Programming TLS in Isabelle/HOL

Andreas Lochbihler Marc Züst

Institute of Information Security
ETH Zurich, Switzerland

Isabelle Workshop 2014

supported by SNF

Isabelle/HOL as a programming language

Isabelle/HOL as a programming language

```
datatype color = R | B
datatype ('a, 'b) rbt
  = Empty
  | Branch color "('a, 'b) rbt" 'a 'b "('a, 'b) rbt"
```

datatypes

Isabelle/HOL as a programming language

```
fun merge :: "('b ⇒ 'a) ⇒ 'b list ⇒ 'b list ⇒ 'b list" where
  "merge key (a#as) (b#bs) = (if key a > key b
    then b # merge key (a#as) bs
    else a # merge key as (b#bs))"
| "merge key [] bs = bs"
| "merge key as [] = as"
```

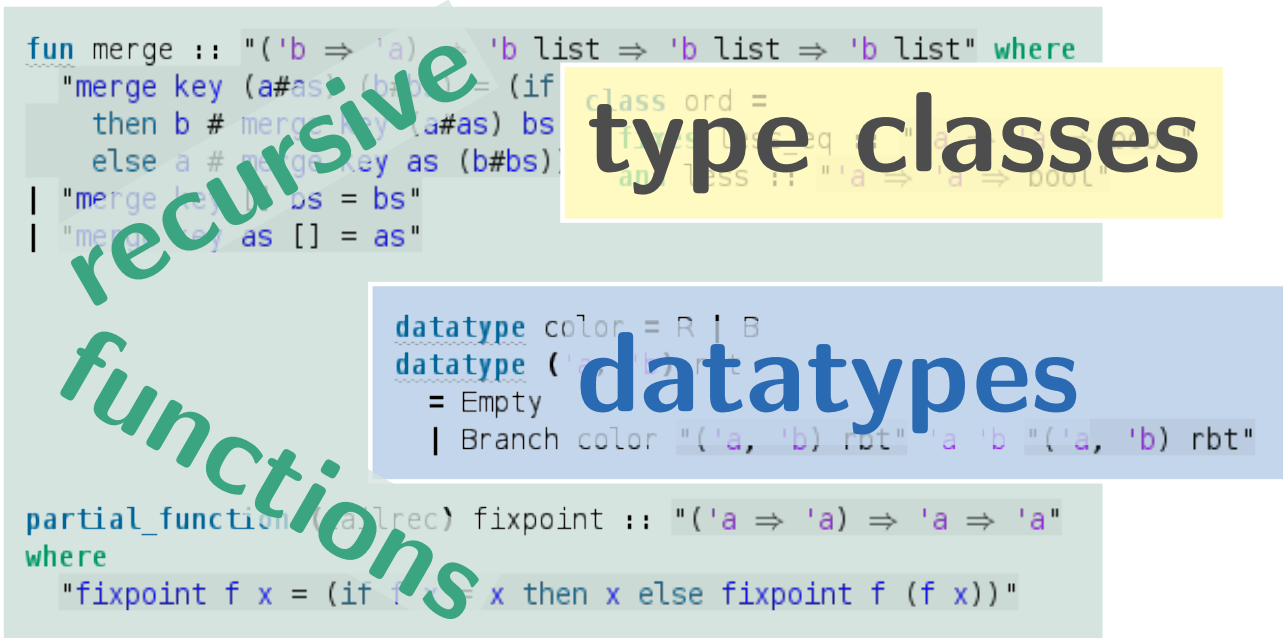
recursive functions

```
datatype color = R | B
datatype ('a, 'b) rbt
  = Empty
  | Branch color "('a, 'b) rbt" 'a 'b "('a, 'b) rbt"
```

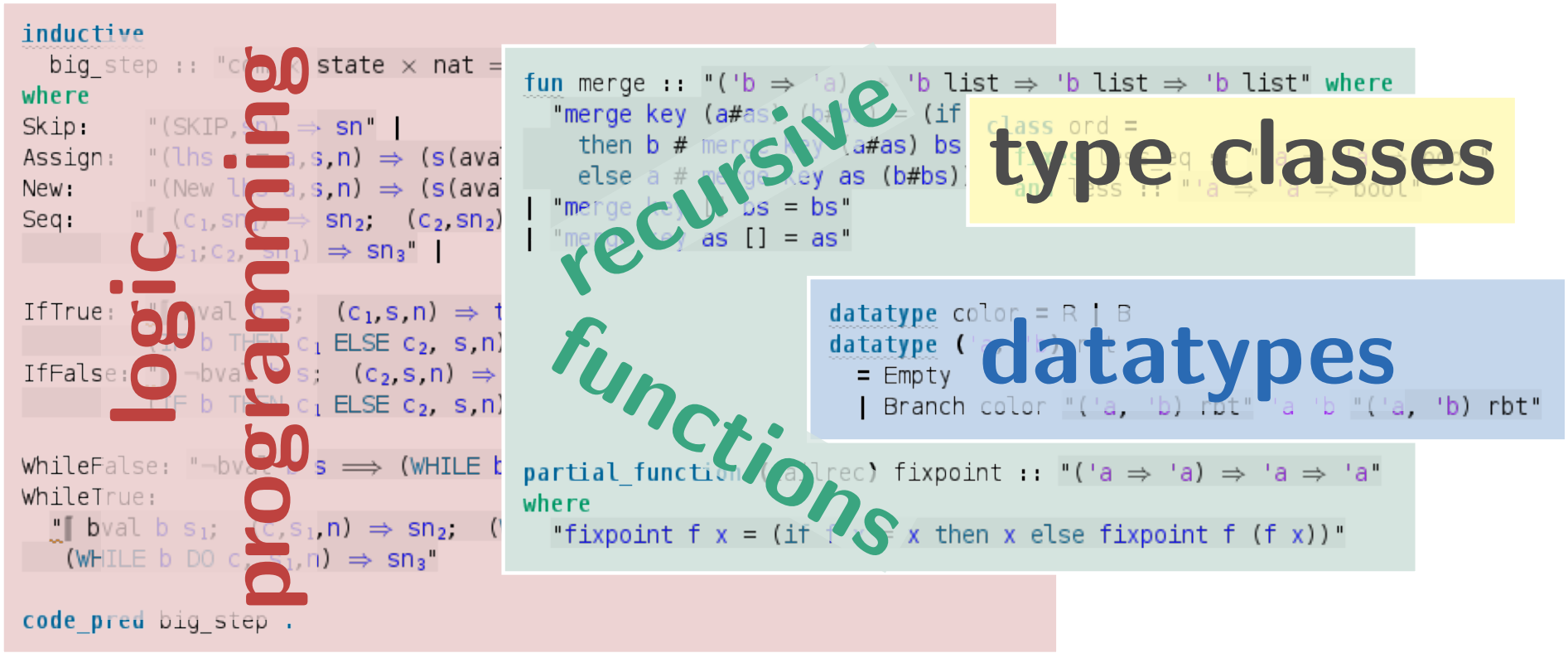
datatypes

```
partial_function (tailrec) fixpoint :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a"
where
  "fixpoint f x = (if f x = x then x else fixpoint f (f x))"
```

Isabelle/HOL as a programming language



Isabelle/HOL as a programming language



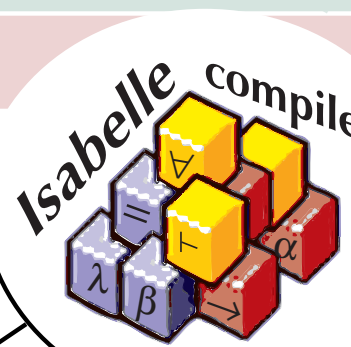
Isabelle/HOL as a programming language

logic programming

recursive functions

type classes

datatypes



Standard ML

Caml

Haskell

Scala

Isabelle/HOL as a programming language

```
inductive
  big_step :: "com × state ⇒ state ⇒ bool"
where
Skip:    "(SKIP,s) ⇒ s" |
Assign:  "(lhs := a,s) ⇒ (s(ava
New:     "(New lhs a,s) ⇒ (s(ava
Seq:     "⟦ (c₁,s₁) ⇒ s₂;  (c₂,s₂)
           (c₁;c₂, s₁) ⇒ s₃" |

IfTrue:  "⟦ bval b s;  (c₁,s,n) ⇒ t
           (IF b THEN c₁ ELSE c₂, s,n)
IfFalse: "⟦ ¬bval b s;  (c₂,s,n) ⇒
           (IF b THEN c₁ ELSE c₂, s,n)

WhileFalse: "¬bval b s ⟹ (WHILE b
WhileTrue:
  "⟦ bval b s₁;  (c,s₁,n) ⇒ s₂;
    (WHILE b DO c,s₂,n) ⇒ s₃"

code_pred big_step .
```

logic programming

```
fun merge :: "('b ⇒ 'a) ⇒ 'b list ⇒ 'b list ⇒ 'b list" where
  "merge key (a#as) (b#bs) = (if
    then b # merge key (a#as) bs
    else a # merge key as (b#bs))"
| "merge key bs = bs"
| "merge key as [] = as"
```

