Deductive verification of distributed systems with PVS theorem prover—Part 1
CS141a: Distributed Systems Laboratory

sayan mitra
mitras@caltech.edu

CALIFORNIA INSTITUTE OF TECHNOLOGY

January 2007
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
verifying distributed systems

> proving that a distributed system behaves correctly is difficult
  > concurrency
  > failures
  > real-time constraints

> simulations and tests can be used to find bugs but cannot guarantee correctness

> model based distributed system development
  > describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
  - describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton
  - write the correctness requirements of the system as formulas in some logic, e.g., propositional logic, temporal logic
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
  - describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton
  - write the correctness requirements of the system as formulas in some logic, e.g., propositional logic, temporal logic
  - prove that the automaton satisfies the requirements
verifying distributed systems

- proving that a distributed system behaves correctly is difficult
  - concurrency
  - failures
  - real-time constraints
- simulations and tests can be used to find bugs but cannot guarantee correctness
- model based distributed system development
  - describe distributed system and its environment as a mathematical object, e.g., a state machine or an automaton
  - write the correctness requirements of the system as formulas in some logic, e.g., propositional logic, temporal logic
  - prove that the automaton satisfies the requirements
  - generate verified executable code from automaton through successive refinements
overview of tutorial

- quick introduction to PVS—a theorem prover for high-order logic
  - PVS specification language
  - prover commands
- specifying distributed algorithms in PVS
- proving properties of algorithms using PVS
propositional logic

\[ P := \text{true} \mid \text{false} \mid \neg P_1 \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2 \]
propositional logic

\[ P := \text{true} \mid \text{false} \mid \neg P_1 \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2 \]

sentences are built from \textit{finitely} many atomic propositions \{P_i\}
propositional logic

\[ P := \text{true} \mid \text{false} \mid \neg P_1 \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2 \]

sentences are built from \textit{finitely} many atomic propositions \( \{P_i\} \)

\textit{validity} and \textit{satisfiability} of any propositional sentence can be checked by constructing the truth table
propositional logic

\[ P := \text{true} \mid \text{false} \mid \neg P_1 \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2 \]

sentences are built from \textit{finitely} many atomic propositions \( \{P_i\} \)

\textit{validity} and \textit{satisfiability} of any propositional sentence can be checked by constructing the truth table

propositional logic is \textit{decidable}
propositional logic

\[ P \coloneqq true \mid false \mid \neg P_1 \mid P_1 \land P_2 \mid P_1 \lor P_2 \mid P_1 \implies P_2 \mid P_1 \iff P_2 \]

Sentences are built from finitely many atomic propositions \( \{P_i\} \).

Validity and satisfiability of any propositional sentence can be checked by constructing the truth table.

Propositional logic is decidable.

Many interesting problems can be expressed in propositional logic, e.g., circuit design, hardware verification.
first and higher order logic

▶ most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals

▶ higher order logic (HOL):
  ▶ more expressive ⇒ allows natural description of systems
  ▶ harder to decide ⇒ fully automatic verification not possible
first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. \( \forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x \)
  - functions: unary \( f(x) \), n-ary \( g(x_1, \ldots, x_n) \)
  - cannot quantify over functions and predicates
  - only certain fragments of FOL are decidable
    - E.g., monadic formulas: no function symbols, only unary predicates
- higher order logic (HOL):
  - more expressive \( \Rightarrow \) allows natural description of systems
  - harder to decide \( \Rightarrow \) fully automatic verification not possible
first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. \( \forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x \)
most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals

first order logic (FOL):
  - quantification over variables: e.g. $\forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals

first order logic (FOL):
  - quantification over variables: e.g. \( \forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x \)
  - functions: unary \( f(x) \), n-ary \( g(x_1, \ldots, x_n) \)
  - cannot quantify over functions and predicates

higher order logic (HOL):
  - more expressive \( \Rightarrow \) allows natural description of systems
  - harder to decide \( \Rightarrow \) fully automatic verification not possible
first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
first and higher order logic

