# State Spaces: The Locale Way

Norbert Schirmer

August 27, 2014

Contents

1 Introduction 1

2 Distinctness of Names in a Binary Tree 2
   2.1 The Binary Tree . . . . . . . . . . . . . . . . . . . . . . . . 2
   2.2 Distinctness of Nodes . . . . . . . . . . . . . . . . . . . . 2
   2.3 Containment of Trees . . . . . . . . . . . . . . . . . . . . . 3

3 State Space Representation as Function 14

4 Setup for State Space Locales 16

5 Syntax for State Space Lookup and Update 17

6 Examples 18
   6.1 Benchmarks . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

1 Introduction

These theories introduce a new command called `statespace`. It’s usage is similar to `records`. However, the command does not introduce a new type but an abstract specification based on the locale infrastructure. This leads to extra flexibility in composing state space components, in particular multiple inheritance and renaming of components.

The state space infrastructure basically manages the following things:

- distinctness of field names
- projections / injections from / to an abstract `value` type
- syntax translations for lookup and update, hiding the projections and injections
- simplification procedure for lookups / updates, similar to `records`
Overview In Section 2 we define distinctness of the nodes in a binary tree and provide the basic prover tools to support efficient distinctness reasoning for field names managed by state spaces. The state is represented as a function from (abstract) names to (abstract) values as introduced in Section 3. The basic setup for state spaces is in Section 4. Some syntax for lookup and updates is added in Section 5. Finally Section 6 explains the usage of state spaces by examples.

2 Distinctness of Names in a Binary Tree

theory DistinctTreeProver
imports Main
begin

A state space manages a set of (abstract) names and assumes that the names are distinct. The names are stored as parameters of a locale and distinctness as an assumption. The most common request is to proof distinctness of two given names. We maintain the names in a balanced binary tree and formulate a predicate that all nodes in the tree have distinct names. This setup leads to logarithmic certificates.

2.1 The Binary Tree

datatype 'a tree = Node 'a tree 'a bool 'a tree | Tip

The boolean flag in the node marks the content of the node as deleted, without having to build a new tree. We prefer the boolean flag to an option type, so that the ML-layer can still use the node content to facilitate binary search in the tree. The ML code keeps the nodes sorted using the term order. We do not have to push ordering to the HOL level.

2.2 Distinctness of Nodes

primrec set-of :: 'a tree ⇒ 'a set
where
  set-of Tip = {}
| set-of (Node l x d r) = (if d then {} else {x}) ∪ set-of l ∪ set-of r

primrec all-distinct :: 'a tree ⇒ bool
where
  all-distinct Tip = True
| all-distinct (Node l x d r) =
  ((d ∨ (x ∉ set-of l ∧ x ∉ set-of r)) ∧
  set-of l ∩ set-of r = {} ∧
  all-distinct l ∧ all-distinct r)
Given a binary tree $t$ for which \textit{all-distinct} holds, given two different nodes contained in the tree, we want to write a ML function that generates a logarithmic certificate that the content of the nodes is distinct. We use the following lemmas to achieve this.

\begin{itemize}
  \item \textbf{lemma} \textit{all-distinct-left}: \textit{all-distinct} ($\text{Node } l \ x \ b \ r$) $\implies$ \textit{all-distinct} $l$
    \hspace{1cm} by simp
  \item \textbf{lemma} \textit{all-distinct-right}: \textit{all-distinct} ($\text{Node } l \ x \ b \ r$) $\implies$ \textit{all-distinct} $r$
    \hspace{1cm} by simp
  \item \textbf{lemma} \textit{distinct-left}: \textit{all-distinct} ($\text{Node } l \ x \ False \ r$) $\implies$ $y \in \text{set-of } l \implies x \neq y$
    \hspace{1cm} by auto
  \item \textbf{lemma} \textit{distinct-right}: \textit{all-distinct} ($\text{Node } l \ x \ False \ r$) $\implies$ $y \in \text{set-of } r \implies x \neq y$
    \hspace{1cm} by auto
  \item \textbf{lemma} \textit{distinct-left-right}:
    \hspace{1cm} \textit{all-distinct} ($\text{Node } l \ z \ b \ r$) $\implies$ $x \in \text{set-of } l \implies y \in \text{set-of } r \implies x \neq y$
    \hspace{1cm} by auto
  \item \textbf{lemma} \textit{in-set-root}: $x \in \text{set-of } (\text{Node } l \ x \ False \ r)$
    \hspace{1cm} by simp
  \item \textbf{lemma} \textit{in-set-left}: $y \in \text{set-of } l \implies y \in \text{set-of } (\text{Node } l \ x \ False \ r)$
    \hspace{1cm} by simp
  \item \textbf{lemma} \textit{in-set-right}: $y \in \text{set-of } r \implies y \in \text{set-of } (\text{Node } l \ x \ False \ r)$
    \hspace{1cm} by simp
  \item \textbf{lemma} \textit{swap-neq}: $x \neq y \implies y \neq x$
    \hspace{1cm} by blast
  \item \textbf{lemma} \textit{neq-to-eq-False}: $x \neq y \implies (x = y) \equiv \text{False}$
    \hspace{1cm} by simp
\end{itemize}