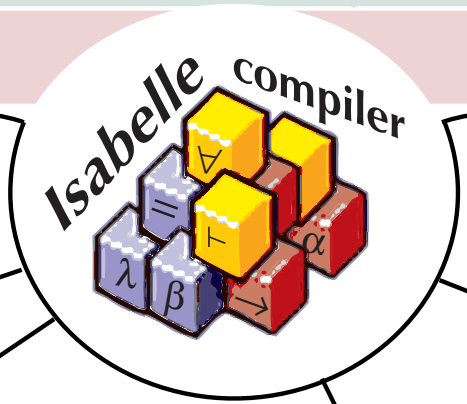
recursive functions

type classes

```
datatype color = R | B
datatype ('a, 'b) rbt
  = Empty
  | Branch color "('a, 'b) rbt" 'a 'b "('a, 'b) rbt"
```

datatypes

```
partial_function (tailrec) fixpoint :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a"
where
  "fixpoint f x = (if f x = x then x else fixpoint f (f x))"
```



Quickcheck
value

Standard ML

Caml

Haskell

Scala

Isabelle/HOL as a programming language

```
inductive
  big_step :: "com × state × nat ⇒ state × nat ⇒ bool"
where
Skip:     "(SKIP,s,n) ⇒ (s,n)" |
Assign:   "(lhs := a,s,n) ⇒ (s(ava
New:      "(New lhs a,s,n) ⇒ (s(ava
Seq:      "⟦ (c₁,s₁,n) ⇒ s₂;  (c₂,s₂
           (c₁;c₂,s₁,n) ⇒ s₃" |

IfTrue:   "⟦ bval b s;  (c₁,s,n) ⇒ t
           (IF b THEN c₁ ELSE c₂, s,n)
IfFalse:  "⟦ ¬bval b s;  (c₂,s,n) ⇒
           (IF b THEN c₁ ELSE c₂, s,n)

WhileFalse: "¬bval b s ⟹ (WHILE b
WhileTrue:
  "⟦ bval b s₁;  (c,s₁,n) ⇒ s₂;  (
   (WHILE b DO c,s₂,n) ⇒ s₃"

code_pred big_step .
```

logic programming

```
fun merge :: "('b ⇒ 'a) ⇒ 'b list ⇒ 'b list ⇒ 'b list" where
  "merge key (a#as) (b#bs) = (if
    then b # merge key (a#as) bs
    else a # merge key as (b#bs))
| "merge key bs = bs"
| "merge key as [] = as"
```

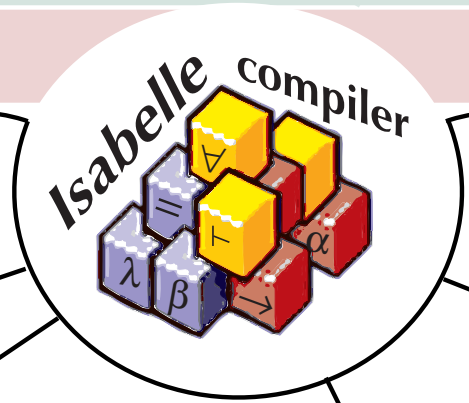
recursive functions

```
datatype color = R | B
datatype ('a, 'b) rbt
  = Empty
  | Branch color "('a, 'b) rbt" 'a 'b "('a, 'b) rbt"
```

datatypes

```
partial_function (tailrec) fixpoint :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a"
where
  "fixpoint f x = (if f x = x then x else fixpoint f (f x))"
```

type classes



Quickcheck
value

CAVA, CeTA
CoCon

Standard ML

Caml

Haskell

Scala

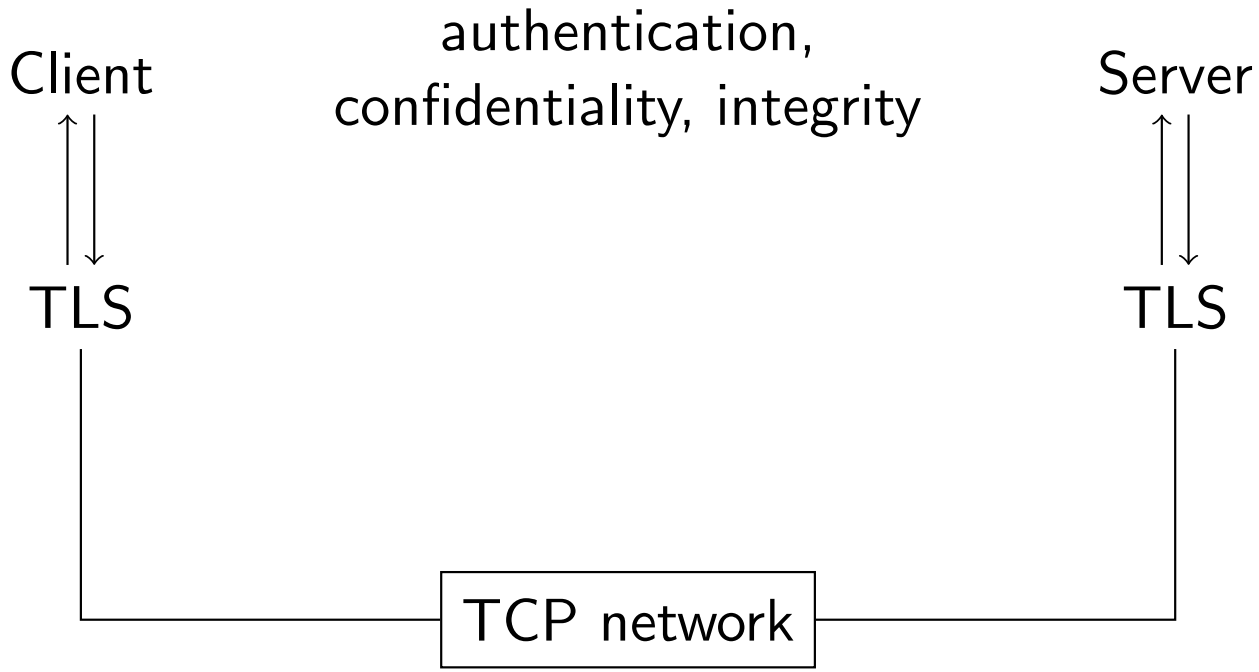
Case Study: Transport Layer Security (TLS)

What happens when we go beyond
self-contained batch-style functional programs?

Case Study: Transport Layer Security (TLS)

What happens when we go beyond
self-contained batch-style functional programs?

