# Fortgeschrittene funktionale Programmierung

## LVA 185.A05, VU 2.0, ECTS 3.0
## SS 2020

(Stand: 22.04.2020)

Jens Knoop

Technische Universität Wien
Information Systems Engineering
Compilers and Languages

# Lecture 5

Part III: Quality Assurance

- (Chapter 4: Equational Reasoning... (topic of Lecture 3)
- Chapter 5: Testing
- Chapter 6: Verification

# Outline in more Detail (1)

## Part III: Quality Assurance

# Outline in more Detail (2)

- Chap. 6: Verification
  - 6.1 Inductive Proof Principles on Natural Numbers
    - 6.1.1 Natural Induction
    - 6.1.2 Strong Induction
    - 6.1.3 Excursus: Fibonacci and The Golden Ratio
  - 6.2 Inductive Proof Principles on Structured Data
    - 6.2.1 Induction and Recursion
    - 6.2.2 Structural Induction
  - 6.3 Inductive Proofs on Algebraic Data Types
    - 6.3.1 Inductive Proofs on Haskell Trees
    - 6.3.2 Inductive Proofs on Haskell Lists
    - 6.3.3 Inductive Proofs on Partial Haskell Lists
  - 6.4 Proving Properties of Streams
    - 6.4.1 Inductive Proofs on Haskell Stream Approximants
    - 6.4.2 Inductive Proofs on Haskell List and Stream Approximants

# Outline in more Detail (3)

- Chap. 6: Verification (cont'd)

# Chapter 5

## Testing

# Chapter 5.1

## Motivation

# Gaining Confidence in Correctness of Programs

...essentially, three means are at our disposal:

1. Correctness by Construction (*a priori*, cf. Chapter 4)
   - Exemplified by the development of functional pearls.
2. Verification (*a posteriori*, cf. Chapter 6)
   - Rigoros, formal correctness proofs (soundness of the specification, soundness of the implementation).
   - High confidence, high effort (typically).
3. Testing (*a posteriori*, Chapter 5)
   - *Ad hoc*: Controllable effort but usually no quantifiable quality statement; hence, a questionable overall value.
   - Systematically: Controllable effort, quantifiable quality statement.

...even if conducted systematically, we should keep in mind:

> Testing can only show the presence of errors.
> Not their absence.
>
> Edsger W. Dijkstra (11.5.1930-6.8.2002)
> *1972 Recipient of the ACM Turing Award*

...nonetheless, testing is often amazingly successful in revealing errors.

# Specifications: Basis of any Kind of Testing

...specifications (shall) describe and fix the meaning of programs:

- Informally, e.g., as commentary in the program or separately in another document.
  - $\rightsquigarrow$ Disadvantage: often ambiguous, open to interpretation.
- Formally, e.g., in terms of pre- and post-conditions, in a formal specification language with a precise semantics.
  - $\rightsquigarrow$ Advantage: precise and rigorous, unambiguous.

# Requirements for Systematic Testing

'Must' features:

- Specification-based
- Tool-supported
- Automatic

'Nice-to-Have' features:

- Reporting
  - What has been tested?
  - How thoroughly, how comprehensively has been tested?
  - How was success defined?
- Reproducibility, Repeatability
  - Reproducibility of tests
  - Repeatability of tests after program modifications

# We consider QuickCheck

...for systematic testing, a combinator library enabling tool-supported specification-based in Haskell.

QuickCheck

- defines a formal specification language
  ...allowing property definitions inside of the Haskell source code.
- defines a test data generator language
  ...allowing a simple and concise description of a large number of tests.
- allows tests to be repeated at will
  ...ensuring reproducibility.
- allows automatic testing of all properties specified in a module, including the delivery of success/failure reports
  ...with tests and reports automatically generated.

# It is worth noting

...that QuickCheck and its property specification and test data generator languages are

- examples of domain-specific embedded languages
  ...a special strength of functional programming.
- implemented as a combinator library in Haskell
  ...allowing us to make use of the full expressiveness of Haskell when defining properties and test data generators.
- part of the standard distribution of Haskell (for both GHC and Hugs; see module `QuickCheck`)
  ...ensuring easy access and immediate usability.

# Chapter 5.2

## Defining Properties

# Defining Simple Properties w/ QuickCheck (1)



...simple properties can be defined in terms of Boolean valued functions, so-called predicates.

Example:

Define inside of a Haskell program the (predicate) property:

```
prop_PlusAssociative :: Int -> Int -> Int -> Bool
prop_PlusAssociative x y z = (x+y)+z == x+(y+z)
```

Double-checking prop_PlusAssociative with Hugs yields:

```
Main>quickCheck prop_PlusAssociative
OK, passed 100 tests
```

# Defining Simple Properties w/ QuickCheck (2)

... slightly varying the introductory example.

Replace `Int` by `Float` in the property definition:

```
prop_PlusAssociative' :: Float -> Float -> Float -> Bool
prop_PlusAssociative' x y z = (x+y)+z == x+(y+z)
```

Double-checking `prop_PlusAssociative'` with Hugs might yield:

```
Main>quickCheck prop_PlusAssociative'
Falsifiable, after 13 tests:
1.0
-5.16667
-3.71429
```

# Note

- The type signatures for `prop_PlusAssociative` and `prop_PlusAssociative'` are necessary because of the overloading of `(+)`.
- If the type signatures were missing, error messages on ambiguous overloading would be issued; intuitively, QuickCheck needs to know which test data to generate.
- Type signatures in predicate definitions allow the type-specific generation of test data.
- Associativity of addition is falsifiable for type `Float`; think e.g. of rounding errors.
- Success/error reports are automatically issued and provide information on
  - the number of tests successfully passed
  - a counter example.

# A more Advanced Example

...illustrating limitations of property definitions as predicates.

Given:

- A function `insert :: Int -> [Int] -> [Int]`
- A predicate `is_ordered :: [Int] -> Bool`

To be tested:

- Correctness of the insertion operation: After inserting an element, the list shall be sorted.

Property definition as a Predicate:

```
prop_InsertOrdered :: Int -> [Int] -> Bool
prop_InsertOrdered x xs = is_ordered (insert x xs)
```

This property, however, is falsifiable: It is naive, since the argument list `xs` is not required to be sorted itself, and thus too strong.

# Advanced Features for Property Definitions (1)

...using new syntactic features for property definitions:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs
  = is_ordered xs ==> is_ordered (insert x xs)
```

Note:

- '`is_ordered xs ==>`' adds a precondition to the property definition; generated test data, which do not match the precondition, are discarded.
- '`==>`' is thus not a Boolean operator but affects the selection of test data; all such operators in `QuickCheck` have the result type `Property`.
- Using `==>` amounts to a trial-and-error approach for test data generation: 'Generate, then check whether the precondition is matched; if not, drop; repeat.'

# Advanced Features for Property Definitions (2)

...QuickCheck provides further features for property definitions to improve on this:

```
prop_InsertOrdered :: Int -> Property
prop_InsertOrdered x =
 forAll orderedLists $ \xs -> is_ordered (insert x xs)
```

*generates randomly a set of sorted lists*
*tested to satisfy*: `is_ordered (insert x xs)`

Note:

- While the preceding definition of `prop_InsertOrdered x xs = is_ordered xs ==>...` quantifies over all lists, the above property definition quantifies explicitly over the subset of ordered lists (cf. Chapter 5.5).
- Quantifying over subsets of values of a domain avoids test data generation in a trial-and-error fashion. Only 'meaningful' test data are generated.

# Looking ahead

...QuickCheck supports also the specification of much more sophisticated properties, e.g.:

- *The list resulting from insertion coincides with the argument list (except of the inserted element).*

as well as testing of

- more than one property at the same time.

  This is achieved by running a (small) program (also called quickCheck) from the command line. E.g., the call:

  – `Main>quickCheck Module.hs`

  checks all properties defined in `Module.hs` at the same time.

# Chapter 5.3

## Testing against Abstract Models

# Objective

Testing the correctness (or: soundness) of an

– implementation

against a

– reference implementation

of a so-called

– abstract model (or: reference model).

We demonstrate this considering an extended example:

– Testing soundness of an efficient implementation of queues against a less efficient reference implementation of an abstract model of queues.

# The Abstract Model of Queues

...defined in terms of an:

(Executable) Specification:

```
type Queue a = [a]

emptyQ        = []
enQ x q       = q ++ [x] -- Inefficient due to (++)!
is_emptyQ q   = null q   -- Cost of enQ proportional
frontQ (x:q)  = x        -- to number of list elements.
deQ (x:q)     = q
```

...in the following, this executable specification of 'first-in-first-out (FIFO)' queues serves as the reference implementation for queues; an implementation, which is simple but inefficient.

# Implementing Queues more Efficiently

...than by the reference implementation of the abstract model:

Key idea (due to F. Warren Burton, 1982):

- Split a queue into two portions (a queue front and a queue back).
- Store the back of the queue in reverse order.

This queue representation ensures:

- Efficient access to both queue front and queue back: (++) is replaced by (:) (so-called strength reduction).

Example:

- Queue representations: `[7,2,9,4,1,6,8,3]` $\widehat{=}$ `([7,2,9,4],[3,8,6,1])`, `([7,2],[3,8,6,1,4,9])`, `([7,2,9,4,1],[3,8,6])`,...
- Abstract model enqueuing, (++): `[7,2,9,4,1,6,8,3]++[5]`
- Implementation enqueuing, (:): `([7,2,9,4],5:[3,8,6,1])`, `([7,2],5:[3,8,6,1,4,9])`, `([7,2,9,4,1],5:[3,8,6])`,...

# Implementing the Abstract Model of Queues

Implementation:

```
type QueueI a      = ([a],[a])

emptyQI            = ([],[])
enQI x (f,b)       = (f,x:b)  -- (:) instead of (++)!
                              -- Therefore, more
                              -- efficient!
is_emptyQI (f,b) = null f
frontQI (x:f,b)  = x

deQI (x:f,b)       = flipQI (f,b)
 where
  flipQI ([],b)    = (reverse b,[])  -- 'back' be-
  flipQI q         = q               -- comes 'front'
                                     -- when 'front'
                                     -- gets empty.
```

# Relating Implementation and Abstract Model

...by means of the function `retrieve`:

```
retrieve :: QueueI a -> Queue a
retrieve (f,b) = f ++ reverse b
```

Note, `retrieve` transforms each of the (usually many)

- 'concrete' representations of an 'abstract' queue into their unique canonical representation as an 'abstract' queue, i.e., it transforms values of `(QueueI a)` into their unique matching value of `(Queue a)`.

Example:

```
retrieve ([7,2,9,4],[5,3,8,6,1]) ->> [7,2,9,4,1,6,8,3,5]
retrieve ([7,2],[5,3,8,6,1,4,9]) ->> [7,2,9,4,1,6,8,3,5]
retrieve ([7,2,9,4,1],[5,3,8,6]) ->> [7,2,9,4,1,6,8,3,5]
...
```

# Now

...we want to test whether operations defined on `(QueueI a)` behave in the same way as their specifying counterparts defined on `(Queue a)`.

For convenience, we focus on queues of integer values (i.e., `(QueueI Int)` and `(Queue Int)`). For this reason, we omit giving (the actually required) type signatures in property definitions.

Using `retrieve :: QueueI Int -> Queue Int` we can check, whether the results of applying

- the efficient operations on `(QueueI Int)` match the ones of their abstract counterparts on `(Queue Int)`.

# Soundness Properties: Initial Definitions

Defining five soundness properties:

```
prop_emptyQ      = retrieve emptyQI == emptyQ
prop_enQ x q     = retrieve (enQI x q)
                    == enQ x (retrieve q)
prop_isemptyQ q = is_emptyQI q
                    == is_emptyQ (retrieve q)
prop_frontQ q    = frontQI q == frontQ (retrieve q)
prop_deQ q       = retrieve (deQI q)
                    == deQ (retrieve q)
```

...which can reasonably be expected to hold, if the implementation of queues over `(QueueI Int)` is correct wrt their abstract model over `(Queue Int)`.

However, this is not true! Three (out of five) properties can be falsified!

# Falsifiability of prop_isemptyQ

Testing prop_isemptyQ using QuickCheck, e.g., yields:

```
Main>quickCheck prop_isemptyQ
Falsifiable, after 4 tests:
([],[-1])
```

Cause of failure: The definition of is_emptyQI implicitly assumes that the following invariant holds:

– (Silently assumed) invariant: The front of a list is only empty, if its back is empty, too:

$$\texttt{is\_emptyQI (f,b)} \Rightarrow \texttt{null b}$$

since is_emptyQI (f,b) = null f, emptyQI = ([],[]).

This invariant, however, is not enforced by the implementation!

# Falsifiability of `frontQI` and `deQI`

...the definitions of `frontQI` and `deQI` rely on the very same assumption as the one of `is_emptyQI` that the front of a queue is only empty, if its back is empty, too.

Thus, in addition to `prop_isemptyQ` the properties

- `prop_frontQ`
- `prop_deQ`

are falsifiable, too!

Remedy: The silently made assumption on the invariant, which we took care of when defining `deQI`, must be made explicit in the property definitions.

# Soundness Properties: 1st Refinement (1)

We define the invariant as follows:

```
invariant :: QueueI Int -> Bool
invariant (f,b) = (not (null f)) || null b
```

...and adjust the property definitions accordingly:

```
prop_emptyQ      = retrieve emptyQI == emptyQ
-------------------------------------------------
prop_enQ x q     = invariant q ==>
          retrieve (enQI x q) == enQ x (retrieve q)
prop_isemptyQ q = invariant q ==>
              is_emptyQI q == is_emptyQ (retrieve q)
prop_frontQ q    = invariant q ==>
                    frontQI q == frontQ (retrieve q)
prop_deQ q       = invariant q ==>
               retrieve (deQI q) == deQ (retrieve q)
```

# Soundness Properties: 1st Refinement (2)

Now, testing prop_isemptyQ using QuickCheck yields:

```
Main>quickCheck prop_isemptyQ
OK, passed 100 tests
```

However, testing prop_frontQ still fails:

```
Main>quickCheck prop_frontQ
Program error: front ([],[])
```

Cause of failure: frontQI (as well as deQI) may only be applied to non-empty lists.

...so far, we did not take care of this.

# Soundness Properties: 2nd Refinement

...to fix this, add `not (is_emptyQI q)` to the precondition of the challenged properties.

This leads to:

```
prop_emptyQ      = retrieve emptyQI == emptyQ
prop_enQ x q     = invariant q ==>
        retrieve (enQI x q) == enQ x (retrieve q)
prop_isemptyQ q = invariant q ==>
            is_emptyQI q == is_emptyQ (retrieve q)
--------------------------------------------------
prop_frontQ q = invariant q && not (is_emptyQI q) ==>
                  frontQI q == frontQ (retrieve q)
prop_deQ q    = invariant q && not (is_emptyQI q) ==>
             retrieve (deQI q) == deQ (retrieve q)
```

# Soundness Issues Reconsidered

After this 2nd refinement, all five properties pass now the `QuickCheck` test successfully!

However, we are not yet done. So far we only tested that

- operations on queues behave correctly on queues which satisfy the invariant:

```
invariant :: QueueI Int -> Bool
invariant (f,b) = (not (null f)) || null b
```

Additionally, we need to check that

- operations producing a queue do only produce queues which satisfy the invariant.

# Additional Soundness Properties

...for operations producing queues:

```
prop_inv_emptyQ  = invariant emptyQI
prop_inv_enQ x q = invariant q ==>
                            invariant (enQI x q)
prop_inv_deQ q   = invariant q &&
                    not (is_emptyQI q) ==>
                            invariant (deQI q)
```

# Testing the Additional Soundness Properties

Testing the additional properties with `QuickCheck` yields:

```
Main>quickCheck prop_inv_enQ
Falsifiable, after 0 tests:
0
([],[])
```

Cause of failure: The implementation of `enQI` does not ensure the validity of the invariant when applied to the empty list:

- Adding to the back of the empty queue breaks the invariant!

This means:

- The implementation of `enQI` by `enQI x (f,b) = (f,x:b)` is faulty and needs to be fixed!

# Fixing the Faulty Implementation of enQI

...by replacing the faulty implementation of `enQI`:

```
enQI x (f,b) = (f,x:b)
```

by the sound one:

```
enQI x (f,b) = flipQ (f,x:b)
 where
  flipQI ([],b) = (reverse b,[])
  flipQI q      = q
```

Now, all 8 properties pass the QuickCheck test successfully!

# Summary

...reconsidering the development of the example, testing revealed

- (only) one bug in the implementation (this was in function `enQI`; for `deQI`, we were keen to get handling empty back queues right from the very beginnings)
- several missing preconditions and one missing invariant in the initial property definitions.

This is typical, and both revealing flaws in implementations and property definitions is valuable:

- The initially missing preconditions and the invariant are now explicitly given in the program text as part of the property definitions.
- They add to understanding the program and are valuable as documentation, both for the program developer and for future users (think of program maintainance!).

# Chapter 5.4

## Testing against Algebraic Specifications

# Objective

Testing the correctness (or: soundness) of an

– implementation

against

– equational constraints

the operations ought to satisfy, a so-called

– algebraic specification.

...testing against an algebraic specification is (often) a useful alternative to testing against an abstract model. In the following, we demonstrate this considering queues as an example.

# Algebraic Specification of Queue Operations

...any proper definition of queue operations can be expected to satisfy the following equational constraints:

```
prop_isemptyQ q =
 invariant q ==> isEmptyQI q == (q == emptyQI)

prop_front_emptyQ x = frontQI (enQI x emptyQI) == x
prop_front_enQ x q =
 invariant q && not (is_emptyQI q) ==>
                frontQI (enQI x q) == frontQI q

prop_deQ_emptyQ x = deQI (enQI x emptyQI) == emptyQI
prop_deQ_enQ x q =
 invariant q && not (is_emptyQI q) ==>
             deQI (enQI x q) == enQI x (deQI q)
```

Compare these property definitions with the behaviour specification of the abstract data type (ADT) queue in Chapter 8.3!

# Testing against the Algebraic Specification

...testing the equational constraint `prop_deQ_enQ` using `QuickCheck` yields:

```
Main>quickCheck prop_deQ_enQ
Falsifiable, after 1 tests:
0
([1],[0])
```

Cause of failure: Evaluating

- the left hand side expression yields:
  `deQI (enQI 0 ([1],[0])) ->> deQI ([1],[0,0]) ->> flipQI ([],[0,0]) ->> ([0,0],[])`
- the right hand side expression yields:
  `enQI 0 (deQI ([1],[0])) ->> enQI 0 (flipQI ([],[0])) ->> enQI 0 ([0],[]) ->> ([0],[0])`
- `([0,0],[])` and `([0],[0])` are equivalent (they represent the abstract queue `[0,0]`) but are not exactly equal!


Lecture 5
Detailed Outline
Chap. 5
5.1
5.2
5.3
5.4
5.5
5.6
5.7
5.8
5.9
Chap. 6
Final Note

# Refining the Algebraic Specification

...by replacing testing for equality by testing for equivalence:

```
q `equiv` q' = invariant q && invariant q' &&
               retrieve q == retrieve q'
```

Replacing the initial formulation of:

```
prop_deQ_enQ x q =
 invariant q && not (is_emptyQI q) ==>
             deQI (enQI x q) == enQI x (deQI q)
```

by the new one:

```
prop_deQ_enQ x q =
 invariant q && not (is_emptyQI q) ==>
         deQI (enQI x q) `equiv` enQI x (deQI q)
```

the QuickCheck test of prop_deQ_enQ passes successfully!

# Testing further Equational Constraints

Analogously to Chapter 5.3, we also need to check that

– operations producing a queue do only produce queues which are equivalent, if the arguments are.

To this end, we introduce additional soundness properties for the operations `enQI` and `deQI`:

```
prop_enQ_equivQ q q' x =
 q `equiv` q' ==> enQI x q `equiv` enQI x q'
prop_deQ_equivQ q q' =
 q `equiv` q' && not (null q) && not (null q') ==>
                                      deQI q `equiv` deQI q'
```

# Note

...though mathematically sound, the usability of the property definitions `prop_enQ_equiv` and `prop_deQ_equiv` for testing with `QuickCheck` is limited.

Testing them with `QuickCheck`, we might observe, e.g.:

```
Main>quickCheck prop_enQ_equiv
     Arguments exhausted after 58 tests.
```

...which is due to an implementation feature of `QuickCheck`:

- `QuickCheck` generates the two lists `q` und `q'` randomly.
- Most of the generated pairs of lists will thus not be equivalent, and hence be discarded as test cases.
- `QuickCheck` makes a maximum number of tries of generating test cases (default: 1.000); afterwards, it stops, possibly before the number of 100 test cases is reached.

# Looking ahead

...QuickCheck provides features to cope with such problems of test case generation; providing especially support for

- Quantifying over subsets of value domains by means of
  - filters
  - generators (type-based, weighted, size controlled,...)
- ...
- Test case monitoring

...which we are going to illustrate next, mostly driven by examples.

# Chapter 5.5

## Controlling Test Data Generation

# Controlling Test Data Domains and Sizes (1)



...or: How to shape the 1) value domains test data are drawn from, and 2) the size of individual test data generated?

1) Value Domains of Test Data and Quantifying over Them

- By default, the parameters of `QuickCheck` properties are quantified over all values of the underlying data type (e.g., all integers, all lists of integers; not: all even integers, all sorted lists of integers, etc.).

  As we have seen, however, it is often preferable or even necessary to only quantify over subsets of a value domain (e.g., all sorted lists of integers).

# Controlling Test Data Domains and Sizes (2)

## 2) Size of Individual Test Data

– A set of test data drawn from a value domain should be a 'fair mix of smaller and larger values' avoiding the generation of extremely large values as well as of (too many) duplicates, in particular, of 'trivial' values (e.g., empty list, lists of length 1, empty trees, etc.).

  Meeting the 'fairness' requirement is especially challenging for data domains whose values are recursively defined (e.g., trees, lists, etc.).

QuickCheck offers several means for controlling

- quantification over sets and subsets of sets of value domains (cf. Chapter 5.5.1).
- the size of generated values (cf. Chapter 5.5.2).

# Chapter 5.5.1

## Controlling Quantification over Value Domains

# Controlling Quantification over Value Domains

Discussed so far (cf. Chapter 5.3, 5.4):

1. Boolean functions: Used as preconditions in property definitions act as test case filters selecting useful ones:
   - Works well, if most elements of the underlying value domain are members of the relevant subset, too.
   - Works poorly, if only a few elements of the underlying domain are members of the relevant subset.

Discussed next:

2. Generators: Used for targeted generation of test data of the subset of interest:
   - Generators of the monadic type `(Gen a)` generate random values of type `a`; conceptually, generators can be identified with the set of values they can generate.
   - Generators are used together with the property `forall set p`, which tests property `p` for all randomly generated elements of the set `set`.

# Note

...Boolean functions as test case filters and generators differ in their strengths and limitations for particular tasks, e.g., representing relations of values like equivalence of values. Representing value equivalence by a

- Boolean function makes it easy to check whether two values are equivalent, but difficult to generate values which are equivalent.
- Generator, i.e., a function mapping a value to a set of related (e.g., equivalent) values, makes it easy to generate equivalent values, but difficult to check if two given values are equivalent.

...we now continue with the generator approach.

# The 1-Ary Type Constructor `Gen`

...values generated by QuickCheck are of type `Gen a`.

The type constructor `Gen` is an instance of the type constructor class `Monad` (cf. Chapter 13), which eases the definition of concrete data generators.

Consider e.g. the two generator expressions `return a` and `do {x <- s; e}` of type `Gen a`:

- `return a` can be thought of to represent the singleton set {`a`}, and to generate value `a`.
- `do {x <- s; e}` can be thought of to represent the set {`e` | `x` $\in$ `s`}.

# The Function choose

...for random element generation.

choose is the most basic function of QuickCheck supporting to make a choice:

```
choose :: Random a => (a,a) -> Gen a
```

Note:

- Random denotes a type class provided by the library module Random of Haskell; its operations support the generation of pseudo-random numbers.
- choose generates a 'random' element of domain a of the specified range.
- Conceptually, choose (1,n), e.g., represents the set {1,...,n}, and randomly selects one element of it.

# Defining Generators using choose

...illustrated by defining the generator equivQ, which, given a queue value q, generates a new queue value q′ equivalent to q:

```
equivQ :: QueueI a -> Gen (QueueI a)
equivQ q =
 do k <- choose (0,0 `max` (n-1))
    return (take (n-k) els,reverse (drop (n-k) els))
    where els = retrieve q
          n   = length els
```

Note:

- Given a (QueueI a)-value q, equivQ generates randomly a queue q′ with the same elements as q.
- The number k of elements in the back queue of q′ is chosen properly smaller than the total number of elements of q′ (supposed this total number is different from 0).

# Property Definitions with Generators (1)

...using equivQ, we define soundness property:

```
prop_equivQ q = invariant q ==>
  forAll (equivQ q) $ \q' -> q `equiv` q'
```

...allowing to test, whether equivQ produces in fact queues, which are equivalent to the argument it is applied to.

Note:

- ($) means function application allowing the omission of parentheses (see the anonymous $\lambda$-expression in the definition of prop_equivQ).
- The property dual to prop_equivQ, whether all queues equivalent to some queue can be generated by equivQ, cannot in general be established by testing.

# Property Definitions with Generators (2)

...using equivQ, we can define counterparts of the properties prop_enQ_equivQ and prop_deQ_equivQ allowing to test, whether enQ and deQ map equivalent queues to equivalent queues:

```
prop_enQ_equivQ q x = invariant q ==>
  forAll (equivQ q) $ \q' -> enQI x q `equiv` enQI x q'
prop_deQ_equivQ q = invariant q && not (null q) ==>
  forAll (equivQ q) $ \q' -> deQI q `equiv` deQI q'
```

For comparison, we recall the initial definitions (cf. Chapter 5.4):

```
prop_enQ_equivQ q q' x =
  q `equiv` q' ==> enQI x q `equiv` enQI x q'
prop_deQ_equivQ q q' =
  q `equiv` q' && not (null q) && not (null q') ==>
                                    deQI q `equiv` deQI q'
```

# Type-based Generation of Value Sets

...is enabled by the overloaded generator `arbitrary`, e.g., for generating the argument values of properties:

Example: Generating (and testing) over unrestriced sets of numerical values:

```
prop_max_le =
  forAll arbitrary $ \x ->
    forAll arbitrary $ \y -> x <= x `max` y
```

This definition is equivalent to the short-hand form:

```
prop_max_le x y = x <= x `max` y
```

# Type-based Generation of Subsets of Value Sets

...can be achieved by `arbitrary` followed by a suitable value modification:

Example: The generator `atLeast` defined on top of `arbitrary` generates the set of numerical values $\{ y \mid y \geq x \}$:

```
atLeast x = do diff <- arbitrary
               return (x + abs diff)
```

Note, the definition of `atLeast` makes use of the equality of the sets:

$$\{ y \mid y \geq x \} = \{ x + abs\ d \mid d \in \mathbb{Z} \}$$

which is valid for numerical values (note, the idea underlying the definition of `atLeast` can be adapted to types other than numerical ones).

# Selecting a Generator

...is enabled by the generator oneof which can conceptually be thought of as set union operator.

Example: The generator orderedLists (cf. Chapter 5.2) for generating sorted lists is based on the idea that a sorted list is either 1) empty or 2) the result of attaching a new head element to a sorted list of larger elements:

```
orderedLists = do x <- arbitrary
                  listsFrom x
  where
   listsFrom x
    = oneof [return [],            -- either: empty
             do y <- atLeast x     -- or: a list of elems > x
                liftM (x:) (listsFrom y)] -- extended
                                  -- by x as new head element
```

# Note

...the `oneof` generator picks alternatives with

- equal probability.

This can impact the generation of test data unduly. E.g., the generator `orderedLists` will produce

- the empty list far too often

questioning its usability as an adequate test data generator for ordered lists.

QuickCheck offers thus means for a weighted selection of generators.

# Weighted Selection of a Generator

...is enabled by the generator `frequency`, which allows assigning weights to a set of selectable generators controlling their relative likelihood of being actually selected:

```
frequency :: [(Int,Gen a)] -> Gen a
```

Example:

```
listsFrom x
   = frequency [(1,return []),
                (4,do y <- atLeast x
                      liftM (x:) (listsFrom y))]
```

Note:

- `QuickCheck` generators correspond actually to a probability distribution over a set, rather than just the set itself.
- The assignment of weights above gives the cons case a weight of 4; generated lists will thus have an average length of 4 elements.

# Pragmatics: Generators as Default Generators

...if a generator like `orderedLists` is used frequently, this generator should be made the default generator for values of the generated type. To this end, define a new type for the value type generated and make this new type an instance of the type class `Arbitrary` as shown below:

```
newtype OrderedList a = OL [a]

instance (Num a, Arbitrary a) =>
                  Arbitrary (OrderedList a) where
  arbitrary = liftM OL orderedLists
```

Example: Redefining `insert` with the new type `OrderedList`

```
insert :: Ord a => a -> OrderedList a
                                  -> OrderedList a
```

ensures that arguments generated for `insert` will automatically be ordered.

# Chapter 5.5.2

## Controlling the Size of Test Data

# Controlling the Size of Test Data

...is usually necessary in order to avoid the generation of unreasonably large test cases; QuickCheck provides support for this.

QuickCheck generators are parameterized on an

- integer valued parameter `size`, which is gradually increased during testing (first tests explore small cases, later tests larger and larger ones).

The interpretation of the `size` parameter is up to the

- implementor of a test case generator (the default generator for lists, e.g., interpretes `size` as an upper bound on the length of lists).

Generators depending on `size` are defined using function:

```
sized :: (Int -> Gen a) -> Gen a
```

# Example

...the default generator vector for list values:

```
vector n = sequence [arbitrary | i <- [1..n]]
```

...calling `vector` with argument `length` generates lists of random values of length `length`.

`vector` in concert with function `sized`:

```
sized $ \n -> do length <- choose (0,n)
                 vector length
```

# The Function `resize`

...allows to supply an explicit size argument to a generator:

```
resize :: Int -> Gen a -> Gen a
```

Example: Generating a list of lists while bounding the total number of elements by the size parameter:

```
sized $ \n -> resize (round (sqrt (fromInt n))) arbitrary
```

Note: The definition uses the default generator but replaces the size parameter by its square root. The list of lists is generated by the default generator `arbitrary` but with a smaller size parameter.

# Chapter 5.5.3

## Example: Test Data Generators at Work

# Generators for Built-in and User-defined Types



Note, test data generators for

- predefined ('built-in') types of Haskell
  - are provided by QuickCheck.
- user-defined types
  - must be provided by the user in terms of defining suitable instances of the type class `Arbitrary`.
  - require usually measures to control the size of generated test data, especially for values of inductively defined types.

This is illustrated next considering binary trees as example:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

# A User-defined Generator for Binary Trees

...we make the type (Tree a) straightforwardly an instance of the type class Arbitrary:

```
instance Arbitrary a => Arbitrary (Tree a) where
 arbitrary =
  frequency [(1,return Leaf),
             (3,liftM3 Branch
                    arbitrary arbitrary arbitrary)]
```

Note: Assigning the weights (1 resp. 3) to the two subgenerators shall ensure that not too many trivial trees of size 1 are generated.

# Analyzing the Arbitrary Instance (`Tree a`)

Fact:

- The likelihood that a finite tree is generated, is only one third because termination is only possible, if all subtrees which are generated are finite.

Problem:

- With increasing breadth of the tree under generation, the requirement of selecting the 'terminating' branch must be satisfied simultaneously at ever more places pushing the likelihood for this towards 0.

Remedy: Using the `size` parameter in order to ensure

- termination.
- generation of 'reasonably' sized trees.

# The Refined Generator for Binary Trees

...replace the initial `instance`-declaration for `(Tree a)` by:

```
instance Arbitrary a => Arbitrary (Tree a) where
 arbitrary = sized arbTree

arbTree 0 = return Leaf
arbTree n | n>0 =
 frequency [(1,return Leaf),
            (3,liftM3 Branch shrub arbitrary shrub)]
  where shrub = arbTree (n `div` 2)
```

Note:

- `shrub` is a generator for 'small(er)' trees. It is not bound to a special tree; the two occurrences of `shrub` will usually generate different trees.
- Since the size limit for subtrees is halved, their total size is bounded by the argument value of `arbTree`.
- Generators for values of recursive types must usually be handled like in this example.

# A Note on Lift Functions

...lift functions used throughout Chapter 5.5 are provided by the library module `Monad` (cf. Chapter 13):

```
liftM  :: Monad m => (a -> b) -> (m a -> m b)
liftM2 :: Monad m => (a -> b -> c) -> (m a -> m b -> m c)
liftM3 :: Monad m => (a -> b -> c -> d) ->
                        (m a -> m b -> m c -> m d)
liftM4 :: Monad m => (a -> b -> c -> d -> e) ->
                 (m a -> m b -> m c -> m d -> m e)
liftM5 :: Monad m => (a -> b -> c -> d -> e -> f) ->
          (m a -> m b -> m c -> m d -> m e -> m f)
```

# Chapter 5.6

## Monitoring, Reporting, and Coverage

# Why is Test-Data Monitoring Useful?

...reconsider the example of inserting into a sorted list:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs
  = is_ordered xs ==> is_ordered (insert x xs)
```

`QuickCheck` checks `prop_InsertOrdered` by

- randomly generating lists

and checking each of them being sorted (used as a test case) or not (discarded).

# Analyzing Potential Risks

Fact:

- The likelihood that a randomly generated list is sorted decreases with its length.

  Conversely: The likelihood of being sorted is the higher the shorter the list is.

Risk:

- Property `prop_InsertOrdered` is likely to be mostly tested with lists of length one or two.
- Even QuickCheck runs run to completion are not meaningful.

# Test Data Monitoring and Reporting

...can thus provide useful hints on the

- quality and coverage of the test cases

of a QuickCheck run.

QuickCheck provides a variety of

- monitoring and reporting possibilities

for this purpose.

Instrumental are the QuickCheck combinators:

1. `trivial`
2. `classify`
3. `collect`

# The QuickCheck Combinator trivial

...allows monitoring and reporting the percentage of test cases which are considered trivial, where the meaning of

- 'trivial' is user-definable, e.g., lists up to a length of 2.

Example:

```
prop_InsertOrdered :: Int -> [Int] -> Property
prop_InsertOrdered x xs = is_ordered xs ==>
 trivial (length xs <= 2) $ is_ordered (insert x xs)
```

Double-checking the property with Hugs might yield:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests (91% trivial).
```

# Analyzing the QuickCheck Run

...reveals:

- 91% of the test cases were trivial checking lists of length 2 or shorter.
- These are far too many in order to ensure that the test run is meaningful.
- This shows again that the operator `==>` must be used with care in test case generators.

Remedy:

- Replacing the default means of test case generation by a user-defined generator, e.g., by proper quantification as sketched in Chapter 5.2.

Note:

- The combinator `trivial` is defined in terms of the more general combinator `classify`:

```
trivial p = classify p "trivial"
```

# The QuickCheck Combinator `classify`

...supports a more refined test-case monitoring and reporting than `trivial` by allowing to define sets of interesting test case classes:

Example:

```
prop_InsertOrdered x xs = is_ordered xs ==>
 classify (null xs) "empty lists" $
  classify (length xs == 1) "unit lists" $
   is_ordered (insert x xs)
```

Double-checking this property might yield:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
42% unit lists.
40% empty lists.
```

# The QuickCheck Combinator collect

...goes beyond the monitoring and reporting capabilities of even classify by delivering histograms of test case values.

Example:

```
prop_InsertOrdered x xs = is_ordered xs ==>
 collect (length xs) $ is_ordered (insert x xs)
```

Double-checking this property might yield:

```
Main>quickCheck prop_InsertOrdered
OK, passed 100 tests.
46% 0.
34% 1.
15% 2.
 5% 3.
```

# Chapter 5.7

## Implementation of QuickCheck

# QuickCheck: Facts and Figures

## QuickCheck

- consists in total of about 300 lines of code.
- has been developed by Koen Claessen and John Hughes.
- was initially presented in:
  - Koen Claessen, John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
- This chapter is mostly based on:
  - Koen Claessen, John Hughes. Specification-based Testing with QuickCheck. In Jeremy Gibbons, Oege de Moor (Eds.), The Fun of Programming. Palgrave MacMillan, 17-39, 2003.

# A Glimpse of the QuickCheck Code

```haskell
newtype Property = Prop (Gen Result)

class Testable a where
 property :: a -> Property

instance Testable Bool where
 property b = Prop (return (resultBool b))

instance Testable Property where
 property p = p

instance (Arbitrary a, Show a, Testable b) =>
                             Testable (a -> b) where
 property f = forAll arbitrary f

quickCheck :: Testable a => a -> IO ()
```

# For further Details

...including applications, refer to e.g.:

– Koen Claessen, John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.

– Koen Claessen, John Hughes. Testing Monadic Code with QuickCheck. In Proceedings of the ACM SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.

# Chapter 5.8

## Summary

# On the Relevance and Value

...of specifications, testing, and tools like QuickCheck.

Specifications: Experience shows

- Formalizing specifications is meaningful (even if they are not used for a formal proof of soundness).
- Specifications provided are (initially) often faulty themselves.

Testing: Investigations of Richard Hamlet reported in

– Richard Hamlet. Random Testing. Encyclopedia of Software Engineering, Wiley, 970-978, 1994.

indicate that

- results from a high number of test cases are meaningful even if test cases are randomly generated.
- random test case generation is often 'cheap.'

# On the Value of Tools like QuickCheck

Together, the findings on specifications and testing provide good reasons for using tools like QuickCheck on a

- routine basis.

Experience actually shows that QuickCheck is effective for

- disclosing bugs in programs and specifications with little effort.
- reducing test costs while at the same time testing more thoroughly.

Note that there is a range of other combinator libraries supporting the lightweight testing of Haskell programs, e.g.:

- EasyCheck
- SmallCheck
- Lazy SmallCheck
- Hat (for tracing Haskell programs)

# In Closing: Another Independent Confirmation

...of the relevance of testing:

> ...the success of tests is that they test
> the programmer, not the program.
> Rigorous testing regimes rapidly persuade
> error-prone programmers (like me) to remove
> themselves from the profession.
> [...]
> ...programmers who have survived the rigors of
> testing are what make programs of the present day
> useful, efficient, and (nearly) correct.

C. Antony Hoare (* 1934)
Recipient of the 1980 ACM A.M. Turing Award:
For his fundamental contributions to the definition and design of programming languages.

# Background: An Influential Work

...of Tony Hoare, advocating rigor and correctness from the very beginnings in software development:

– Charles A.R. Hoare. An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10): 576-580, 1969.

and a retrospective written 40 years later:

– Charles A.R. Hoare. Retrospective: An Axiomatic Basis for Computer Programming. Communications of the ACM 52(10):30-32, 2009.

# Ext. Quote from Hoare's Retrospective Article

"One thing I got spectacularly wrong. I could see that programs were getting larger, and I thought that testing would be an increasingly ineffective way of removing errors from them. I did not realize that the success of tests is that they test the programmer, not the program. Rigorous testing regimes rapidly persuade error-prone programmers (like me) to remove themselves from the profession. Failure in test immediately punishes any lapse in programming concentration, and (just as important) the failure count enables implementers to resist management pressure for premature delivery of unreliable code [...]. The experience, judgment, and intuition of programmers who have survived the rigors of testing are what make programs of the present day useful, efficient, and (nearly) correct. Formal methods for achieving correctness must support the intuitive judgment of programmers, not replace it. My basic mistake was to set up proof in opposition to testing, where in fact both of them are valuable and mutually supportive ways of accumulating evidence of the correctness and serviceability of programs."

# Chapter 5.9

## References, Further Reading

# Chapter 5: Basic Reading (1)


- Koen Claessen, John Hughes. *Specification-based Testing with QuickCheck*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 17-39, 2003.
- Koen Claessen, John Hughes. *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. In Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), 268-279, 2000.
- Koen Claessen, John Hughes. *Testing Monadic Code with QuickCheck*. In Proceedings of the ACM SIGPLAN 2002 Haskell Workshop (Haskell 2002), 65-77, 2002.

# Chapter 5: Basic Reading (2)


- Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 18.2, QuickCheck)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 19.6, DSLs for computation: generating data in QuickCheck)
- Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Chapter 11, Testing and Quality Assurance; Chapter 26, Advanced Library Design: Building a Bloom Filter – Testing with QuickCheck)

# Chapter 5: Selected Further Reading (1)


- Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, Malcolm Wallace. *Testing and Tracing Lazy Functional Programs Using `QuickCheck` and `Hat`*. In Johan Jeuring, Simon Peyton Jones (Eds.) *Advanced Functional Programming – Revised Lectures*. Springer-V., LNCS Tutorial 2638, 59-99, 2003.
- Jan Christiansen, Sebastian Fischer. *Easycheck – Test Data for Free*. In Proceedings of the 9th International Symposium on Functional and Logic Programming (SFLP 2008), Springer-V., LNCS 4989, 322-336, 2008.
- Colin Runciman, Matthew Naylor, Fredrik Lindblad. *SmallCheck and Lazy SmallCheck*. In Proceedings of the ACM SIGPLAN 2008 Haskell Workshop (Haskell 2008), 37-48, 2008. (Available from `http://hackage.haskell.org`)

# Chapter 5: Selected Further Reading (2)


- F. Warren Burton. *An Efficient Implementation of FIFO Queues*. Information Processing Letters 14(5):205-206, 1982.
- Richard Hamlet. *Random Testing*. In J. Marciniak (Ed.), Encyclopedia of Software Engineering, Wiley, 970-978, 1994.
- Charles A.R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM 12(10):576-580, 1969.
- Charles A.R. Hoare. *Retrospective: An Axiomatic Basis for Computer Programming*. Communications of the ACM 52(10):30-32, 2009.

This software comes “without warranty of any kind, expressed or implied, including but not limited to, the implied warranties of merchantability and fitness for a particular purpose.”

# Chapter 6

## Verification

...[both] proof [and] testing [...] are valuable and mutually supportive ways of accumulating evidence of the correctness and serviceability of programs.

C. Antony Hoare (* 1934)

Recipient of the 1980 ACM A.M. Turing Award:
For his fundamental contributions to the definition and design of programming languages.

...while coinciding in their overall goal, testing and verification (proof!) are of different rigor.

Testing, even if it can be amazingly effective, is limited to

- showing the presence of errors; it can not show their absence (except of the most simple scenarios).

while verification can

- prove the absence of errors!

# Important Proof Techniques

...for proving properties of functional programs so far:

- Equational reasoning (cf. Chapter 4)

In this chapter, we complement equational reasoning with proof techniques based on important inductive proof principles (not limited to functional programs) which may operate on:

- Unstructured data
  - integers
  - chars
  - Booleans
  - ...
- Structured data
  - lists (finite by definition)
  - streams (infinite by definition)
  - trees (finite or infinite)
  - ...

# Outline of Inductive Proof Principles

...we will consider:

- Inductive proof principles on natural numbers
  - Natural (or: mathematical) induction
    (dtsch. vollständige Induktion)
  - Strong induction (dtsch. verallgemeinerte Induktion)
- Inductive proof principles on structured data
  - Structural induction (dtsch. strukturelle Induktion)

  In particular:
  - Structural induction on lists
  - Structural induction on stream approximants
- Coinduction
- Fixed point induction

Ohne Mathematik tappt man doch immer im Dunkeln.

Werner von Siemens (1816-1892)
dt. Erfinder und Unternehmer

# Chapter 6.1

## Inductive Proof Principles on Natural Numbers

# Chapter 6.1.1

## Natural Induction

# The Principle of Natural Induction

Let $\mathbb{N}$ be the set of natural numbers, and $P$ be a property of natural numbers.

The Principle of Natural (or: Mathematical) Induction

$$\underbrace{P(1)}_{\text{Base Case}} \wedge [\overbrace{\forall n \in \mathbb{N}.\ \underbrace{P(n)}_{\text{Induction Hypothesis}} \ \underbrace{\Rightarrow P(n+1)}_{\text{Induction Step}}}^{\text{Inductive Case}}] \quad \Rightarrow \quad \underbrace{\forall\, n \in \mathbb{N}.\ P(n)}_{\text{Conclusion}}$$

(dtsch. Prinzip der vollständigen Induktion)

# Example: Illustrating Natural Induction



Lemma 6.1.1.1

$$\forall\, n \in \mathrm{I\!N}.\ \sum_{k=1}^{n} (2k-1) = n^2$$

Proof (by means of natural (mathematical) induction).

# Proof of Lemma 6.1.1.1 (1)

Base case: Let $n = 1$. In this case we obtain the equality of the left and right hand side expression straightforwardly by equational reasoning:

$$\begin{aligned}
\sum_{k=1}^{n}(2k-1) &= \sum_{k=1}^{1}(2k-1) \\
&= 2*1-1 \\
&= 2-1 \\
&= 1 \\
&= 1^2 \\
&= n^2
\end{aligned}$$

# Proof of Lemma 6.1.1.1 (2)

Inductive case: Let $n \in \mathbb{N}$. By means of the induction hypothesis (IH) we can assume $\sum_{k=1}^{n}(2k-1) = n^2$. This allows us to complete the proof by equational reasoning:

$$\begin{aligned}
\sum_{k=1}^{n+1}(2k-1) &= 2(n+1) - 1 + \sum_{k=1}^{n}(2k-1) \\
\text{(IH)} &= 2(n+1) - 1 + n^2 \\
&= 2n + 2 - 1 + n^2 \\
&= 2n + 1 + n^2 \\
&= n^2 + 2n + 1 \\
&= n^2 + n + n + 1 \\
&= (n+1)(n+1) \\
&= (n+1)^2
\end{aligned}$$

$\square$

# Chapter 6.1.2

## Strong Induction

# The Principle of Strong Induction

Let $\mathbb{N}$ be the set of natural numbers, and $P$ be a property of natural numbers.