\section{2.3 Containment of Trees}

When deriving a state space from other ones, we create a new name tree which contains all the names of the parent state spaces and assume the predicate \textit{all-distinct}. We then prove that the new locale interprets all parent locales. Hence we have to show that the new distinctness assumption on all names implies the distinctness assumptions of the parent locales. This proof is implemented in ML. We do this efficiently by defining a kind of containment check of trees by “subtraction”. We subtract the parent tree from the new tree. If this succeeds we know that \textit{all-distinct} of the new tree implies \textit{all-distinct} of the parent tree. The resulting certificate is of the order $n \ast \log m$ where $n$ is the size of the (smaller) parent tree and $m$ the
size of the (bigger) new tree.

```plaintext
primrec delete :: 'a ⇒ 'a tree ⇒ 'a tree option
where
  delete x Tip = None
| delete x (Node l y d r) = (case delete x l of
    Some l′ ⇒
      (case delete x r of
        Some r′ ⇒ Some (Node l′ y (d ∨ (x=y)) r′)
        | None ⇒ Some (Node l′ y (d ∨ (x=y)) r))
        | None ⇒
          (case delete x r of
            Some r′ ⇒ Some (Node l y (d ∨ (x=y)) r′)
            | None ⇒ if x=y ∧ ¬d then Some (Node l y True r)
            else None))
```

```
lemma delete-Some-set-of: delete x t = Some t′ ⇒ set-of t′ ⊆ set-of t
proof (induct t arbitrary: t′)
  case Tip thus ?case by simp
next
  case (Node l y d r)
  have del: delete x (Node l y d r) = Some t′ by fact
  proof (cases delete x l)
    case (Some l′)
    note x-l-Some = this
    with Node.hyps
    have l′-l: set-of l′ ⊆ set-of l
      by simp
    show ?thesis
    proof (cases delete x r)
      case (Some r′)
      with Node.hyps
      have set-of r′ ⊆ set-of r
        by simp
      with l′-l Some x-l-Some del
      show ?thesis
        by (auto split: split-if-asm)
    next
    case None
    with l′-l Some x-l-Some del
    show ?thesis
      by (fastforce split: split-if-asm)
  qed
next
  case None
  note x-l-None = this
  show ?thesis
  proof (cases delete x r)
```
case (Some $r'$)  
with Node.hyps  
have set-of $r'$ \subseteq set-of $r$  
  by simp  
with Some $x$-l-None del  
show ?thesis  
  by (fastforce split: split-if-asm)
next  
case None  
with $x$-l-None del  
show ?thesis  
  by (fastforce split: split-if-asm)
qed

qed

lemma delete-Some-all-distinct:
  delete $x$ $t$ = Some $t'$ \implies all-distinct $t$ \implies all-distinct $t'$
proof (induct $t$ arbitrary: $t'$)
  case Tip thus ?case by simp
next  
case (Node $l$ $y$ $d$ $r$)  
have del: delete $x$ (Node $l$ $y$ $d$ $r$) = Some $t'$ by fact  
have all-distinct (Node $l$ $y$ $d$ $r$) by fact
then obtain
  dist-l: all-distinct $l$ and
  dist-r: all-distinct $r$ and
  $d$: $d$ \lor ($y$ /\notin set-of $l$ \land $y$ /\notin set-of $r$) and
  dist-l-r: set-of $l$ \cap set-of $r$ = \{\}
  by auto
show ?case
proof (cases delete $x$ $t$)
  case (Some $l'$)  
  note $x$-l-Some = this  
  from Node.hyps (1) [OF Some dist-l]
  have dist-l': all-distinct $l'$
    by simp  
  from delete-Some-set-of [OF $x$-l-Some]
  have $l'$-l: set-of $l'$ \subseteq set-of $l$.
  show ?thesis
  proof (cases delete $x$ $r$)
    case (Some $r'$)
    from Node.hyps (2) [OF Some dist-r]
    have dist-r': all-distinct $r'$
      by simp  
    from delete-Some-set-of [OF Some]
    have set-of $r'$ \subseteq set-of $r$.
    with dist-\textit{l'} dist-r' $l'$-l Some $x$-l-\textit{Some} del $d$ dist-l-r
show ?thesis 
  by fastforce
next
case None 
  with l'|l dist-l' x-l-Some del d dist-l-r dist-r
  show ?thesis 
    by fastforce
qed
next
case None 
note x-l-None = this
show ?thesis 
proof (cases delete x r)
case (Some r')
  with Node.hyps (2) [OF Some dist-r]
  have dist-r': all-distinct r'
    by simp
  from delete-Some-set-of [OF Some]
  have set-of r' ⊆ set-of r.
  with Some dist-r' x-l-None del dist-l d dist-l-r
  show ?thesis 
    by fastforce
next
case None 
  with x-l-None del dist-l dist-r d dist-l-r
  show ?thesis 
    by (fastforce split: split-if-asm)
qed 
qed
qed

lemma delete-None-set-of-cone: delete x t = None = (x ∉ set-of t)
proof (induct t)
case Tip thus ?case by simp
next
case (Node l y d r)
  thus ?case
    by (auto split: option.splits)
qed

lemma delete-Some-x-set-of: 
  delete x t = Some t' ⇒ x ∈ set-of t ∧ x ∉ set-of t'
proof (induct t arbitrary: t')
case Tip thus ?case by simp
next
case (Node l y d r)
  have del: delete x (Node l y d r) = Some t' by fact
  show ?case 
  proof (cases delete x l)

