Provably Correct Software with Dependent Types

Andrea Laretto

Università di Pisa
Dipartimento di Informatica

11 febbraio 2022

Outline

- The main topics presented in this seminar:
  - **Introduction to dependent type theory**
    - Simple types
    - Curry-Howard isomoprhism
    - Dependent type systems and logics
  - **Verification and proofs using Agda**
    - Dependent datatypes and pattern matching
    - Inductive relations
    - Types-as-specifications
    - Theorem proving
    - Examples of verification
  - **Characteristics of type-based verification**
    - Overview and discussion
    - Internal and external verification
    - Automation and solvers
  - **Conclusion and related work**

A Quick Introduction to Type Theory

- Consider a statically-typed functional programming language (e.g. Haskell, OCaml, Rust, TypeScript, Scala, etc.)
- *Type systems:* terms belong to types, just like sets contain elements
- `3 : Int`, `"hello" : String`, `Int : Set`, `String : Set`, etc.
- Type systems are classified by the kind of types we can construct:

$$\begin{array}{lllll}
\textbf{Types:} & A, B & ::= & \texttt{Int} \mid \texttt{Bool} \mid \dots & \text{(Base types)} \\
 & & \mid & A \times B & \text{(Product type)} \\
 & & \mid & A + B & \text{(Sum type)} \\
 & & \mid & A \rightarrow B & \text{(Function type)} \\
 & & \mid & \top & \text{(Unit type)} \\
 & & \mid & \bot & \text{(Empty type)}
\end{array}$$

- Type system presented here: *simple types*

Curry-Howard Correspondence

- *What is the use of type systems for verification?*
- Core idea: types are *propositions*, terms are their *proofs*:

Type theory		**Logic**	
Types		Propositions	
Terms		Proofs	
Product type	$A \times B$	Conjunction	$A \wedge B$
Sum type	$A + B$	Disjunction	$A \vee B$
Function type	$A \rightarrow B$	Implication	$A \Rightarrow B$
Unit type	$\top$	True	T
Empty type	$\bot$	False	F

- *Constructing* a well-typed term with type T is equivalent to *proving* T
- Simple types correspond to **(intuitionistic) propositional logic**
- *How do we construct terms (i.e. proofs) of each type?*

Type Systems

$$
\begin{array}{llll}
\textbf{Terms:}\quad s, t & ::= & \mathrm{zero} \mid \mathrm{succ}(t) & \text{(Integers)} \\
 & \mid & \mathrm{true} \mid \mathrm{false} & \text{(Booleans)} \\
 & \mid & x & \text{(Variables)} \\
 & \mid & \lambda x.t & \text{(Lambda abstraction)} \\
 & \mid & s\ t & \text{(Function application)} \\
 & \mid & \langle s, t \rangle & \text{(Product pairing)} \\
 & \mid & \mathrm{proj}_\ell(t) \mid \mathrm{proj}_r(t) & \text{(Product projections)} \\
 & \mid & \mathrm{inj}_\ell(t) \mid \mathrm{inj}_r(t) & \text{(Sum injections)} \\
 & \mid & () & \text{(Unit)}
\end{array}
$$

- Inductively define a relation of well-typed terms, $\Gamma \vdash t : A$
- *How do we keep track of variables in our functional terms?*
- A *context* is a list of the variables in scope along with their type:

$$
\begin{array}{llll}
\textbf{Context:}\quad \Gamma & ::= & \emptyset & \text{(Empty context)} \\
 & \mid & \Gamma, (x : A) & \text{(Context extension)}
\end{array}
$$

Simple Types - Typing Rules

$$\frac{}{\Gamma \vdash \mathrm{zero} : \mathtt{Int}}\ (\text{Zero}) \qquad \frac{\Gamma \vdash a : \mathtt{Int}}{\Gamma \vdash \mathrm{succ}(a) : \mathtt{Int}}\ (\text{Succ})$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \langle s, t \rangle : A \times B}\ (\text{Pair})$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathrm{proj}_\ell(t) : A}\ (\text{Proj}_\ell) \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathrm{proj}_r(t) : B}\ (\text{Proj}_r)$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathrm{inj}_\ell(t) : A + B}\ (\text{Sum}_\ell) \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathrm{inj}_r(t) : A + B}\ (\text{Sum}_r)$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma \vdash t : A}{\Gamma, x : B \vdash t : A}\ (\text{Weaken})$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B}\ (\text{Fun}) \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\,t : B}\ (\text{App})$$

$$\frac{}{\Gamma \vdash () : \top}\ (\text{Unit})$$

Dependent Function and Product Types

- *What about universal and existential quantification?*
- Two new constructors, and we allow *variables* to also appear *inside types*
- **Dependent function** ($\forall$)