The Principle of Strong Induction

$$\forall n \in \mathbb{N}.\ [\overbrace{\underbrace{(\forall m < n.\ P(m))}_{\text{Induction Hypothesis}} \ \underbrace{\Rightarrow P(n)}_{\text{Induction Step}}}^{\text{(Inductive) Case}}] \quad \Rightarrow \quad \underbrace{\forall\, n \in \mathbb{N}.\ P(n)}_{\text{Conclusion}}$$

(dtsch. Prinzip der verallgemeinerten Induktion)

Note: For the smallest natural number $\hat{n}$ ($\mathbb{N}_0$ vs. $\mathbb{N}_1$), the induction hypothesis boils down to 'true', i.e., $P(\hat{n})$ has to be proven without relying on anything special.

# Example: Illustrating Strong Induction

The Fibonacci function $fib : \mathbb{N}_0 \to \mathbb{N}_0$ is defined by:

$$\forall\, n \in \mathbb{N}_0.\ fib(n) =_{df} \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n \geq 2 \end{cases}$$

## Lemma 6.1.2.1

$$\forall\, n \in \mathbb{N}_0.\ fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

Proof (by means of strong induction).

# Key for Proving Lemma 6.1.2.1 for $n \geq 2$

...is to assume for $m = n - 1$ and $m = n - 2$ the equality:

$$fib(m) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^m - \left(\frac{1-\sqrt{5}}{2}\right)^m}{\sqrt{5}}$$

according to the induction hypothesis (IH).

(Note: For $n \geq 2$, the induction hypothesis would allow us to use this equality even for all $m < n$ (not just for $m = n - 1$ and $m = n - 2$). This, however, is not required to complete the proof.)

# Proof of Lemma 6.1.2.1 (1)

Case 1: Let $n = 0$. Equational reasoning yields straightforwardly the desired equality:

$$fib(0) = 0 = \frac{0}{\sqrt{5}} = \frac{1-1}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^0 - \left(\frac{1-\sqrt{5}}{2}\right)^0}{\sqrt{5}}$$

(Note: For proving Case 1, the induction hypothesis allows nothing to assume on the validity of the statement. Fortunately, nothing is required.)

Case 2: Let $n = 1$. Again, equational reasoning yields directly the desired equality:

$$fib(1) = 1 = \frac{\sqrt{5}}{\sqrt{5}} = \frac{\frac{1}{2} + \frac{\sqrt{5}}{2} - (\frac{1}{2} - \frac{\sqrt{5}}{2})}{\sqrt{5}} = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}}$$

(Note: For proving Case 2, we could have used the statement for $n = 0$ by means of the induction hypothesis. This, however, is not required.)

# Proof of Lemma 6.1.2.1 (2)

Case 3: Let $n \geq 2$. Using the Ind. Hypothesis for $n-2$, $n-1$ we obtain:

$$\begin{aligned}
fib(n) &= fib(n-2) + fib(n-1) \\
\text{(2x IH)} &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}}{\sqrt{5}} \\
&= \frac{\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1+\sqrt{5}}{2}\right)^{n-1}\right] - \left[\left(\frac{1-\sqrt{5}}{2}\right)^{n-2} + \left(\frac{1-\sqrt{5}}{2}\right)^{n-1}\right]}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1+\sqrt{5}}{2}\right] - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left[1 + \frac{1-\sqrt{5}}{2}\right]}{\sqrt{5}} \\
(*) &= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}\left(\frac{1+\sqrt{5}}{2}\right)^{2} - \left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\left(\frac{1-\sqrt{5}}{2}\right)^{2}}{\sqrt{5}} \\
&= \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n} - \left(\frac{1-\sqrt{5}}{2}\right)^{n}}{\sqrt{5}} \qquad \square
\end{aligned}$$

# Proof of $(*)$

Equality $(*)$ follows from equalities (1) and (2):

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 = 1+\frac{1+\sqrt{5}}{2} \tag{1}$$

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 = 1+\frac{1-\sqrt{5}}{2} \tag{2}$$

...which can be proved by equational reasoning and the binomial formulae $(BF)$:

$$\left(\frac{1+\sqrt{5}}{2}\right)^2 \stackrel{(BF)}{=} \frac{1+2\sqrt{5}+5}{4} = \frac{6+2\sqrt{5}}{4} = \frac{3+\sqrt{5}}{2} = 1+\frac{1+\sqrt{5}}{2}$$

$$\left(\frac{1-\sqrt{5}}{2}\right)^2 \stackrel{(BF)}{=} \frac{1-2\sqrt{5}+5}{4} = \frac{6-2\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2} = 1+\frac{1-\sqrt{5}}{2}$$

# Chapter 6.2

## Inductive Proof Principles on Structured Data

# Chapter 6.2.1

## Induction and Recursion

# Induction and Recursion

...two closely related notions.

Induction

- describes things starting from something very simple, and building up from there: A bottom-up principle.

Recursion

- starts from the whole thing, working backward to the simple case(s): A top-down principle.

Induction and recursion can thus be considered

- the two sides of the same coin.

# The Context-Dependent Preferred Usage

...of induction over recursion resp. vice versa

- e.g., defining data structures (induction)
- e.g., defining algorithms (recursion)

is often mostly due to historical reasons.

Data types (inductive view):

```
data Tree = Leaf Int | Node Tree Int Tree
```

Algorithms (recursive view):

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
```

# Illustration

- Inductive definition of arithmetic expressions:
  - (r1) Each numeral $n$ and variable $v$ is an (atomic) arithmetic expression.
  - (r2) If $e_1$ and $e_2$ are arithmetic expressions, then also $(e_1 + e_2)$, $(e_1 - e_2)$, $(e_1 * e_2)$, and $(e_1/e_2)$.
  - (r3) Every arithmetic expression is inductively constructed by means of rules (r1) and (r2).

- Recursive definition of the merge sort algorithm:

  A list of integers $l$ is sorted by the following 3 steps:
  - (ms1) Split $l$ into two sublists $l_1$ and $l_2$.
  - (ms2) Sort the sublists $l_1$ and $l_2$ recursively obtaining their sorted counterparts $sl_1$ and $sl_2$, respectively.
  - (ms3) Merge $sl_1$ and $sl_2$ into the sorted list $sl$ of $l$.

# In Closing

Data structures often follow an

- inductive definition pattern, e.g.:
  - A list is either empty or a pair consisting of an element and another list.
  - A (binary) tree is either a leaf or it is composed of a node and a left and a right subtree.
  - An arithmetic expression is either a numeral or a variable, or it is composed of (two) arithmetic expressions by means of a (binary) arithmetic operator.

Algorithms (functions) on data structures often follow a

- recursive definition pattern, e.g.:
  - The function `length` computing the length of a list.
  - The function `depth` computing the depth of a tree.
  - The function `evaluate` computing the value of an arithmetic expression (given a valuation of its variables).

# Chapter 6.2.2

## Structural Induction

# The Principle of Structural Induction

Let $A$ and $O$ be a set of atoms and operators, respectively; let $S$ be the set of elements inductively constructed from $A$ and $O$. Let $sub(s) \subseteq S$, $s \in S$, denote the set of elements $s$ is composed of, and let $P$ be a property of the elements of $S$.

**The Principle of Structural Induction**

$$\forall\, s \in S.\ [\overbrace{\underbrace{(\forall\, s' \in sub(s).\ P(s'))}_{\text{Induction Hypothesis}}\ \underbrace{\Rightarrow P(s)}_{\text{Induction Step}}}^{\text{(Inductive) Case}}] \quad \Rightarrow \quad \underbrace{\forall\, s \in S.\ P(s)}_{\text{Conclusion}}$$

(dtsch. Prinzip der strukturellen Induktion)

Note: For the atoms $\hat{s}$ of $S$, the 'simplest' elements of $S$, we have $sub(\hat{s}) = \emptyset$. For these elements the induction hypothesis boils down to 'true,' i.e., $P(\hat{s})$ has to be proven without relying on anything special.

# Example: Illustrating Structural Induction

...the set of (simple) arithmetic expressions $\mathcal{AE}$ is defined by the BNF rule:

$$e \ ::= \ n \mid v \mid (e_1 + e_2) \mid (e_1 - e_2) \mid (e_1 * e_2) \mid (e_1 / e_2)$$

where $n$ and $v$ stand for (integer) numerals and variables, respectively.

## Lemma 6.2.2.1

Let $p_e$ and $op_e$, $e \in \mathcal{AE}$, denote the number of parentheses and operators of $e$, respectively. Then:

$$\forall\, e \in \mathcal{AE}.\ p_e \ = \ 2 * op_e$$

Proof (by means of structural induction).

# Proof of Lemma 6.2.2.1 (1)

(Base) case: Let $e \equiv n$, $n$ a numeral, or $e \equiv v$, $v$ a variable.

In both cases $e$ is free of parentheses and operators, i.e.:

$$p_e = 0 = op_e \qquad (*)$$

Using $(*)$, equational reasoning yields directly the desired equality:

$$\begin{array}{rcl} & & p_e \\ (*) & = & 0 \\ & = & 2 * 0 \\ (*) & = & 2 * op_e \end{array}$$

# Proof of Lemma 6.2.2.1 (2)

(Inductive) case: Let $e \equiv (e_1 \circ e_2)$, $\circ \in \{+, -, *, /\}$, and $e_1, e_2 \in \mathcal{AE}$. By means of the induction hypothesis (IH), we can assume $p_{e_1} = 2 * op_{e_1}$ and $p_{e_2} = 2 * op_{e_2}$. The equality of $p_e$ and $2 * op_e$ follows then by equational reasoning:

$$
\begin{array}{rcl}
 & & p_e \\
(\, e \equiv (e_1 \circ e_2)\,) & = & p_{(e_1 \circ e_2)} \\
 & = & 1 + p_{e_1} + p_{e_2} + 1 \\
\text{(2x IH)} & = & 2 * op_{e_1} + 2 + 2 * op_{e_2} \\
 & = & 2 * op_{e_1} + 2 * 1 + 2 * op_{e_2} \\
 & = & 2 * (\, op_{e_1} + 1 + op_{e_2}\,) \\
 & = & 2 * op_{(e_1 \circ e_2)} \\
(\,(e_1 \circ e_2) \equiv e\,) & = & 2 * op_e
\end{array}
$$

$\square$

# Note

...the principles of

- natural (math.) induction (dtsch. vollständige Induktion)

  $P(1) \wedge [\forall n \in \mathbb{N}.\, P(n) \Rightarrow P(n+1)] \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\, P(n)$

- strong induction (dtsch. verallgemeinerte Induktion)

  $\forall n \in \mathbb{N}.\, [\,(\forall m < n.\, P(m)) \Rightarrow P(n)\,] \quad \Rightarrow \quad \forall\, n \in \mathbb{N}.\, P(n)$

- structural induction (dtsch. strukturelle Induktion)

  $\forall\, s \in S.\, [\,(\forall\, s' \in sub(s).\, P(s')) \Rightarrow P(s)\,] \;\Rightarrow\; \forall\, s \in S.\, P(s)$

are equally expressive.

# Chapter 6.3

## Inductive Proofs on Algebraic Data Types

# Chapter 6.3.1

## Inductive Proofs on Haskell Trees

# Inductive Proofs on Finite Trees

A tree is called

- finite, if every path originating at its root has finite length.
- maximum, if it is finite and all paths from a leaf to its root have the same length.

Let

```
data Tree = Leaf Int | Node Tree Tree
```

## Lemma 6.3.1.1

Let *depth*(t) and *leaves*(t) denote the depth and the number of leaves of any finite tree value `t :: Tree`, respectively. Then:

$$\forall\, \texttt{t :: Tree}.\ \texttt{t}\ \mathit{maximum} \Rightarrow \mathit{leaves}(\texttt{t}) = 2^{\mathit{depth}(\texttt{t})}$$

Proof (by means of structural induction).

# Proof of Lemma 6.3.1.1 (1)

Base case: Let $\texttt{t} \equiv (\texttt{Leaf k})$ for some integer value $\texttt{k}$.

Here, we have $depth(\texttt{t}) = 0$ and $leaves(\texttt{t}) = 1$. Equational reasoning yields the desired equality of $leaves(\texttt{t})$ and $2^{depths(\texttt{t})}$:

$$
\begin{array}{rcl}
 & & leaves(\texttt{t}) \\
(\texttt{t} \equiv (\texttt{Leaf k})) & = & leaves(\texttt{Leaf k}) \\
 & = & 1 \\
 & = & 2^0 \\
 & = & 2^{depths(\texttt{t})}
\end{array}
$$

# Proof of Lemma 6.3.1.1 (2)

Inductive case: Let $\texttt{t} \equiv (\texttt{Node t1 t2})$ maximum. This implies $\texttt{t1}$, $\texttt{t2}$ are maximum themselves, $\mathit{depth}(\texttt{t1}) = \mathit{depth}(\texttt{t2})$, and $\mathit{depth}(\texttt{t}) = \mathit{depth}(\texttt{t1}) + 1 = \mathit{depth}(\texttt{t2}) + 1$. By means of the inductive hypothesis (IH) we can assume $\mathit{leaves}(\texttt{t1}) = 2^{\mathit{depth}(\texttt{t1})}$ and $\mathit{leaves}(\texttt{t2}) = 2^{\mathit{depth}(\texttt{t2})}$. This allows us to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \mathit{leaves}(\texttt{t}) \\
(\,\texttt{t} \equiv (\texttt{Node t1 t2})\,) & = & \mathit{leaves}(\texttt{Node t1 t2}) \\
 & = & \mathit{leaves}(\texttt{t1}) + \mathit{leaves}(\texttt{t2}) \\
(\text{2x IH}) & = & 2^{\mathit{depth}(\texttt{t1})} + 2^{\mathit{depth}(\texttt{t2})} \\
(\mathit{depth}(\texttt{t1}) = \mathit{depth}(\texttt{t2})) & = & 2^{\mathit{depth}(\texttt{t1})} + 2^{\mathit{depth}(\texttt{t1})} \\
 & = & 2 * 2^{\mathit{depth}(\texttt{t1})} \\
 & = & 2^{\mathit{depth}(\texttt{t1+1})} \\
 & = & 2^{\mathit{depth}(\texttt{t})}
\end{array}
$$

$\square$

# Chapter 6.3.2

## Inductive Proofs on Haskell Lists

# Lists

...can contain

- defined values only, e.g.:
  `[], (1 : 2 : 3 : []), (1 : 4 `div` 2 : 3 : []),...`
- defined and undefined values, e.g.:
  `(1 : 4 `div` 0 : 3 : fac (-1) : []),...`

  ```
  head (1 : 4 `div` 0 : 3 : fac (-1) : []) ->> 1
  head (tail (1 : 4 `div` 0 : 3 : fac (-1) : [])) ->> 'error'
  head (tail (tail (tail (1 : 4 `div` 0 : 3 : fac (-1) : []))))
   ->> 'non-termination'
  ```

We thus consider

- defined and undefined values

in more detail and distinguish structural induction on lists with

- (only) defined values.
- defined and undefined values.

# Chapter 6.3.2.1

## Defined and Undefined Values

# Defined and Undefined Values

A computation is

- faulty, if it
  - produces an error.
  - does not (regularly) terminate.

  The value of a faulty computation is called
  - undefined (or: the undefined value)

  and usually denoted by the symbol $\perp$ (read: 'bottom').
- non-faulty if its value
  - is different from $\perp$.

  The value of a non-faulty computation is called
  - defined (or: a defined value).

# Example

The function

```
buggy_div :: Int -> Int
buggy_div n = div n 0
```

...produces an error for every argument called with.

The function

```
buggy_fac :: Int -> Int
buggy_fac n = (n-1) * buggy_fac n
buggy_fac 0 = 1
```

...does not (regularly) terminate for any argument called with.

# Simple Haskell Terms

...with value $\bot$:

- Error: The Prelude definition

```
undefined :: a                        -- polymorphic
undefined | False = undefined
```

