Script **generated by TTT**

Title: FDS (07.06.2019)

Date: Fri Jun 07 08:37:02 CEST 2019

Duration: 69:18 min

Pages: 104

Chapter 7

Binary Trees

5. Binary Trees
6. Basic Functions
7. Complete and Balanced Trees

`HOL/Library/Tree.thy`

Binary trees

datatype $'a\ tree = Leaf \mid Node\ ('a\ tree)\ 'a\ ('a\ tree)$

Tree traversal

$inorder :: 'a\ tree \Rightarrow 'a\ list$

Tree traversal

$inorder :: 'a\ tree \Rightarrow 'a\ list$
$inorder\ \langle\rangle = []$
$inorder\ \langle l,\ x,\ r\rangle = inorder\ l\ @\ [x]\ @\ inorder\ r$

Tree traversal

$inorder :: 'a\ tree \Rightarrow 'a\ list$
$inorder\ \langle\rangle = []$
$inorder\ \langle l,\ x,\ r\rangle = inorder\ l\ @\ [x]\ @\ inorder\ r$

$preorder :: 'a\ tree \Rightarrow 'a\ list$

Size

$size :: 'a\ tree \Rightarrow nat$

$|\langle\rangle| = 0$
$|\langle l,\ \_,\ r\rangle| = |l| + |r| + 1$

$size1 :: 'a\ tree \Rightarrow nat$

$|\langle\rangle|_1 = 1$
$|\langle l,\ \_,\ r\rangle|_1 = |l|_1 + |r|_1$

Size

$size :: 'a\ tree \Rightarrow nat$

$|\langle\rangle| = 0$
$|\langle l,\ \_,\ r\rangle| = |l| + |r| + 1$

$size1 :: 'a\ tree \Rightarrow nat$

$|\langle\rangle|_1 = 1$
$|\langle l,\ \_,\ r\rangle|_1 = |l|_1 + |r|_1$

Lemma $|t|_1 = |t| + 1$

Size

$size :: 'a\ tree \Rightarrow nat$

$|\langle\rangle| = 0$
$|\langle l,\ \_,\ r\rangle| = |l| + |r| + 1$

$size1 :: 'a\ tree \Rightarrow nat$

$|\langle\rangle|_1 = 1$
$|\langle l,\ \_,\ r\rangle|_1 = |l|_1 + |r|_1$

Lemma $|t|_1 = |t| + 1$

Warning: $|.|$ and $|.|_1$ only on slides

Height

$height :: 'a\ tree \Rightarrow nat$

$h(\langle\rangle) = 0$
$h(\langle l,\ \_,\ r\rangle) = max\ (h(l))\ (h(r)) + 1$

Height

height :: *'a tree* ⇒ *nat*

$h(\langle\rangle) = 0$
$h(\langle l,\ \_,\ r\rangle) = max\ (h(l))\ (h(r)) + 1$

Warning: $h(.)$ only on slides

Lemma $h(t) \leq |t|$

Lemma $|t|_1 \leq 2^{h(t)}$

Minimal height

min_height :: *'a tree* ⇒ *nat*

Minimal height

min_height :: *'a tree* ⇒ *nat*

$mh(\langle\rangle) = 0$
$mh(\langle l,\ \_,\ r\rangle) = min\ (mh(l))\ (mh(r)) + 1$

Minimal height

min_height :: *'a tree* ⇒ *nat*

$mh(\langle\rangle) = 0$
$mh(\langle l,\ \_,\ r\rangle) = min\ (mh(l))\ (mh(r)) + 1$

Warning: $mh(.)$ only on slides

Lemma $mh(t) \leq h(t)$

Lemma $2^{mh(t)} \leq |t|_1$

Minimal height

$min\_height :: 'a\ tree \Rightarrow nat$

$mh(\langle\rangle) = 0$
$mh(\langle l,\ \_,\ r\rangle) = min\ (mh(l))\ (mh(r)) + 1$

Warning: $mh(.)$ only on slides

Lemma $mh(t) \leq h(t)$

Complete tree

$complete :: 'a\ tree \Rightarrow bool$

Complete tree

$complete :: 'a\ tree \Rightarrow bool$
$complete\ \langle\rangle = True$
$complete\ \langle l,\ \_,\ r\rangle =$
$(complete\ l \wedge complete\ r \wedge h(l) = h(r))$

Complete tree

$complete :: 'a\ tree \Rightarrow bool$
$complete\ \langle\rangle = True$
$complete\ \langle l,\ \_,\ r\rangle =$
$(complete\ l \wedge complete\ r \wedge h(l) = h(r))$