6
case (Some l')
note x-l-Some = this
from Node.hyps (1) [OF Some]
obtain x-l: x ∈ set-of l x /∈ set-of l'
  by simp
show ?thesis
proof (cases delete x r)
case (Some r')
from Node.hyps (2) [OF Some]
obtain x-r: x ∈ set-of r x /∈ set-of r'
  by simp
from x-r x-l Some x-l-Some del
show ?thesis
  by (clarsimp split: split-if-asm)
next
case None
then have x /∈ set-of r
  by (simp add: delete-None-set-of-conv)
with x-l None x-l-Some del
show ?thesis
  by (clarsimp split: split-if-asm)
qed
next
case None
note x-l-None = this
then have x-notin-l: x /∈ set-of l
  by (simp add: delete-None-set-of-conv)
show ?thesis
proof (cases delete x r)
case (Some r')
from Node.hyps (2) [OF Some]
obtain x-r: x ∈ set-of r x /∈ set-of r'
  by simp
from x-r x-notin-l Some x-l-None del
show ?thesis
  by (clarsimp split: split-if-asm)
next
case None
then have x /∈ set-of r
  by (simp add: delete-None-set-of-conv)
with None x-l-None x-notin-l del
show ?thesis
  by (clarsimp split: split-if-asm)
qed
qed

primrec subtract :: 'a tree ⇒ 'a tree ⇒ 'a tree option
where
\begin{align*}
\text{subtract Tip } t &= \text{Some } t \\
| \text{subtract } (\text{Node } l x b r) \ t &= \\
&\quad (\text{case } \text{delete } x \ t \ of \\
&\quad \quad \text{Some } t' \Rightarrow (\text{case } \text{subtract } l \ t' \ of \\
&\quad \quad \quad \quad \text{Some } t'' \Rightarrow \text{subtract } r \ t'' \\
&\quad \quad \quad \quad | \text{None } \Rightarrow \text{None}) \\
&\quad | \text{None } \Rightarrow \text{None})
\end{align*}

\text{lemma } \text{subtract-}\text{Some-set-of-res}:
\begin{align*}
\text{subtract } t_1 t_2 = \text{Some } t \implies \text{set-of } t \subseteq \text{set-of } t_2
\end{align*}

\text{proof} (\text{induct } t_1 \text{ arbitrary: } t_2 \ t)
\begin{enumerate}
\item \text{case Tip thus } ?\text{case by simp}
\item \text{next}
\begin{enumerate}
\item \text{case } (\text{Node } l x b r)
\begin{align*}
&\text{have sub: } \text{subtract } (\text{Node } l x b r) \ t_2 = \text{Some } t \text{ by fact} \\
&\text{show } ?\text{case}
\end{align*}
\begin{enumerate}
\begin{align*}
&\text{proof } (\text{cases delete } x \ t_2)
&\quad \text{case } (\text{Some } t_2')
&\qquad \text{note del-x-}\text{Some } = \text{this}
&\qquad \text{from delete-}\text{Some-set-of } \text{[OF Some]}
&\qquad \text{have } t_2'-t_2: \text{set-of } t_2' \subseteq \text{set-of } t_2.
&\qquad \text{show } ?\text{thesis}
&\quad \text{proof } (\text{cases subtract } l \ t_2')
&\qquad \text{case } (\text{Some } t_2'')
&\qquad \qquad \text{note sub-l-}\text{Some } = \text{this}
&\qquad \qquad \text{from Node.hyps } (1) \text{ [OF Some]}
&\qquad \qquad \text{have } t_2''-t_2: \text{set-of } t_2'' \subseteq \text{set-of } t_2'.
&\qquad \qquad \text{show } ?\text{thesis}
&\qquad \qquad \text{proof } (\text{cases subtract } r \ t_2'')
&\qquad \qquad \quad \text{case } (\text{Some } t_2''')
&\qquad \qquad \quad \quad \text{from Node.hyps } (2) \text{ [OF Some]}
&\qquad \qquad \quad \quad \text{have } \text{set-of } t_2''' \subseteq \text{set-of } t_2''.
&\qquad \qquad \quad \quad \text{with } \text{Some sub-l-}\text{Some del-x-}\text{Some sub } t_2''-t_2' \ t_2'-t_2
&\qquad \qquad \quad \text{show } ?\text{thesis}
&\qquad \qquad \quad \text{by simp}
&\quad \text{next}
&\quad \text{case } \text{None}
&\quad \text{with } \text{del-x-}\text{Some sub-l-}\text{Some sub}
&\quad \text{show } ?\text{thesis}
&\quad \quad \text{by simp}
&\quad \text{qed}
&\quad \text{next}
&\quad \text{case } \text{None}
&\quad \text{with } \text{del-x-}\text{Some sub}
&\quad \text{show } ?\text{thesis}
&\quad \quad \text{by simp}
&\quad \text{qed}
&\quad \text{next}
\end{enumerate}
\end{enumerate}
\end{enumerate}
case None
  with sub show \texttt{thesis} by simp
qed