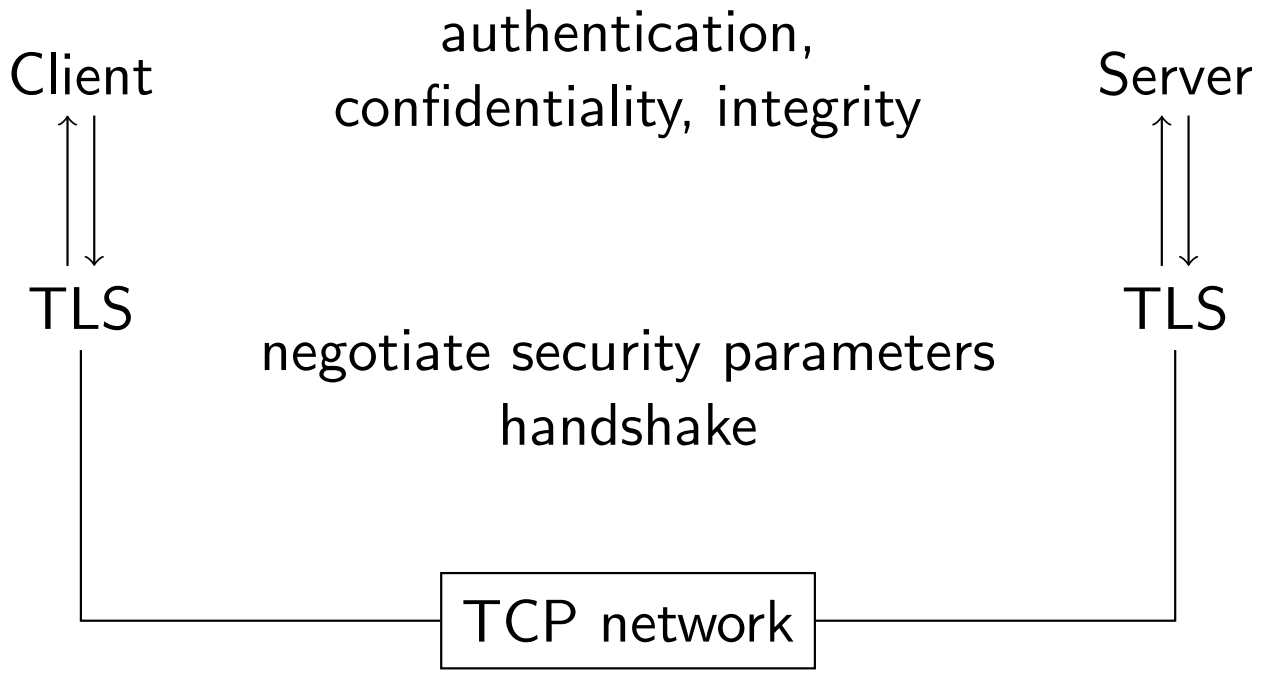
Program TLS in Isabelle/HOL



Case Study: Transport Layer Security (TLS)

What happens when we go beyond
self-contained batch-style functional programs?

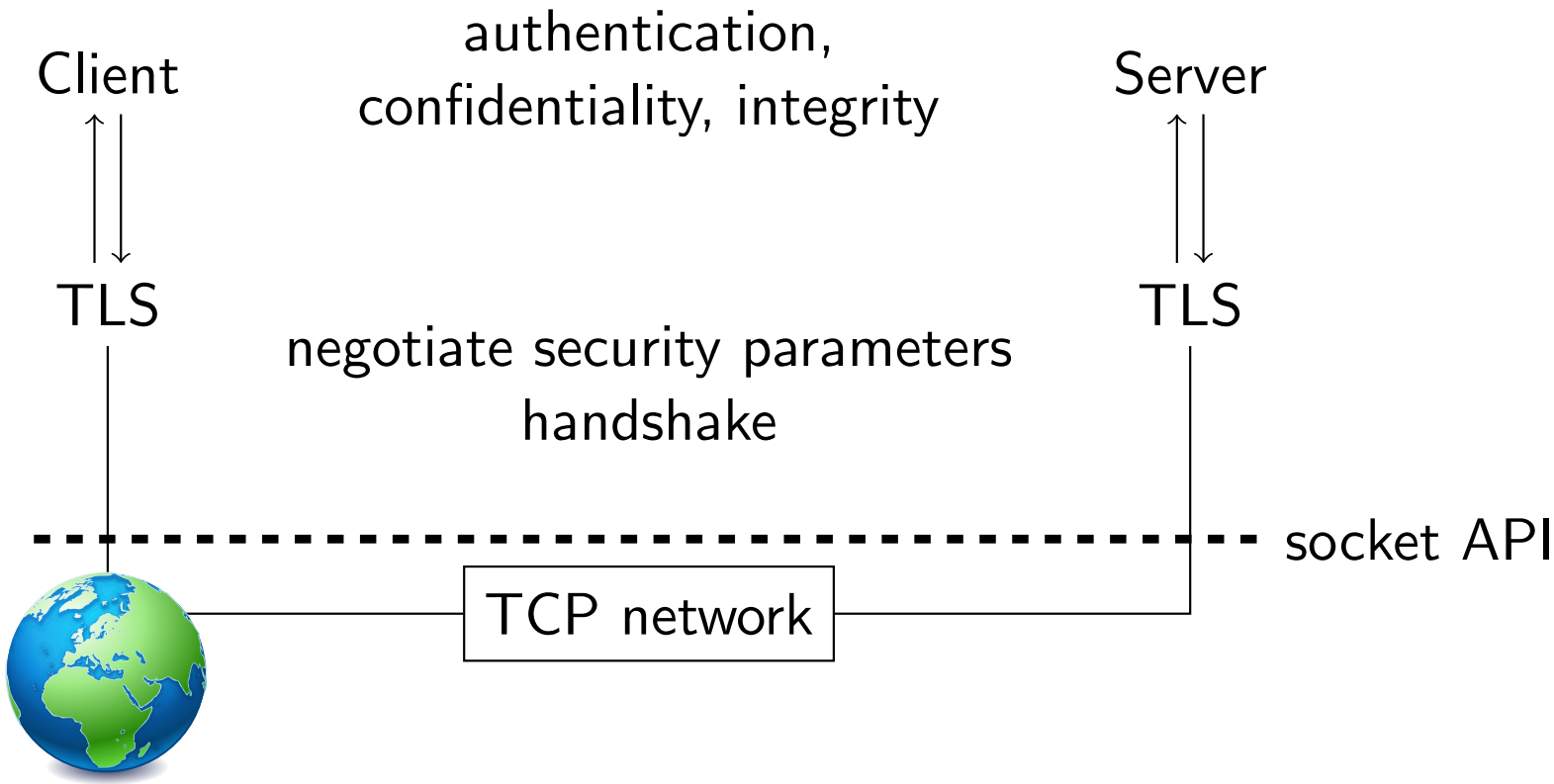
Program TLS in Isabelle/HOL



Case Study: Transport Layer Security (TLS)

What happens when we go beyond
self-contained batch-style functional programs?