Lemma $complete\ t = (mh(t) = h(t))$

Complete tree

complete :: *'a tree* ⇒ *bool*
$complete\ \langle\rangle = True$
$complete\ \langle l, \_, r\rangle =$
$(complete\ l \land complete\ r \land h(l) = h(r))$

Lemma $complete\ t = (mh(t) = h(t))$

Lemma $complete\ t \Longrightarrow |t|_1 = 2^{h(t)}$

Complete tree

complete :: *'a tree* ⇒ *bool*
$complete\ \langle\rangle = True$
$complete\ \langle l, \_, r\rangle =$
$(complete\ l \land complete\ r \land h(l) = h(r))$

Complete tree

complete :: *'a tree* ⇒ *bool*
$complete\ \langle\rangle = True$
$complete\ \langle l, \_, r\rangle =$
$(complete\ l \land complete\ r \land h(l) = h(r))$

Lemma $complete\ t = (mh(t) = h(t))$

Lemma $complete\ t \Longrightarrow |t|_1 = 2^{h(t)}$

Lemma $|t|_1 = 2^{h(t)} \Longrightarrow complete\ t$

Complete tree

complete :: *'a tree* ⇒ *bool*
$complete\ \langle\rangle = True$
$complete\ \langle l, \_, r\rangle =$
$(complete\ l \land complete\ r \land h(l) = h(r))$

Lemma $complete\ t = (mh(t) = h(t))$

Lemma $complete\ t \Longrightarrow |t|_1 = 2^{h(t)}$

Lemma $|t|_1 = 2^{h(t)} \Longrightarrow complete\ t$
Lemma $|t|_1 = 2^{mh(t)} \Longrightarrow complete\ t$

Complete tree

$complete :: {}'a\ tree \Rightarrow bool$

$complete\ \langle\rangle = True$

$complete\ \langle l, \_, r\rangle =$

$(complete\ l \wedge complete\ r \wedge h(l) = h(r))$

Lemma $complete\ t = (mh(t) = h(t))$

Lemma $complete\ t \Longrightarrow |t|_1 = 2^{h(t)}$

Lemma $|t|_1 = 2^{h(t)} \Longrightarrow complete\ t$

Lemma $|t|_1 = 2^{mh(t)} \Longrightarrow complete\ t$

Corollary $\neg\ complete\ t \Longrightarrow |t|_1 < 2^{h(t)}$

Complete tree

$complete :: {}'a\ tree \Rightarrow bool$

$complete\ \langle\rangle = True$

$complete\ \langle l, \_, r\rangle =$

$(complete\ l \wedge complete\ r \wedge h(l) = h(r))$

Lemma $complete\ t = (mh(t) = h(t))$

Lemma $complete\ t \Longrightarrow |t|_1 = 2^{h(t)}$

Lemma $|t|_1 = 2^{h(t)} \Longrightarrow complete\ t$

Lemma $|t|_1 = 2^{mh(t)} \Longrightarrow complete\ t$

Corollary $\neg\ complete\ t \Longrightarrow |t|_1 < 2^{h(t)}$

Corollary $\neg\ complete\ t \Longrightarrow 2^{mh(t)} < |t|_1$

Complete tree

$complete :: {}'a\ tree \Rightarrow bool$

$complete\ \langle\rangle = True$

$complete\ \langle l, \_, r\rangle =$

$(complete\ l \wedge complete\ r \wedge h(l) = h(r))$

Lemma $complete\ t = (mh(t) = h(t))$

Lemma $complete\ t \Longrightarrow |t|_1 = 2^{h(t)}$

Lemma $|t|_1 = 2^{h(t)} \Longrightarrow complete\ t$

Lemma $|t|_1 = 2^{mh(t)} \Longrightarrow complete\ t$

Corollary $\neg\ complete\ t \Longrightarrow |t|_1 < 2^{h(t)}$

Corollary $\neg\ complete\ t \Longrightarrow 2^{mh(t)} < |t|_1$

41

Balanced tree

$balanced :: 'a\ tree \Rightarrow bool$

Balanced tree

$balanced :: 'a\ tree \Rightarrow bool$

$balanced\ t = (h(t) - mh(t) \leq 1)$

Balanced tree

$balanced :: 'a\ tree \Rightarrow bool$

$balanced\ t = (h(t) - mh(t) \leq 1)$

Balanced trees have optimal height:

Lemma If $balanced\ t$ and $|t| \leq |t'|$ then $h(t) \leq h(t')$.