- most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
- first order logic (FOL):
  - quantification over variables: e.g. $\forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x$
  - functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
  - cannot quantify over functions and predicates
- only certain fragments of FOL are decidable
  - E.g., monadic formulas: no function symbols, only unary predicates
most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals

first order logic (FOL):
- quantification over variables: e.g. $\forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x$
- functions: unary $f(x)$, n-ary $g(x_1, \ldots, x_n)$
- cannot quantify over functions and predicates

only certain fragments of FOL are decidable
- E.g., monadic formulas: no function symbols, only unary predicates

higher order logic (HOL):

more expressive $\Rightarrow$ allows natural description of systems
harder to decide $\Rightarrow$ fully automatic verification not possible
most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals

first order logic (FOL):

- quantification over variables: e.g. \( \forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x \)
- functions: unary \( f(x) \), n-ary \( g(x_1, \ldots, x_n) \)
- cannot quantify over functions and predicates

only certain fragments of FOL are decidable

- E.g., monadic formulas: no function symbols, only unary predicates

higher order logic (HOL):

- more expressive \( \Rightarrow \) allows natural description of systems
first and higher order logic

▶ most systems cannot be finitely axiomatized in propositional logic e.g., Archimedean property of reals
▶ first order logic (FOL):
  ▶ quantification over variables: e.g. \( \forall x \in \mathbb{R}, \exists n \in \mathbb{N}, n > x \)
  ▶ functions: unary \( f(x) \), n-ary \( g(x_1, \ldots, x_n) \)
  ▶ cannot quantify over functions and predicates
▶ only certain fragments of FOL are decidable
  ▶ E.g., monadic formulas: no function symbols, only unary predicates
▶ higher order logic (HOL):
  ▶ more expressive \( \Rightarrow \) allows natural description of systems
  ▶ harder to decide \( \Rightarrow \) fully automatic verification not possible
PVS

Prototype Verification System (Version 4.1)

http://pvs.csl.sri.com/
Prototype Verification System (Version 4.1)
http://pvs.csl.sri.com/

- a specification language, a theorem prover, and much more...
Prototype Verification System (Version 4.1)
http://pvs.csl.sri.com/
a specification language, a theorem prover, and much more ...
the PVS specification language is based on HOL; typed lambda calculus
Prototype Verification System (Version 4.1)
http://pvs.csl.sri.com/

- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures
Prototype Verification System (Version 4.1)
http://pvs.csl.sri.com/

- a specification language, a theorem prover, and much more ...
- the PVS specification language is based on HOL; typed lambda calculus
- the PVS prover is an interactive theorem prover with built-in semi-decision procedures
- relatively easy to plug in new proof strategies and decision procedures
Prototype Verification System (Version 4.1)
http://pvs.csl.sri.com/

a specification language, a theorem prover, and much more ...

the PVS specification language is based on HOL; typed lambda calculus

the PVS prover is an interactive theorem prover with built-in semi-decision procedures

relatively easy to plug in new proof strategies and decision procedures

written in LISP, version 4.1 is open source
Theorem proving and other areas of CS
theorem proving and other areas of CS

verification

theorem proving

functional logic

programming

PVS

proof methods

decision procedures

LISP
theorem proving and other areas of CS
theorem proving and other areas of CS
Theorem proving and other areas of CS

- Verification
- Logic
- Functional programming

PVS
theorem proving and other areas of CS

- verification
- proof methods
- decision procedures
- logic
- PVS
- functional programming
- LISP
example 1: a theory of stack of integers

Stack: theory begin

Stack: type = [# length: nat, seq: [below[length] -> nat] #]
example 1: a theory of stack of integers

\[\text{Stack: theory begin}\]

\[\text{Stack: type } = \left[\# \text{ length: nat, seq: [below[length] } \rightarrow \text{ nat } \right] \#\]\n
\[\text{NonEmptyStack?(c:Stack): bool } = c'\text{ length } \neq 0\]

\[\text{NonEmptyStack: type } = (\text{NonEmptyStack?})\]
example 1: a theory of stack of integers

Stack: theory begin

Stack: type = [# length: nat, seq: [below[length] -> nat ] #]

NonEmptyStack?(c:Stack): bool = c\'length l= 0

NonEmptyStack: type = (NonEmptyStack?)

length(c:Stack):nat = c\'length

top(c:NonEmptyStack):nat = q\'seq(length(c)-1)
example 1: a theory of stack of integers

Stack: theory begin

Stack: type = [# length: nat, seq: [below[length] -> nat] #]

NonEmptyStack?(c:Stack): bool = c′length /= 0

NonEmptyStack: type = (NonEmptyStack?)

length(c:Stack):nat = c′length

top(c:NonEmptyStack):nat = q′seq(length(c)-1)

push(c:stack, a:nat):NonEmptyStack = (# length := c′length + 1,
seq := seq(c) with [(c′length) := a] #)

pop(c:NonEmptyStack):[Stack,nat]

end Stack
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·

All functions are total.