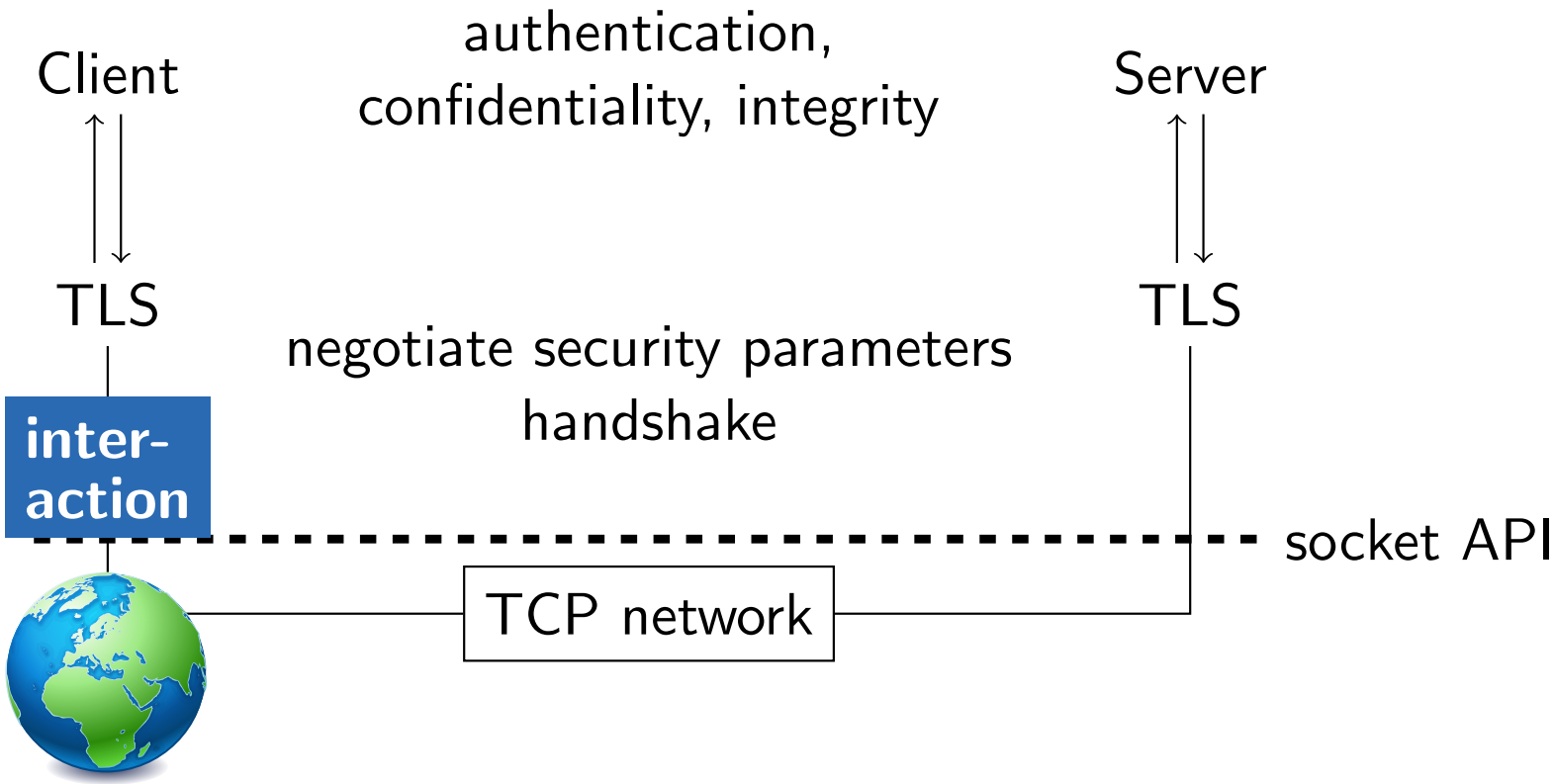
Program TLS in Isabelle/HOL



Case Study: Transport Layer Security (TLS)

What happens when we go beyond
self-contained batch-style functional programs?

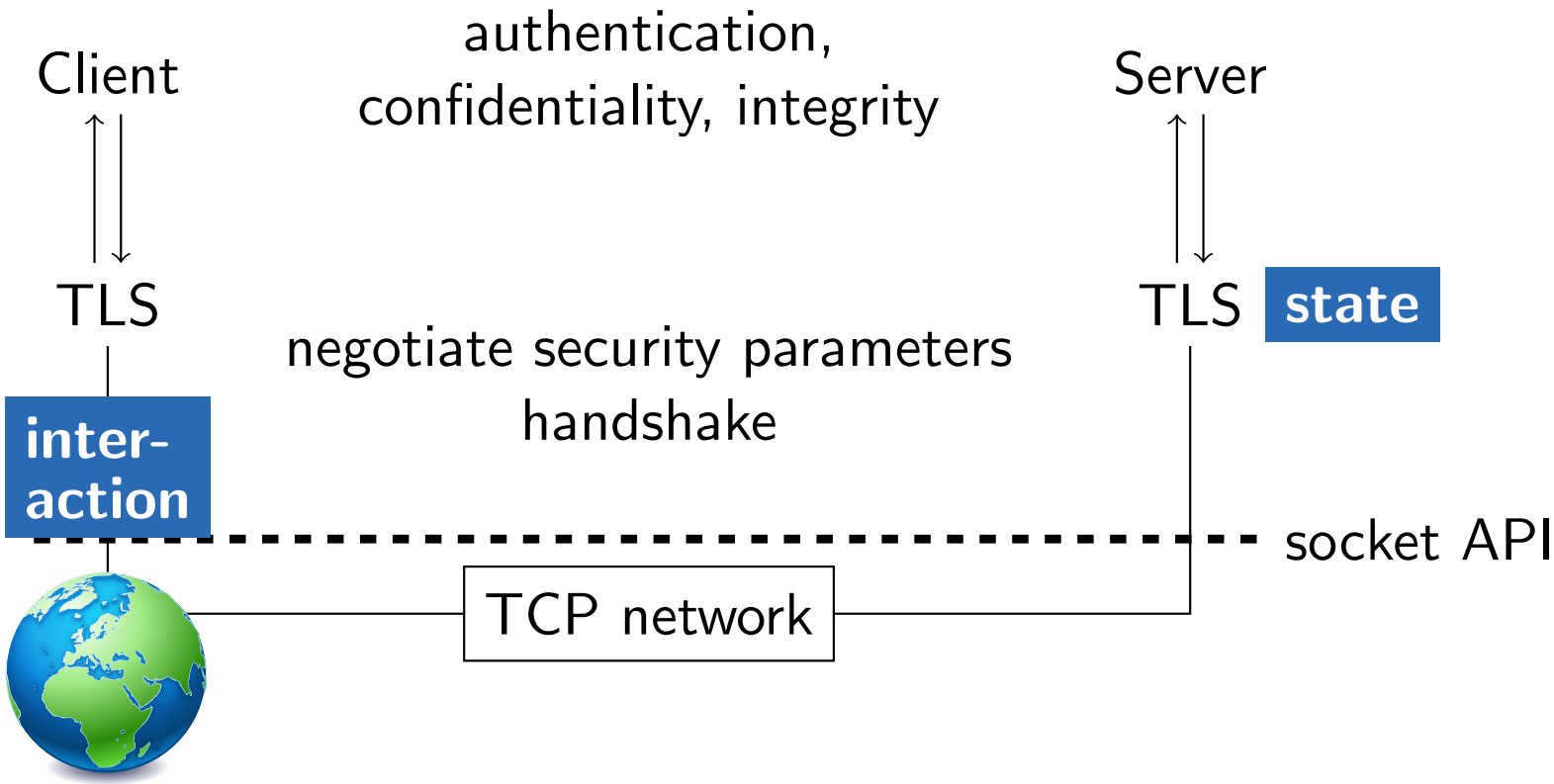
Program TLS in Isabelle/HOL



Case Study: Transport Layer Security (TLS)

What happens when we go beyond
self-contained batch-style functional programs?