lemma subtract-Some-set-of:
  \texttt{subtract} \, \texttt{t1 \, t2} = Some \, \texttt{t} \implies \texttt{set-of} \, \texttt{t1} \subseteq \texttt{set-of} \, \texttt{t2}
proof (induct \texttt{t1} arbitrary: \texttt{t2 \, t})
  case Tip thus \texttt{case} by simp
next
  case (Node \, \texttt{l \, x \, d \, r})
  have \texttt{sub} : \texttt{subtract} (Node \, \texttt{l \, x \, d \, r}) \, \texttt{t2} = Some \, \texttt{t} by fact
  show \texttt{thesis}
    proof (cases delete \texttt{x \, t2})
      case (Some \, \texttt{t2}')
      note \texttt{del-x-Some} = this
      from delete-Some-set-of [OF Some]
      have \texttt{t2'\,-t2}: set-of \texttt{t2'} \subseteq set-of \texttt{t2}.
      from delete-None-set-of-conv [of \texttt{x \, t2}] Some
      have \texttt{x-t2}: \texttt{x} \in set-of \texttt{t2}
        by simp
      show \texttt{thesis}
        proof (cases subtract \texttt{l \, t2'})
          case (Some \, \texttt{t2''})
          note \texttt{sub-l-Some} = this
          from Node.hyps (1) [OF Some]
          have \texttt{l-t2'}: set-of \texttt{l} \subseteq set-of \texttt{t2'}.
          from subtract-Some-set-of-res [OF Some]
          have \texttt{t2''\,-t2'}: set-of \texttt{t2''} \subseteq set-of \texttt{t2'}.
          show \texttt{thesis}
            proof (cases subtract \texttt{r \, t2''})
              case (Some \, \texttt{t2'''})
              from Node.hyps (2) [OF Some]
              have \texttt{r-t2'''\,-t2''}: set-of \texttt{r} \subseteq set-of \texttt{t2''}.
              from Some sub-l-Some del-x-\texttt{Some} sub \texttt{r-t2''} \texttt{l-t2'} \texttt{t2'\,-t2} \texttt{t2''\,-t2'} \texttt{x-t2}
              show \texttt{thesis}
                by auto
            next
              case None
              with del-x-\texttt{Some} sub-l-\texttt{Some} sub
              show \texttt{thesis}
                by simp
            qed
        next
          case None
          with del-x-\texttt{Some} sub
          show \texttt{thesis}
            by simp
      qed
  qed
next
  case None
    with sub show ?thesis by simp
qed
def

lemma subtract-Some-all-distinct-res:
  \( \text{subtract } t_1 \ t_2 = \text{Some } t \implies \text{all-distinct } t_2 \implies \text{all-distinct } t \)
proof (induct \( t_1 \) arbitrary: \( t_2 \) \( t \))
case Tip thus ?case by simp
next
case (Node \( l \ x \ d \ r \))
  have sub: \( \text{subtract } (\text{Node } l \ x \ d \ r) \ t_2 = \text{Some } t \) by fact
  have dist-t2: \( \text{all-distinct } t_2 \) by fact
  show ?case
  proof (cases delete \( x \) \( t_2 \))
    case (Some \( t_2' \))
    note del-x-Some = this
    from delete-Some-all-distinct [OF Some dist-t2]
    have dist-t2': \( \text{all-distinct } t_2' \).
    show ?thesis
    proof (cases \( \text{subtract } l \ t_2' \))
      case (Some \( t_2'' \))
      note sub-l-Some = this
      from Node.hyps (1) [OF Some dist-t2]
      have dist-t2'': \( \text{all-distinct } t_2'' \).
      show ?thesis
      proof (cases \( \text{subtract } r \ t_2'' \))
        case (Some \( t_2''' \))
        from Node.hyps (2) [OF Some dist-t2']
        have dist-t2'''': \( \text{all-distinct } t_2''' \).
        from Some sub-l-Some del-x-Some sub
        dist-t2'''
        show ?thesis
        by simp
      next
      case None
      with del-x-Some sub-l-Some sub
      show ?thesis
      by simp
    qed
  next
  case None
  with del-x-Some sub-l-Some sub
  show ?thesis
  by simp
  qed
next
case None
with sub show ?thesis by simp
qed
qed

lemma subtract-Some-dist-res:
  \(\text{subtract } t_1 \ t_2 = \text{Some } t \implies \text{set-of } t_1 \cap \text{set-of } t = \{\}\)
proof (induct \(t_1\) arbitrary: \(t_2\) \(t\))
  case Tip thus ?case by simp
next
  case (Node \(l\) \(x\) \(d\) \(r\))
  have sub: \(\text{subtract } (\text{Node } l \ x \ d \ r) \ t_2 = \text{Some } t\) by fact
  show ?thesis
  proof (cases delete \(x\) \(t_2\))
    case (Some \(t_2'\))
    note del-x-Some = this
    from delete-Some-x-set-of [OF Some]
    obtain \(x\)-t2: \(x \in \text{set-of } t_2\) and \(x\)-not-t2': \(x \notin \text{set-of } t_2'\)
    by simp
    from delete-Some-set-of [OF Some]
    have \(t_2'\)-t2: \(\text{set-of } t_2' \subseteq \text{set-of } t_2\).
    show ?thesis
    proof (cases subtract \(l\) \(t_2'\))
      case (Some \(t_2''\))
      note del-l-Some = this
      from Node.hyps (1) [OF Some]
      have dist-l-t2'': \(\text{set-of } l \cap \text{set-of } t_2'' = \{\}\).
      from subtract-Some-set-of-res [OF Some]
      have \(t_2''\)-t2': \(\text{set-of } t_2'' \subseteq \text{set-of } t_2'\).
      show ?thesis
      proof (cases subtract \(r\) \(t_2''\))
        case (Some \(t_2'''\))
        from Node.hyps (2) [OF Some]
        have dist-r-t2''': \(\text{set-of } r \cap \text{set-of } t_2''' = \{\}\).
        from subtract-Some-set-of-res [OF Some]
        have \(t_2''''\)-t2': \(\text{set-of } t_2''' \subseteq \text{set-of } t_2'\).
        from Some sub-l-Some del-x-Some sub \(t_2''''\)-t2' dist-l-t2'' dist-r-t2'''
        \(t_2''''\)-t2' \(t_2'\)-t2 x-not-t2'
        show ?thesis
        by auto
      next
      case None
      with del-x-Some sub-l-Some sub
      show ?thesis
      by simp
    qed
  next
  case None
next