```
undefined ->> 'error' ≙ ⊥
```

  is an expression (of arbitrary type) whose evaluation leads to an error due to case exhaustion.

- Non-termination: The co-recursive definition

```
loop :: a                             -- polymorphic
loop = loop
```

```
loop ->> loop ->> loop ->> ... ≙ ⊥
```

  is an expression (of arbitrary type) whose evaluation does not (regularly) terminate.

# The Undefined Value $\perp$

- is an element of every Haskell data type, i.e.: $\perp$ :: a.
- is the value of faulty or non-terminating computations.
- can be considered an approximation (the 'least accurate' one) of any ordinary value of a data type.

This gives rise to:

## Definition 6.3.2.1.1 (Defined, Undefined Values)

The value of any data type representing the result of a faulty or non-terminating computation is called undefined and denoted by $\perp$; all other values of a data type are called defined.

# Lists

...are finite sequences of values built from the empty list.

## Definition 6.3.2.1.2 (List)

A list is a possibly empty finite sequence of

- (defined or undefined) values of the same type
- built from the empty list `[]`.

It is called

- defined, if none of its values equals $\perp$.
- a list with possibly undefined values, if some of its values can equal $\perp$.

# Illustration

Haskell lists are

- possibly empty finite sequences of values of the same type.
  Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...
- built from the empty list.
  Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...
- composed of defined and undefined values.
  Examples: `[]`, `(1:2:[])`, `(1:`$\perp$`:3:[])`, `(`$\perp$`:`$\perp$`:3:[])`,...

Haskell lists are

- defined, if all their values are defined.
  Examples: `[]`, `(1:[])`, `(1:2:3:[])`,...
- lists with undefined values, if some of their values equal the undefined value.
  Examples: `(`$\perp$`:[])`,`(1:`$\perp$`:[])`, `(`$\perp$`:2:`$\perp$`:[])`,...

# Chapter 6.3.2.2

## Structural Induction over Defined Lists

# Structural Induction over Defined Lists

Let $P$ be a property of defined lists.

Proof pattern of structural induction over defined lists

1. Base case: Prove that $P(\texttt{[]})$ is true.
2. Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\texttt{x:xs})$ is true (induction step).

Note: This pattern is an instance of the more general pattern of structural induction, specialized here for defined lists.

# Example: Induction over Defined Lists

Let

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

## Lemma 6.3.2.2.1

$$\forall\, \mathtt{xs}, \mathtt{ys}\ \textit{defined} :: [\mathtt{a}].$$

$$\mathtt{length}\ (\mathtt{xs} +\!\!+\, \mathtt{ys}) = \mathtt{length}\ \mathtt{xs}\ +\ \mathtt{length}\ \mathtt{ys}$$

Proof (by induction on the structure of xs).

# Proof of Lemma 6.3.2.2.1 (1)

Let `ys :: [a]` be a defined list.

Base case: Let $xs \equiv []$. As desired, we obtain by means of equational reasoning:

```
    length (xs ++ ys)
 =  length ([] ++ ys)
 =  length ys
 =  0 + length ys
 =  length [] + length ys
 =  length xs + length ys
```

# Proof of Lemma 6.3.2.2.1 (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x:xs}')$, $\texttt{xs}$ defined. This implies $\texttt{xs}'$ (and $\texttt{x}$) is defined, too. By means of the induction hypothesis (IH), we can thus assume $\texttt{length (xs}' \mathbin{++} \texttt{ys)} = \texttt{(length xs}' \texttt{ + length ys)}$. This allows to complete the proof as follows:

$$
\begin{array}{lll}
 & & \texttt{length (xs ++ ys)} \\
 & = & \texttt{length ((x:xs}'\texttt{) ++ ys)} \\
 & = & \texttt{length (x:(xs}' \mathbin{++} \texttt{ys))} \\
 & = & \texttt{1 + length (xs}' \mathbin{++} \texttt{ys)} \\
\text{(IH)} & = & \texttt{1 + (length xs}' \texttt{ + length ys)} \\
 & = & \texttt{(1 + length xs}'\texttt{) + length ys} \\
 & = & \texttt{length (x:xs}'\texttt{) + length ys} \\
 & = & \texttt{length xs + length ys}
\end{array}
$$

$\square$

# Chapter 6.3.2.3

## Structural Induction over Lists with Undefined Values

# Structural Induction for Lists w/ Undef. Values

Let $P$ be a property of lists with possibly undefined values.

Proof pattern of structural induction over lists with possibly undefined values:

1. Base case: Prove that $P(\texttt{[]})$ is true.
2. Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\bot\texttt{:xs})$ and $P(\texttt{x:xs})$, $\texttt{x}$ a defined value, are true (induction step).

Note: This pattern is an instance of the more general pattern of structural induction, specialized here for lists with possibly undefined values.

# Example: Induct. over Lists w/ Undef. Values

Let

```
length :: [a] -> Int
length []      = 0
length (_:xs) = 1 + length xs
```

## Lemma 6.3.2.3.1

$$\forall\, \texttt{xs}, \texttt{ys} \textit{ with possibly undefined values} :: \texttt{[a]}.$$
$$\texttt{length (xs ++ ys)} = \texttt{length xs + length ys}$$

Proof (by induction over the structure of xs).

# Proof of Lemma 6.3.2.3.1 (1)

Let `ys :: [a]` be a list with possibly undefined values.

Base case: Let $\texttt{xs} \equiv \texttt{[]}$. As desired, we obtain by means of equational reasoning:

```
  length (xs ++ ys)
= length ([] ++ ys)
= length ys
= 0 + length ys
= length [] + length ys
= length xs + length ys
```

# Proof of Lemma 6.3.2.3.1 (2)

Inductive case 1: Let $\texttt{xs} \equiv (\perp\!:\!\texttt{xs}')$. By means of the induction hypothesis (IH), we can assume $\texttt{length (xs}' \texttt{++ ys)} = \texttt{(length xs}' \texttt{ + length ys)}$. This allows to complete the proof as follows:

$$
\begin{array}{rrl}
 & & \texttt{length (xs ++ ys)} \\
 & = & \texttt{length ((}\perp\texttt{:xs}'\texttt{) ++ ys)} \\
 & = & \texttt{length (}\perp\texttt{:(xs}'\texttt{ ++ ys))} \\
 & = & \texttt{1 + length (xs}'\texttt{ ++ ys)} \\
\text{(IH)} & = & \texttt{1 + (length xs}'\texttt{ + length ys)} \\
 & = & \texttt{(1 + length xs}'\texttt{) + length ys} \\
 & = & \texttt{length (}\perp\texttt{:xs}'\texttt{) + length ys} \\
 & = & \texttt{length xs + length ys}
\end{array}
$$

# Proof of Lemma 6.3.2.3.1 (3)

Inductive case 2: Let $\texttt{xs} \equiv (\texttt{x:xs}')$, $\texttt{x}$ defined. By means of the induction hypothesis (IH), we can assume $\texttt{length (xs}' \texttt{++ ys)} = \texttt{(length xs}' \texttt{ + length ys)}$. This allows to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \texttt{length (xs ++ ys)} \\
 & = & \texttt{length ((x:xs}'\texttt{) ++ ys)} \\
 & = & \texttt{length (x:(xs}' \texttt{++ ys))} \\
 & = & \texttt{1 + length (xs}' \texttt{++ ys)} \\
\text{(IH)} & = & \texttt{1 + (length xs}' \texttt{ + length ys)} \\
 & = & \texttt{(1 + length xs}'\texttt{) + length ys} \\
 & = & \texttt{length (x:xs}'\texttt{) + length ys} \\
 & = & \texttt{length xs + length ys}
\end{array}
$$

$\square$

# Chapter 6.3.3

## Inductive Proofs on Partial Haskell Lists

# Chapter 6.3.3.1

## Partial Lists

# Partial Lists

...are finite sequences of values built from the undefined list.

## Definition 6.3.3.1.1 (Partial List)

A partial list is a possibly empty finite sequence of

- (defined or undefined) values of the same type
- built from the undefined list $\perp$.

It is called

- defined, if none of its values equals $\perp$ (and there is at least one).
- a partial list with possibly undefined values, if some of its values can equal $\perp$.

# Illustration

Partial Haskell lists are

- possibly empty finite sequences of values built from the undefined list.
  Examples: $\perp$, `(1:`$\perp$`)`, `(1:2:`$\perp$`)`, `(1:2:3:`$\perp$`)`,...
- partial lists with undefined values, if some of their values equal the undefined value.
  Examples: `(1:`$\perp$`:3:`$\perp$`)`, `(1:`$\perp$`:`$\perp$`:`$\perp$`)`. `(`$\perp$`:`$\perp$`:`$\perp$`:`$\perp$`)`,...

Note the different types of $\perp$ and $\perp$ in the above examples:

$\perp$ :: `Int`

$\perp$ :: `[Int]`

# Chapter 6.3.3.2

## Computing with Lists and Partial Lists

# Some Examples of Lists and Partial Lists

...with and without undefined values:

```
empty = []                    -- Empty list
ns = 2 : 3 : 5 : 7 : []       -- Defined list
ms = 2 : loop : 5 : 7 : []    -- List w/ undefined
                              -- values

pempty = loop :: [Int]        -- Empty partial list
xs = 2 : 3 : 5 : 7 : loop     -- Def. partial list
ys = 2 : loop : 5 : 7 : loop  -- Partial list w/
                              -- undefined values
```

Note: All occurrences of `loop` in `ns`, `ms`, `xs`, `ys`, and `pempty` have value $\perp$ but of different type:

– `loop` = $\perp$ `:: Int` in `ms` and `ys`.

– `loop` = $\perp$ `:: [Int]` in `pempty`, `xs`, and `ys` .

# Using the Definitions (1)

...introduced before, we get:

```
reverse ns ->> [7,5,3,2]
reverse ms ->> [7,5  ...followed by an infinite wait
reverse xs ->> ...infinite wait
reverse ys ->> ...infinite wait

head (reverse ms) ->> 7          -- thanks to lazy eval.
head (tail (reverse ms)) ->> 5  -- thanks to lazy eval.
head (tail (tail (reverse ms))) ->> ...infinite wait

head (tail (reverse xs)) ->> ...infinite wait

last ms ->> 7
last xs ->> ...infinite wait

reverse (reverse ms) ->> [2 ...followed by an
                                    infinite wait
head (reverse (reverse ms)) ->> 2
```

# Using the Definitions (2)

...introduced before, we also get:

```
length ns ->> 4
length ms ->> 4
length xs ->> ...infinite wait
length ys ->> ...infinite wait

length (take 4 ns) ->> 4
length (take 3 ms) ->> 3
length (take 2 xs) ->> 2
length (take 3 ys) ->> 3

length (take 5 ns) ->> 4
length (take 4 xs) ->> 4
length (take 5 xs) ->> ...infinite wait
```

# The Different Evaluation Behaviour

...of `length` and `reverse` is due to requiring or not-requiring a pattern match on the values of the argument:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs      -- No pattern match
                                   -- on the head of the
                                   -- argument list!

reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x] -- Pattern match on
                                   -- the head of the
                                   -- argument list!

reverse :: [a] -> [a]
reverse = foldl (flip (:)) []      -- Same here, even if
                                   -- pointfree defined!
```

# Chapter 6.3.3.3

## Inductive Proof Patterns for Partial Lists

# The Inductive Proof Patterns

...introduced in Chapter 6.3.2.2 and 6.3.2.3 apply to lists (possibly with undefined values), which (by definition) are built from the empty list `[]`.

By contrast, partial lists (possibly with undefined values, too) are built from the undefined list $\perp$.

We thus need to adapt the inductive proof principles for lists to work for partial lists (with possibly undefined values).

# Inductive Proofs on Partial Lists

Let $P$ be a property of partial lists.

A) Proof pattern for defined partial lists:

1. Base case: Prove that $P(\perp)$ is true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\mathtt{x}:\mathtt{xs})$ is true (induction step).

B) Proof pattern for partial lists w/ possibly undefined values:

1. Base case: Prove that $P(\perp)$ is true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\perp:\mathtt{xs})$ and $P(\mathtt{x}:\mathtt{xs})$, $\mathtt{x}$ a defined value, are true (induction step).

# Inductive Proofs on Lists and Partial Lists

Let $P$ be a property defined of lists and partial lists.

C) Proof pattern for lists and partial lists w/ possibly undefined values:

1. Base case: Prove that $P(\perp)$ and $P([\,])$ are true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\perp\!:\!\mathtt{xs})$ and $P(\mathtt{x}\!:\!\mathtt{xs})$, $\mathtt{x}$ a defined value, are true (induction step).

# Chapter 6.4

## Proving Properties of Streams

# Chapter 6.4.1

## Inductive Proofs on Haskell Stream Approximants

# Streams

...are infinite sequences of values of the same type.

## Definition 6.4.1.1 (Stream)

A stream is an infinite sequence of (defined or undefined) values of the same type.

## Definition 6.4.1.2 (Def. Stream, S. w/ Undef. Values)

A stream is called

1. defined, if all its values are defined.
2. a stream with possibly undefined values, if some of its values can be equal to $\perp$.

Homework: Does it make sense to say, a stream were built from the empty stream or the undefined stream?

# Comparing Partial Lists: Approximation Order

...intuitively, a partial list `xs` approximates a partial list `ys`, if `xs` is 'equal to but less defined' than `ys`, `xs` $\sqsubseteq$ `ys`:

$$
\begin{array}{rcl}
\bot & \sqsubseteq & 0 : \bot \\
0 : \bot & \sqsubseteq & 0 : 1 : \bot \\
0 : 1 : \bot & \sqsubseteq & 0 : 1 : 1 : \bot \\
0 : 1 : 1 : 2 : \bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : \bot \\
 & \ldots & \\
\bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot \\
0 : \bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot \\
0 : 1 : 2 : \bot & \sqsubseteq & 0 : 1 : 1 : 2 : 3 : 5 : 8 : \bot \\
 & \ldots &
\end{array}
$$

Streams can be approximated by infinite sequences of

– increasingly more accurate partial lists, called PL-approximants.

# Illustrating Stream Approximation

...the stream of natural numbers

$$\texttt{[1..]} = 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots$$

is approximated by the infinite sequence of more and more accurate PL-approximants, whose limit is the stream itself:

$$
\begin{array}{l}
\quad \perp \\
\sqsubseteq 1 : \perp \\
\sqsubseteq 1 : 2 : \perp \\
\sqsubseteq 1 : 2 : 3 : \perp \\
\sqsubseteq 1 : 2 : 3 : 4 : \perp \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : \perp \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : \perp \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : \perp \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : \perp \\
\ldots \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots = \texttt{[1..]}
\end{array}
$$

# Intuitively

...the undefined list $\perp$ is the 'least defined' partial list; hence, the 'least accurate' approximant of a stream. Sequences of more and more 'defined' approximants are getting more and more 'accurate.'

...considering partial lists (which are finite by definition)

- approximations of streams equals in spirit the approach of outputting/printing a stream prefix by interrupting the printing of the stream after some period of time by hitting `Ctrl-C`.

Extending this period of time further and further yields

- more and more accurate approximants of the stream.

# Partial Orders, Chains

...towards formalizing the idea of approximation:

## Definition 6.4.1.3 (Partially Ordered Set)

A set $M$ with a binary relation $R$ is called a partially ordered set iff $R$ is reflexive, transitive, and anti-symmetric; the pair $(M, R)$ is called a partial order, and $R$ a partial order on $M$.

## Definition 6.4.1.4 (Chain)

A subset $C \subseteq P$ of a partial order $(P, \sqsubseteq)$ is called a chain, if $C$ is totally ordered.

