State Spaces: The Locale Way

Norbert Schirmer

April 19, 2009

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*
⟨*proof*⟩

lemma *all-distinct-right*: *all-distinct* (*Node l x b r*) $\Longrightarrow$ *all-distinct r*
⟨*proof*⟩

lemma *distinct-left*: ⟦*all-distinct* (*Node l x False r*); *y* ∈ *set-of l* ⟧ $\Longrightarrow$ *x*≠*y*
⟨*proof*⟩

lemma *distinct-right*: ⟦*all-distinct* (*Node l x False r*); *y* ∈ *set-of r* ⟧ $\Longrightarrow$ *x*≠*y*
⟨*proof*⟩

lemma *distinct-left-right*: ⟦*all-distinct* (*Node l z b r*); *x* ∈ *set-of l*; *y* ∈ *set-of r*⟧
$\Longrightarrow$ *x*≠*y*
⟨*proof*⟩

lemma *in-set-root*: *x* ∈ *set-of* (*Node l x False r*)
⟨*proof*⟩

lemma *in-set-left*: *y* ∈ *set-of l* $\Longrightarrow$ *y* ∈ *set-of* (*Node l x False r*)
⟨*proof*⟩

lemma *in-set-right*: *y* ∈ *set-of r* $\Longrightarrow$ *y* ∈ *set-of* (*Node l x False r*)
⟨*proof*⟩

lemma *swap-neq*: *x* ≠ *y* $\Longrightarrow$ *y* ≠ *x*
⟨*proof*⟩

lemma *neq-to-eq-False*: *x*≠*y* $\Longrightarrow$ (*x*=*y*)≡*False*
⟨*proof*⟩

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*⟩

lemma *delete-Some-all-distinct*:
⋀*t'*. ⟦*delete x t* = *Some t'*; *all-distinct t*⟧ ⟹ *all-distinct t'*
⟨*proof*⟩

lemma *delete-None-set-of-conv*: *delete x t* = *None* = (*x* ∉ *set-of t*)
⟨*proof*⟩

lemma *delete-Some-x-set-of*:
⋀*t'*. *delete x t* = *Some t'* ⟹ *x* ∈ *set-of t* ∧ *x* ∉ *set-of t'*
⟨*proof*⟩

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*:
⋀*t*$_2$ *t*. *subtract t*$_1$ *t*$_2$ = *Some t* ⟹ *set-of t* ⊆ *set-of t*$_2$
⟨*proof*⟩

lemma *subtract-Some-set-of*:
⋀*t*$_2$ *t*. *subtract t*$_1$ *t*$_2$ = *Some t* ⟹ *set-of t*$_1$ ⊆ *set-of t*$_2$
⟨*proof*⟩

lemma *subtract-Some-all-distinct-res*:
$\bigwedge t_2\ t.\ [\![\mathit{subtract}\ t_1\ t_2 = \mathit{Some}\ t;\ \mathit{all\text{-}distinct}\ t_2]\!] \Longrightarrow \mathit{all\text{-}distinct}\ t$
⟨*proof*⟩

lemma *subtract-Some-dist-res*:
$\bigwedge t_2\ t.\ \mathit{subtract}\ t_1\ t_2 = \mathit{Some}\ t \Longrightarrow \mathit{set\text{-}of}\ t_1 \cap \mathit{set\text{-}of}\ t = \{\}$
⟨*proof*⟩

lemma *subtract-Some-all-distinct*:
$\bigwedge t_2\ t.\ [\![\mathit{subtract}\ t_1\ t_2 = \mathit{Some}\ t;\ \mathit{all\text{-}distinct}\ t_2]\!] \Longrightarrow \mathit{all\text{-}distinct}\ t_1$
⟨*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.

constdefs *K-statefun*:: $'a \Rightarrow 'b \Rightarrow 'a$ *K-statefun c x* $\equiv c$

lemma *K-statefun-apply* [*simp*]: *K-statefun c x* $= c$
⟨*proof*⟩

lemma *K-statefun-comp* [*simp*]: (*K-statefun c* $\circ f$) = *K-statefun c*
⟨*proof*⟩

lemma *K-statefun-cong* [*cong*]: *K-statefun c x* = *K-statefun c x*
⟨*proof*⟩

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*))
⟨*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*:

$n{\neq}m \Longrightarrow lookup\ destr\ n\ (update\ destr'\ constr\ m\ f\ s) = lookup\ destr\ n\ s$
⟨*proof*⟩

lemma *id-id-cancel*: $id\ (id\ x) = x$
⟨*proof*⟩

lemma *destr-contstr-comp-id*:
$(\bigwedge v.\ destr\ (constr\ v) = v) \Longrightarrow destr \circ constr = id$
⟨*proof*⟩

lemma *block-conj-cong*: $(P \wedge Q) = (P \wedge Q)$
⟨*proof*⟩

lemma *conj1-False*: $(P{\equiv}False) \Longrightarrow (P \wedge Q) \equiv False$
⟨*proof*⟩

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

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

lemma *conj-cong*: $[\![P{\equiv}P';\ Q{\equiv}Q']\!] \Longrightarrow (P \wedge Q) \equiv (P' \wedge Q')$
⟨*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*: $\exists x.\ id\ x = y$
⟨*proof*⟩

lemma *swap-ex-eq*:
$\exists s.\ f\ s = x \equiv True \Longrightarrow$
$\exists s.\ x = f\ s \equiv True$
⟨*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*
uses *state-space.ML state-fun.ML*
begin

⟨*ML*⟩

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*
 ⟨*proof*⟩

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

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

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*

$s\langle x{:=}y\rangle$ == *-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 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 $\bigwedge n$. *project-nat* (*inject-nat* n) = n

  fixes *project-bool*::'*value* ⇒ *bool* **and** *inject-bool*::*bool* ⇒ '*value*
  assumes $\bigwedge b$. *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 $s(n := \mathit{inject\text{-}nat}\ 2)$. We can now establish the following lemma:

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

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*: $s\langle N := 2\rangle{\cdot}x = s{\cdot}x$ ⟨*proof*⟩

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 $s\langle a := i\rangle \cdot a = i$
  ⟨*proof*⟩

lemma (**in** *foo*) *foo2*:
  shows $(s\langle a{:=}i\rangle) \cdot a = i$
  ⟨*proof*⟩

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

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

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

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

⟨*proof*⟩
thm *foo1*⟨*proof*⟩
thm *bar1*⟨*proof*⟩

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

lemma (**in** *dup*)
shows *s*<*a* := *i*>·*x* = *s*·*x*
⟨*proof*⟩

lemma (**in** *dup*)
shows *s*<*A* := *i*>·*a* = *s*·*a*
⟨*proof*⟩

lemma (**in** *dup*)
shows *s*<*A* := *i*>·*x* = *s*·*x*
⟨*proof*⟩

Hmm, I hoped this would work now...

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.

end