with del-x-Some sub
show ?thesis
by simp
qed
next
case None
with sub show ?thesis by simp
qed
qed

lemma subtract-Some-all-distinct:
subtract t₁ t₂ = Some t ⟹ all-distinct t₂ ⟹ all-distinct t₁
proof (induct t₁ arbitrary: t₂ t)
case Tip thus ?case by simp
next
case (Node l x d r)
have sub: subtract (Node l x d r) t₂ = Some t by fact
have dist-t₂: all-distinct t₂ by fact
show ?case
proof (cases delete x t₂)
case (Some t₂')
  note del-x-Some = this
  from delete-Some-all-distinct [OF Some dist-t₂']
  have dist-t₂'': all-distinct t₂''.
  from delete-Some-set-of [OF Some]
  have t₂'-t₂: set-of t₂' ⊆ set-of t₂.
  from delete-Some-x-set-of [OF Some]
  obtain x-t₂: x ∈ set-of t₂ and x-not-t₂': x /∈ set-of t₂'
  by simp
show ?thesis
proof (cases subtract l t₂')
  case (Some t₂'')
  note sub-l-Some = this
  from Node.hyps (1) [OF Some dist-t₂'']
  have dist-l: all-distinct l .
  from subtract-Some-all-distinct-res [OF Some dist-t₂'']
  have dist-t₂''': all-distinct t₂'''.
  from subtract-Some-set-of [OF Some]
  have l-t₂': set-of l ⊆ set-of t₂' .
  from subtract-Some-set-of-res [OF Some]
  have t₂''-t₂': set-of t₂'' ⊆ set-of t₂'.
  from subtract-Some-dist-res [OF Some]
  have dist-l-t₂'': set-of l ∩ set-of t₂'' = {} .
  show ?thesis
proof (cases subtract r t₂'')
  case (Some t₂'''')
  from Node.hyps (2) [OF Some dist-t₂''']
  have dist-r: all-distinct r .
from subtract-Some-set-of [OF Some]
    have r-t2"': set-of r ⊆ set-of t2"'.
from subtract-Some-dist-res [OF Some]
    have dist-r-t2'''': set-of r ∩ set-of t2''' = {}.

from dist-l dist-r Some sub-l-Some del-x-Some r-t2'' l-t2' x-t2 x-not-t2'
t2''-t2' dist-l-t2'' dist-r-t2''' show ?thesis
    by auto
next
    case None
    with del-x-Some sub-l-Some sub
    show ?thesis
    by simp
qed
next
    case None
    with del-x-Some sub
    show ?thesis
    by simp
qed
next
    case None
    with sub show ?thesis by simp
qed

lemma delete-left:
assumes dist: all-distinct (Node l y d r)
assumes del-l: delete x l = Some l'
shows delete x (Node l y d r) = Some (Node l' y d r)
proof –
    from delete-Some-x-set-of [OF del-l]
    obtain x: x ∈ set-of l
        by simp
    with dist
    have delete x r = None
        by (cases delete x r) (auto dest:delete-Some-x-set-of)
    with x
    show ?thesis
        using del-l dist
        by (auto split: option.splits)
qed

lemma delete-right:
assumes dist: all-distinct (Node l y d r)
assumes del-r: delete x r = Some r'

shows $\text{delete } x \ (\text{Node } l \ y \ d \ r) = \text{Some} \ (\text{Node } l \ y \ d \ r')$

proof –
  from delete-Some-x-set-of [OF del-r]
  obtain $x$: $x \in \text{set-of } r$
    by simp
  with dist
  have $\text{delete } x \ l = \text{None}$
    by (cases $\text{delete } x \ l$) (auto dest: delete-Some-x-set-of)

  with $x$
  show $\text{thesis}$
    using del-r dist
    by (auto split: option.splits)
qed

lemma delete-root:
  assumes $\text{dist: all-distinct } (\text{Node } l \ x \ False \ r)$
  shows $\text{delete } x \ (\text{Node } l \ x \ False \ r) = \text{Some} \ (\text{Node } l \ x \ True \ r)$

proof –
  from $\text{dist}$ have $\text{delete } x \ r = \text{None}$
    by (cases $\text{delete } x \ r$) (auto dest: delete-Some-x-set-of)

  moreover
  from $\text{dist}$ have $\text{delete } x \ l = \text{None}$
    by (cases $\text{delete } x \ l$) (auto dest: delete-Some-x-set-of)

  ultimately show $\text{thesis}$
    using $\text{dist}$
    by (auto split: option.splits)
qed

lemma subtract-Node:
  assumes $\text{del: delete } x \ t = \text{Some } t'$
  assumes $\text{sub-l: subtract } l \ t' = \text{Some } t''$
  assumes $\text{sub-r: subtract } r \ t'' = \text{Some } t'''$
  shows $\text{subtract } (\text{Node } l \ x \ False \ r) \ t = \text{Some } t'''

using del sub-l sub-r
by simp

lemma subtract-Tip: $\text{subtract } \text{Tip } t = \text{Some } t$
  by simp

Now we have all the theorems in place that are needed for the certificate generating ML functions.