Type/function definitions can be concrete, e.g., `top`, or uninterpreted, e.g., `pop`.

A predicate `B` on type `T` automatically defines a subtype `(B)` of `T`, e.g., `(NonEmptyStack)` is a subtype of `Stack`.

All assignments and definitions must be type-correct.

Typechecking is in general undecidable; PVS generates proof obligations or type correctness conditions (TCCs). E.g., application of `pop(c)` generates the TCC `NonEmptyStack?`.
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat, bool, real, ...`
- **type constructors**: `finite_sequences, records, sets, arrays, ...`
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·
- **type constructors**: `finite_sequences`, `records`, `sets`, `arrays`, · · ·
- all functions are **total**
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·
- **type constructors**: `finite_sequences`, `records`, `sets`, `arrays`, · · ·
- all functions are **total**
- type/function definitions can be **concrete**, e.g., `top`, or **uninterpreted**, e.g., `pop`

A predicate `B` on type `T` automatically defines a subtype `(B)` of `T`, e.g., `(NonEmptyStack)` is a subtype of `Stack`.

All assignments and definitions must be **type-correct**.

Typechecking is in general undecidable; PVS generates proof obligations or type correctness conditions (TCCs). E.g., application of `pop(c)` generates the TCC `NonEmptyStack?(c)`.
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat`, `bool`, `real`, · · ·
- **type constructors**: `finite.sequences`, `records`, `sets`, `arrays`, · · ·
- **all functions are total**
- **type/function definitions can be concrete**, e.g., `top`, or **uninterpreted**, e.g., `pop`
- **a predicate** $B$ on type $T$ **automatically defines a subtype** $(B)$ of $T$, e.g., `(NonEmptyStack?)` is a subtype of `Stack`
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
- **built in types**: `nat, bool, real, · · ·`
- **type constructors**: `finite_sequences, records, sets, arrays, · · ·`
- all functions are **total**
- type/function definitions can be **concrete**, e.g., `top`, or **uninterpreted**, e.g., `pop`
- a predicate `B` on type `T` automatically defines a **subtype** `(B)` of `T`, e.g., `(NonEmptyStack?)` is a subtype of `Stack`
- all assignments and definitions must be type-correct
basic concepts

- **theory**: a collection of type and function definitions, axioms, and theorems
- built in types: `nat`, `bool`, `real`, · · ·
- **type constructores**: `finite_sequences`, `records`, `sets`, `arrays`, · · ·
- all functions are **total**
- type/function definitions can be **concrete**, e.g., `top`, or **uninterpreted**, e.g., `pop`
- a predicate $B$ on type $T$ automatically defines a **subtype** ($B$) of $T$, e.g., `(NonEmptyStack?)` is a subtype of `Stack`
- all assignments and definitions must be type-correct
- typechecking is in general **undecidable**; PVS generates proof obligations or **type correctness conditions** (TCCs). E.g., application of `pop(c)` generates the TCC `NonEmptyStack?(c)`
some properties of stacks

**Stack**: theory begin

... 

c: var Stack 
a: var nat

*nonempty*: lemma forall (c,a): NonEmptyStack?(push(c,a))