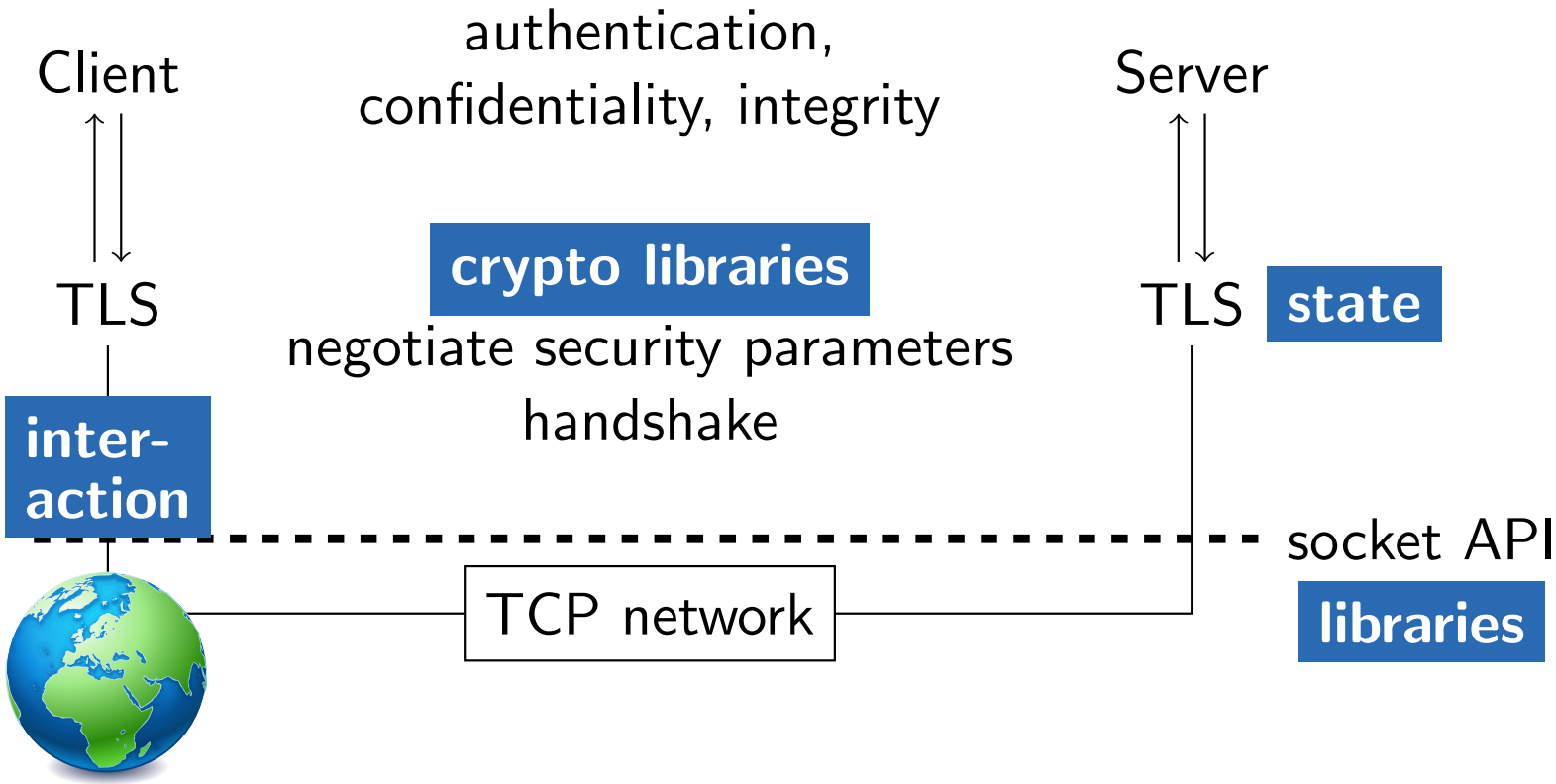
Program TLS in Isabelle/HOL



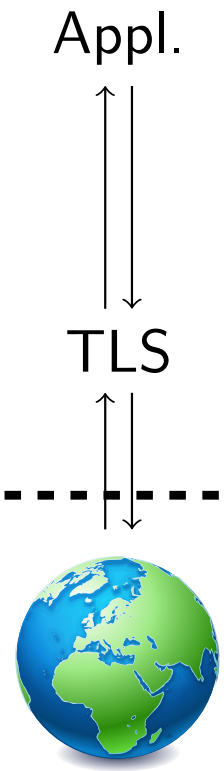
Case Study: Transport Layer Security (TLS)

What happens when we go beyond
self-contained batch-style functional programs?

Program TLS in Isabelle/HOL

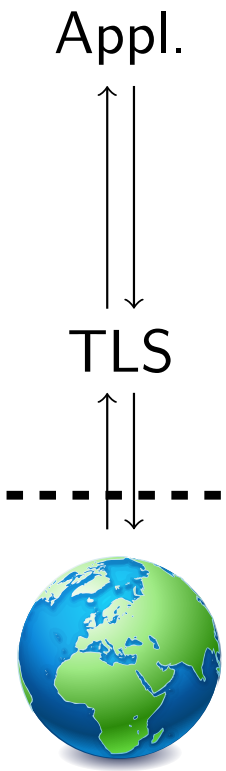


Interactive values



- cannot model socket API with ordinary HOL functions

Interactive values

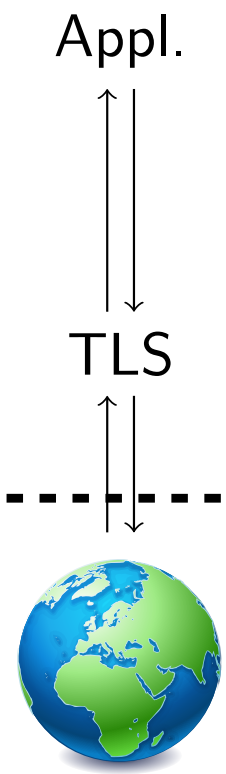


- cannot model socket API with ordinary HOL functions
- make interaction explicit in HOL

codatatype (α, o, ι) *resumption* =

Pure (*result*: α)
| *IO* (*output*: o) (*continuation*: $\iota \Rightarrow (\alpha, o, \iota)$ *resumption*)

Interactive values

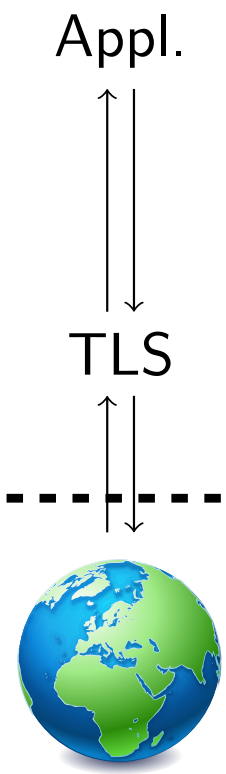


- cannot model socket API with ordinary HOL functions
- make interaction explicit in HOL

codatatype (α, o, ι) *resumption* $=$
Fail
| *Pure* (*result*: α)
| *IO* (*output*: o) (*continuation*: $\iota \Rightarrow (\alpha, o, \iota)$ *resumption*)

- add monad structure and setup for **partial-function**

Interactive values



- cannot model socket API with ordinary HOL functions
- make interaction explicit in HOL

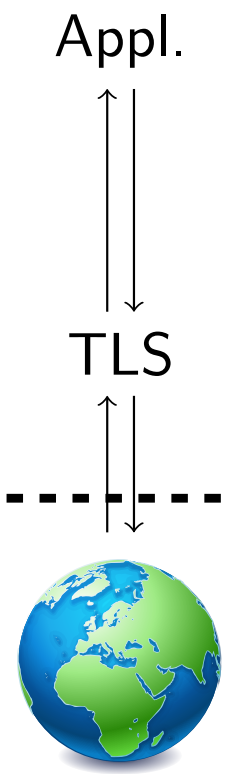
codatatype (α, o, ι) *resumption* =
  Fail
| *Pure* (*result*: α)
| *IO* (*output*: o) (*continuation*: $\iota \Rightarrow (\alpha, o, \iota)$ *resumption*)

- add monad structure and setup for **partial-function**

stdin = *IO StdIn* (λ*in*. *case in of Receive s* ⇒ *Pure s* | _ ⇒ *Fail*)
stdout s = *IO* (*StdOut s*) (λ*in*. *case in of Ack* ⇒ *Pure* () | _ ⇒ *Fail*)

definition *hello* where *hello* = *do* { *s* ← *stdin*; *stdout* (*"Hello, " @ s*) }

Interactive values



- cannot model socket API with ordinary HOL functions
- make interaction explicit in HOL

probabilistic interactive values

- cryptography requires a good source of randomness
- combine resumption with reader monad over coin flips
- ensures that randomness is used at most once

- add monad structure and setup for **partial-function**

$$stdin = IO\ StdIn\ (\lambda in.\ case\ in\ of\ Receive\ s \Rightarrow Pure\ s \mid \_ \Rightarrow Fail)$$
$$stdout\ s = IO\ (StdOut\ s)\ (\lambda in.\ case\ in\ of\ Ack \Rightarrow Pure\ () \mid \_ \Rightarrow Fail)$$

definition *hello* where $hello = do\ \{\ s \leftarrow stdin;\ stdout\ (''Hello,\ ''\ @\ s)\ \}$

Semantics and compilation of interactive values

Trace semantics

codatatype (α, ε) *trace* $= [\,]_{(\alpha\ \mathit{option})} \mid \mathit{TCons}\ \varepsilon\ ((\alpha, \varepsilon)\ \mathit{trace})$

$$\begin{aligned}
\mathit{traces}\ \mathit{Fail} &= \{\,[\,]_{\mathit{None}}\,\} \\
\mathit{traces}\ (\mathit{Pure}\ x) &= \{\,[\,]_{\mathit{Some}\ x}\,\} \\
\mathit{traces}\ (\mathit{IO}\ \mathit{out}\ c) &= \bigcup_{\mathit{in} \in \mathit{wf\text{-}responses}\ \mathit{out}} \mathit{TCons}\ (\mathit{out}, \mathit{in})\ `\ \mathit{traces}\ (c\ \mathit{in})
\end{aligned}$$

