State Spaces: The Locale Way

Norbert Schirmer

November 22, 2007

Contents

1 Introduction

These theories introduce a new command called **statespace**. It's usage is similar to **record**s. 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*
uses (*distinct-tree-prover.ML*)
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 $'a\ tree = Node\ 'a\ tree\ 'a\ bool\ 'a\ tree \mid Tip$

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

consts *set-of*:: $'a\ tree \Rightarrow 'a\ set$
primrec
$set\text{-}of\ Tip = \{\}$
$set\text{-}of\ (Node\ l\ x\ d\ r) = (if\ d\ then\ \{\}\ else\ \{x\}) \cup set\text{-}of\ l \cup set\text{-}of\ r$

consts *all-distinct*:: $'a\ tree \Rightarrow bool$
primrec
$all\text{-}distinct\ Tip = True$
$all\text{-}distinct\ (Node\ l\ x\ d\ r) = ((d \vee (x \notin set\text{-}of\ l \wedge x \notin set\text{-}of\ r)) \wedge$
$set\text{-}of\ l \cap set\text{-}of\ r = \{\} \wedge$
$all\text{-}distinct\ l \wedge all\text{-}distinct\ r)$

Given a binary tree t for which *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.

lemma *all-distinct-left*:
all-distinct (*Node l x b r*) $\Longrightarrow$ *all-distinct l*
by *simp*

lemma *all-distinct-right*: *all-distinct* (*Node l x b r*) $\Longrightarrow$ *all-distinct r*
by *simp*

lemma *distinct-left*: $[\![$*all-distinct* (*Node l x False r*); $y \in$ *set-of l* $]\!] \Longrightarrow x \neq y$
by *auto*

lemma *distinct-right*: $[\![$*all-distinct* (*Node l x False r*); $y \in$ *set-of r* $]\!] \Longrightarrow x \neq y$
by *auto*

lemma *distinct-left-right*: $[\![$*all-distinct* (*Node l z b r*); $x \in$ *set-of l*; $y \in$ *set-of r*$]\!]$
$\Longrightarrow x \neq y$
by *auto*

lemma *in-set-root*: $x \in$ *set-of* (*Node l x False r*)
by *simp*

lemma *in-set-left*: $y \in$ *set-of l* $\Longrightarrow$ $y \in$ *set-of* (*Node l x False r*)
by *simp*

lemma *in-set-right*: $y \in$ *set-of r* $\Longrightarrow$ $y \in$ *set-of* (*Node l x False r*)
by *simp*

lemma *swap-neq*: $x \neq y \Longrightarrow y \neq x$
by *blast*

lemma *neq-to-eq-False*: $x \neq y \Longrightarrow (x = y) \equiv$ *False*
by *simp*

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 assumme the predicate *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 *all-distinct* of the new tree implies *all-distinct* of the parent tree. The resulting certificate is of the