$(x : A) \to B$ — *If you provide any term x of type A,*
I provide you a proof that $B(x)$ holds for x

- **Dependent product** ($\exists$)

$(x : A) \times B$ — *A concrete pair with a term x of type A (on the left),*
and a proof that $B(x)$ holds for that specific x

- *Intuitively, how does quantification work?*

Dependent Types in Practice

- A form of *universal quantification* can already be found *in the wild!*
- We obtain different logics depending on *what we can quantify on*
- Quantify on Set ⇒ parametric polymorphic types **(second-order prop. logic)**

```
sort :: (A : Set) → List A → List A
sort A xs = ...

sortBools :: List Bool → List Bool
sortBools xs = sort Bool xs
```

- Quantify on *any type* ⇒ dependent types **(higher-order predicate logic)**

```
sort :: (A : Set) → (n : Int) → Vector n A → Vector n A
sort A n xs = ...

sortFiveBools :: Vector 5 Bool → Vector 5 Bool
sortFiveBools xs = sort Bool 5 xs
```

- Why *dependent types*? Types also depend on *terms*, not just types

Dependent Types

- With quantification, *variables* and *terms* can appear inside a type!
- We *unify terms and types* in a single definition:

Terms + Types:

$$
\begin{array}{lcll}
s, t, A, B & ::= & x & \text{(Variables)} \\
 & \mid & \lambda x.t & \text{(Lambda abstraction)} \\
 & \mid & s\ t & \text{(Function application)} \\
 & \mid & \langle s, t \rangle & \text{(Product pairing)} \\
 & \mid & \mathrm{proj}_{\ell}(t) \mid \mathrm{proj}_{r}(t) & \text{(Product projections)} \\
 & \mid & \dots & \\
 & \mid & (x : A) \to B & \text{(Dependent function type)} \\
 & \mid & (x : A) \times B & \text{(Dependent product type)} \\
 & \mid & \mathrm{Set} & \text{(Universe)}
\end{array}
$$

Dependent Types - Typing Rules

$$\frac{}{\Gamma \vdash \mathsf{Int} : \mathrm{Set}}\ (\mathsf{IntType}) \qquad \frac{}{\Gamma \vdash \mathrm{Set} : \mathrm{Set}}\ (\mathrm{SetInSet}^*)$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \qquad \Gamma, x : A \vdash B : \mathrm{Set}}{(x : A) \to B : \mathrm{Set}}\ (\mathsf{FunctionType})$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \qquad \Gamma, x : A \vdash B : \mathrm{Set}}{(x : A) \times B : \mathrm{Set}}\ (\mathsf{ProductType})$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B} \overset{(\mathsf{App})}{\Longrightarrow} \frac{\Gamma \vdash f : (x : A) \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x := a]}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B} \overset{(\mathsf{Fun})}{\Longrightarrow} \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : (x : A) \to B}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \langle a, b \rangle : A \times B} \overset{(\mathsf{Pair})}{\Longrightarrow} \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B[x := a]}{\Gamma \vdash \langle a, b \rangle : (x : A) \times B}$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathrm{proj}_r(p) : B} \overset{(\mathsf{Proj}_r)}{\Longrightarrow} \frac{\Gamma \vdash p : (x : A) \times B}{\Gamma \vdash \mathrm{proj}_r(p) : B[x := \mathrm{proj}_\ell(p)]}$$

Why Intuitionistic/Constructive Logic?