Semantics and compilation of interactive values

Trace semantics

codatatype (α, ε) *trace* $=$ $[]_{(\alpha\ option)}$ | *TCons* ε $((\alpha, \varepsilon)$ *trace*)

$$
\begin{aligned}
&\mathit{traces}\ \mathit{Fail} &&= \{\, []_{\mathit{None}} \,\} \\
&\mathit{traces}\ (\mathit{Pure}\ x) &&= \{\, []_{\mathit{Some}\ x} \,\} \\
&\mathit{traces}\ (\mathit{IO}\ \mathit{out}\ c) &&= \bigcup_{\mathit{in} \in \mathit{wf\text{-}responses}\ \mathit{out}} \mathit{TCons}\ (\mathit{out}, \mathit{in})\ `\ \mathit{traces}\ (c\ \mathit{in})
\end{aligned}
$$

Compilation via an interpreter into the IO monad

typedecl 🌍

typedef α *IO* $\cong$ 🌍 $\rightharpoonup \alpha \times$ 🌍

Semantics and compilation of interactive values

Trace semantics

codatatype (α, ε) *trace* $=$ $[]_{(\alpha\ option)}$ | *TCons* ε $((\alpha, \varepsilon)$ *trace*)

$$
\begin{aligned}
&\mathit{traces}\ \mathit{Fail} &&= \{\, []_{\mathit{None}} \,\} \\
&\mathit{traces}\ (\mathit{Pure}\ x) &&= \{\, []_{\mathit{Some}\ x} \,\} \\
&\mathit{traces}\ (\mathit{IO}\ \mathit{out}\ c) &&= \bigcup_{\mathit{in} \in \mathit{wf\text{-}responses}\ \mathit{out}} \mathit{TCons}\ (\mathit{out}, \mathit{in})\ `\ \mathit{traces}\ (c\ \mathit{in})
\end{aligned}
$$

Compilation via an interpreter into the IO monad

typedecl 🌍

typedef α *IO* $\cong$ 🌍 $\rightharpoonup \alpha \times$ 🌍

consts *stdin-impl* :: *string IO*

stdout-impl :: *string* $\Rightarrow$ *unit IO*

$$
\begin{aligned}
&\mathit{interp}\ \mathit{StdIn} &&= \mathit{stdin\text{-}impl} &&\gg\!= \mathit{return} \circ \mathit{Receive} \\
&\mathit{interp}\ (\mathit{StdOut}\ s) &&= \mathit{stdout\text{-}impl}\ s &&\gg \mathit{return}\ \mathit{Ack}
\end{aligned}
$$

Semantics and compilation of interactive values

Trace semantics

codatatype (α, ε) *trace* $= [\,]_{(\alpha\ option)} \mid TCons\ \varepsilon\ ((\alpha, \varepsilon)\ trace)$

$$
\begin{aligned}
traces\ Fail &= \{\,[\,]_{None}\,\} \\
traces\ (Pure\ x) &= \{\,[\,]_{Some\ x}\,\} \\
traces\ (IO\ out\ c) &= \bigcup_{in \in wf\text{-}responses\ out} TCons\ (out, in)\ `\ traces\ (c\ in)
\end{aligned}
$$

Compilation via an interpreter into the IO monad

typedecl 🌍
typedef $\alpha\ IO \cong$ 🌍 $\rightharpoonup \alpha \times$ 🌍

consts *stdin-impl* :: *string IO*
stdout-impl :: *string* $\Rightarrow$ *unit IO*

$$
\begin{aligned}
interp\ StdIn &= stdin\text{-}impl \gg= return \circ Receive \\
interp\ (StdOut\ s) &= stdout\text{-}impl\ s \gg return\ Ack
\end{aligned}
$$

$$
\begin{aligned}
interp\text{-}resumption\ Fail &= fail\text{-}with\text{-}exception \\
interp\text{-}resumption\ (Pure\ x) &= return\ x \\
interp\text{-}resumption\ (IO\ out\ c) &= interp\ out \gg= (interp\text{-}resumption \circ c)
\end{aligned}
$$

Semantics and compilation of interactive values

Trace semantics

codatatype (α, ε) *trace* $= [\,]_{(\alpha\ option)} \mid TCons\ \varepsilon\ ((\alpha, \varepsilon)\ trace)$

$$
\begin{aligned}
traces\ Fail &= \{\,[\,]_{None}\,\} \\
traces\ (Pure\ x) &= \{\,[\,]_{Some\ x}\,\} \\
traces\ (IO\ out\ c) &= \bigcup_{in \in wf\text{-}responses\ out} TCons\ (out, in)\ `\ traces\ (c\ in)
\end{aligned}
$$

Compilation via an interpreter into the IO monad

typedecl 🌍
typedef $\alpha\ IO \cong$ 🌍 $\rightharpoonup \alpha \times$ 🌍

consts *stdin-impl* :: *string IO*
stdout-impl :: *string* $\Rightarrow$ *unit IO*

Interpreter separates execution and proving

- ⊕ Application-specific models of the environment possible
- ⊕ Efficient: interactions are interpreted, computation is native
- ⊖ Generated code is hardly readable

Foreign function interface

Import and re-use library functions

- crypto algorithms
- I/O, socket API
- parser combinators

Foreign function interface

Import and re-use library functions

1. Declare unspecified types and constants
   typedecl *byte-string*
   consts *hash-sha1* :: *byte-string* ⇒ *byte-string*
2. Adapt code serialisation with **code-printing**

- crypto algorithms
- I/O, socket API
- parser combinators

Foreign function interface

Import and re-use library functions

1. Declare unspecified types and constants
   typedecl *byte-string*
   consts *hash-sha1* :: *byte-string* ⇒ *byte-string*
2. Adapt code serialisation with **code-printing**

- crypto algorithms
- I/O, socket API
- parser combinators

Types for data exchange

- *unit*, *bool*, *integer*, *char*, _ *option*, _ *list*, *String.literal*
- ⊕ machine arithmetic *uint8*, *uint16*, . . . — Native Word in the AFP
- ⊖ string buffers — convert to *string*

Foreign function interface

Import and re-use library functions

1. Declare unspecified types and constants
   typedecl *byte-string*
   consts *hash-sha1* :: *byte-string* ⇒ *byte-string*
2. Adapt code serialisation with **code-printing**

- crypto algorithms
- I/O, socket API
- parser combinators

Types for data exchange

- *unit*, *bool*, *integer*, *char*, _ *option*, _ *list*, *String.literal*
- ⊕ machine arithmetic *uint8*, *uint16*, . . . — Native Word in the AFP
- ⊖ string buffers — convert to *string*

Challenges

- need uniform interface to different target language APIs
- ⊖ evaluation, quickcheck no longer work
- ⊖ cannot prove anything meaningful

Foreign function interface

Import and re-use library functions

1. Declare unspecified types and constants
   typedecl *byte-string*
   consts *hash-sha1* :: *byte-string* ⇒ *byte-string*
2. Adapt code serialisation with **code-printing**

- crypto algorithms
- I/O, socket API
- ~~parser combinators~~

Types for data exchange

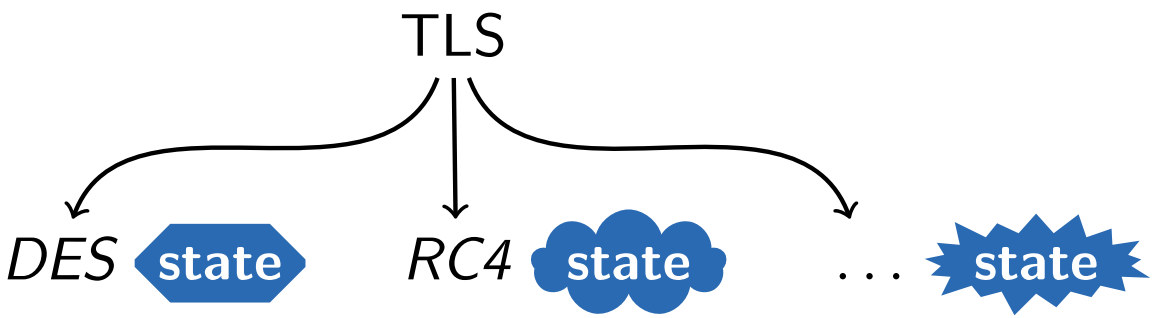
- *unit*, *bool*, *integer*, *char*, _ *option*, _ *list*, *String.literal*
- ⊕ machine arithmetic *uint8*, *uint16*, . . . — Native Word in the AFP
- ⊖ string buffers — convert to *string*

