is-trail* :: *'a pre-path* ⇒ *bool*
is-trail p ≡ *is-walk p* ∧ *distinct* (*edge-list p*)

A circuit is a trail whose start and end vertices are the same. We wish to exclude the path consisting of a single vertex from being a circuit. The loop free condition excludes a circuit from having length 2. The edge distinctness condition for a trail excludes the length from being 3. Thus any circuit must have length greater or equal to 4. Note that due to the above considerations, we could define a circuit as having length greater of equal to 2, which would make proving that a given list was a circuit easier. Instead, we derive this alternative definition, since we feel the current definition conforms more closely with what the reader expects.

constdefs *is-circuit* :: *'a pre-path* ⇒ *bool*
is-circuit p ≡ *is-trail p* ∧ *last p* = *hd p* ∧ *4* ≤ *length p*

A path is a trail whose vertices are distinct.

constdefs *is-path* :: *'a pre-path* ⇒ *bool*
is-path p ≡ *is-trail p* ∧ *distinct p*

Of course, if the vertices are distinct, the *edge-list* must also be distinct, and so this trail condition can be omitted when checking whether something is a path or not. We derive an alternative definition of a path as a non-empty distinct list.

lemma (**in** *trails-etc*) *is-path-def-2*: *is-path p* = (*p* ≠ [] ∧ *distinct p*)

Finally we define a cycle as a circuit whose vertices are distinct, excepting that the first and last vertices are the same.

constdefs *is-cycle* :: *'a pre-path* ⇒ *bool*
is-cycle p ≡ *is-circuit p* ∧ *is-path* (*tl p*)

Again, we derive a definition that takes account of the redundancy in some of the conditions.

lemma (**in** *trails-etc*) *is-cycle-def-2*: *is-cycle p* = (*hd p* = *last p* ∧ *4* ≤ *length p* ∧ *distinct* (*tl p*))

5 Acyclic, Connected

Our main development is concerned with trees. In [Bol79], trees are defined as acyclic, connected graphs. We formalise the notion of acyclicity as follows.

constdefs *is-acyclic* :: $('a,'b)$ *pre-graph-scheme* $\Rightarrow$ *bool*
is-acyclic $G \equiv \neg$ ($\exists$ *p*. *is-cycle p* $\wedge$ *graph-of p* $\leq$ *G*)

Connectedness is defined in the obvious way.

constdefs *is-connected* :: $('a,'b)$ *pre-graph-scheme* $\Rightarrow$ *bool*
is-connected G $\equiv \forall$ *v1* $\in$ *Verts G*. $\forall$ *v2* $\in$ *Verts G*.
((*v1* $\neq$ *v2*) $\longrightarrow$ ($\exists$ *p*. *is-path p* $\wedge$ *hd p* = *v1* $\wedge$ *last p* = *v2* $\wedge$ *graph-of p* $\leq$ *G*))

Before introducing trees, we define some related concepts that assist the proofs that follow.

The *cp* of two paths is the maximal common prefix of paths.

consts *cp* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
primrec
cp xxs [] = []
cp xxs (*y*#*ys*) = (*case xxs of* [] $\Rightarrow$ []
| (*x*#*xs*) $\Rightarrow$ *if x* = *y then* (*x*# *cp xs ys*) *else* [])

Conversely, the remainder of paths $p1$ and $p2$ is that part of $p1$ left after removing the common prefix.

consts *rem* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list*
primrec
rem xxs [] = *xxs*
rem xxs (*y*#*ys*) = (*case xxs of* [] $\Rightarrow$ []
| (*x*#*xs*) $\Rightarrow$ *if x* = *y then* (*rem xs ys*) *else xxs*)

These two minor concepts are used extensively in further proofs. For instance, the first proof concerns decomposing two paths into their common prefixes, vertices $v1$ and $v2$ which are different, and the remainder of the two paths.