# Domains

## Definition 6.4.1.5 (Domain)

A partial order $(D, \sqsubseteq)$ is called a domain (or: complete partial order (CPO)), if

1. $D$ has a least element $\perp$.
2. $\bigsqcup C$ exists for every chain $C$ in $D$.

The relation $\sqsubseteq$ is then called approximation order of $(D, \sqsubseteq)$.

Example: Let $\mathcal{P}(\mathbb{N})$ be the power set of $\mathbb{N}$. Then: $(\mathcal{P}(\mathbb{N}), \sqsubseteq)$, $\sqsubseteq =_{df} \subseteq$, is a domain with:

- least element $\emptyset$
- $\bigsqcup C = \bigcup C$ for every chain $C \subseteq \mathcal{P}(\mathbb{N})$

# Approximation Order on Partial Lists, Streams

...let $S_{(PL,St)} =_{df} \{s \mid s \text{ partial list or stream}\}$ be the set of partial lists and streams.

## Lemma 6.4.1.6 (Partial Order on $S_{(PL,St)}$)

The relation $\sqsubseteq$ on $S_{(PL,St)}$ defined by:

$$\begin{array}{lcll} \bot & \sqsubseteq & \texttt{xs} & \\ \texttt{x : xs} & \sqsubseteq & \texttt{y : ys} & \Longleftrightarrow_{df} \; \texttt{x} \equiv \texttt{y} \wedge \texttt{xs} \sqsubseteq \texttt{ys} \end{array}$$

is a partial order on $S_{(PL,St)}$, where $\equiv$ denotes equality on list resp. stream entries.

## Lemma 6.4.1.7 (Domain $((S_{(PL,St)}, \sqsubseteq))$)

$(S_{(PL,St)}, \sqsubseteq)$ with $\sqsubseteq$ as in Lemma 6.4.1.6 is a domain with least element $\bot$ and approximation order $\sqsubseteq$.

# Partial Lists as Stream Approximants

## Definition 6.4.1.8 (PL-Approximants)

Let xs be a defined stream. The set of PL-approximants of xs is the set *PL-Approx*(xs) $=_{df}$ { take′ n xs | n $\in \mathbb{N}_0$ }, where

```
take' :: Integer -> [a] -> [a]
take' n _      | n <= 0  = undefined
take' n (x:xs)           = x : take' (n-1) xs
```

Note: PL-approximants are built from the undefined list, not the empty list; they all have finite length.

Examples:

– *PL-Approx*([1..]) = {⊥, 1:⊥, 1:2:⊥, 1:2:3:⊥, ...}
– *PL-Approx*([1,1..]) = {⊥,1:⊥,1:1:⊥,1:1:1:⊥,...}

# Main Result: Approximation

## Lemma 6.4.1.9 (PL-Approximants Chain)

The set *PL-Approx*(`xs`), `xs` a defined stream, is a chain.

## Theorem 6.4.1.10 (Approximation)

Let `xs` be a defined stream. Then `xs` is equal to the least upper bound of its PL-approximants set, its so-called limit:

$$\texttt{xs} = \bigsqcup \textit{PL-Approx}(\texttt{xs}) = \bigsqcup_{n=0}^{\infty} \texttt{take}' \ \texttt{n} \ \texttt{xs}$$

Note: Refer to Appendix A for the definition of technical terms and illustrating examples, if required.

# Streams as Limit of their PL-Approximants Sets

...the set of PL-approximants of a defined stream is a chain with the stream itself as its least upper bound (cf. Approximation Theorem 6.4.1.10) as illustrated below:

$$
\begin{array}{l}
\quad \bot \\
\sqsubseteq 1 : \bot \\
\sqsubseteq 1 : 2 : \bot \\
\sqsubseteq 1 : 2 : 3 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : \bot \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : \bot \\
\ldots \\
\sqsubseteq 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : \ldots = [1..]
\end{array}
$$

# Finite and Infinite Sequences of Values

...are quite diverse objects enjoying different properties.

Properties valid for lists (i.e., finite sequences) might hold or might not hold for streams (i.e., infinite sequences) and vice versa, e.g.:

- $\forall\, z \in \mathbb{Z}.$ `take n xs ++ drop n xs = xs`
  ...does hold for defined lists and streams.
- `reverse (reverse xs)) = xs`
  ...does hold for defined lists but not for streams.
- $\forall\, n \in \mathbb{N}.$ `drop n xs` $\neq$ `[]`
  ...does hold for streams but not for lists.

# Finite PL-Approximants and Streams

...are quite diverse objects, too.

Properties which are valid for every partial list of the infinite set of finite PL-approximants of a stream might hold or might not hold for its limit, the stream itself, and vice versa, e.g.:

- `map (f . g) xs = (map f . map g) xs`
  does hold for all PL-approximants of a defined stream and the stream itself.
- *'This sequence is partial'*
  ...does hold for all PL-approximants of a stream but not for the stream itself.
- `tail xs` *'is a stream'*
  ...does hold for a stream but not for any of its PL-approximants.

# Reconsidering the Induction Principles

...considered so far.

The induction principles of Chapter 6.3.2 and 6.3.3 apply to

- finite sequences of (possibly undefined) values

This allows proving properties for all finite lists and/or all finite partial lists (with possibly undefined values).

Streams, however, are by definition

- infinite sequences of values.

The induction principles of Chapter 6.3.2 and 6.3.3 are thus not directly applicable for proving properties on streams, especially in the light of the fact that properties being valid for every PL-approximant of a stream need not hold for the stream itself.

# Fortunately

...the induction principle for partial lists (with and without possibly undefined values) of Chapter 6.3.3 can be used to prove so-called (in analogy to Definition 6.4.4.1)

– admissible properties of approximant sets

for streams.

A property of a PL-approximants set is admissible if it holds for its limit, if it holds for each of its elements.

Equational properties are admissible.

Together with Approximation Theorem 6.4.1.10, this justifies the inductive proof principles considered next.

# Inductive Proofs on PL-Approximants Sets

...for proving 'admissible' properties of streams.

Let $P$ be an equational property defined on PL-approximants and streams.

A) Proof pattern for defined PL-approximants:

1. Base case: Prove that $P(\bot)$ is true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\mathtt{x}:\mathtt{xs})$ is true (induction step).

B) Proof pattern for PL-approximants w/ possibly undef. values:

1. Base case: Prove that $P(\bot)$ is true.
2. Inductive case: Assuming that $P(\mathtt{xs})$ is true (induction hypothesis), prove that $P(\bot:\mathtt{xs})$ and $P(\mathtt{x}:\mathtt{xs})$, $\mathtt{x}$ a defined value, are true (induction step).

# Example: Induction on PL-Approximants

Lecture 5
Detailed Outline
Chap. 5
Chap. 6
6.1
6.2
6.3
6.4
6.4.1
6.4.2
6.5
6.6
6.7
6.8
Final Note

## Lemma 6.4.1.11

We have:

$$(\forall\, \texttt{xs} \in \texttt{[a]}\,.\ \texttt{xs}\ \textit{defined stream})\ \forall\, \texttt{n} \in \mathbb{N}.$$
$$\texttt{take n xs ++ drop n xs = xs}$$

Proof by cases and induction on the structure of `xs`.

# Proof of Lemma 6.4.1.11 (1)

Case 1: Let $n \in \mathbb{N}$, $n = 0$, and xs be some defined stream. Equational reasoning yields the desired equality:

$$\begin{array}{rrl} & & \texttt{take n xs ++ drop n xs} \\ & = & \texttt{take 0 xs ++ drop 0 xs} \\ (\text{Def. } \texttt{take}) & = & \texttt{[] ++ xs} \\ & = & \texttt{xs} \end{array}$$

Case 2: Let $n \in \mathbb{N}$, $n \geq 1$ be some natural number. We now proceed by induction on the structure of xs.

Base case: Let $\texttt{xs} \equiv \perp$. Equational reasoning yields as desired:

$$\begin{array}{rrl} & & \texttt{take n xs ++ drop n xs} \\ & = & \texttt{take n} \perp \texttt{ ++ drop n} \perp \\ (\text{Def. } \texttt{take}, \text{ case exh.}) & = & \perp \texttt{ ++ } \perp \\ & = & \perp \\ & = & \texttt{xs} \end{array}$$

# Proof of Lemma 6.4.1.11 (2)

Inductive case: Let $\texttt{xs} \equiv (\texttt{x}:\texttt{xs}')$ be a defined PL-approximant. Then $\texttt{x}$ is defined and $\texttt{xs}'$ is a defined PL-approximant, too. By means of Case 1 (if $\texttt{n=1}$) and the induction hypothesis (IH) (if $\texttt{n}>1$), we can assume for all $n \in \mathbb{N}$ the equality $(\texttt{take (n-1) xs}' \texttt{ ++ drop (n-1) xs}') = \texttt{xs}'$. This allows us to complete the proof as follows:

$$
\begin{array}{rcl}
 & & \texttt{take n xs ++ drop n xs} \\
 & = & \texttt{take n } (\texttt{x}:\texttt{xs}') \texttt{ ++ drop n } (\texttt{x}:\texttt{xs}') \\
 & = & \texttt{x} : (\texttt{take (n-1) xs}' \texttt{ ++ drop (n-1) xs}') \\
\text{(Case 1, IH)} & = & \texttt{x} : \texttt{xs}' \\
 & = & (\texttt{x}:\texttt{xs}') \\
 & = & \texttt{xs}
\end{array}
$$

$\square$

# Chapter 6.4.2

## Inductive Proofs on Haskell List and Stream Approximants

# Approximation Order on Lists, Part. Lists, Streams

Let $S_{(L,PL,St)} =_{df} \{s \mid s \text{ list or partial list or stream}\}$ be the set of lists, partial lists and streams.

## Lemma 6.4.2.1 (Partial Order)

The relation $\sqsubseteq$ on $S_{(L,PL,St)}$ defined by:

$$
\begin{array}{lcll}
\bot & \sqsubseteq & \texttt{xs} & \\
[\,] & \sqsubseteq & \texttt{xs} & \Longleftrightarrow_{df} \; xs = [\,] \\
\texttt{x : xs} & \sqsubseteq & \texttt{y : ys} & \Longleftrightarrow_{df} \; \texttt{x} \equiv \texttt{y} \wedge \texttt{xs} \sqsubseteq \texttt{ys}
\end{array}
$$

is a partial order on $S_{(L,PL,St)}$, where $\equiv$ denotes equality on list resp. stream entries.

## Lemma 6.4.2.2 (Domain ($S_{(L,PL,St)}$))

$(S_{(L,PL,St)}, \sqsubseteq)$ with $\sqsubseteq$ as in Lemma 6.4.2.1 is a domain with least element $\bot$ and approximation order $\sqsubseteq$.

# Partial Lists as List and Stream Approximants



## Definition 6.4.2.3 (LPL-Approximants)

Let xs be a defined list or a defined stream. The set of LPL-approximants of xs is the set

$$\mathit{LPL\text{-}Approx}(\texttt{xs}) =_{df} \{\, \texttt{approx n xs} \mid \texttt{n} \in \mathbb{N}_0 \,\}$$

where

```
approx :: Integer -> [a] -> [a]
approx (n+1) []     = []
approx (n+1) (x:xs) = x : approx n xs
```

Note: There are LPL-approximants built from the undefined list and others built from the empty list; all of them are of finite length.

# Notes on approx

```
approx :: Integer -> [a] -> [a]
approx (n+1) []     = []
approx (n+1) (x:xs) = x : approx n xs
```

Pattern `n+1` matches only positive integers $\geq 1$. Thus:

1. `approx m ys ->> ys`,
   if $\texttt{m} > \texttt{len ys}$.
2. `approx m ys ->>` $\texttt{y}_0 : \texttt{y}_1 : \ldots : \texttt{y}_{m-1} : \perp$,
   if $\texttt{m} \leq \texttt{len ys}$

   (i.e., `approx` will cause an error after generating the first $m$ elements of `ys`).

Thus, `approx` being similar to `take'` used in Definition 6.4.1.8 behaves differently when applied to lists (which, by definition, are built from the empty list, not the undefined list).

# Examples: Applying approx

```
approx 0 [1,2]    ->> ⊥
approx 1 [1,2]    ->> approx (0+1) [1,2]
                  ->> 1 : approx 0 [2]
                  ->> 1 : ⊥
approx 2 [1,2]    ->> approx (1+1) [1,2]
                  ->> 1 : approx 1 [2]
                  ->> 1 : approx (0+1) [2]
                  ->> 1 : 2 : approx 0 []
                  ->> 1 : 2 :  ⊥
approx 3 [1,2]    ->> approx (2+1) [1,2]
                  ->> 1 : approx 2 [2]
                  ->> 1 : approx (1+1) [2]
                  ->> 1 : 2 : approx 1 []
                  ->> 1 : 2 : approx (0+1) []
                  ->> 1 : 2 : []
approx 7 [1,2..]  ->> 1 : 2 : 3 : 4 : 5 : 6 : 7 : ⊥
```

# Examples: PL-Approximants Sets

Lists:

$$\mathit{LPL\text{-}Approx}(\bot) = \{\bot\}$$
$$\mathit{LPL\text{-}Approx}(\texttt{[]}) = \{\bot,\ \texttt{[]}\}$$
$$\mathit{LPL\text{-}Approx}(\texttt{[1,2]}) = \{\bot, \texttt{1:}\bot, \texttt{1:2:}\bot, \texttt{1:2:[]}\}$$

Streams:

$$\mathit{LPL\text{-}Approx}(\texttt{[1..]}) = \{\bot, \texttt{1:}\bot, \texttt{1:2:}\bot, \texttt{1:2:3:}\bot, \ldots\}$$
$$\mathit{LPL\text{-}Approx}(\texttt{[1,1..]}) = \{\bot,\ \texttt{1:}\bot, \texttt{1:1:}\bot,\ \texttt{1:1:1:}\bot, \ldots\}$$

# Main Result: Approximation

## Lemma 6.4.2.4 (LPL-Approximants Chain)

The set *LPL-Approx*($\mathtt{xs}$), $\mathtt{xs}$ a defined list or a defined stream, is a chain.