Challenges

- need uniform interface to different target language APIs
- ⊖ evaluation, quickcheck no longer work
- ⊖ cannot prove anything meaningful **(termination!)**

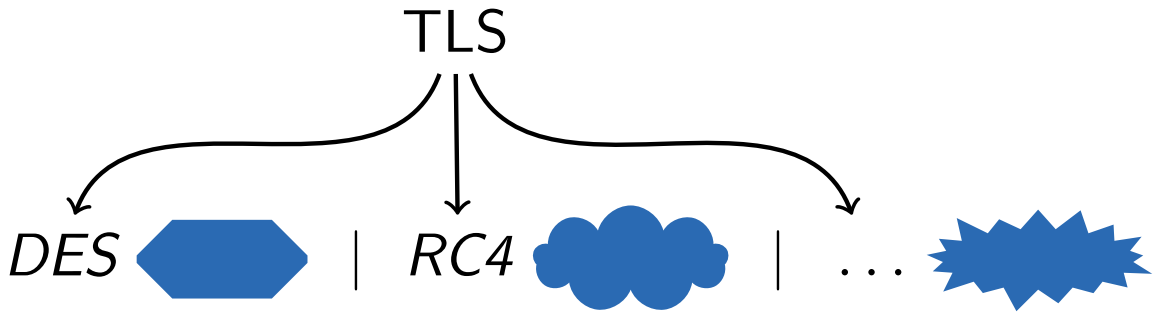
Existential state types

TLS state stores varying ciphersuites



Existential state types

TLS state stores varying ciphersuites

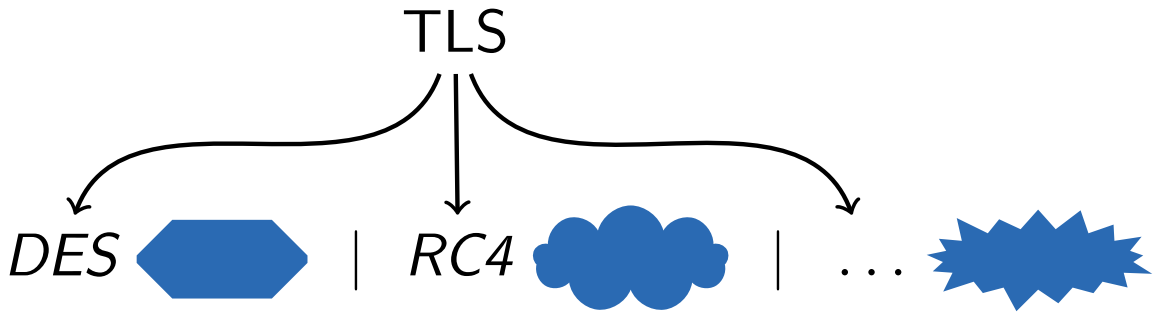


datatype *cipher* =

$$encrypt :: cipher \Rightarrow string \Rightarrow (cipher \times string)$$
$$encrypt\ (DES\ s)\ m = apfst\ DES\ (encrypt\text{-}des\ s\ m)$$
$$encrypt\ (RC4\ s)\ m = apfst\ RC4\ (encrypt\text{-}rc4\ s\ m)$$

Existential state types

TLS state stores varying ciphersuites



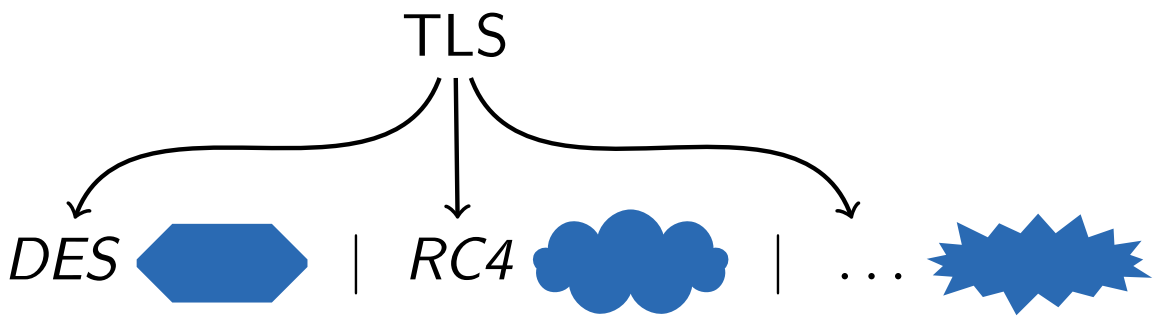
datatype *cipher* =

encrypt :: *cipher* ⇒ *string* ⇒ (*cipher* × *string*)
encrypt (*DES s*) *m* = *apfst DES* (*encrypt-des s m*)
encrypt (*RC4 s*) *m* = *apfst RC4* (*encrypt-rc4 s m*)

difficult to add new ciphersuites

Existential state types

TLS state stores varying ciphersuites



datatype *cipher* =

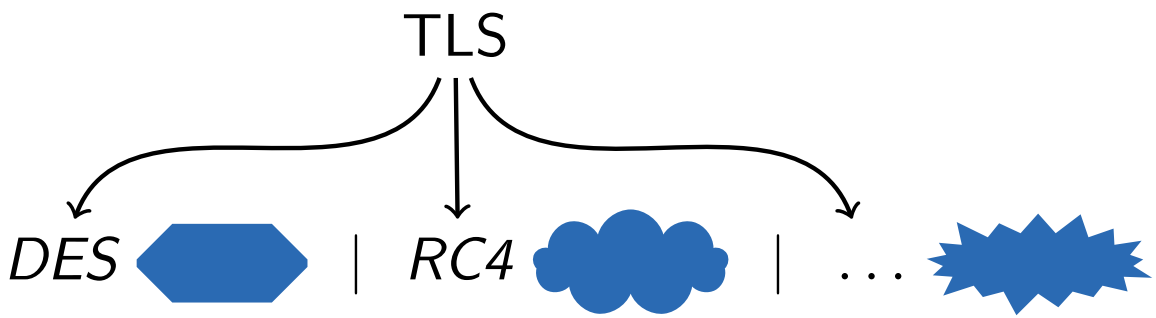
encrypt :: *cipher* ⇒ *string* ⇒ (*cipher* × *string*)
encrypt (*DES s*) *m* = *apfst DES* (*encrypt-des s m*)
encrypt (*RC4 s*) *m* = *apfst RC4* (*encrypt-rc4 s m*)

difficult to add new ciphersuites

datatype *cipher* = *Cipher* σ (σ ⇒ *string* ⇒ σ × *string*)
encrypt (*Cipher s enc*) *m* = (λ(*s*′, *m*′). (*Cipher s*′ *enc*, *m*′)) (*enc s m*)

Existential state types

TLS state stores varying ciphersuites



datatype *cipher* =

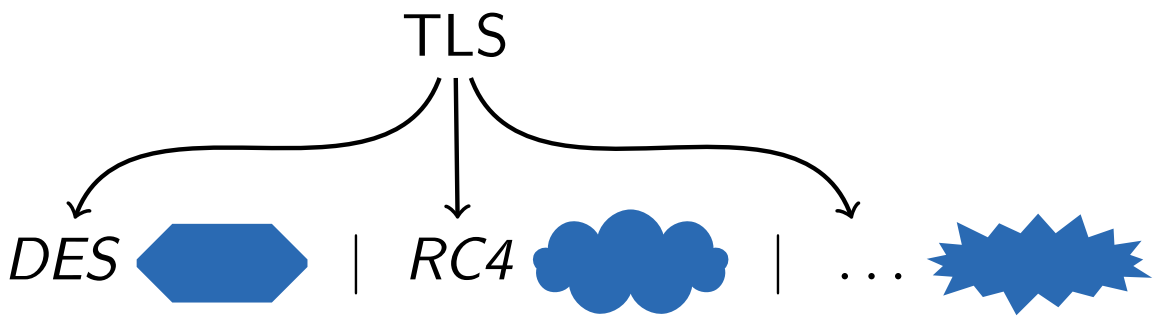
encrypt :: *cipher* ⇒ *string* ⇒ (*cipher* × *string*)
encrypt (*DES s*) *m* = *apfst DES* (*encrypt-des s m*)
encrypt (*RC4 s*) *m* = *apfst RC4* (*encrypt-rc4 s m*)

difficult to add new ciphersuites

datatype σ *cipher* = *Cipher* σ (σ ⇒ *string* ⇒ σ × *string*)
encrypt (*Cipher s enc*) *m* = (λ(*s*′, *m*′). (*Cipher s*′ *enc*, *m*′)) (*enc s m*)