- *Constructionism*: proofs are concrete objects/programs
- All the logics mentioned so far: only their *intuitionistic* fragments
- Why is the logic we defined called *intuitionistic/constructive*?
- Because the *law of excluded middle* $P \vee \neg P$ *does not hold*:

```
lem : (P : Set) → P + (¬ P)
lem = ?
```

- Proving LEM with a constructive proof amounts to saying:

> *There is a program* `lem` *that, for any proposition P,*
> *can provide a proof of P or a proof its negation ¬P*

- $\implies$ A constructive proof for LEM would decide the halting problem!
- $\neg\neg A \not\to A, \qquad A \to B \not\equiv \neg A \vee B, \qquad \forall A \not\equiv \neg \exists \neg A, \qquad \ldots$

Extending the Language

- *So far*: the *basic blocks* of our constructive logic
- *Can we define additional types/propositions on top of this logic?*

Agda

- A brief overview of the *dependently typed functional programming language and proof assistant Agda* (Norell 2007, Chalmers University)
- Construct and verify properties of programs using *propositions-as-types*

Inductive Datatypes using Agda

- Defining new types and data structures using Agda: *inductive datatypes*
- Basic inductive type:

```
data Nat : Set where
  zero : Nat
  succ : Nat → Nat
```

- Polymorphic type (quantification on types, **Set**)

```
data List : Set → Set where
  nil  : (A : Set) → List A
  cons : (A : Set) → A → List A → List A
```

- Dependent type (quantification on terms, Nat)

```
data Vect : Nat → Set → Set where
  nilV  : (A : Set) → Vect zero A
  consV : (n : Nat) → (A : Set)
        → A → Vect n A → Vect (succ n) A
```

Implicit Quantification

- How does `Vect` work, and how do we construct new vectors?

```
data Vect : Nat → Set → Set where
  nilV  : (A : Set) → Vect zero A
  consV : (n : Nat) → (A : Set)
       → A → Vect n A → Vect (succ n) A

-- Example of a vector with [6,2,8,3]
v1 : Vect 4 Nat
v1 = consV 3 Nat 6
   ( consV 2 Nat 2
   ( consV 1 Nat 8
   ( consV 0 Nat 3
   ( nilV Nat
   ))))
```

Implicit Quantification

- How does `Vect` work, and how do we construct new vectors?

```
data Vect : Nat → Set → Set where
  nilV  : {A : Set} → Vect zero A
  consV : {n : Nat} → {A : Set}
       → A → Vect n A → Vect (succ n) A

-- Example of a vector with [6,2,8,3]
v2 : Vect 4 Nat
v2 = consV 6 (consV 2 (consV 8 (consV 3 nilV)))
```

Implicit Quantification

- How does `Vect` work, and how do we construct new vectors?

```
data Vect : Nat → Set → Set where
  nilV  : ∀ {A} → Vect zero A
  consV : ∀ {n A}
       → A → Vect n A → Vect (succ n) A

-- Example of a vector with [6,2,8,3]
v3 : Vect 4 Nat
v3 = consV 6 (consV 2 (consV 8 (consV 3 nilV)))
```

Implicit Quantification

- How does `Vect` work, and how do we construct new vectors?

```
data Vect : Nat → Set → Set where
  []  : ∀ {A} → Vect zero A
  _::_ : ∀ {n A}
      →  A → Vect n A → Vect (succ n) A

-- Example of a vector with [6,2,8,3]
v4 : Vect 4 Nat
v4 = 6 :: 2 :: 8 :: 3 :: []
```

Dependent Pattern Matching

- The expressivity of dependent types also influences *how datatypes can be manipulated*, and the *static guarantees* they provide
- The core mechanism behind pattern matching: *dependent unification*
- Extract the first element of a list, in Haskell:

```
head :: [a] -> a
head (a : as) = a
head [] = error "Head on empty list"
```

- Extract the first element of a list, in Agda:

```
head : ∀ {n A} → Vect (succ n) A → A
head (a :: as) = a
```

- The `Vect` constructor for the empty list

```
[] : ∀ {A} → Vect zero A
```

is impossible because `zero` cannot be unified with `succ n`!

