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 6

4 Setup for State Space Locales 7

5 Syntax for State Space Lookup and Update 8

6 Examples 9
   6.1 Benchmarks 12

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.

\textbf{lemma all-distinct-left:} \textbf{all-distinct} (\( \text{Node} \ l \ x \ b \ r \)) \( \Rightarrow \) \textbf{all-distinct} \( l \) \\
\textit{⟨proof⟩}

\textbf{lemma all-distinct-right:} \textbf{all-distinct} (\( \text{Node} \ l \ x \ b \ r \)) \( \Rightarrow \) \textbf{all-distinct} \( r \) \\
\textit{⟨proof⟩}

\textbf{lemma distinct-left:} \textbf{all-distinct} (\( \text{Node} \ l \ x \ False \ r \)) \( \Rightarrow \) \( y \in \text{set-of} \ l \Rightarrow x \neq y \) \\
\textit{⟨proof⟩}

\textbf{lemma distinct-right:} \textbf{all-distinct} (\( \text{Node} \ l \ x \ False \ r \)) \( \Rightarrow \) \( y \in \text{set-of} \ r \Rightarrow x \neq y \) \\
\textit{⟨proof⟩}

\textbf{lemma distinct-left-right:} \\
\textbf{all-distinct} (\( \text{Node} \ l \ z \ b \ r \)) \( \Rightarrow \) \( x \in \text{set-of} \ l \Rightarrow y \in \text{set-of} \ r \Rightarrow x \neq y \) \\
\textit{⟨proof⟩}

\textbf{lemma in-set-root:} \( x \in \text{set-of} \ (\text{Node} \ l \ x \ False \ r) \) \\
\textit{⟨proof⟩}

\textbf{lemma in-set-left:} \( y \in \text{set-of} \ l \Rightarrow y \in \text{set-of} \ (\text{Node} \ l \ x \ False \ r) \) \\
\textit{⟨proof⟩}

\textbf{lemma in-set-right:} \( y \in \text{set-of} \ r \Rightarrow y \in \text{set-of} \ (\text{Node} \ l \ x \ False \ r) \) \\
\textit{⟨proof⟩}

\textbf{lemma swap-neq:} \( x \neq y \Rightarrow y \neq x \) \\
\textit{⟨proof⟩}

\textbf{lemma neq-to-eq-False:} \( x \neq y \Rightarrow (x=y)\equiv \text{False} \) \\
\textit{⟨proof⟩}

\textbf{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.

**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`

**lemma** `delete-Some-all-distinct`:

```
delete x t = Some t' ⇒ all-distinct t ⇒ all-distinct t'
```

**lemma** `delete-None-set-of-cone`:

```
delete x t = None = (x ∉ set-of t)
```

**lemma** `delete-Some-x-set-of`:

```
delete x t = Some t' ⇒ x ∈ set-of t ∧ x ∉ set-of t'
```

**primrec** `subtract :: 'a tree ⇒ 'a tree ⇒ 'a tree option`

**where**

```
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`:

```
subtract t_1 t_2 = Some t ⇒ set-of t ⊆ set-of t_2
```

**lemma** `subtract-Some-set-of`:

```
subtract t_1 t_2 = Some t ⇒ set-of t_1 ⊆ set-of t_2
```
lemma subtract-Some-all-distinct-res:
subtract $t_1 \ t_2 = \text{Some } t \implies \text{all-distinct } t_2 \implies \text{all-distinct } t$

⟨proof⟩

lemma subtract-Some-dist-res:
subtract $t_1 \ t_2 = \text{Some } t \implies \text{set-of } t_1 \cap \text{set-of } t = \{\}$

⟨proof⟩

lemma subtract-Some-all-distinct:
subtract $t_1 \ t_2 = \text{Some } t \implies \text{all-distinct } t_2 \implies \text{all-distinct } t_1$

⟨proof⟩

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⟩

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⟩

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⟩

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'''

⟨proof⟩

lemma subtract-Tip: subtract Tip t = Some t

⟨proof⟩

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

⟨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.

definition K-statefun :: 'a ⇒ 'b ⇒ 'a where K-statefun c x ≡ c

lemma K-statefun-apply [simp]: K-statefun c x = c
(proof)

lemma K-statefun-comp [simp]: (K-statefun c o f) = K-statefun c
(proof)

lemma K-statefun-cong [cong]: K-statefun c x = K-statefun c x
(proof)

definition lookup :: ('v ⇒ 'a) ⇒ 'n ⇒ ('n ⇒ 'v) ⇒ 'a
where lookup destr n s = destr (s n)

definition update :: ('v ⇒ 'a1) ⇒ ('a2 ⇒ 'v) ⇒ 'n ⇒ ('a1 ⇒ 'a2) ⇒ ('n ⇒ 'v) ⇒ ('n ⇒ 'v)
where update destr constr n f s = s(n := constr (f (destr (s n))))

lemma lookup-update-same: (∀v. destr (constr v) = v) → lookup destr n (update destr constr n f s) = f (destr (s n))
(proof)

lemma lookup-update-id-same: lookup destr n (update destr' id n (K-statefun (lookup id n s'))) s) = lookup destr n s'
(proof)

lemma lookup-update-other: n ≠ m → lookup destr n (update destr' constr m f s) = lookup destr n s
(proof)
lemma id-id-cancel: id (id x) = x
⟨proof⟩

lemma destr-contstr-comp-id: (∀v. destr (constr v) = v) ⇒ destr ◦ constr = id
⟨proof⟩

lemma block-conj-cong: (P ∧ Q) = (P ∧ Q)
⟨proof⟩

lemma conj1-False: P ≡ False ⇒ (P ∧ Q) ≡ False
⟨proof⟩

lemma conj2-False: Q ≡ False ⇒ (P ∧ Q) ≡ False
⟨proof⟩

lemma conj-True: P ≡ True ⇒ Q ≡ True ⇒ (P ∧ Q) ≡ True
⟨proof⟩

lemma conj-cong: P ≡ P' ⇒ Q ≡ Q' ⇒ (P ∧ Q) ≡ (P' ∧ Q')
⟨proof⟩

lemma update-apply: (update destr constr n f s x) =
  (if x=n then constr (f (destr (s n))) else s x)
⟨proof⟩

lemma ex-id: ∃x. id x = y
⟨proof⟩

lemma swap-ex-eq: 
  ∃s. f s = x ≡ True ≡
  ∃s. x = f s ≡ True
⟨proof⟩

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

lemma update d c n (K-statespace (lookup d n s)) s = s
⟨proof⟩
end

4 Setup for State Space Locales

theory StateSpaceLocale imports StateFun
keywords statespace :: thy-decl
begin


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⟩

lemma project-inject-comp-id [statefun-simp]: project ∘ inject = id
  ⟨proof⟩

lemma project-inject-comp-cancel [statefun-simp]: f ∘ project ∘ inject = f
  ⟨proof⟩

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

⟨ML⟩

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

print-locale vars-namespace
print-locale vars-valuetypes
print-locale vars

This resembles a record definition, but introduces sophisticated locale in-
frastructure 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 \Rightarrow \text{nat} and inject-nat::\text{nat} \Rightarrow 'value
  assumes \( \forall n. \text{project-nat} (\text{inject-nat} n) = n \)  
  fixes project-bool::'value \Rightarrow \text{bool} and inject-bool::\text{bool} \Rightarrow 'value
  assumes \( \forall b. \text{project-bool} (\text{inject-bool} b) = b \)

The HOL predicate distinct describes distinctness of all names in the con-
text. Locale \( \text{vars}' \) 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 sim-
  plifications.

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 \texttt{project-nat (s n)}, and an update as $s \langle n := 2 \rangle$, which is translated to $s(n := \texttt{inject-nat 2})$. We can now establish the following lemma:

\textbf{lemma (in vars) foo:} $s < n := 2 > \cdot b = s \cdot b$ \langle proof \rangle

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 \texttt{vars} for later use and is automatically propagated to all its interpretations. Here is another example:

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

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 $x$ 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 distinct \texttt{[N, B, n, b, x]}, from this we can derive both distinct \texttt{[N, B]} and distinct \texttt{[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:} $s(N := 2) \cdot B = s \cdot B$ as well as $s(n := 2) \cdot b = s \cdot b$. 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 $n = N$:

\textbf{lemma (in varsX) foo:} $s(N := 2) \cdot x = s \cdot x$ \langle proof \rangle

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(a := i)·a = i
  ⟨proof⟩

lemma (in foo) foo2:
  shows (s{a := i})·a = i
  ⟨proof⟩

lemma (in foo) foo3:
  shows (s{a := i})·b = s·b
  ⟨proof⟩

lemma (in foo) foo4:
  shows (s{a := i,b := j,c := k,a := x}) = (s(b := j,c := k,a := x))
  ⟨proof⟩

statespace bar =
  b::bool
  c::string

lemma (in bar) bar1:
  shows (s{b := True})·c = s·c
  ⟨proof⟩

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(a := i)·B = s·B
  ⟨proof⟩
  thm foo1 ⟨proof⟩
  thm bar1 ⟨proof⟩

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

lemma (in dup)
  shows s< a := i >·x = s·x
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.

\begin{ML}
0.2s
\textbf{statespace 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
\end{ML}

2.4s

\textbf{statespace 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
statespace benchmark1000 = 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 A119::nat A120::nat A121::nat A122::nat A123::nat A124::nat A125::nat A126::nat A127::nat A128::nat A129::nat A130::nat A131::nat A132::nat A133::nat A134::nat A135::nat A136::nat A137::nat A138::nat A139::nat A140::nat A141::nat A142::nat A143::nat A144::nat A145::nat A146::nat A147::nat A148::nat A149::nat A150::nat A151::nat A152::nat A153::nat A154::nat A155::nat A156::nat A157::nat A158::nat A159::nat A160::nat A161::nat A162::nat A163::nat A164::nat A165::nat A166::nat A167::nat A168::nat A169::nat A170::nat A171::nat A172::nat A173::nat A174::nat A175::nat A176::nat A177::nat A178::nat A179::nat A180::nat A181::nat A182::nat A183::nat A184::nat A185::nat A186::nat A187::nat A188::nat A189::nat A190::nat A191::nat A192::nat A193::nat A194::nat A195::nat A196::nat A197::nat A198::nat A199::nat A200::nat A201::nat A202::nat A203::nat A204::nat A205::nat A206::nat A207::nat A208::nat A209::nat
lemma (in benchmark100) test: s<A1 := a>-A100 = s·A100 ⟨proof⟩
lemma (in benchmark500) test: s<A1 := a>-A100 = s·A100 ⟨proof⟩
lemma (in benchmark1000) test: s<A1 := a>-A100 = s·A100 ⟨proof⟩

end