ML-file distinct-tree-prover.ML

end

3 State Space Representation as Function

theory StateFun imports DistinctTreeProver
The state space is represented as a function from names to values. We neither fix the type of names nor the type of values. We define lookup and update functions and provide simprocs that simplify expressions containing these, similar to HOL-records.

The lookup and update function get constructor/destructor functions as parameters. These are used to embed various HOL-types into the abstract value type. Conceptually the abstract value type is a sum of all types that we attempt to store in the state space.

The update is actually generalized to a map function. The map supplies better compositionality, especially if you think of nested state spaces.

**definition** $K$-statefun :: $\forall a \Rightarrow b \Rightarrow a$ where $K$-statefun $c \; x \equiv c$

**lemma** $K$-statefun-apply [simp]: $K$-statefun $c \; x = c$
by (simp add: $K$-statefun-def)

**lemma** $K$-statefun-comp [simp]: $(K$-statefun $c \; \circ \; f) = K$-statefun $c$
by (rule ext) (simp add: comp-def)

**lemma** $K$-statefun-cong [cong]: $K$-statefun $c \; x = K$-statefun $c \; x$
by (rule refl)

**definition** lookup :: $(\forall v \Rightarrow a) \Rightarrow (n \Rightarrow (n \Rightarrow \forall v) \Rightarrow a$
where $lookup$ $destr$ $n$ $s = destr \; (s \; n)$

**definition** update :: $(\forall v \Rightarrow a1) \Rightarrow (a2 \Rightarrow \forall v) \Rightarrow (n \Rightarrow (a1 \Rightarrow a2) \Rightarrow (n \Rightarrow \forall v) \Rightarrow (n \Rightarrow \forall v)$
where $update$ $destr$ $constr$ $n$ $f$ $s = s(n := constr \; (f \; (destr \; (s \; n))))$

**lemma** lookup-update-same:
$(\forall v. \; destr \; (constr \; v) = v) \implies lookup \; destr \; n \; (update \; destr \; constr \; n \; f \; s) = f \; (destr \; (s \; n))$
by (simp add: lookup-def update-def)

**lemma** lookup-update-id-same:
$lookup \; destr \; n \; (update \; destr \; \id \; n \; (K$-statefun \; (lookup \; id \; n \; s' \;)) \; s) = lookup \; destr \; n \; s'$
by (simp add: lookup-def update-def)

**lemma** lookup-update-other:
$n \neq m \implies lookup \; destr \; n \; (update \; destr \; constr \; m \; f \; s) = lookup \; destr \; n \; s$
by (simp add: lookup-def update-def)

**lemma** id-id-cancel: $\id \; (id \; x) = x$
by (simp add: id-def)

15
lemma destr-contr-comp-id: (\forall v. destr (contr v) = v) \implies destr \circ contr = id
by (rule ext) simp

lemma block-conj-cong: (P \land Q) = (P \land Q)
by simp

lemma conj1-False: P \equiv False \implies (P \land Q) \equiv False
by simp

lemma conj2-False: Q \equiv False \implies (P \land Q) \equiv False
by simp

lemma conj-True: P \equiv True \implies Q \equiv True \implies (P \land Q) \equiv True
by simp

lemma conj-cong: P \equiv P' \implies Q \equiv Q' \implies (P \land Q) \equiv (P' \land Q')
by simp

lemma update-apply: (update destr constr n f s x)
  = (if x = n then constr (f (destr (s n))) else s x)
by (simp add: update-def)

lemma ex-id: \exists x. id x = y
by (simp add: id-def)

lemma swap-ex-eq:
  \exists s. f s = x \equiv True \implies
  \exists s. x = f s \equiv True
apply (rule eq-reflection)
apply auto
done

lemmas meta-ext = eq-reflection [OF ext]

lemma update d c n (K-statespace (lookup d n s)) s = s
apply (simp add: update-def lookup-def)
apply (rule ext)
apply simp
oops

end

4 Setup for State Space Locales

theory StateSpaceLocale imports StateFun
For every type that is to be stored in a state space, an instance of this locale is imported in order convert the abstract and concrete values.

locale project-inject =
  fixes project :: 'value ⇒ 'a
  and inject :: 'a ⇒ 'value
  assumes project-inject-cancel [statefun-simp]: project (inject x) = x
begin

lemma ex-project [statefun-simp]: ∃ v. project v = x
proof
  show project (inject x) = x
  by (rule project-inject-cancel)
qed

lemma project-inject-comp-id [statefun-simp]: project o inject = id
  by (rule ext) (simp add: project-inject-cancel)

lemma project-inject-comp-cancel[statefun-simp]: f o project o inject = f
  by (rule ext) (simp add: project-inject-cancel)
end

end

5 Syntax for State Space Lookup and Update

theory StateSpaceSyntax
  imports StateSpaceLocale
begin

The state space syntax is kept in an extra theory so that you can choose if you want to use it or not.

syntax
  -statespace-lookup :: ('a ⇒ 'b) ⇒ 'a ⇒ 'c  (-- [60, 60] 60)
  -statespace-update :: ('a ⇒ 'b) ⇒ 'a ⇒ 'c ⇒ ('a ⇒ 'b)
  -statespace-updates :: ('a ⇒ 'b) ⇒ updbinds ⇒ ('a ⇒ 'b)  (--> [900, 0] 900)

translations
  -statespace-updates f (-updbinds b bs) ==
  -statespace-update (-statespace-updates f b) bs
  s<x:=y> == -statespace-update s x y
Did you ever dream about records with multiple inheritance? Then you should definitely have a look at statespaces. They may be what you are dreaming of. Or at least almost . . .