Dependent Pattern Matching (2)

- Zip of two lists, in Haskell:

```
zip :: [a] -> [b] -> [(a, b)]
zip [] [] = []
zip (a : as) (b : bs) = (a, b) : zip as bs
zip [] _ = error "Right list longer than left"
zip _ [] = error "Left list longer than right"
```

- Zip of two lists, in Agda:

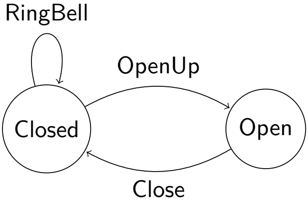
```
zip : ∀ {A B n} → Vect n A → Vect n B → Vect n (A × B)
zip [] [] = []
zip (a :: as) (b :: bs) = ⟨ a , b ⟩ :: zip as bs
```

- Core idea: inductive datatypes to *restrict* and *express* the validity of data
- Programs reason on *valid* cases: *"make invalid states unrepresentable"*

Transition Systems with Dependent Types

- A common use of dependent types: *embedding transition systems into code*
- States are tracked as types, programs are *compositions of transitions*

```
data DoorState : Set where
  DoorClosed : DoorState
  DoorOpen   : DoorState
```



```
data DoorCommand : DoorState → DoorState → Set where
  Open     : DoorCommand DoorClosed DoorOpen
  Close    : DoorCommand DoorOpen   DoorClosed
  RingBell : DoorCommand DoorClosed DoorClosed
  _>>_     : ∀ {S₁ S₂ S₃}
           → DoorCommand S₁ S₂
           → DoorCommand S₂ S₃
           → DoorCommand S₁ S₃
```

Transition Systems - Example

- Some simple examples of programs:

```
prog1 : DoorCommand DoorClosed DoorOpen
prog1 = do RingBell
           Open
           Close
           RingBell
           Open
```

- Programs can assume to start from any state, and can be composed:

```
prog2 : DoorCommand DoorOpen DoorOpen
prog2 = do Close
           RingBell
           prog1
           Close
           Open
```

Transition Systems - Vending Machine

```
VendState : Set          -- Transition system for a vending machine
VendState = Nat × Nat    -- States = (coins inserted, bars available)

data VendCom : VendState → VendState → Set where
  InsertCoin : ∀ {c b} → VendCom ⟨      c ,      b ⟩ ⟨ succ c , b ⟩
  Vend       : ∀ {c b} → VendCom ⟨ succ c , succ b ⟩ ⟨      c , b ⟩
  EmptyCoins : ∀ {c b} → VendCom ⟨      c ,      b ⟩ ⟨   zero , b ⟩
  Refill     : ∀ {c b} → (new : Nat)
                       → VendCom ⟨      c ,      b ⟩ ⟨ c , b + new ⟩
  _>>_       : ∀ {S₁ S₂ S₃}
             → VendCom S₁ S₂ → VendCom S₂ S₃ → VendCom S₁ S₃
```

```
vend1 : VendCom (0, 3) (1, 4)
vend1 = do InsertCoin
           Vend
           Refill 2
           InsertCoin
```

```
vend2 : ∀ {c b}
        → VendCom (c + 1, b) (0, b + 2)
vend2 = do Refill 4
           Vend
           InsertCoin
           Vend
           EmptyCoins
```

Inductive Relations

- *So far*: basic blocks of our logic + concrete dependent datatypes
- *Can we define new propositions/properties on top of dependent types?*
- We can use *inductive datatypes* to also encode **inductive relations**
- Less-than relation for naturals:

```
data _≤_ : Nat → Nat → Set where
  zn : ∀ {n}
       ---------
     → zero ≤ n
  ss : ∀ {a b}
     → a ≤ b
       ----------------
     → succ a ≤ succ b
```

- How do we prove that this kind of proposition holds? *Construct a term!*

```
three-less-seven : 3 ≤ 7
three-less-seven = ss (ss (ss zn))
```

Using Inductive Relations

- *Types can be used to express preconditions and properties of programs*
- Core idea: properties are *given* and *returned* by functions
- An example; get the *n*-th element in a vector:

```
data _<_ : Nat → Nat → Set where
  zs : ∀ {n}             → zero   < succ n
  ss : ∀ {a b} → a < b → succ a < succ b

nth : ∀ {A n}
    → (i : Nat)
    → Vect n A
    → i < n
    → A
nth zero     (x :: v) zs     = x
nth (succ i) (x :: v) (ss p) = nth i v p
```