Warning

- The terms *complete* and *balanced* are not defined uniquely in the literature.

Warning

- The terms *complete* and *balanced* are not defined uniquely in the literature.
- For example, Knuth calls *complete* what we call *balanced*.

Chapter 8

Search Trees

BSTs represent sets

Any tree represents a set:

$set\_tree :: 'a\ tree \Rightarrow 'a\ set$

$set\_tree\ \langle\rangle = \{\}$
$set\_tree\ \langle l,\ x,\ r\rangle = set\_tree\ l \cup \{x\} \cup set\_tree\ r$

A BST represents a set that can be searched in time $O(h(t))$

Function set_tree is called an *abstraction function* because it maps the implementation to the abstract mathematical object

bst

$bst :: {'a}\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$
$bst\ \langle l,\ a,\ r\rangle =$
$(bst\ l \wedge$
$bst\ r \wedge$
$(\forall x \in set\_tree\ l.\ x < a) \wedge (\forall x \in set\_tree\ r.\ a < x))$

bst

$bst :: {'a}\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$
$bst\ \langle l,\ a,\ r\rangle =$
$(bst\ l \wedge$
$bst\ r \wedge$
$(\forall x \in set\_tree\ l.\ x < a) \wedge (\forall x \in set\_tree\ r.\ a < x))$

Type $'a$ must be in class $linorder$ ($'a :: linorder$) where $linorder$ are *linear orders* (also called *total orders*).

bst

$bst :: {'a}\ tree \Rightarrow bool$

$bst\ \langle\rangle = True$
$bst\ \langle l,\ a,\ r\rangle =$
$(bst\ l \wedge$
$bst\ r \wedge$
$(\forall x \in set\_tree\ l.\ x < a) \wedge (\forall x \in set\_tree\ r.\ a < x))$

Type $'a$ must be in class $linorder$ ($'a :: linorder$) where $linorder$ are *linear orders* (also called *total orders*).

Note: nat, int and $real$ are in class $linorder$

Set interface

An implementation of sets of elements of type $'a$ must provide

Set interface

An implementation of sets of elements of type $'a$ must provide

- An implementation type $'s$
- $empty :: 's$
- $insert :: 'a \Rightarrow 's \Rightarrow 's$

Set interface

An implementation of sets of elements of type $'a$ must provide

- An implementation type $'s$
- $empty :: 's$
- $insert :: 'a \Rightarrow 's \Rightarrow 's$
- $delete :: 'a \Rightarrow 's \Rightarrow 's$
- $isin :: \quad 's \Rightarrow 'a \Rightarrow bool$

Map interface

Instead of a set, a search tree can also implement a map from $'a$ to $'b$:

Map interface

Instead of a set, a search tree can also implement a map from $'a$ to $'b$:

- An implementation type $'m$
- $empty :: 'm$
- $update :: 'a \Rightarrow 'b \Rightarrow 'm \Rightarrow 'm$

Map interface

Instead of a set, a search tree can also implement a map from $'a$ to $'b$:

- An implementation type $'m$
- $empty :: 'm$
- $update :: 'a \Rightarrow 'b \Rightarrow 'm \Rightarrow 'm$
- $delete :: 'a \Rightarrow 'm \Rightarrow 'm$
- $lookup :: 'm \Rightarrow 'a \Rightarrow 'b\ option$

Map interface

Instead of a set, a search tree can also implement a map from $'a$ to $'b$:

- An implementation type $'m$
- $empty :: 'm$
- $update :: 'a \Rightarrow 'b \Rightarrow 'm \Rightarrow 'm$
- $delete :: 'a \Rightarrow 'm \Rightarrow 'm$
- $lookup :: 'm \Rightarrow 'a \Rightarrow 'b\ option$

Sets are a special case of maps

Comparison of elements

Comparison of elements

We assume that the element type $'a$ is a linear order

Instead of using $<$ and $\leq$ directly:

datatype $cmp\_val = LT \mid EQ \mid GT$

Comparison of elements

We assume that the element type $'a$ is a linear order

Instead of using $<$ and $\leq$ directly:

datatype $cmp\_val = LT \mid EQ \mid GT$

$cmp\ x\ y =$
(if $x < y$ then LT else if $x = y$ then EQ else GT)