lemma (**in** *trails-etc*) *cp-lem-2*: **assumes** *a*: *length p1* $\leq$ *length p2* **and** *b*: $\neg$ ($\exists$ *ys*. *p2* = *p1* @ *ys*)
shows $\exists$ *ys1 ys2 v1 v2*. *p1* = *cp p1 p2* @ [*v1*] @ *ys1* $\wedge$ *p2* = *cp p1 p2* @ [*v2*] @ *ys2* $\wedge$ *v1* $\neq$ *v2*

We are building up to the definition of a tree. One of the main features of a tree is that for any two nodes, there is a unique path between the two nodes. Conversely, if there are two non-equal paths from one node to another node, then we can find a cycle lying in these two paths. The main lemma is the following.

lemma *acyclic-unique-path-lem*:

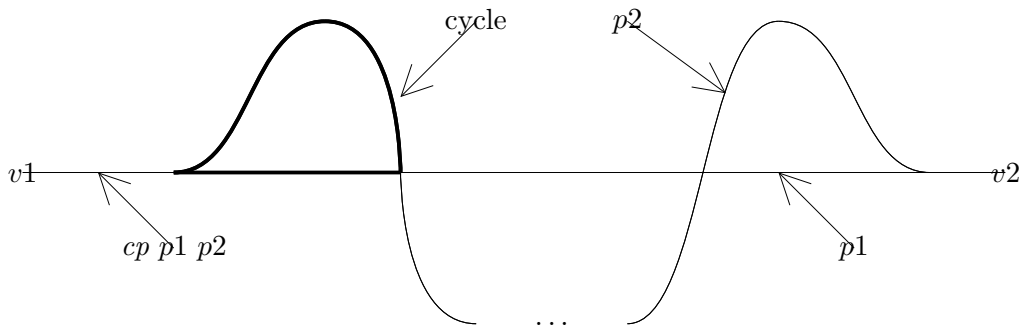

Figure 1: Constructing a Cycle

assumes *b*: *is-path p1* **and** *b'*: *hd p1 = v1 ∧ last p1 = v2*
and *c*: *is-path p2* **and** *c'*: *hd p2 = v1 ∧ last p2 = v2*
and *d*: *p1 ≠ p2* **and** *d1*: *length p1 ≤ length p2* **and** *d2*: ¬ (∃ *ys*. *p2 = p1 @ ys*)
and *e*: *v1 ≠ v2*
shows ∃ *c*. *is-cycle c ∧ set* (*edge-list c*) ⊆ *set* (*edge-list p1*) ∪ *set* (*edge-list p2*) (**is** ∃ *c*. *?Goal c*)

By way of explanation, we include a diagram, Fig. 1.

The idea is simply that if two paths $p1, p2$ start at $v1$ and end at $v2$, then one can consider the cycle formed from the first place they part, to the subsequent point at which they meet.

We use this lemma to prove the following.

lemma *acyclic-unique-path*:
assumes *b*: *is-path p1* **and** *b'*: *hd p1 = v1 ∧ last p1 = v2*
and *c*: *is-path p2* **and** *c'*: *hd p2 = v1 ∧ last p2 = v2*
and *d*: *p1 ≠ p2*
shows ∃ *c*. *is-cycle c ∧ set* (*edge-list c*) ⊆ *set* (*edge-list p1*) ∪ *set* (*edge-list p2*) (**is** ∃ *c*. *?Goal c*)

6 Trees

With these preliminaries in place, we are finally in a position to define and prove properties of trees. A tree is an acyclic, connected graph.

constdefs *is-tree* :: ('*a*,'*b*) *pre-tree-scheme* ⇒ *bool*
is-tree t ≡ *is-graph t ∧ is-connected t ∧ is-acyclic t*

In a tree, every two vertices are joined by a unique path.

lemma (**in** *is-tree-T*) *unique-path-exists*: **assumes** *v1*: *v1* ∈ *V* **and** *v2*: *v2* ∈ *V*
shows ∃! *p*. *is-path p ∧ hd p = v1 ∧ last p = v2 ∧ graph-of p* ≤ *T*