First-class Propositions

- *How do we construct propositions?* ⇒ Functions that give us proofs
- Given any two a and b, decide whether a < b holds or not:

```
data _<_ : Nat → Nat → Set where
  zs : ∀ {n}             → zero   < succ n
  ss : ∀ {a b} → a < b → succ a < succ b

_<?_ : (a b : Nat) → (a < b) + (¬ (a < b))
a      <? zero   = injr <false-because-impossible>
zero   <? succ b = injl zs
succ a <? succ b
  with a <? b
     | injl  a<b = injl ( ss  a<b)
     | injr ¬a<b = injr (¬ss ¬a<b) -- ¬a<b ⇒ ¬(a+1)<(b+1)
```

- The idea: always validate and check indices before using nth!
- *(and if they are valid, we must provide the evidence to the function)*

Propositions as Invariants

- We can combine *propositions inside inductive datatypes* to express essential invariants of data structures, e.g. *binary search trees*:

```
data BST : (min : Nat) → (max : Nat) → (n : Nat) → Set where
  empty : ∀ {min max}
        → BST min max zero
  node  : ∀ {l r n m}
        → (a : Nat)
        → l < a
        → a < r
        → BST l a n
        → BST a r m
        → BST l r (n + m + 1)
```

Examples of Inductive Relations

- Some more examples of inductive relations for verification
- "A given predicate P holds for all elements of a vector":

```
data All : {A : Set} → {n : Nat}
        → (A → Set) → Vect n A → Set where
  empty : ∀ {A} {P : A → Set}
          ---------
        → All P []
  holds : ∀ {A P a n} {as : Vect n A}
        → P a
        → All P as
          --------------
        → All P (a :: as)
```

Examples of Inductive Relations

- An element "m" belongs to a list "xs":

```
data _∈_ : ∀ {A} → A → List A → Set where
  here : ∀ {A x} {xs : List A}
         -----------
       → x ∈ x :: xs
  next : ∀ {A x} (m : A) {xs : List A}
       → x ∈ xs
         -----------
       → x ∈ m :: xs
```

Examples of Inductive Relations

- The list "as" is a subset of the list "bs":

```
data _⊆_ : ∀ {A} → List A → List A → Set where
    emptysub : ∀ {A} {bs : List A}
              ----------------
        →         [] ⊆ bs
    skip : ∀ {A} {x : A} {as : List A} {bs : List A}
        →         as ⊆ bs
              ----------------
        →         as ⊆ x :: bs
    take : ∀ {A} {x : A} {as : List A} {bs : List A}
        →         as ⊆ bs
              ----------------
        → x :: as ⊆ x :: bs
```

Verifying the `filter` Function

- *How do we express properties of programs?*
- A function can return a proposition that *proves its correctness*
- A `filter` function for lists, *proving all elements satisfy* `P`:

```
filter : ∀ {A} {P : A → Set}
      → ((a : A) → P a + ¬ P a)
      → List A
      → (out : List A) × All P out
filter D [] = ⟨ [] , empty ⟩
filter D (x ∷ v) =
  let ⟨ out , all ⟩ = filter D v in
  with D x
     | injl yes = ⟨ x ∷ out , holds yes all ⟩
     | injr no  = ⟨     out ,           all ⟩
```

Verifying the `filter` Function (2)

- *Can we add another specification to* `filter`*?*
- "The list returned is a subset of the one in output":

```
filter : ∀ {A} {P : A → Set}
       → ((a : A) → P a + ¬ P a)
       → (input : List A)
       → (out : List A) × All P out × out ⊆ input
filter D [] = ⟨ [] , empty , emptysub ⟩
filter D (x ∷ v) =
  let ⟨ out , all , subs ⟩ = filter D v in
  with D x
     | injl yes = ⟨ x ∷ out , holds yes all , take subs ⟩
     | injr no  = ⟨     out ,           all , skip subs ⟩
```

Theorem Proving

- ...but if `filter` always returned the empty list it would also respect the specification!

  ```
  filter : ∀ {A} {P : A → Set}
         → ((a : A) → P a + ¬ P a)
         → List A
         → (out : List A) × All P out
  filter D xs = ⟨ [] , empty ⟩
  ```
- Specifying more and more properties also becomes unmanageable
- Can we verify a ”completeness” property of `filter` on its input?
- *Yes*, by verifying `filter` separately using *theorem proving!*
- Main advantage: theorems can be proven and checked *independently*
- A *theorem*: start from some premises, prove some conclusion
- With proposition-as-types, *theorems are just standard functions!*