*idem*: lemma forall (c, a): pop(push(c , a))‘1 = c

*pushpop*: lemma forall (c, a): pop(push(c,a))‘2 = a

end Stack
a polymorphic stack

\[
\text{Stack}[T:\text{type}]: \text{theory begin}
\]

\[
\text{Stack: type} = [\# \text{length: nat}, \text{seq: [below[\text{length}] -\to T]} \#]
\]

\[
\ldots
\]

\[
c: \text{var Stack}
\]

\[
a: \text{var } T
\]

\[
\text{nonempty: lemma forall } (c,a): \text{NonEmptyStack}(\text{push}(c,a))
\]

\[
\text{idem: lemma forall } (c, a): \text{pop}(\text{push}(c, a))'1 = c
\]

\[
\text{pushpop: lemma forall } (c, a): \text{pop}(\text{push}(c,a))'2 = a
\]

\[
\text{end Stack}
\]
inductive definitions and recursive functions

\[ \text{even}(n: \text{nat}): \text{inductive bool} = n = 0 \text{ or } n > 1 \text{ and } \text{even}(n-2) \]
inductive definitions and recursive functions

\[
even(n : \text{nat}) : \text{inductive bool} = n = 0 \text{ or } n > 1 \text{ and } \text{even}(n-2)
\]

\[
\text{fact}(n : \text{nat}) : \text{recursive nat} = \text{if } n = 0 \text{ then 1 else } n \ast \text{fact}(n-1) \text{ endif}
\]

\[
\text{measure lambda } (n : \text{nat}) : n
\]
inductive definitions and recursive functions

\[ \text{even}(n:\text{nat}) : \text{inductive bool} = n = 0 \text{ or } n > 1 \text{ and even}(n-2) \]

\[ \text{fact}(n:\text{nat}) : \text{recursive nat} = \text{if } n = 0 \text{ then } 1 \text{ else } n \ast \text{fact}(n-1) \text{ endif} \]

measure lambda \((n:\text{nat}):n\)

- inductive definitions cannot be used as rewrite rules
inductive definitions and recursive functions

\textit{even}(n: \text{nat}): \text{inductive bool} = n = 0 \text{ or } n > 1 \text{ and } \text{even}(n-2)

\textit{fact}(n: \text{nat}): \text{recursive nat} = \text{if } n = 0 \text{ then } 1 \text{ else } n \ast \text{fact}(n-1) \text{ endif}

\text{measure lambda } \lambda (n: \text{nat}): n

\begin{itemize}
\item inductive definitions cannot be used as rewrite rules
\item mutual recursion not allowed
\end{itemize}
inductive definitions and recursive functions

\textit{even}\,(n:nat): \textbf{inductive bool} = n = 0 \textbf{or} n > 1 \textbf{and} even\,(n-2)

\textit{fact}\,(n:nat): \textbf{recursive nat} = \textbf{if} \ n = 0 \ \textbf{then} \ 1 \ \textbf{else} \ n \ast \textit{fact}\,(n-1) \ \textbf{endif}

\textbf{measure lambda} \ (n:nat): n

- Inductive definitions cannot be used as rewrite rules
- Mutual recursion not allowed
- Domain of the \textbf{measure} function is the same domain as the recursive function being defined and its range must be a well-founded set with an order relation
polymorphic theory of automata

simplemachine[
states, actions: type,
enabled: [actions, states -> bool],
trans: [actions, states -> states],
start: [states -> bool]
]: theory
polymorphic theory of automata

simplemachine[
  states, actions: type,
  enabled: [actions, states -> bool],
  trans: [actions, states -> states],
  start: [states -> bool]
]: theory

reachable_hidden(s, n): recursive bool =
  if n = 0 then start(s)
  else (exists a, s1 : reachable_hidden(s1, n - 1) and enabled(a, s1) and s = trans(a, s1))
  endif
polymorphic theory of automata

simplemachine[
states, actions: type,
enabled: [actions, states -> bool],
trans: [actions, states -> states],
start: [states -> bool]
]: theory

reachable_hidden(s, n): recursive bool =
if n = 0 then start(s)
else (exists a, s1 : reachable_hidden(s1, n - 1) and enabled(a, s1) and s = trans(a, s1))
endif

measure (lambda s, n: n)
polymorphic theory of automata

simplemachine[
states, actions: type,
enabled: [actions,states -> bool ],
trans: [actions,states -> states],
start: [states -> bool ]
]: theory

reachable_hidden(s,n): recursive bool =
if n = 0 then start(s)
else (exists a, s1 : reachable_hidden(s1,n -1) and
enabled(a,s1) and s = trans(a,s1))
endif

measure (lambda s,n: n)

reachable(s): bool = exists n : reachable_hidden(s,n)
polymorphic theory of automata