Implementation type: $'a\ tree$

$empty = Leaf$

$insert\ x\ \langle\rangle = \langle\langle\rangle,\ x,\ \langle\rangle\rangle$
$insert\ x\ \langle l,\ a,\ r\rangle =$ (case $cmp\ x\ a$ of
$\quad LT \Rightarrow \langle insert\ x\ l,\ a,\ r\rangle$
$\quad \mid EQ \Rightarrow \langle l,\ a,\ r\rangle$
$\quad \mid GT \Rightarrow \langle l,\ a,\ insert\ x\ r\rangle$)

$isin\ \langle\rangle\ x = False$
$isin\ \langle l,\ a,\ r\rangle\ x =$ (case $cmp\ x\ a$ of
$\quad LT \Rightarrow isin\ l\ x$
$\quad \mid EQ \Rightarrow True$
$\quad \mid GT \Rightarrow isin\ r\ x$)

$delete\ x\ \langle\rangle = \langle\rangle$

$delete\ x\ \langle\rangle = \langle\rangle$
$delete\ x\ \langle l,\ a,\ r\rangle =$
(case $cmp\ x\ a$ of
  $LT \Rightarrow \langle delete\ x\ l,\ a,\ r\rangle$
  | $EQ \Rightarrow$ if $r = \langle\rangle$ then l
        else let $(a',\ r') = split\_min\ r$ in $\langle l,\ a',\ r'\rangle$
  | $GT \Rightarrow \langle l,\ a,\ delete\ x\ r\rangle$)

8 Unbalanced BST
- Implementation
- Correctness
- Correctness Proof Method Based on Sorted Lists

Why is this implementation correct?

Because	$empty$	$insert$	$delete$	$isin$
simulate	$\{\}$	$\cup\ \{.\}$	$-\ \{.\}$	$\in$

Why is this implementation correct?

Because	$empty$	$insert$	$delete$	$isin$
simulate	$\{\}$	$\cup \{.\}$	$- \{.\}$	$\in$

$$set\_tree\ empty = \{\}$$

Why is this implementation correct?

Because	$empty$	$insert$	$delete$	$isin$
simulate	$\{\}$	$\cup \{.\}$	$- \{.\}$	$\in$

$$set\_tree\ empty = \{\}$$

$$set\_tree\ (insert\ x\ t) = set\_tree\ t \cup \{x\}$$

Why is this implementation correct?

Because	$empty$	$insert$	$delete$	$isin$
simulate	$\{\}$	$\cup \{.\}$	$- \{.\}$	$\in$

$$set\_tree\ empty = \{\}$$

$$set\_tree\ (insert\ x\ t) = set\_tree\ t \cup \{x\}$$

$$set\_tree\ (delete\ x\ t) = set\_tree\ t - \{x\}$$

$$isin\ t\ x = (x \in set\_tree\ t)$$

Why is this implementation correct?

Because	$empty$	$insert$	$delete$	$isin$
simulate	$\{\}$	$\cup \{.\}$	$- \{.\}$	$\in$

$$set\_tree\ empty = \{\}$$

$$set\_tree\ (insert\ x\ t) = set\_tree\ t \cup \{x\}$$

$$set\_tree\ (delete\ x\ t) = set\_tree\ t - \{x\}$$

$$isin\ t\ x = (x \in set\_tree\ t)$$

Under the assumption $bst\ t$

Also: bst must be invariant

$bst\ empty$
$bst\ t \Longrightarrow bst\ (insert\ x\ t)$
$bst\ t \Longrightarrow bst\ (delete\ x\ t)$

Also: bst must be invariant

$bst\ empty$
$bst\ t \Longrightarrow bst\ (insert\ x\ t)$
$bst\ t \Longrightarrow bst\ (delete\ x\ t)$

Key idea

Local definition:

$sorted$ means sorted w.r.t. $<$

Key idea

Local definition:

$sorted$ means sorted w.r.t. $<$

No duplicates!

$\Longrightarrow$ $bst\ t$ can be expressed as $sorted(inorder\ t)$

Key idea

Local definition:

$sorted$ means sorted w.r.t. $<$

No duplicates!

$\Longrightarrow$ $bst\ t$ can be expressed as $sorted(inorder\ t)$

Conduct proofs on sorted lists, not sets

Two kinds of invariants

- Unbalanced trees only need the invariant bst

Two kinds of invariants

- Unbalanced trees only need the invariant bst
- More efficient search trees come with additional *structural invariants* = balance criteria.

Correctness via sorted lists

Correctness proofs of (almost) all search trees
covered in this course
can be automated.

Correctness via sorted lists

Correctness proofs of (almost) all search trees
covered in this course
can be automated.