Completeness of `filter`

```
filter-complete : ∀ {A} {xs : List A} {P : A → Set}
               → (D : (a : A) → P a + ¬ P a)
               → (a : A)
               → P a
               → a ∈ xs
                 ----------------
               → a ∈ filter D xs
filter-complete D a p here
  with D a
     | injl yes = here
     | injr no  = contradiction p no
filter-complete D a p (next m a∈xs)
  with filter-complete D a p a∈xs | D m
     | a∈x::xs                    | injl yes = next m a∈x::xs
     | a∈x::xs                    | injr no  =        a∈x::xs
```

Verifying sorting algorithms

- More advanced; verifying the correctness of sorting algorithms:
- Defining and proving the *correctness of insertion sort*:

```
data Sorted : List Nat → Set where
  sorted-[] : Sorted []
  sorted-::  : ∀ {x xs}
          → All (x ≤_) xs
          → Sorted xs
          → Sorted (x :: xs)

insert : Nat → List Nat → List Nat
insert x [] = [ x ]
insert x (y :: ys)
  with x ≤? y
     | injl x≤y = x :: y :: ys
     | injr y≤x = y :: insert x ys

insertion-sort : List Nat → List Nat
insertion-sort [] = []
insertion-sort (x :: xs) = insert x (insertion-sort xs)
```

Insertion sort (2)

```
≤-trans : ∀ {z x y} → x ≤ z → z ≤ y → x ≤ y
≤-trans zn y = zn
≤-trans (ss x) (ss y) = ss (≤-trans x y)

All≤-trans : ∀ {ys : List Nat} {x y ys}
          → y ≤ x
          → All (x ≤_) ys
          → All (y ≤_) ys
All≤-trans x≤y empty              = empty
All≤-trans x≤y (holds y≤z y≤*zs) = holds (≤-trans x≤y y≤z)
                                          (All≤-trans {as} x≤y y≤*zs)
All≤-insert : ∀ {ys y x}
           →      x ≤ y
           → All (x ≤_)           ys
           → All (x ≤_) (insert y ys)
All≤-insert {[]}        x≤y empty = holds x≤y empty
All≤-insert {z ∷ _} {y} x≤y (holds x≤z x≤*zs)
  with y ≤? z
     | injl y≤z = holds x≤y (holds x≤z x≤*zs)
     | injr z≤y = holds x≤z (All≤-insert x≤y x≤*zs)
```

Insertion sort (3)

```
insert-preserves-sorted : (x xs : List Nat)
                        → Sorted xs
                        → Sorted (insert x xs)
insert-preserves-sorted _ [] sorted-[] = sorted-:: empty sorted-[]
insert-preserves-sorted x (y :: ys) (sorted-:: y≤*ys sys)
  with x ≤? y
    | injl x≤y =
      sorted-:: (holds x≤y (All≤-trans x≤y y≤*ys))
                (sorted-:: y≤*ys sys)
    | injr y≤x =
      sorted-:: (All≤-insert y≤x y≤*ys)
                (insert-preserves-sorted x ys sys)

insertion-sort-sorts : (xs : List Nat) → Sorted (insertion-sort xs)
insertion-sort-sorts [] = sorted-[]
insertion-sort-sorts (x :: xs) =
  insert-preserves-sorted x (insertion-sort xs)
                            (insertion-sort-sorts xs)
```

Characteristics of Type-Based Verification

Type-Based Verification

- A unified language for both *programming, specification, and verification*
- Properties and their proofs become *first-class objects* of our language
- Verification is essentially providing *types-as-specifications*
- *How are specifications used in practice?*
- Two main approaches to software verification with dependent types:
    - **Internal verification**
    - **External verification**

Internal Verification

Internal verification:

Intertwine programs with properties and correctness proofs

- ☺ Development can be *guided* by the structure of invariants
- ☺ Descriptive types help in documenting programs
- ☹ Hard to change with changing requirements

Common use cases:

- Essential invariants of datatypes
- Enforcing correct-by-construction programming
- Structural properties checked with pattern matching

External Verification

External verification:

Correctness and properties are proved separately from programs

- ☺ No performance penalty on execution when passing proofs around
- ☺ Can be added, and verified/typechecked on-demand
- ☹ Specifications cannot be exploited when writing programs

Common use cases:

- Algebraic properties (e.g. associativity)
- As properties used for internal verification
- As specification to help describe the intended behaviour

Automation and Solvers

- Powerful and expressive logics, which also allow for *theorem proving*
- Obvious but crucial drawback: *everything is undecidable!*
- Big advantage of model checking verification: *push-button technology*
- *Can automation help us construct proofs and prove properties?*
- *Often automation can help!* Three general approaches:
    - **External tool integration**
    - **Metaprogramming solvers**
    - **Tactics and heuristics**

Automation and Solvers

- **Agsy**: *synthesize* (fragments of) proofs and programs using constructors

```
≤-trans : ∀ {x y z} → x ≤ z → z ≤ y → x ≤ y
≤-trans zn         _          = zn
≤-trans (ss x≤z) (ss z≤y) = ss (≤-trans x≤z z≤y)
```

- **Schmitty**: integrate external SMT solvers, such as Z3

```
thm1 : ∀ (x y : ℤ) → x - y ≤ x + y → x ≡ y
thm1 = solveZ3
```

- **agda-stdlib**: reflection-based monoid and ring solvers in Agda

```
thm2 : ∀ (a b c : ℤ) → a + b * c + 1 ≡ 1 + c * b + a
thm2 = solveRing
```

- **Coq, Isabelle**: emphasis on using tactics and user-directed automation; specify the general idea using imperative-style commands

Tactics with Coq

- *What does using tactics and automation look like?*
- Verifying the `filter` function in Coq:

```
Lemma filter_In :
    forall x l f, In x (filter l) <-> In x l /\ f x = true.
Proof.
    intros x l f;
    induction l as [|a ? ?];
    simpl.
  - tauto.
  - intros.
    case_eq (f a); intros; simpl;
    intuition congruence.
Qed.
```

- An underlying *proof term* is still constructed, and can be inspected!

Conclusion and Further Work

Dependent types provide an elegant and expressive framework combining functional programming and its verification using types

Other interesting aspects for type-based verification:

- proof irrelevance and proof-relevant compilation [Gilbert, 2019]
- extracting functional programs from proofs of correctness [Letouzey, 2008]
- verification of concurrent programs and protocols: session types [Ciccone, Padovani, 2020]
- resource usage with linear types: quantitative type theory [Atkey, 2019]
- combining CTL model checking with dependent types; [O'Connor, 2016]
- more expressive type theories: cubical type theory [Vezzosi et al., 2021]

Thank you for your attention!

Bibliography I

Ulf Norell.
Towards a practical programming language based on dependent type theory.
PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, 2007.

Ulf Norell.
Dependently typed programming in Agda.
In *Proceedings of TLDI'09: 2009 ACM SIGPLAN, 2009*, pages 1–2.

Aaron Stump.
Verified Functional Programming in Agda.
Association for Computing Machinery, 2016.

Bibliography II

Adam Chlipala.
Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.
The MIT Press, 2013.

Ana Bove, Peter Dybjer, and Ulf Norell.
A brief overview of Agda – a functional language with dependent types.
In *Theorem Proving in Higher Order Logics, 22nd International Conference, 2009. Proceedings*, volume 5674, pages 73–78.

James McKinna.
Why dependent types matter.
In *Proceedings of the 33rd ACM SIGPLAN-SIGACT, POPL 2006*.

Bibliography III

Gaëtan Gilbert.
A type theory with definitional proof-irrelevance.
PhD thesis, Mines ParisTech, France, 2019.

Luca Ciccone and Luca Padovani.
A dependently-typed linear π-calculus in Agda.
Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming, 2020.

Liam O'Connor.
Applications of applicative proof search.
In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016. ACM, 2016.

Bibliography IV

- Pierre Letouzey.
  Extraction in Coq: an overview.
  volume 5028 of *Lecture Notes in Computer Science*, pages 359–369.
  Springer, 2008.

- Zhaohui Luo.
  Computation and Reasoning - A Type Theory for Computer Science.
  Clarendon Press, 1994.