\[\text{base}(\text{Inv}) : \text{bool} = \forall s : \text{start}(s) \implies \text{Inv}(s)\]

\[\text{inductstep}(\text{Inv}) : \text{bool} = \forall s, a : \text{reachable}(s) \land \text{Inv}(s) \land \text{enabled}(a, s) \implies \text{Inv}(\text{trans}(a, s))\]
polymorphic theory of automata

\[
\text{base}(\text{Inv}) : \text{bool} = \forall s: \text{start}(s) \implies \text{Inv}(s)
\]

\[
\text{inductstep}(\text{Inv}) : \text{bool} = \forall s, a: \text{reachable}(s) \text{ and } \text{Inv}(s) \text{ and } \text{enabled}(a, s) \implies \text{Inv}(\text{trans}(a, s))
\]

\[
\text{inductthm}(\text{Inv}): \text{bool} = \text{base}(\text{Inv}) \text{ and } \text{inductstep}(\text{Inv}) \implies (\forall s : \text{reachable}(s) \implies \text{Inv}(s))
\]
Example: specifying an automaton

an automaton is specified by the following components:

- **states**: type+

---

does this force transitions to be deterministic?

No! push internal nondeterministic choices to (external) choice over actions
example: specifying an automaton

an automaton is specified by the following components:

- \textit{states}: \texttt{type}\+
- \textit{actions}: \texttt{type}

\begin{itemize}
  \item \textit{enabled}: \([\text{states}, \text{actions}] \rightarrow \texttt{bool}\)
  \item \textit{trans}: \([\text{states}, \text{actions}] \rightarrow \text{states}\)
\end{itemize}

does this force transitions to be deterministic?

\textit{No!} push internal nondeterministic choices to (external) choice over actions
example: specifying an automaton

an automaton is specified by the following components:

- **states**: type+
- **actions**: type
- **enabled**: [states, actions -> bool]
example: specifying an automaton

an automaton is specified by the following components:

- **states**: \( \text{type}^+ \)
- **actions**: \( \text{type} \)
- **enabled**: \( \text{[states, actions} \rightarrow \text{bool]} \)
- **trans**: \( \text{[states, actions} \rightarrow \text{states]} \)

does this force transitions to be deterministic? No! Push internal nondeterministic choices to (external) choice over actions.
example: specifying an automaton

an automaton is specified by the following components:

- **states:** type+
- **actions:** type
- **enabled:** [states, actions -> bool]
- **trans:** [states, actions -> states]

does this force transitions to be deterministic?
an automaton is specified by the following components:

- `states: type^+`
- `actions: type`
- `enabled: [states, actions -> bool]`
- `trans: [states, actions -> states]`

does this force transitions to be deterministic?

no! push internal nondeterministic choices to (external) choice over actions
many more types of types

- enumerations
  - `color: type = [red, orange, green]`
many more types of types

- **enumerations**  
  \[ \text{color: type} = [\text{red, orange, green}] \]

- **tuple**  
  \[ \text{states: type} = [\text{nat, real, color}] \]
many more types of types

- **enumerations** `color: type = [red, orange, green]`
- **tuple** `states: type = [nat, real, color]`
- **record** `states2: type = [# counter:nat, timer:real, light:color #]`
many more types of types

- **enumerations**  
  \[ \text{color: type} = [\text{red, orange, green}] \]

- **tuple**  
  \[ \text{states: type} = [\text{nat, real, color}] \]

- **record**  
  \[ \text{states2: type} = [\# \text{counter: nat, timer: real, light: color}] \]

- **functions**
  \[ \text{Values: type} = [l \rightarrow \text{nat}] \]
  \[ \text{Values: type} = \text{function} [l \rightarrow \text{nat}] \]
  \[ \text{Values: type} = \text{array} [l \rightarrow \text{nat}] \]
many more types of types

- **enumerations**  
  `color: type = [red, orange, green]`

- **tuple**  
  `states: type = [nat, real, color]`

- **record**  
  `states2: type = [# counter:nat, timer:real, light:color #]`

- **functions**
  - `Values: type = [l -> nat ]`
  - `Values: type = function [l -> nat ]`
  - `Values: type = array [l -> nat ]`

- **dependent types**
  - `Queue: [# length: nat, seq:[{n:nat | n < length} -> t] #]`
many more types of types