Except for the structural invariants.

Correctness via sorted lists

Correctness proofs of (almost) all search trees
covered in this course
can be automated.

Except for the structural invariants.

Therefore we concentrate on the latter.

A methodological interlude:

A closer look at ADT principles
and their realization in Isabelle

Set and binary search tree as examples
(ignoring *delete*)

Example (Set interface)

$empty :: 's$
$insert :: 'a \Rightarrow 's \Rightarrow 's$
$isin :: \quad 's \Rightarrow 'a \Rightarrow bool$

We assume that each ADT describes one

Type of Interest T

Model-oriented specification

Specify type T via a model $=$ existing HOL type A

Model-oriented specification

Specify type T via a model $=$ existing HOL type A
Motto: T should behave like A

Model-oriented specification

Specify type T via a model $=$ existing HOL type A
Motto: T should behave like A

Specification of "behaves like" via an

- *abstraction function* $\alpha :: T \Rightarrow A$

Model-oriented specification

Specify type T via a model = existing HOL type A
Motto: T should behave like A

Specification of "behaves like" via an

- *abstraction function* $\alpha :: T \Rightarrow A$

Only some elements of T represent elements of A:

- *invariant* $invar :: T \Rightarrow bool$

Model-oriented specification

Specify type T via a model = existing HOL type A
Motto: T should behave like A

Specification of "behaves like" via an

- *abstraction function* $\alpha :: T \Rightarrow A$

Only some elements of T represent elements of A:

- *invariant* $invar :: T \Rightarrow bool$

α and $invar$ are part of the interface,
but only for specification and verification purposes

Example (Set ADT)

$empty :: \ldots$
$insert :: \ldots$
$isin :: \ldots$
$set :: \quad 's \Rightarrow 'a\ set$ (name arbitrary)
$invar :: \quad 's \Rightarrow bool$ (name arbitrary)

Example (Set ADT)

$empty :: \ldots$
$insert :: \ldots$
$isin :: \ldots$
$set :: \quad 's \Rightarrow 'a\ set$ (name arbitrary)
$invar :: \quad 's \Rightarrow bool$ (name arbitrary)

$$set\ empty = \{\}$$

Example (Set ADT)

$empty$ :: ...
$insert$ :: ...
$isin$ :: ...
set :: $'s \Rightarrow 'a\ set$ (name arbitrary)
$invar$ :: $'s \Rightarrow bool$ (name arbitrary)

$$set\ empty = \{\}$$
$$set(insert\ x\ s) = set\ s \cup \{x\}$$
$$isin\ s\ x = (x \in set\ s)$$

In Isabelle: **locale**

locale Set =

In Isabelle: **locale**

locale Set =
fixes $empty :: 's$
fixes $insert :: 'a \Rightarrow 's \Rightarrow 's$
fixes $isin :: 's \Rightarrow 'a \Rightarrow bool$

In Isabelle: **locale**

locale Set =
fixes $empty :: 's$
fixes $insert :: 'a \Rightarrow 's \Rightarrow 's$
fixes $isin :: 's \Rightarrow 'a \Rightarrow bool$
fixes $set :: 's \Rightarrow 'a\ set$
fixes $invar :: 's \Rightarrow bool$
assumes $set\ empty = \{\}$
assumes $invar\ s \Longrightarrow isin\ s\ x = (x \in set\ s)$
assumes $invar\ s \Longrightarrow set(insert\ x\ s) = set\ s \cup \{x\}$

In Isabelle: **locale**

locale $Set =$

fixes $empty :: 's$
fixes $insert :: 'a \Rightarrow 's \Rightarrow 's$
fixes $isin :: 's \Rightarrow 'a \Rightarrow bool$
fixes $set :: 's \Rightarrow 'a\ set$
fixes $invar :: 's \Rightarrow bool$
assumes $set\ empty = \{\}$
assumes $invar\ s \Longrightarrow isin\ s\ x = (x \in set\ s)$
assumes $invar\ s \Longrightarrow set(insert\ x\ s) = set\ s \cup \{x\}$
assumes $invar\ empty$
assumes $invar\ s \Longrightarrow invar(insert\ x\ s)$

In Isabelle: **locale**

locale $Set =$

fixes $empty :: 's$
fixes $insert :: 'a \Rightarrow 's \Rightarrow 's$
fixes $isin :: 's \Rightarrow 'a \Rightarrow bool$
fixes $set :: 's \Rightarrow 'a\ set$
fixes $invar :: 's \Rightarrow bool$