order $n * log\ m$ where n is the size of the (smaller) parent tree and m the size of the (bigger) new tree.

```
consts delete :: 'a ⇒ 'a tree ⇒ 'a tree option
primrec
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: ⋀t'. delete x t = Some t' ⟹ set-of t' ⊆ set-of t
proof (induct 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)
    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 (fastsimp 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' ⊆ set-of r
        by simp
      with Some x-l-None del
      show ?thesis
        by (fastsimp split: split-if-asm)
    next
      case None
      with x-l-None del
      show ?thesis
        by (fastsimp split: split-if-asm)
    qed
  qed
qed

lemma delete-Some-all-distinct:
⋀t'. ⟦delete x t = Some t'; all-distinct t⟧ ⟹ all-distinct t'
proof (induct 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 ∨ (y ∉ set-of l ∧ y ∉ set-of r) and
    dist-l-r: set-of l ∩ set-of r = {}
    by auto
  show ?case
  proof (cases delete x l)
    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' ⊆ 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' ⊆ set-of r.
```

```
      with dist-l' dist-r' l'-l Some x-l-Some del d dist-l-r
      show ?thesis
        by fastsimp
    next
      case None
      with l'-l dist-l'  x-l-Some del d dist-l-r dist-r
      show ?thesis
        by fastsimp
    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 fastsimp
    next
      case None
      with x-l-None del dist-l dist-r d dist-l-r
      show ?thesis
        by (fastsimp split: split-if-asm)
    qed
  qed
qed

lemma delete-None-set-of-conv: 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:
  ⋀t'. delete x t = Some t' ⟹ x ∈ set-of t ∧ x ∉ set-of t'
proof (induct 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)
    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
qed
```

```
consts subtract :: 'a tree ⇒ 'a tree ⇒ 'a tree option
primrec
subtract Tip t = Some t
subtract (Node l x b r) t =
  (case delete x t of
    Some t' ⇒ (case subtract l t' of
                 Some t'' ⇒ subtract r t''
               | None ⇒ None)
   | None ⇒ None)

lemma subtract-Some-set-of-res:
  ⋀t2 t. subtract t1 t2 = Some t ⟹ set-of t ⊆ set-of t2
proof (induct t1)
  case Tip thus ?case by simp
next
  case (Node l x b r)
  have sub: subtract (Node l x b r) t2 = Some t by fact
  show ?case
  proof (cases delete x t2)
    case (Some t2')
    note del-x-Some = this
    from delete-Some-set-of [OF Some]
    have t2'-t2: set-of t2' ⊆ set-of t2 .
    show ?thesis
    proof (cases subtract l t2')
      case (Some t2'')
      note sub-l-Some = this
      from Node.hyps (1) [OF Some]
      have t2''-t2': set-of t2'' ⊆ set-of t2' .
      show ?thesis
      proof (cases subtract r t2'')
        case (Some t2''')
        from Node.hyps (2) [OF Some ]
        have set-of t2''' ⊆ set-of t2'' .
        with Some sub-l-Some del-x-Some sub t2''-t2' t2'-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
      show ?thesis
        by simp
    qed
```

```
  next
    case None
    with sub show ?thesis by simp
  qed
qed

lemma subtract-Some-set-of:
  ⋀t₂ t. subtract t₁ t₂ = Some t ⟹ set-of t₁ ⊆ set-of t₂
proof (induct t₁)
  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
  show ?case
  proof (cases delete x t₂)
    case (Some t₂')
    note del-x-Some = this
    from delete-Some-set-of [OF Some]
    have t2'-t2: set-of t₂' ⊆ set-of t₂ .
    from delete-None-set-of-conv [of x t₂] Some
    have x-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]
      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' .
      show ?thesis
      proof (cases subtract r t₂'')
        case (Some t₂''')
        from Node.hyps (2) [OF Some ]
        have r-t₂'': set-of r ⊆ set-of t₂'' .
        from Some sub-l-Some del-x-Some sub r-t₂'' l-t2' t2'-t2 t2''-t2' x-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
qed

lemma subtract-Some-all-distinct-res:
  ⋀t2 t. ⟦subtract t1 t2 = Some t; all-distinct t2⟧ ⟹ all-distinct t
proof (induct t1)
  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' .
    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-t2'': all-distinct t2'' .
      show ?thesis
      proof (cases subtract r t2'')
        case (Some t2''')
        from Node.hyps (2) [OF Some dist-t2'']
        have dist-t2''': all-distinct t2''' .
        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
      show ?thesis
        by simp