- **enumerations**  color: type = [red, orange, green]
- **tuple**  states: type = [nat, real, color]
- **record**  states2: type = [# counter:nat, timer:real, light:color #]
- **functions**
  Values: type = [l -> nat]
  Values: type = function [l -> nat]
  Values: type = array [l -> nat]
- **dependent types**
  Queue: [# length: nat, seq:{n:nat | n < length} -> t ] #]

\[\text{ID: type} = \{1,2,3,4\}\]
\[\text{location: type} = [x:real, y:real]\]
\[\text{states: [# pos:[ID -> location], clock:[ID -> posreal], failed:[ID -> bool] #]}\]
abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.
an abstract datatype defines a collection of objects through constructors and recognizers.

actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions
abstract datatypes

an **abstract datatype** defines a collection of objects through **constructors** and **recognizers**.

```plaintext
actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions
```

- defines a new type called **actions**
abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions

▶ defines a new type called actions
▶ $a_f3$: actions = fail(3) is a constant of type action
abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions

- defines a new type called actions

- a_f3: actions = fail(3) is a constant of type action
  - fail?(a_f3) returns true
an abstract datatype defines a collection of objects through constructors and recognizers.

actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions

- defines a new type called actions
- a_f3: actions = fail(3) is a constant of type action
  - fail?(a_f3) returns true
  - time_elapse?(a_f3) returns false
abstract datatypes

An abstract datatype defines a collection of objects through constructors and recognizers.

**actions**: `datatype`
- `fail(i:ID):fail?`
- `time_elapse(t:posreal):time_elapse?`
- `send(i:ID,m:location):send?`
- `receive(i:ID,m:location):receive?`

**end actions**

- Defines a new type called `actions`
- **a_f3**: `actions = fail(3)` is a constant of type `action`
  - `fail?(a_f3)` returns true
  - `time_elapse?(a_f3)` returns false
  - `i(a_f3)` returns 3
abstract datatypes

an abstract datatype defines a collection of objects through constructors and recognizers.

(actions: datatype
fail(i:ID):fail?
time_elapse(t:posreal):time_elapse?
send(i:ID,m:location):send?
receive(i:ID,m:location):receive?
end actions

▶ defines a new type called actions
▶ a_f3: actions = fail(3) is a constant of type action
  ▶ fail?(a_f3) returns true
  ▶ time_elapse?(a_f3) returns false
  ▶ i(a_f3) returns 3
  ▶ what is i(time_elapse(10)) ?
defining enabling conditions and transitions

\[
\text{enabled}(a:\text{actions}, s:\text{states}):\text{bool} = \\
\text{cases } a \text{ of } \\
\text{fail}(i): \not\text{failed}(s)(i)
\]
defining enabling conditions and transitions

\[
\text{enabled}(a: \text{actions}, s: \text{states}): \text{bool} = \\
\text{cases } a \text{ of } \\
\text{fail}(i): \\
\text{not failed}(s)(i) \\
\text{send}(i,m): \\
\text{pos}(s)(i) = m \\
\ldots \\
\text{endcases}
\]
defining enabling conditions and transitions

```haskell
enabled(a: actions, s: states): bool =
  cases a of
  fail(i): not failed(s)(i)

  send(i, m):
  pos(s)(i) = m

  ...
  endcases

trans(a: actions, s: states): states =
  cases a of
  time_elapse(t):
  s with [clock := clock(s) + t]
```
defining enabling conditions and transitions

\[
\text{enabled}(a: \text{actions}, s: \text{states}): \text{bool} =
\begin{cases}
\text{fail}(i): & \text{not failed}(s)(i) \\
\text{send}(i,m): & \text{pos}(s)(i) = m \\
& \ldots
\end{cases}
\]

\[
\text{trans}(a: \text{actions}, s: \text{states}): \text{states} =
\begin{cases}
\text{time_elapse}(t): & s \text{ with } [\text{clock} := \text{clock}(s) + t] \\
\text{fail}(i): & s \text{ with } [\text{failed} := \text{failed}(s) \text{ with } [(i) := \text{true}]] \\
& \ldots
\end{cases}
\]
references

   Read chapter 2 for basic instructions about the user interface