Using this fact, we can pick out, for any 2 vertices in a tree, the path joining them using Hilberts ϵ operator.

constdefs *some-path* :: $('a,'b)$ *pre-graph-scheme* $\Rightarrow$ $'a \Rightarrow 'a \Rightarrow$ $'a$ *pre-path*
some-path g v1 v2 $\equiv$ (*SOME p. is-path p*
$\wedge$ *hd p* $=$ *v1*
$\wedge$ *last p* $=$ *v2*
$\wedge$ *graph-of p* $\leq$ *g*)

In our main development, we are interested in naming channels. For instance, in a rooted tree, given a node, we are interested in the channel that leads from that node to the root of the tree. We first define the node that is adjacent to the first node, and lies in the path to the second.

constdefs *leadsto* :: $('n,'b)$ *pre-tree-scheme* $\Rightarrow$ $'n \Rightarrow 'n \Rightarrow$ $'n$
leadsto t v1 v2 $\equiv$ *hd* (*tl* (*some-path t v1 v2*))

Then the channel from one node leading to another can be defined as follows.

constdefs *chan* :: $('n,'b)$ *pre-tree-scheme* $\Rightarrow$ $'n \Rightarrow 'n \Rightarrow$ $'n$ *edge*
chan t v1 v2 $\equiv$ {*v1*, *leadsto t v1 v2*}

7 Rooted Trees

Our main development is concerned with rooted trees, that is, trees where a node has been singled out as the root of the tree. We extend the graph record type.

record $'a$ *pre-rooted-tree* $=$ $'a$ *pre-graph* $+$
root :: $'a$

And define the associated predicate.

constdefs *is-rooted-tree* :: $('a,'b)$ *pre-rooted-tree-scheme* $\Rightarrow$ *bool*
is-rooted-tree t $\equiv$ *is-tree t* $\wedge$ *root t* $\in$ *Verts t*

Having defined a root, we can orient the tree by placing the root at the top, and the leaves at the bottom (an Australian tree, if one is reading this in England). The parent of a node is adjacent to the node, and closer to the root.

constdefs *parent* :: $('n,'b)$ *pre-rooted-tree-scheme* $\Rightarrow$ $'n \Rightarrow$ $'n$
parent t v $\equiv$ *leadsto t v* (*root t*)

Similarly, the parents of a node are those nodes lying between the node and the root.

constdefs *parents* :: $('n,'b)$ *pre-rooted-tree-scheme* $\Rightarrow$ $'n \Rightarrow$ $'n$ *set*
parents t v $\equiv$ *set* (*root-path t v*)

Analogous to the parent, the children of a node n are those adjacent nodes whose parent is n.

constdefs *children* :: $('n,'b)$ *pre-rooted-tree-scheme* $\Rightarrow$ $'n \Rightarrow$ $'n$ *set*
children t v $\equiv \{c.\ c \in$ *Verts t* $\wedge\ c \neq$ *root t* $\wedge$ *parent t c* $= v\}$

The leaves of a tree are those nodes with no children.

constdefs *leaves* :: $('n,'b)$ *pre-rooted-tree-scheme* $\Rightarrow$ $'n$ *set*
leaves t $\equiv \{v.\ v \in$ *Verts t* $\wedge$ *children t v* $= \{\}\}$

8 Further Definitions

We can also define some notions from [EM03].The channel leading upwards from a node towards the root is:

constdefs *ch-up* :: $('n,'b)$ *pre-rooted-tree-scheme* $\Rightarrow$ $'n \Rightarrow$ $'n$ *edge*
ch-up t v $\equiv \{v$,*parent t v*$\}$

Again, the channels leading downwards from a node are defined as:

constdefs *ch-down* :: $('n,'b)$ *pre-rooted-tree-scheme* $\Rightarrow$ $'n \Rightarrow$ $'n$ *edge set*
ch-down t n $\equiv$ (*ch-up t*) ‘ (*children t n*)

For the next few definitions, we make a type abbreviation.