## Theorem 6.4.2.5 (Approximation)

Let $\mathtt{xs}$ be a defined list or a defined stream. Then $\mathtt{xs}$ is equal to the least upper bound of its LPL-approximants set, its so-called limit:

$$\mathtt{xs} = \bigsqcup \mathit{LPL\text{-}Approx}(\mathtt{xs}) = \bigsqcup_{\mathtt{n}=0}^{\infty} \mathit{approx}\ \mathtt{n}\ \mathtt{xs}$$

# Proof Sketch of Theorem 6.4.2.5 for Lists

Let $\texttt{xs} \equiv (\texttt{x}_0 : \texttt{x}_1 : \texttt{x}_2 : \ldots : \texttt{x}_{\texttt{len(xs)-1}} : [\,])$ be a defined list.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\bigsqcup_{\texttt{n}=0}^{\infty} \mathit{approx}\ \texttt{n}\ \texttt{xs}} \\
= & \bigsqcup \{ \bot, & (\texttt{n = 0}) \\
& \quad \texttt{x}_0 : \bot, & (\texttt{n = 1}) \\
& \quad \texttt{x}_0 : \texttt{x}_1 : \bot, & (\texttt{n = 2}) \\
& \quad \ldots & \\
& \quad \texttt{x}_0 : \texttt{x}_1 : \ldots : \texttt{x}_{\texttt{n-1}} : \bot, & (\texttt{n = len(xs)}) \\
& \quad \texttt{x}_0 : \texttt{x}_1 : \ldots : \texttt{x}_{\texttt{n-1}} : [\,], & (\texttt{n = len(xs)+1}) \\
& \quad \texttt{x}_0 : \texttt{x}_1 : \ldots : \texttt{x}_{\texttt{n-1}} : [\,], & (\texttt{n = len(xs)+2}) \\
& \quad \ldots & \\
& \ \} & \\
= & \texttt{x}_0 : \texttt{x}_1 : \texttt{x}_2 : \ldots : \texttt{x}_{\texttt{n-1}} : [\,] & \\
= & \texttt{x}_0 : \texttt{x}_1 : \texttt{x}_2 : \ldots : \texttt{x}_{\texttt{len(xs)-1}} : [\,] & \\
\equiv & \texttt{xs} &
\end{array}
$$

# Proof Sketch of Theorem 6.4.2.5 for Streams

Let $\texttt{xs} \equiv (\texttt{x}_0 : \texttt{x}_1 : \texttt{x}_2 : \ldots : \texttt{x}_n : \ldots)$ be a defined stream.

$$
\begin{array}{lll}
\bigsqcup_{n=0}^{\infty} \mathit{approx}\ \texttt{n}\ \texttt{xs} & & \\
\quad = & \bigsqcup \{ \bot, & (\texttt{n = 0}) \\
 & \quad \texttt{x}_0 : \bot, & (\texttt{n = 1}) \\
 & \quad \texttt{x}_0 : \texttt{x}_1 : \bot, & (\texttt{n = 2}) \\
 & \quad \ldots & \\
 & \quad \texttt{x}_0 : \texttt{x}_1 : \ldots : \texttt{x}_{m-1} : \bot, & (\texttt{n = m}) \\
 & \quad \texttt{x}_0 : \texttt{x}_1 : \ldots : \texttt{x}_{m} : \bot, & (\texttt{n = m+1}) \\
 & \quad \texttt{x}_0 : \texttt{x}_1 : \ldots : \texttt{x}_{m+1} : \bot, & (\texttt{n = m+2}) \\
 & \quad \ldots & \\
 & \} & \\
\quad = & \texttt{x}_0 : \texttt{x}_1 : \texttt{x}_2 : \ldots : \texttt{x}_n : \ldots & \\
\quad \equiv & \texttt{xs} &
\end{array}
$$

# Inductive Proofs on LPL-Approximants Sets

...for proving 'admissible' properties of streams.

Let $P$ be an equational property defined on LPL-approximants and streams.

A) Proof pattern for defined LPL-approximants:

1. Base case: Prove that $P(\bot)$ and $P(\texttt{[]})$ are true.
2. Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\texttt{x:xs})$ is true (induction step).

B) Proof pattern for LPL-approximants w/ possibly undefined values:

1. Base case: Prove that $P(\bot)$ and $P(\texttt{[]})$ are true.
2. Inductive case: Assuming that $P(\texttt{xs})$ is true (induction hypothesis), prove that $P(\bot\texttt{:xs})$ and $P(\texttt{x:xs})$, $\texttt{x}$ a defined value, are true (induction step).

# Chapter 6.5

## Proving Equality of Streams

# Chapter 6.5.1

## Approximation

# Approximants, Approximants Sets

...allow to reduce proving the equality of streams to proving

1. the equality of sets (cf. Approximation Theorem 6.5.1.7)
2. equivalent statements amenable to mathematical induction (cf. Approximation Theorem 6.5.1.8)

and provide this way an important proof principle for proving the equality of streams.

# L-Approximants of Defined Lists and Streams

## Definition 6.5.1.1 (L-Approximants)

Let xs be a defined list or a defined stream. The set of L-approximants of xs is the set

$$L\text{-}Approx(\texttt{xs}) =_{df} \{\, \texttt{take n xs} \mid \texttt{n} \in \mathbb{N}_0 \,\},$$ where

```
take :: Int -> [a] -> [a]
take n _      | n <= 0 = []
take _ []              = []
take n (x:xs)          = x : take (n-1) xs
```

Note, L-approximants are built from the empty list (not the undefined list); every L-approximant is finite.

Examples:

- $L\text{-}Approx(\texttt{[]}) = \{\,\texttt{[]}\,\}$
- $L\text{-}Approx(\texttt{[1,2,3]}) = \{\,\texttt{[]},\texttt{1:[]},\texttt{1:2:[]},\texttt{1:2:3:[]}\,\}$
- $L\text{-}Approx(\texttt{[1..]}) = \{\,\texttt{[]},\texttt{1:[]},\texttt{1:2:[]},\texttt{1:2:3:[]},\ldots\,\}$

# Finiteness, Infinity of Sequences

...in terms of L-approximants sets.

## Definition 6.5.1.2 (Finite, Infinite Sequences)

A sequence of defined values `xs` is

1. finite (i.e., a list), if *L–Approx*(`xs`) is finite.
2. infinite (i.e., a stream), if *L–Approx*(`xs`) is infinite.

## Lemma 6.5.1.3 (Finite, Infinite Sequences)

A sequence of defined values `xs` is

1. finite, if: $\exists\, m \in \mathbb{N}.\, (\forall\, n \in \mathbb{N}.\, n \geq m).$ `take n xs = take (n+1) xs`
2. infinite, if: $\forall\, n \in \mathbb{N}.$ `take n xs /= take (n+1) xs`

## Corollary 6.5.1.4 (Finite Sequences)

A sequence of defined values `xs` is finite, if:

$\exists\, m \in \mathbb{N}.\, (\forall\, n \in \mathbb{N}.\, n \geq m).$ `take m xs = take (n+1) xs`

# Equality of Sequences and Streams

...in terms of L-approximant sets.

## Definition 6.5.1.5 (Equality of Sequences)

Let `xs` and `ys` be two sequences of defined values. `xs` and `ys` are equal, if their sets of L-approximants are equal:

$$\begin{aligned} \textit{L-Approx}(\texttt{xs}) &= \{\, \texttt{take n xs} \mid \texttt{n} \in \mathbb{N} \,\} \\ &= \{\, \texttt{take n ys} \mid \texttt{n} \in \mathbb{N} \,\} = \textit{L-Approx}(\texttt{ys}) \end{aligned}$$

## Lemma 6.5.1.6 (Equality of Sequences)

Let `xs` and `ys` be two sequences of defined values. `xs` and `ys` are equal, if for every natural number their L-approximants are equal:

$$\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{take n xs = take n ys}$$

# Main Results: Stream Equality by Set Equality

...reducing the proof of stream equality to proving set equality.

## Theorem 6.5.1.7 (Stream Equality, Approximation 1)

Let $\mathtt{xs}, \mathtt{ys}$ be defined streams. Then the following statements are equivalent:

1. $xs = ys$
2. $\mathit{LPL\text{-}Approx}(\mathtt{xs}) = \mathit{LPL\text{-}Approx}(\mathtt{ys})$
3. $\mathit{PL\text{-}Approx}(\mathtt{xs}) = \mathit{PL\text{-}Approx}(\mathtt{ys})$
4. $\mathit{L\text{-}Approx}(\mathtt{xs}) = \mathit{L\text{-}Approx}(\mathtt{ys})$

# Main Results: Stream Equality by Nat. Induct.

...reducing the proof of stream equality to an equivalent statement amenable to natural (or: mathematical) induction.

## Theorem 6.5.1.8 (Stream Equality, Approximation 2)

Let xs, ys be defined streams. Then the following statements are equivalent:

1. $\texttt{xs} = \texttt{ys}$
2. $\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{approx n xs} = \texttt{approx n ys}$
3. $\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{take}'\ \texttt{n xs} = \texttt{take}'\ \texttt{n ys}$
4. $\forall\, \texttt{n} \in \mathbb{N}.\ \texttt{take n xs} = \texttt{take n ys}$
5. $\forall\, \texttt{n} \in \mathbb{N}_0.\ \texttt{xs!!n} = \texttt{ys!!n}$

Note: Proving stream equality is usually technically more convenient using Theorem 6.5.1.8(5) than any of the statements of Theorem 6.5.1.7.

# Example: Applying Theorem 6.5.1.8

Consider the factorial function:

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

and the two stream functions `facs_mp` and `facs_zw`:

```
facs_mp = map fac [0..]
facs_zw = 1 : zipWith (*) [1..] facs_zw
```

which generate the stream of factorials: `1,1,2,6,24,...`

According to Theorem 6.5.1.8(5), proving the equality of `facs_mp` and `facs_zw` boils down to proving Lemma 6.5.1.9, which we prove by natural induction:

## Lemma 6.5.1.9

$$\forall\, n \in \mathbb{N}_0.\ \texttt{facs\_mp!!n} = \texttt{facs\_zw!!n}$$

# Proof by Lemma 6.5.1.9 (1)

Base case: Let n=0. Equational reasoning yields the desired equality in this case:

$$
\begin{array}{rcl}
 & & \texttt{facs\_mp!!n} \\
(n = 0) & = & \texttt{facs\_mp!!0} \\
(\text{Def. }\texttt{facs\_mp}) & = & \texttt{(map fac [0..])!!0} \\
(\text{L. 6.5.1.10(1)}) & = & \texttt{fac ([0..]!!0)} \\
 & = & \texttt{fac 0} \\
(\text{Def. }\texttt{fac}) & = & \texttt{1} \\
(\text{Def. }\texttt{(!!)}) & = & \texttt{(1 : zipWith (*) [1..] facs\_zw)!!0} \\
(\text{Def. }\texttt{facs\_zw}) & = & \texttt{facs\_zw!!0} \\
(n = 0) & = & \texttt{facs\_zw!!n}
\end{array}
$$

# Proof by Lemma 6.5.1.9 (2)

Inductive case: Let $\mathtt{n} \in \mathbb{N}_0$. By means of the induction hypothesis (IH), we can assume `facs_mp!!n = facs_zw!!n`. As desired we get:

| | | |
|---|---|---|
| | | `facs_mp!!(n+1)` |
| (Def. `facs_mp`) | = | `(map fac [0..])!!(n+1)` |
| (L. 6.5.1.10(1)) | = | `fac ([0..]!!(n+1))` |
| (Def. `[0..]`, `(!!)`) | = | `fac (n+1)` |
| (Def. `fac`) | = | `(n+1) * fac n` |
| (L. 6.5.1.10(3)) | = | `(n+1) * (facs_mp!!n)` |
| (IH) | = | `(n+1) * (facs_zw!!n)` |
| (Def. `(!!)`) | = | `([1..]!!n) * (facs_zw!!n)` |
| (Def. `(*)`) | = | `(*) ([1..]!!n) (facs_zw!!n)` |
| (L. 6.5.1.10(2)) | = | `(zipWith (*) [1..] facs_zw)!!n` |
| (Def. `(!!)`) | = | `(1 : zipWith (*) [1..] facs_zw)!!(n+1)` |
| (Def. `facs_zw`) | = | `facs_zw!!(n+1)` |

# Supporting Statement

## Lemma 6.5.1.10

For all natural numbers $n \in \mathbb{N}_0$, we have:

1. `(map f xs)!!n = f (xs!!n)`
2. `(zipWith g xs ys)!!n = g (xs!!n) (ys!!n)`
3. `fac n = facs_mp!!n`

*Proof.* Homework.

# Chapter 6.5.2

## Coinduction

# Proof by Coinduction

...another useful principle for proving equality of infinite objects such as streams which

- complements the principle of proof by approximation of Chapter 6.5.1.
- reduces proving equality of two objects to proving they exhibit the same 'observational behaviour.'

For streams, this boils down to proving

- the heads of the streams are the same.
- their tails exhibit the same 'observational behaviour.'

# Equality of Streams

...let

- $[A]$ denote the set of streams over a set of elements $A$.
- $f, g \in [A]$ be written as $f = [f_0, f_1, f_2, f_3, f_4, f_5, \ldots]$ and $g = [g_0, g_1, g_2, g_3, g_4, g_5, \ldots]$, respectively.

## Definition 6.5.2.1 (Equality of Streams)

$f, g \in [A]$ are equal iff $\forall\, i \in \mathbb{N}_0.\ f_i = g_i$, i.e., $f$ and $g$ have the same 'observational behaviour.'

...in accordance with Theorem 6.5.1.8.

# Reducing Equality of Streams

...to their bisimilarity.

This requires to introduce:

- Labelled transition systems (LTS) for stream representation
- Stream bisimulation relations for capturing the notion of 'same' behaviour of streams

and some supporting notions:

- Expansions of LTS states
- Bisimilar states

# Labelled Transition Systems

### Definition 6.5.2.2 (Labeled Transition System)

A labelled transition system (LTS) is a tripel $(Q, A, T)$ with

- $Q$ a set of states.
- $A$ a set of action labels.
- $T \subseteq Q \times A \times Q$ a ternary transition relation.

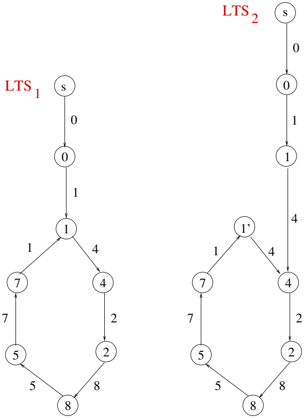
Note: For $(q, a, p) \in T$ we write more conveniently: $q \xrightarrow{a} p$.

# Example: Representing Streams as LTSs

The decimal representation of $\frac{1}{7}$ has numerous representations as streams of digits, e.g.:

- $0.\overline{142857}$, $0.1\overline{428571}$, $0.14\overline{285714}$, $0.142857142\overline{857142}$,...

$LTS_1$, $LTS_2$ are LTS representations of the 2nd and 3rd one:



# Expansion of LTS States

Let $(Q, A, T)$ be an LTS, and $q \in Q$.

## Definition 6.5.2.3 (Expansion of an LTS State)

1. A finite expansion of $q$ is a finite sequence of actions $[a_0, a_1, a_2, a_3, ..., a_n]$ such that

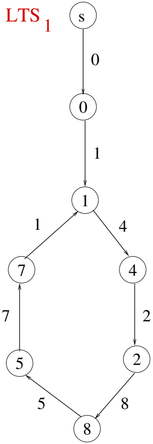
$$(\forall\, i \in \mathbb{N}_0.\ i \leq n).\ \exists\, q_i, q_{i+1} \in Q.\ q_0 = q \wedge q_i \xrightarrow{a_i} q_{i+1}.$$

2. An infinite expansion of $q$ is an infinite sequence of actions $[a_0, a_1, a_2, a_3, ...]$ such that

$$\forall\, i \in \mathbb{N}_0.\ \exists\, q_i, q_{i+1} \in Q.\ q_0 = q \wedge q_i \xrightarrow{a_i} q_{i+1}.$$

# Example: Expansion of Digit Stream States

Consider $LTS_1$ representing digit stream $0.1\overline{428571}$:




The (unique) infinite expansion of state (i.e., node)

– s is $01\overline{428571}$, 0 is $1\overline{428571}$, 1 is $\overline{428571}$, 2 is $\overline{857142}$,...

# Bisimulation Relations, Bisimilar States

Let $(Q, A, T)$ be an LTS, let $p, q \in Q$.

## Definition 6.5.2.4 ((Greatest) Bisimulation Relation)

A bisimulation on $(Q, A, T)$ is a binary relation $R$ on $Q$, which satisfies: If $q \, R \, p$ and $a \in A$ then:

- $q \xrightarrow{a} q' \;\Rightarrow\; \exists\, p' \in Q.\; p \xrightarrow{a} p' \;\wedge\; q' \, R \, p'$
- $p \xrightarrow{a} p' \;\Rightarrow\; \exists\, q' \in Q.\; q \xrightarrow{a} q' \;\wedge\; q' \, R \, p'$

The largest bisimulation on $Q$ (wrt $\subseteq$) is denoted by $\sim$.