    qed
  next
```

```
    case None
    with sub show ?thesis by simp
  qed
qed


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

        from Some sub-l-Some del-x-Some sub t2'''-t2'' dist-l-t2'' dist-r-t2'''
            t2''-t2' t2'-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
    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:
 ⋀t2 t. ⟦subtract t1 t2 = Some t; all-distinct t2⟧ ⟹ all-distinct t1
proof (induct t1)
 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
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 ∈ set-of l
    by simp
  moreover with dist
  have delete x r = None
    by (cases delete x r) (auto dest:delete-Some-x-set-of)

  ultimately
  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 *delete x* (*Node l y d r*) = *Some* (*Node l y d r′*)
proof −
from *delete-Some-x-set-of* [*OF del-r*]
obtain *x* ∈ *set-of r*
by *simp*
moreover with *dist*
have *delete x l* = *None*
by (*cases delete x l*) (*auto dest*:*delete-Some-x-set-of*)

ultimately
show *?thesis*
using *del-r dist*
by (*auto split*: *option.splits*)
qed

lemma *delete-root*:
assumes *dist*: *all-distinct* (*Node l x False r*)
shows *delete x* (*Node l x False r*) = *Some* (*Node l x True r*)
proof −
from *dist* **have** *delete x r* = *None*
by (*cases delete x r*) (*auto dest*:*delete-Some-x-set-of*)
moreover
from *dist* **have** *delete x l* = *None*
by (*cases delete x l*) (*auto dest*:*delete-Some-x-set-of*)
ultimately show *?thesis*
using *dist*
by (*auto split*: *option.splits*)
qed

lemma *subtract-Node*:
assumes *del*: *delete x t* = *Some t′*
assumes *sub-l*: *subtract l t′* = *Some t″*
assumes *sub-r*: *subtract r t″* = *Some t‴*
shows *subtract* (*Node l x False r*) *t* = *Some t‴*
using *del sub-l sub-r*
by *simp*

lemma *subtract-Tip*: *subtract Tip t* = *Some t*
by *simp*

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

use *distinct-tree-prover.ML*

end

3 State Space Representation as Function

theory *StateFun* **imports** *DistinctTreeProver*
begin

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.

constdefs *K-statefun*:: $'a \Rightarrow 'b \Rightarrow 'a$ *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*: *K-statefun-apply comp-def*)

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

constdefs *lookup*:: $('v \Rightarrow 'a) \Rightarrow 'n \Rightarrow ('n \Rightarrow 'v) \Rightarrow 'a$
lookup destr n s $\equiv$ *destr* (*s n*)

constdefs *update*::
$('v \Rightarrow 'a1) \Rightarrow ('a2 \Rightarrow 'v) \Rightarrow 'n \Rightarrow ('a1 \Rightarrow 'a2) \Rightarrow ('n \Rightarrow 'v) \Rightarrow ('n \Rightarrow 'v)$
update destr constr n f s $\equiv$ *s*(*n* := *constr* (*f* (*destr* (*s n*))))

lemma *lookup-update-same*:
($\bigwedge v$. *destr* (*constr v*) = *v*) $\Longrightarrow$ *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*:
$lookup\ destr\ n\ (update\ destr'\ id\ n\ (K\text{-}statefun\ (lookup\ id\ n\ s'))\ s) =$
$lookup\ destr\ n\ s'$
by (*simp add*: *lookup-def update-def*)

lemma *lookup-update-other*:
$n{\neq}m \Longrightarrow lookup\ destr\ n\ (update\ destr'\ constr\ m\ f\ s) = lookup\ destr\ n\ s$
by (*simp add*: *lookup-def update-def*)

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

lemma *destr-contstr-comp-id*:
$(\bigwedge v.\ destr\ (constr\ v) = v) \Longrightarrow destr \circ constr = id$
by (*rule ext*) *simp*

lemma *block-conj-cong*: $(P \wedge Q) = (P \wedge Q)$
by *simp*

lemma *conj1-False*: $(P{\equiv}False) \Longrightarrow (P \wedge Q) \equiv False$
by *simp*

lemma *conj2-False*: $[\![Q{\equiv}False]\!] \Longrightarrow (P \wedge Q) \equiv False$
by *simp*

lemma *conj-True*: $[\![P{\equiv}True;\ Q{\equiv}True]\!] \Longrightarrow (P \wedge Q) \equiv True$
by *simp*

lemma *conj-cong*: $[\![P{\equiv}P';\ Q{\equiv}Q']\!] \Longrightarrow (P \wedge Q) \equiv (P' \wedge Q')$
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*:
$\exists s.\ f\ s = x \equiv True \Longrightarrow$
$\exists s.\ x = f\ s \equiv True$
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*
uses *state-space.ML state-fun.ML*
begin

setup *StateFun.setup*

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*

lemma (**in** *project-inject*)
ex-project [*statefun-simp*]: ∃ *v*. *project v* = *x*
apply (*rule-tac x*= *inject x* **in** *exI*)
apply (*simp add*: *project-inject-cancel*)
done

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

lemma (**in** *project-inject*)