types $'a$ *tree* $=$ $'a$ *pre-rooted-tree*

The receivers, or *res* of a tree, are simply the leaves of a tree.

constdefs *res* :: $'a$ *tree* $\Rightarrow$ $'a$ *set*
res t $\equiv$ *leaves t*

The network elements, or *nes* of a tree, are those nodes which are not the root, and are not the leaves.

constdefs *nes* :: $'a$ *tree* $\Rightarrow$ $'a$ *set*
nes t $\equiv$ *Verts t* $-$ $\{$*root t*$\}$ $-$ *res t*

The vertices of a tree are simply the union of all the nodes. We already have access to the *Verts* of a tree. This definition is made for compatibility with other implementations of the tree library.

constdefs *verts* :: $'a$ *tree* $\Rightarrow$ $'a$ *set*
verts t $\equiv$ $\{$*root t*$\}$ $\cup$ *nes t* $\cup$ *res t*

The *channels-nad* of a tree, given two nodes n, n', is just the channels lying between the two nodes.

constdefs *channels-nad* :: *'a tree* ⇒ *'a* ⇒ *'a* ⇒ *'a edge set*
channels-nad t n' n ≡ *set* (*edge-list* (*some-path t n' n*))

And we can specialise this to the nodes lying between a given node and the root.

constdefs *channels-nad-root* :: *'a tree* ⇒ *'a* ⇒ *'a edge set*
channels-nad-root t ≡ *channels-nad t* (*root t*)

The channels below a given node n, are those channels between n and a receiver. Alternatively, they are the upward channels of all nodes lying below n.

constdefs *channels-below* :: *'a tree* ⇒ *'a* ⇒ *'a edge set*
channels-below t n ≡ *let nodes-below* = {*n'*. *n* ∈ *parents t n'*} − {*n*} *in*
ch-up t ‘ *nodes-below*

The internal channels are simply those channels leading from the root.

constdefs *channels-int* :: *'a tree* ⇒ *'a edge set*
channels-int t ≡ *ch-down t* (*root t*)

For many lemmas we require that there are only a finite number of edges or channels in a given tree.

constdefs *is-fin-rooted-tree* :: *'a tree* ⇒ *bool*
is-fin-rooted-tree t ≡ *is-rooted-tree t* ∧ *finite* (*channels t*)

Finally, we can define the flattening function, for simple forwarding protocols. The flattening of a tree T is another tree $\underline{T}$ which has the same root as T, connected directly to each leaf in T.

constdefs *flatten* :: *'n pre-rooted-tree* ⇒ *'n pre-rooted-tree*
flatten t ≡
(| *Verts* = {*root t*} ∪ *leaves t*, *Edges* = (⋃ *y* ∈ *leaves t* − {*root t*}. { {*root t*,*y*} }), *root* = *root t* |)

9 Further Work

We have yet to merge all the results from previous versions of our tree library into this version. In particular, we have not proved König's lemma for this definition of trees.

Clearly, this work only begins to scratch the surface of elementary graph theory. We hope in the future to polish the work already done, and to

contribute more lemmas so that this may form the basis for a general tree library for Isabelle/HOL.

References

[Bol79] Bela Bollobas. *Graph Theory: An Introductory Course*. Springer Verlag, 1979.

[Cho94] Ching-Tsun Chou. A formal theory of undirected graphs in higher-order logic. In *Lecture Notes in Computer Science 859: 7th Int. Workshop on Higher Order Logic Theorem Proving and Its Applications (HOL'94)*, pages 144–157, 1994.

[EM03] Javier Esparza and Monika Maidl. Simple representative instantiations for multicast protocols. In *TACAS 2003*, pages 128–143. Springer-Verlag LNCS 2619, 2003. Source for mechanisation available at http://homepages.inf.ed.ac.uk/s0128214/doc/monjav-fullversion.ps.gz. Current version available at http://homepages.inf.ed.ac.uk/monika/index.html.

[NPW03] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *A Proof Assistant for Higher Order Logic*, pages 158–164. Springer-Verlag, 2003.