- Andrea Vezzosi, Anders Mörtberg, and Andreas Abel.
  Cubical Agda: A dependently typed programming language with univalence and higher inductive types.
  J. Funct. Program., 31:e8, 2021.

Appendix

LTL in Agda

- A proof-of-concept: embedding Linear Temporal Logic in Agda

```
data LTL : Set where
    ⟨_⟩   : (A → Set) → LTL
    _∧_   : LTL → LTL → LTL
    _∨_   : LTL → LTL → LTL
    ◯_    : LTL → LTL
    _U_   : LTL → LTL → LTL
    ◇_    : LTL → LTL
    □_    : LTL → LTL

record Stream (A : Set) : Set where
  constructor _::_
  coinductive
  field
    head : A
    tail : Stream A
```

LTL - Satisfiability (1)

```
suffix : ℕ → Stream A → Stream A
suffix zero     σ = σ
suffix (succ n) σ = suffix n (tail σ)

data _⊨_ : Stream A → LTL → Set where
    Elem : ∀ {P} {σ : Stream A}
         → P (head σ)
         → σ ⊨ ⟨ P ⟩
    And  : ∀ {σ : Stream A} {φ₁ φ₂}
         → σ ⊨ φ₁
         → σ ⊨ φ₂
         → σ ⊨ φ₁ ∧ φ₂
    OrL  : ∀ {σ : Stream A} {φ₁ φ₂}
         → σ ⊨ φ₁
         → σ ⊨ φ₁ ∨ φ₂
    OrR  : ∀ {σ : Stream A} {φ₁ φ₂}
         → σ ⊨ φ₂
         → σ ⊨ φ₁ ∨ φ₂
```

LTL - Satisfiability (2)

```
...
Next    : ∀ {σ : Stream A} {φ}
      → tail σ ⊨ φ
      →      σ ⊨ ◯ φ
UntilZ : ∀ {σ : Stream A} {φ₁ φ₂}
      →      σ ⊨ φ₂
      →      σ ⊨ φ₁ U φ₂
UntilS : ∀ {σ : Stream A} {φ₁ φ₂}
      →      σ ⊨ φ₁
      → tail σ ⊨ φ₁ U φ₂
      →      σ ⊨ φ₁ U φ₂
Always : ∀ {σ : Stream A} {φ}
      → (∀ i → suffix i σ ⊨ φ)
      → σ ⊨ □ φ
Eventually : ∀ {σ : Stream A} {φ}
      → (∃[ i ] suffix i σ ⊨ φ)
      → σ ⊨ ◊ φ
```

LTL - Examples (1)

```
_ ω : Σ → Stream Σ
_ ω = repeat

ex1 : a ∷ b ω ⊨ ⟨ A ⟩
ex1 = Elem isA

ex2 : a ∷ b ω ⊨ ◯ ⟨ B ⟩
ex2 = Next (Elem isB)

ex3 : a ω ⊨ □ ⟨ A ⟩
ex3 = Always canProve
  where canProve : ∀ i → suffix i (a ω) ⊨ ⟨ A ⟩
        canProve zero = Elem isA
        canProve (succ n) = canProve n
```

LTL - Examples (2)

```
ex4 : ∀ n → (b ^ n) ++ (a ω) ⊨ ◊ □ ⟨ A ⟩
ex4 zero = Eventually ⟨ zero , ex3 ⟩
ex4 (succ n) with ex4 n
... | Eventually ⟨ j , p ⟩ = Eventually ⟨ succ j , p ⟩

(A•B)ω : Stream Σ
(A•B)ω = interleave a b

ex5 : (A•B)ω ⊨ □ ◊ ⟨ B ⟩
ex5 = Always canProve
  where canProve : ∀ i → suffix i (A•B)ω ⊨ ◊ ⟨ B ⟩
        canProve zero               = Eventually ⟨ 1 , Elem isB ⟩
        canProve (succ zero)        = Eventually ⟨ 0 , Elem isB ⟩
        canProve (succ (succ n))    = canProve n

ex6 : ∀ n → (a ^ n) ++ [ b ] ++ (c ω) ⊨ ⟨ A ⟩ U ⟨ B ⟩
ex6 zero     = UntilZ (Elem isB)
ex6 (succ n) = UntilS (Elem isA) (ex6 n)
```