project-inject-comp-cancel[*statefun-simp*]: *f* ∘ *project* ∘ *inject* = *f*
by (*rule ext*) (*simp add*: *project-inject-cancel*)

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 :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (-·- [60,60] 60)
-statespace-update :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c \Rightarrow ('a \Rightarrow 'b)$
-statespace-updates :: $('a \Rightarrow 'b) \Rightarrow updbinds \Rightarrow ('a \Rightarrow 'b)$ (-<-> [900,0] 900)

translations
-statespace-updates f (*-updbinds b bs*) == *-statespace-updates* (*-statespace-updates f b*) *bs*
s<*x*:=*y*> == *-statespace-update s x y*

parse-translation (**advanced**)
⟪
[(*-statespace-lookup*,*StateSpace.lookup-tr*),
(*-get*,*StateSpace.lookup-tr*),
(*-statespace-update*,*StateSpace.update-tr*)]
⟫

print-translation (**advanced**)
⟪
[(*lookup*,*StateSpace.lookup-tr′*),
(*StateFun.lookup*,*StateSpace.lookup-tr′*),
(*update*,*StateSpace.update-tr′*),
(*StateFun.update*,*StateSpace.update-tr′*)]
⟫

end

6 Examples

theory *StateSpaceEx*
imports *StateSpaceLocale StateSpaceSyntax*

begin

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 *nat* and *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 ⋀*n*. *project-nat* (*inject-nat n*) = *n*

  fixes *project-bool*::*′value* ⇒ *bool* **and** *inject-bool*::*bool* ⇒ *′value*
  assumes ⋀*b*. *project-bool* (*inject-bool b*) = *b*

The HOL predicate *distinct* describes distinctness of all names in the context. Locale *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 $s{\cdot}n$, which is translated to *project-nat* (*s n*), and an update as $s\langle n := 2\rangle$, which is translated to $s(n := \textit{inject-nat}\ 2)$. We can now establish the following lemma:

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

Here the simplifier was able to refer to distinctness of *b* and *n* to solve the equation. The resulting lemma is also recorded in locale *vars* for later use

and is automatically propagated to all its interpretations. Here is another example:

statespace *'a varsX = vars [n=N, b=B] + vars + x::'a*

The state space *varsX* imports two copies of the state space *vars*, where one has the variables renamed to upper-case letters, and adds another variable x of type $'a$. 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 *distinct* $[N, B, n, b, x]$, from this we can derive both *distinct* $[N, B]$ and *distinct* $[n, b]$, the distinction assumptions for the two versions of locale *vars* above. Moreover we have all necessary projection and injection assumptions available. These assumptions together allow us to establish state space *varsX* as an interpretation of both instances of locale *vars*. Hence we inherit both variants of theorem *foo*: $s\langle N := 2\rangle \cdot B = s \cdot B$ as well as $s\langle n := 2\rangle \cdot b = s \cdot b$. 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 =*
  f::nat⇒nat
  a::int
  b::nat
  c::'a

lemma (**in** *foo*) *foo1*:
  shows $s\langle a := i\rangle \cdot a = i$
  by *simp*

lemma (**in** *foo*) *foo2*:

shows $(s\langle a{:=}i\rangle)\cdot a = i$
by *simp*

lemma (**in** *foo*) *foo3*:
shows $(s\langle a{:=}i\rangle)\cdot b = s\cdot b$
by *simp*

lemma (**in** *foo*) *foo4*:
shows $(s\langle a{:=}i,b{:=}j,c{:=}k,a{:=}x\rangle) = (s\langle b{:=}j,c{:=}k,a{:=}x\rangle)$
by *simp*

statespace *bar* =
b::*bool*
c::*string*

lemma (**in** *bar*) *bar1*:
shows $(s\langle b{:=}True\rangle)\cdot c = s\cdot c$
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.

statespace $('a,'b)$ *loo* = $'a$ *foo* + *bar* $[b{=}B,c{=}C]$ +
X::$'b$

lemma (**in** *loo*) *loo1*:
shows $s\langle a{:=}i\rangle\cdot B = s\cdot B$
proof −
thm *foo1*

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

$$?s\langle a := ?i\rangle\cdot a = ?i$$

.

have $s{<}a{:=}i{>}\cdot a = i$
by (*rule foo1*)
thm *bar1*

Note the renaming of the parameters in Lemma *bar1*:

$$?s\langle B := True\rangle\cdot C = ?s\cdot C$$

.

have $s{<}B{:=}True{>}\cdot C = s\cdot C$
by (*rule bar1*)
show *?thesis*
by *simp*
qed

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

lemma (**in** *dup*)
shows *s<a := i>·x = s·x*
by *simp*

lemma (**in** *dup*)
shows *s<A := i>·a = s·a*
by *simp*

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.

lemma
includes *foo*
shows *True*
term *s<a := i>·a = i*
by *simp*

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.

end