Isabelle allows to add new top-level commands to the system. Building on the locale infrastructure, we provide a command statespace like this:

```
statespace vars =
n::nat
b::bool
```

This resembles a record definition, but introduces sophisticated locale infrastructure instead of HOL type schemes. The resulting context postulates two distinct names \(n\) and \(b\) and projection / injection functions that convert from abstract values to \(\text{nat}\) and \(\text{bool}\). The logical content of the locale is:

```
locale vars' =
  fixes n::'name and b::'name
  assumes distinct [n, b]

  fixes project-nat::'value ⇒ nat and inject-nat::nat ⇒ 'value
  assumes \(\forall n. \text{project-nat} (\text{inject-nat} n) = n\)
```
The HOL predicate \texttt{distinct} describes distinctness of all names in the context. Locale \texttt{vars'} defines the raw logical content that is defined in the state space locale. We also maintain non-logical context information to support the user:

- Syntax for state lookup and updates that automatically inserts the corresponding projection and injection functions.
- Setup for the proof tools that exploit the distinctness information and the cancellation of projections and injections in deductions and simplifications.

This extra-logical information is added to the locale in form of declarations, which associate the name of a variable to the corresponding projection and injection functions to handle the syntax transformations, and a link from the variable name to the corresponding distinctness theorem. As state spaces are merged or extended there are multiple distinctness theorems in the context. Our declarations take care that the link always points to the strongest distinctness assumption. With these declarations in place, a lookup can be written as \texttt{s\textcdot n}, which is translated to \texttt{project-nat (s n)}, and an update as \texttt{s\langle n:=2\rangle}, which is translated to \texttt{s(n:=inject-nat 2)}. We can now establish the following lemma:

\begin{verbatim}
lemma (in vars) foo: s\langle n:=2\rangle\cdot b = s\cdot b by simp
\end{verbatim}

Here the simplifier was able to refer to distinctness of \texttt{b} and \texttt{n} to solve the equation. The resulting lemma is also recorded in locale \texttt{vars} for later use and is automatically propagated to all its interpretations. Here is another example:

\begin{verbatim}
statespace 'a varsX = NB: vars [n=N, b=B] + vars + x::'a
\end{verbatim}

The state space \texttt{varsX} imports two copies of the state space \texttt{vars}, where one has the variables renamed to upper-case letters, and adds another variable \texttt{x} of type \texttt{'a}. This type is fixed inside the state space but may get instantiated later on, analogous to type parameters of an ML-functor. The distinctness assumption is now \texttt{distinct \[N, B, n, b, x\]}, from this we can derive both \texttt{distinct \[N, B\]} and \texttt{distinct \[n, b\]}, the distinction assumptions for the two versions of locale \texttt{vars} above. Moreover we have all necessary projection and injection assumptions available. These assumptions together allow us to establish state space \texttt{varsX} as an interpretation of both instances of locale \texttt{vars}. Hence we inherit both variants of theorem \texttt{foo}: \texttt{s\langle N:=2\rangle\cdot B = s\cdot B} as well as \texttt{s\langle n:=2\rangle\cdot b = s\cdot b}. These are immediate consequences of the locale interpretation action.
The declarations for syntax and the distinctness theorems also observe the morphisms generated by the locale package due to the renaming $n = N$:

**lemma (in varsX) foo: $s\langle N := 2\rangle \cdot x = s \cdot x$ by simp**

To assure scalability towards many distinct names, the distinctness predicate is refined to operate on balanced trees. Thus we get logarithmic certificates for the distinctness of two names by the distinctness of the paths in the tree. Asked for the distinctness of two names, our tool produces the paths of the variables in the tree (this is implemented in SML, outside the logic) and returns a certificate corresponding to the different paths. Merging state spaces requires to prove that the combined distinctness assumption implies the distinctness assumptions of the components. Such a proof is of the order $m \cdot \log n$, where $n$ and $m$ are the number of nodes in the larger and smaller tree, respectively.

We continue with more examples.