Existential state types

TLS state stores varying ciphersuites



datatype *cipher* =

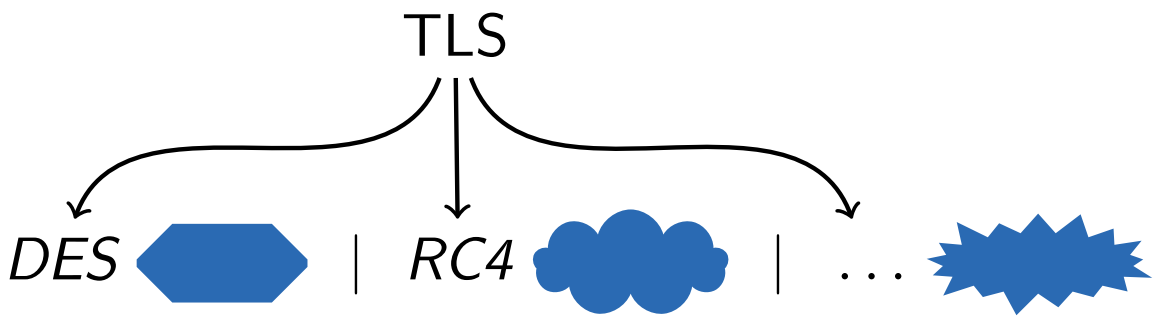
encrypt :: *cipher* ⇒ *string* ⇒ (*cipher* × *string*)
encrypt (*DES s*) *m* = *apfst DES* (*encrypt-des s m*)
encrypt (*RC4 s*) *m* = *apfst RC4* (*encrypt-rc4 s m*)

difficult to add new ciphersuites

datatype *cipher* = ∃σ. *Cipher* σ (σ ⇒ *string* ⇒ σ × *string*)
encrypt (*Cipher s enc*) *m* = (λ(*s*′, *m*′). (*Cipher s*′ *enc*, *m*′)) (*enc s m*)

Existential state types

TLS state stores varying ciphersuites



datatype *cipher* =

encrypt :: *cipher* ⇒ *string* ⇒ (*cipher* × *string*)
encrypt (*DES s*) *m* = *apfst DES* (*encrypt-des s m*)
encrypt (*RC4 s*) *m* = *apfst RC4* (*encrypt-rc4 s m*)

difficult to add new ciphersuites

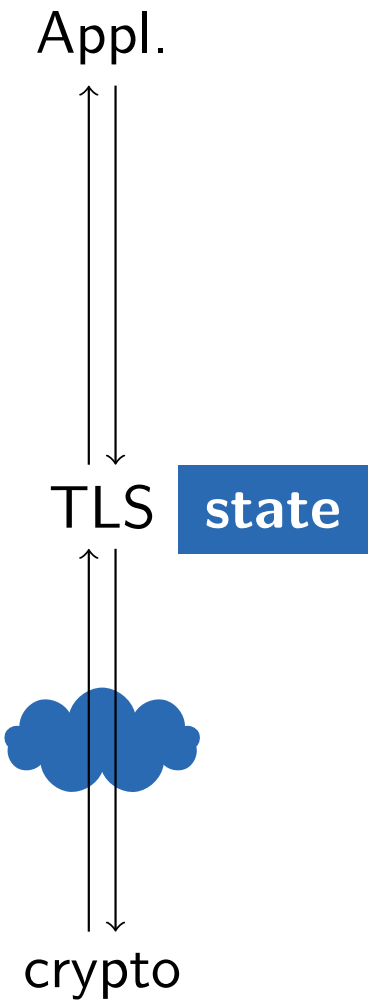
datatype *cipher* = ∃σ. *Cipher* σ (σ ⇒ *string* ⇒ σ × *string*)
encrypt (*Cipher s enc*) *m* = (λ(*s*′, *m*′). (*Cipher s*′ *enc*, *m*′)) (*enc s m*)

Avoid existential type by modelling the observations

codatatype *cipher* = *Cipher* (*encrypt*: *string* ⇒ *cipher* × *string*)

primcorec *mk-rc4* :: ⇒ *cipher* where
encrypt (*mk-rc4 s*) = *map-pair mk-rc4 id* ∘ *encrypt-rc4 s*

Dealing with state



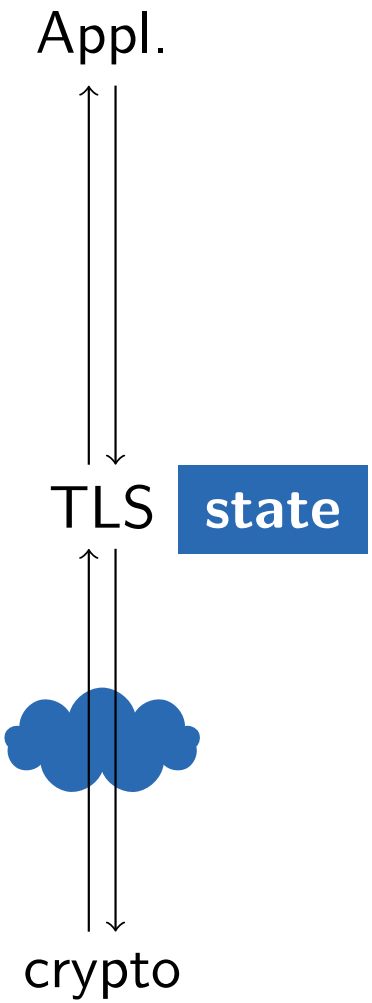
State passing in the monad

- new state type variable σ in $(\alpha, o, \iota, \sigma)$ *resumption*

  hard to combine functions
  that store different kinds of things

- universal type of storable data
  Imperative HOL: countable first-order values

Dealing with state



State passing in the monad

- ⊖ new state type variable σ in $(\alpha, o, \iota, \sigma)$ *resumption*

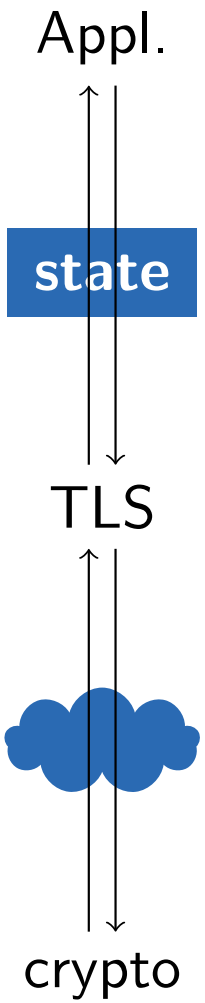
  hard to combine functions
  that store different kinds of things

- ⊖ universal type of storable data
  Imperative HOL: countable first-order values

Explicit state passing

- ⊕ full flexibility
- ⊖ clutter in the program

Dealing with state



State passing in the monad

- ⊖ new state type variable σ in $(\alpha, o, \iota, \sigma)$ *resumption*

  hard to combine functions
  that store different kinds of things

- ⊖ universal type of storable data
  Imperative HOL: countable first-order values

Explicit state passing

- ⊕ full flexibility
- ⊖ clutter in the program

Summary

It is feasible to program TLS in Isabelle/HOL

although some parts are still missing . . .

Summary

It is feasible to program TLS in Isabelle/HOL

although some parts are still missing . . .

Insights from the case study:

FFI imports **code-printing** offers basic functionality,
but types for data exchange are under-developed.

Unit tests instead of proofs,
but existing evaluation tools do not work with FFI imports.

Type system is very restrictive.

- monad transformers
- abstraction and information hiding

BNF The new (co)datatype package is great!