Formally, in general

To ease notation, generalize α and $invar$ (conceptually):

Formally, in general

To ease notation, generalize α and $invar$ (conceptually):
α is the identity and $invar$ is $True$
on types other than T

Formally, in general

To ease notation, generalize α and $invar$ (conceptually):
α is the identity and $invar$ is $True$
on types other than T

Specification of each interface function f (on T):

- f must behave like some function f_A (on A):
  $invar\ t_1 \wedge \ldots \wedge invar\ t_n \Longrightarrow$
  $\alpha(f\ t_1\ \ldots\ t_n) = f_A\ (\alpha\ t_1)\ \ldots\ (\alpha\ t_n)$

Formally, in general

To ease notation, generalize α and $invar$ (conceptually):
α is the identity and $invar$ is $True$
on types other than T

Specification of each interface function f (on T):

- f must behave like some function f_A (on A):
  $invar\ t_1 \wedge \ldots \wedge invar\ t_n \Longrightarrow$
  $\alpha(f\ t_1\ \ldots\ t_n) = f_A\ (\alpha\ t_1)\ \ldots\ (\alpha\ t_n)$
  (α is a homomorphism)

Formally, in general

To ease notation, generalize α and $invar$ (conceptually):
α is the identity and $invar$ is $True$
on types other than T

Specification of each interface function f (on T):

- f must behave like some function f_A (on A):
  $invar\ t_1 \wedge \ldots \wedge invar\ t_n \Longrightarrow$
  $\alpha(f\ t_1\ \ldots\ t_n) = f_A\ (\alpha\ t_1)\ \ldots\ (\alpha\ t_n)$
  (α is a homomorphism)
- f must preserve the invariant:
  $invar\ t_1 \wedge \ldots \wedge invar\ t_n \Longrightarrow invar(f\ t_1\ \ldots\ t_n)$

The purpose of an ADT is to provide a context
for implementing generic algorithms
parameterized with the interface functions of the ADT.

Example

locale Set =
fixes ...
assumes ...
begin

fun set_of_list **where**
$set\_of\_list\ [] = empty\ |$
$set\_of\_list\ (x\ \#\ xs) = insert\ x\ (set\_of\_list\ xs)$

lemma $invar(set\_of\_list\ xs)$
by$(induction\ xs)$
$(auto\ simp:\ invar\_empty\ invar\_insert)$

end

9 Abstract Data Types
Defining ADTs
Using ADTs
Implementing ADTs

Formally, in general

To ease notation, generalize α and $invar$ (conceptually):
α is the identity and $invar$ is $True$
on types other than T

1. Implement interface
2. Prove specification

Example

Define functions $isin$ and $insert$ on type $'a\ tree$ with invariant bst.

In Isabelle: **interpretation**

In Isabelle: **interpretation**

interpretation *Set*
where $empty = Leaf$ **and** $isin = isin$
and $insert = insert$ **and** $set = set\_tree$ **and** $invar = bst$

In Isabelle: **interpretation**

interpretation *Set*
where $empty = Leaf$ **and** $isin = isin$
and $insert = insert$ **and** $set = set\_tree$ **and** $invar = bst$
proof

In Isabelle: **interpretation**

interpretation *Set*
where $empty = Leaf$ **and** $isin = isin$
and $insert = insert$ **and** $set = set\_tree$ **and** $invar = bst$
proof
  show $set\_tree\ empty = \{\}$ $\langle proof \rangle$
next
  fix s **assume** $bst\ s$
  show $set\_tree\ (insert\_tree\ x\ s) = set\_tree\ s \cup \{x\}$
  $\langle proof \rangle$

In Isabelle: **interpretation**

interpretation *Set*
where $empty = Leaf$ **and** $isin = isin$
and $insert = insert$ **and** $set = set\_tree$ **and** $invar = bst$
proof
  show $set\_tree\ empty = \{\}$ $\langle proof \rangle$
next
  fix s **assume** $bst\ s$
  show $set\_tree\ (insert\_tree\ x\ s) = set\_tree\ s \cup \{x\}$
  $\langle proof \rangle$
next
⋮
qed

Interpretation of *Set* also yields

- function $set\_of\_list :: 'a\ list \Rightarrow 'a\ tree$
- theorem $bst\ (set\_of\_list\ xs)$

Interpretation of *Set* also yields

- function $set\_of\_list :: 'a\ list \Rightarrow 'a\ tree$
- theorem $bst\ (set\_of\_list\ xs)$

Now back to search trees ...