**statespace 'a foo =**

\[
\begin{align*}
  f &: \text{nat} \Rightarrow \text{nat} \\
  a &: \text{int} \\
  b &: \text{nat} \\
  c &: 'a
\end{align*}
\]

**lemma (in foo) foo1:**

shown $s\langle a := i\rangle \cdot a = i$
by simp

**lemma (in foo) foo2:**

shown $(s\langle a := i\rangle) \cdot a = i$
by simp

**lemma (in foo) foo3:**

shown $(s\langle a := i\rangle) \cdot b = s \cdot b$
by simp

**lemma (in foo) foo4:**

shown $(s\langle a := i, b := j, c := k, a := x\rangle) = (s\langle b := j, c := k, a := x\rangle)$
by simp

**statespace bar =**

\[
\begin{align*}
  b &: \text{bool} \\
  c &: \text{string}
\end{align*}
\]

**lemma (in bar) bar1:**

shown $(s\langle b := \text{True}\rangle) \cdot c = s \cdot c$
by simp
You can define a derived state space by inheriting existing state spaces, renaming of components if you like, and by declaring new components.

```plaintext
statespace ('a,'b) loo = 'a foo + bar [b=B,c=C] +
X::'b
```

```plaintext
lemma (in loo) loo1:
  shows s⟨a:=i⟩·B = s·B
proof -  
  thm foo1
```

The Lemma foo1 from the parent state space is also available here:

```plaintext
?s⟨a := ?i⟩·a = ?i
```

```plaintext
have s⟨a:=i⟩·a = i
  by (rule foo1)
thm bar1
```

Note the renaming of the parameters in Lemma bar1:

```plaintext
?s⟨B := True⟩·C = ?s·C
```

```plaintext
have s⟨B:=True⟩·C = s·C
  by (rule bar1)
show ?thesis
  by simp
qed
```

```plaintext
statespace 'a dup = FA: 'a foo [f=F, a=A] + 'a foo +
x::int
```

```plaintext
lemma (in dup)
  shows s⟨a := i⟩·x = s·x
  by simp
```

```plaintext
lemma (in dup)
  shows s⟨A := i⟩·a = s·a
  by simp
```

```plaintext
lemma (in dup)
  shows s⟨A := i⟩·x = s·x
  by simp
```

There are known problems with syntax-declarations. They currently only work, when the context is already built. Hopefully this will be implemented correctly in future Isabelle versions.

It would be nice to have nested state spaces. This is logically no problem. From the locale-implementation side this may be something like an 'includes'
into a locale. When there is a more elaborate locale infrastructure in place this may be an easy exercise.

6.1 Benchmarks

Here are some bigger examples for benchmarking.

ML (fun make-benchmark n = 
    writeln (Active.sendback-markup []) 
    (statespace benchmark ` string-of-int n ` =\n     (cat-lines (map (fn i => A ` string-of-int i ` ::nat) (1 upto n)))); 
)

0.2s

\texttt{statespace \texttt{benchmark100} = A1::nat A2::nat A3::nat A4::nat A5::nat A6::nat A7::nat A8::nat A9::nat A10::nat A11::nat A12::nat A13::nat A14::nat A15::nat A16::nat A17::nat A18::nat A19::nat A20::nat A21::nat A22::nat A23::nat A24::nat A25::nat A26::nat A27::nat A28::nat A29::nat A30::nat A31::nat A32::nat A33::nat A34::nat A35::nat A36::nat A37::nat A38::nat A39::nat A40::nat A41::nat A42::nat A43::nat A44::nat A45::nat A46::nat A47::nat A48::nat A49::nat A50::nat A51::nat A52::nat A53::nat A54::nat A55::nat A56::nat A57::nat A58::nat A59::nat A60::nat A61::nat A62::nat A63::nat A64::nat A65::nat A66::nat A67::nat A68::nat A69::nat A70::nat A71::nat A72::nat A73::nat A74::nat A75::nat A76::nat A77::nat A78::nat A79::nat A80::nat A81::nat A82::nat A83::nat A84::nat A85::nat A86::nat A87::nat A88::nat A89::nat A90::nat A91::nat A92::nat A93::nat A94::nat A95::nat A96::nat A97::nat A98::nat A99::nat A100::nat

2.4s

\texttt{statespace \texttt{benchmark500} = A1::nat A2::nat A3::nat A4::nat A5::nat A6::nat A7::nat A8::nat A9::nat A10::nat A11::nat A12::nat A13::nat A14::nat A15::nat A16::nat A17::nat A18::nat A19::nat A20::nat A21::nat A22::nat A23::nat A24::nat A25::nat A26::nat A27::nat A28::nat A29::nat A30::nat A31::nat A32::nat A33::nat A34::nat A35::nat A36::nat A37::nat A38::nat A39::nat A40::nat A41::nat A42::nat A43::nat A44::nat A45::nat A46::nat A47::nat A48::nat A49::nat A50::nat A51::nat A52::nat A53::nat A54::nat A55::nat A56::nat A57::nat A58::nat A59::nat A60::nat A61::nat A62::nat A63::nat A64::nat A65::nat A66::nat A67::nat A68::nat A69::nat A70::nat A71::nat A72::nat A73::nat A74::nat A75::nat A76::nat A77::nat A78::nat A79::nat A80::nat A81::nat A82::nat A83::nat A84::nat A85::nat A86::nat A87::nat A88::nat A89::nat A90::nat A91::nat A92::nat A93::nat A94::nat A95::nat A96::nat A97::nat A98::nat A99::nat A100::nat A101::nat A102::nat A103::nat A104::nat A105::nat A106::nat A107::nat A108::nat A109::nat A110::nat A111::nat A112::nat A113::nat A114::nat A115::nat A116::nat A117::nat A118::nat
lemma (in benchmark100) test: $s \cdot A100 = s \cdot A100$ by simp
lemma (in benchmark500) test: $s \cdot A100 = s \cdot A100$ by simp
lemma (in benchmark1000) test: $s \cdot A100 = s \cdot A100$ by simp
end