## Definition 6.5.2.5 (Bisimilar States)

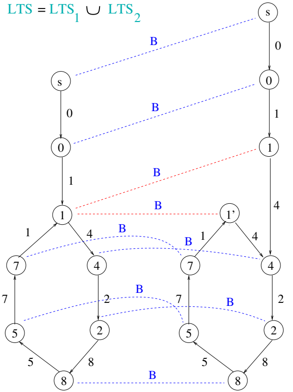
$p$ and $q$ are called bisimilar, if there is a bisimulation $R$ on $Q$ with $q \, R \, p$.

# Example: A Bisimulation for Digit Streams

Consider $LTS = (Q, A, T)$ defined as union of $LTS_1$, $LTS_2$.

We define relation $B$ on $Q$ as follows:

$\forall\, q, q' \in Q.\ q\, B\, q'$ iff $q$, $q'$ have the same infinite expansion




Note: $B$ is the largest bisimulation on $Q$, i.e.: $B = \sim$.

# Streams as Labeled Transition Systems

...for $f = [f_0, f_1, f_2, f_3, f_4, \ldots] \in [A]$ a stream, let

- $f_0$ denote the head
- $\bar{f}$ denote the tail

of $f$, i.e., $f = f_0 : \bar{f}$.

Using this notation, $f$ is represented by the below labelled transition system (which unfolds $f$ partially):

$$f \quad \bullet \xrightarrow{\quad f_0 \quad} \bullet \quad \overline{f}$$

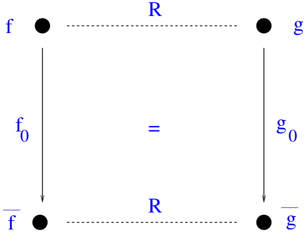
LTS representation of $f$

# Stream Bisimulation

## Definition 6.5.2.6 (Stream Bisimulation)

A stream bisimulation on $[A]$ is a binary relation $R$ on the set of streams $[A]$, which satisfies:

$$\forall f, g \in [A].\ f\, R\, g \ \Rightarrow\ f_0 = g_0 \wedge \bar{f}\, R\, \bar{g}$$




Let $\sim$ denote the largest stream bisimulation on $[A]$.

# Reducing Stream Equality

...to largest stream bisimulation.

Let $f = [f_0, f_1, f_2, f_3, f_4, \ldots]$, $g = [g_0, g_1, g_2, g_3, g_4, \ldots] \in [A]$ be two streams with

$$f \xrightarrow{f_0} \bar{f}, \quad g \xrightarrow{g_0} \bar{g}.$$

Then:

**Theorem 6.5.2.7 (Stream Equality as Stream Bisim.)**

$f$ and $g$ are equal iff $f \sim g$ (i.e., $f_0 = g_0$ and $\bar{f} \sim \bar{g}$).

# Reducing Stream Equality

...further to stream bisimulation.

Since $\sim$ is the largest stream bisimulation, we get:

## Lemma 6.5.2.8

$$f \sim g \;\Leftrightarrow\; \exists\, B.\ B \text{ stream bisimulation on } [A] \;\wedge\; f\, B\, g$$

Together, Theorem 6.4.3.7 and Lemma 6.4.3.8 imply:

## Corollary 6.5.2.9

$f$ and $g$ are equal iff

$$\exists\, B.\ B \text{ stream bisimulation on } [A] \;\wedge\; f\, B\, g$$

# The Coinductive Proof Pattern

...using Corollary 6.5.2.9, proving the equality of two streams $f$ and $g$ of $[A]$ requires:

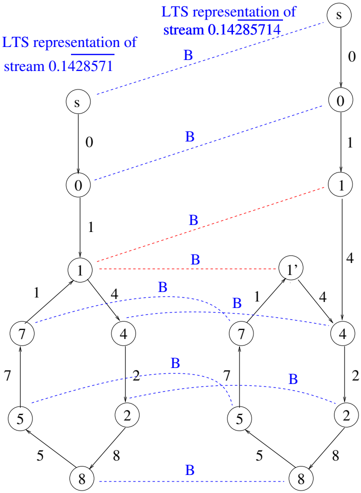
1. Finding a relation $B$ on $[A]$.
2. Proving that $B$ is a stream bisimulation and $f\,B\,g$.

...considering Haskell streams, this means proving the equality of two Haskell streams `xs` and `ys` requires:

1. Finding a relation $B$ on the set of Haskell streams.
2. Proving that $B$ is a stream bisimulation and `xs` $B$ `ys`.

# Example: Stream Bisimulation $B \subseteq \sim$

...for the two streams $0.1\overline{428571}$ and $0.14\overline{285714}$:




...$0.1\overline{428571}$, $0.14\overline{285714}$ are stream bisimilar and hence equal.

# Chapter 6.6

## Fixed Point Induction

# Fixed Point Induction

...a useful proof principle allowing us to prove properties of the

- least fixed point of continuous functions

on complete partial orders or stronger complete lattices, which are both specific partially ordered sets (refer to Appendix A for definitions of terms, if required).

# Admissible Predicates

Let $(C, \sqsubseteq)$ be a complete partial order (CPO) (or :domain), and $\psi$ be a predicate on $C$, i.e., $\psi : C \to I\!B$.

## Definition 6.6.1 (Admissible Predicate)

$\psi$ is called admissible iff for every chain $D \subseteq C$ holds:

$$(\forall\, d \in D.\ \psi(d)) \;\Rightarrow\; \psi(\bigsqcup D)$$

## Lemma 6.6.2

$\psi$ is admissible, if it is expressible as an equation.

# Example: Streams, Sequences of Approximants

Recalling that $(S_{(PL,St)}, \sqsubseteq)$ with

- $S_{(PL,St)}$: Set of partial lists and streams
- $\sqsubseteq$: Approximation order on $S_{(PL,St)}$ (cf. Lemma 6.4.1.6)

is a domain (or: complete partial order) (cf. Lemma 6.4.1.7), we get:

## Corollary 6.6.3

Let $\psi$ be a predicate on the set of partial lists and streams $S_{(PL,St)}$ expressible as an equation, let $s$ be a stream, and $S' \subseteq S$ the infinite chain of its PL-approximants (cf. Definition 6.4.1.8) with $\bigsqcup S' = s$. Then:

$$(\forall\, s' \in S'.\ \psi(s')) \;\Rightarrow\; \psi(\bigsqcup S') \quad (\Leftrightarrow \psi(s))$$

# Monotonic and Continuous Functions on CPOs

Let $(C, \sqsubseteq_C)$ and $(D, \sqsubseteq_D)$ be CPOs, and let $f \in [C \to D]$ be a map from $C$ to $D$.

## Definition 6.6.4 (Monotonic, Continuous Maps)

$f$ is called

1. monotonic (or: order preserving) iff
$$\forall\, c, c' \in C.\; c \sqsubseteq_C c' \Rightarrow f(c) \sqsubseteq_D f(c')$$
(Preservation of the ordering of elements)
2. continuous iff $f$ is monotonic and
$$(\forall\, C' \subseteq C.\; C' \neq \emptyset \wedge C' \; chain).\; f(\bigsqcup_C C') =_D \bigsqcup_D f(C')$$
(Preservation of least upper bounds)

# Fixed Points, Least Fixed Points

...of continuous functions on complete partial orders (CPOs).

## Definition 6.6.5 (Fixed Point, Least Fixed Point)

Let $(C, \sqsubseteq)$ be a complete partial order, let $f \in [C \stackrel{con}{\to} C]$ be a continuous function on $C$, and let $c \in C$ be an element of $C$. Then:

1. $c$ is a fixed point of $f$ iff $f(c) = c$.
2. $c$ is the least fixed point of $f$, denoted by $\mu f$,
   iff $\forall\, d \in C.\ f(d) = d \Rightarrow c \sqsubseteq d$

Note: The Fixed Point Theorem A.5.1.3 of Knaster, Tarski, and Kleene ensures the existence of least fixed points of continuous functions on CPOs.

# Fixed Point Induction

...the general pattern of fixed point induction:

## Theorem 6.6.6 (Fixed Point Induction)

Let $(C, \sqsubseteq)$ be a complete partial order (CPO), let $f : C \to C$ be a continuous function on $C$, and let $\psi : C \to I\!B$ be an admissible predicate on $C$. Then:

$$(\forall\, c \in C.\ \psi(c) \ \Rightarrow\ \psi(f(c))) \ \Rightarrow\ \psi(\mu f)$$

where $\mu f$ denotes the least fixed point of $f$.

# Chapter 6.7

## Verified Programming, Verification Tools

# Chapter 6.7.1

## Correctness by Construction

# Correctness by Construction

...strives for ensuring correctness of a program on the fly of developing it by proving the result of every step of the development process correct.

Conceptually, correctness by construction is an

- *a priori* (or: *on-the-fly*) approach.

This is dual to testing and verification, which conceptually are

- *a posteriori* approaches

as they are applied to a program after its development is finished.

# Techniques for Correctness by Construction

...in principle, every proof technique can be made use of by approaches aiming at correctness by construction, among these

- (inductive) proof principles (cf. Chapter 6)
- equational reasoning, sometimes also called proof by program calculation (cf. Chapter 4).

Particularly important, however, are approaches based on

- transformation rules

which are proven correct and ensure equivalence of the program they are applied to and the one resulting from them.

# Example: Functional Pearls

...developing functional pearls starting with a program being

- obviously correct (but usually inefficient)

by a sequence of transformation steps into a program being

- still correct and (hopefully) more efficient

where (ideally) every transformation step is proved correct (cp. Chapter 4), can be considered an approach in the spirit of

- correctness by construction.

# Chapter 6.7.2

## Provers, Proof-Assistents, Verified Programming

# Provers, Proof-Assistents, Verified Prog. (1)

Provers, proof-assistents for verifying

- Equational properties of functional programs (Sonnex et al., TACAS 2012)
  - Tool Zeno: Proof search is based on induction and equality reasoning which are driven by syntactic heuristics.
- First-order and call-by-value recursive functional programs (Suter et al., SAS 2011)
  - Tool Leon: Based on extending SMT to recursive programs.
- Higher-order functional programs (Unno et al., POPL 2013)
  - Tool MoCHi-X: Prototype implementation of a type inference algorithm as extension of the software model checker MoChi (Kobayashi et al., PLDI 2011).

# Provers, Proof-Assistents, Verified Prog. (2)

- Lazy Haskell (Mitchell et al., Haskell 2008)
  - Tool Catch: Based on static analysis; can prove absence of pattern matching failures; evaluated on 'real' programs.
- ...

Language integrated approaches:

- Programming by contracts (Vytiniotis et al., POPL 2013)
- Verified functional programming in Agda (see next slide)
- ...

# Verified Functional Programming in Agda

Aaron Stump. Verified Functional Programming in Agda. ACM Books Series, No. 9, 2016.

...a text snippet from the book:

'Agda is an advanced programming language based on Type Theory. Agda's type system is expressive enough to support full functional verification of programs, in two styles.

In external verification, we write pure functional programs and then write proofs of properties about them. The proofs are separate external artifacts, typically using structural induction.

In internal verification, we specify properties of programs through rich types for the programs themselves. This often necessitates including proofs inside code, to show the type checker that the specified properties hold.

The power to prove properties of programs in these two styles is a profound addition to the practice of programming, giving programmers the power to guarantee the absence of bugs, and thus improve the quality of software more than previously possible.'

# Chapter 6.8

## References, Further Reading

# Chapter 6: Basic Reading (1)


- Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Chapter 6, Proofs)
- Richard Bird, Philip Wadler. *An Introduction to Functional Programming*. Prentice Hall, 1988. (Chapter 5, Induction and Recursion)
- Kees Doets, Jan van Eijck. *The Haskell Road to Logic, Maths and Programming*. Texts in Computing, Vol. 4, King's College, UK, 2004. (Chapter 3, The Use of Logic: Proof – Proof Style, Proof Recipes, Strategic (Proof) Guidelines; Chapter 7, Induction and Recursion; Chapter 10, Corecursion; Chapter 10.3, Proof by Approximation; Chapter 10.4, Proof by Coinduction; Chapter 11.1, More on Mathematical Induction)

# Chapter 6: Basic Reading (2)


- Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Chapter 11, Proof by Induction; Chapter 14.6, Inductive Properties of Infinite Lists)
- Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2nd edition, 2016. (Chapter 16, Reasoning about programs)
- David Makinson. *Sets, Logic and Maths for Computing*. Springer-V., 2008. (Chapter 4, Recycling Outputs as Inputs: Induction and Recursion; Chapter 4.1, What are Induction and Recursion? Chapter 4.6, Structural Recursion and Induction; Chapter 4.7, Recursion and Induction on Well-Founded Sets)

# Chapter 6: Basic Reading (3)


- Bernhard Steffen, Oliver Rüthing, Michael Huth. *Mathematical Foundations of Advanced Informatics: Inductive Approaches*. Springer-V., 2018. (Chapter 4, Inductive Definitions; Chapter 5, Inductive Proofs; Chapter 6, Inductive Approach: Potential, Limitations, and Pragmatics)
- Simon Thompson. *Haskell – The Craft of Functional Programming*. Addison-Wesley/Pearson, 3rd edition, 2011. (Chapter 9, Reasoning about programs; Chapter 17.9, Proof revisited)
- Antonie J.T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Chapter 9, Correctness)

# Chapter 6: Selected Further Reading (1)


- André Arnold, Irène Guessarian. *Mathematics for Computer Science*. Prentice Hall, 1996. (For inductive proof principles in general)
- Roderick Chapman. *Correctness by Construction: A Manifesto for High Integrity Software*. In Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, Vol. 55, 43-46, 2006.
- Henning Dierks, Michael Schenke. *A Unifying Framework for Correct Program Construction*. In Proceedings of the 4th International Conference on the Mathematics of Program Construction (MPC'98). Springer-V., LNCS 1422, 122-150, 1998.
- Charles A.R. Hoare. *The Ideal of Program Correctness*. The Computer Journal 50(3):254-260, 2007.

# Chapter 6: Selected Further Reading (2)


- Bart Jacobs, Jan Rutten. *A Tutorial on (Co)algebras and (Co)induction*. EATCS Bulletin 62:222-259, 1997.
- Steve King, Jonathan Hammond, Roderick Chapman, Andy Pryor. *Is Proof More Cost-Effective than Testing?* IEEE Transactions on Software Engineering 26(8):675-686, 2000.
- Derrick G. Kourie, Bruce W. Watson. *The Correctness-by-Construction Approach to Programming*. Springer-V., 2012.
- Robin Milner. *Communications and Concurrency*. Prentice Hall, 1989. (Chapter 4 for an introduction to coinductive proofs.)
- Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

# Chapter 6: Selected Further Reading (3)


- Aaron Stump. *Verified Functional Programming in Agda*. ACM Books Series, No. 9, 2016.
- Daniel J. Velleman. *How to Prove It. A Structured Approach*. Cambridge University Press, 3rd edition, 2019.
- Mitchell Wand. *Induction, Recursion, and Programming*. Elsevier, 1980.
- Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993. (Chapter 1, Basic set theory; Chapter 3, Some principles of induction; Chapter 4, Inductive definitions; Chapter 8, Introduction to domain theory; Chapter 8.2, Streams – an example; Chapter 10.2, Fixed-point induction)

# Final Note

...for additional information and details refer to

- full course notes

available at the homepage of the course at:

http:://www.complang.tuwien.ac.at/knoop/ffp185A05_ss2020.html