FPV-Tutorial - SS23

Materialien für Manuel's FPV-Tutorium im Sommersemester 2023

Manuel Lerchner

Zuletzt aktualisiert: 31. Mai 2023

FPV Tutorial - SS23

About

Materialien für Manuel's FPV-Tutorium im Sommersemester 2023.

- **Die Materialien sind privat erstellt und können Fehler enthalten. Im Zweifelsfall haben immer die *offiziellen* Lehrunterlagen Vorrang.**
- Alle Zusammenfassungen dieses Repositories können über manuellerchner.github.io/fpv-tutorial-SS23/summary.pdf heruntergeladen werden.
- Die Tutor-Slides sind unter manuellerchner.github.io/fpv-tutorial-SS23/slides.pdf verfügbar.

Found an error, or want to add something?

1. Fork this Repository
2. Commit and push your changes to **your** forked repository
3. Open a Pull Request to this repository
4. Wait until the changes are merged
5. A Github Action will automatically render the PDFs and deploy the static content to Github Pages

Contributors

Inhaltsverzeichnis

FPV Tutorial - SS23 1
About 1
Found an error, or want to add something? 1
Contributors 1

Week 1: Implications, Assertions and Stronges Postconditions **4**
Implications 4
Definition of Implications 4
Truth Table 4
Examples 4
Assertions 5
Example for MiniJava 5
Strength of Assertions 6
Definition of Assertions-Strength 6
Special Assertions 6
Strongest Postconditions 6
Example 6

Week 2: Preconditions, Postconditions and Local Consistency **8**
Weakest Preconditions 8
Example 8
Local Consistency 8
Example Local Consistency 9

Week 3: Loop Invariants **11**
What is a loop invariant? 11
The Problem with Loops 11
Finding a Loop Invariant 11
Evaluating different Loop Invariants 12
Example Loop Invariants 12
Tips for Finding a Loop Invariant 12

Week 4: Termination Proofs **13**
Why are Termination Proofs Necessary? 13
What is a Termination Proof? 13
How to do a Termination Proof? 13

OCaml **15**
Basic Syntax 15
Comments 15
Variables 15
Functions 15
Tuples 15
Lists 15
Records 15
If-then-else 15
Pattern Matching 15
Example Programs 16

Advanced Hello World 16
Debuging OCaml 16
Using the `#use` command in `utop` 16

Ocaml Exercises **17**
List Module 17
Lists in OCaml 17
Binary Search Tree 17
Binary Search Tree in OCaml 17

Week 1: Implications, Assertions and Stronges Postconditions

Implications

Implications are the the key for understanding FPV. They show up in topics such as *Weakest Preconditions*, *Strongest Postconditions*, *Proof by Induction / Structural Induction…*

Definition of Implications

As you remeber from the "Diskrete Strukturen" course, an implication is a statement of the form $A \implies B$. It is read as:

- "A implies B"
- "If A is true, then B is true"

It's syntactic sugar for the following statement:

$$A \implies B \iff \neg A \lor B$$

This is a very important statement, because it can be used to simplify complex statements, if you can't remember the specific rules for implications.

Truth Table

A	B	$A \implies B$
F	F	T
F	T	T
T	F	F
T	T	T

Examples

Example 1:

$$\begin{aligned} & x = 1 \implies x \geq 0 \\ \iff & \neg(x = 1) \lor (x \geq 0) \\ \iff & (x \neq 1) \lor (x \geq 0) \\ \iff & \mathit{true} \end{aligned}$$

Example 2:

$$
\begin{aligned}
& A \implies (B \implies A) \\
\iff & \neg A \vee (B \implies A) \\
\iff & \neg A \vee (\neg B \vee A) \\
\iff & \neg A \vee A \vee \neg B \\
\iff & \mathit{true} \vee \neg B \\
\iff & \mathit{true}
\end{aligned}
$$

Assertions

Assertions are used to **annotate** specific points in a program and to **check** if a given expression is true at that point. If the expression is false, the program will terminate.

This is usefull if you only want to allow certain values for a variable, because otherwise the program would not work as expected. They can also be used to prove the correctness of a program. Which is the main topic of this course.

Example for MiniJava

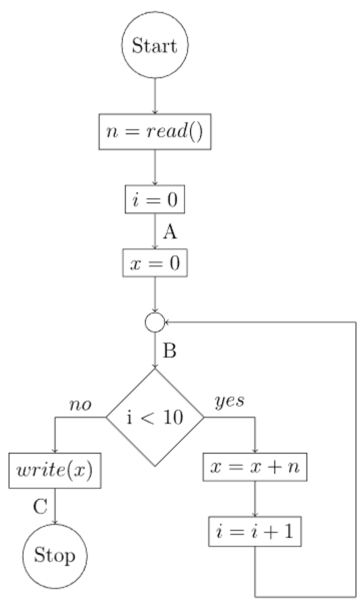

Abbildung 1: Flow Diagram

This corresponds to the following program:

```
void main() {
    var n = read(); //reads an arbitrary integer
    var i = 0;
    assert(A);

    var x = 0;
    assert(B)
    while (i < 10) {
        x = x + n;
        i = i + 1;
        assert(B);
    };

    write(x);
```

```
    assert(C);
}
```

The challenge is to find strong and precise assertions for the specific points in the program which allow us to prove the correctness of the program. In this case, we want to prove that the program **always** prints $n \cdot 10$ to the console. This corresponds to Assertion $C \iff x = n \cdot 10$.

Remember that whenever the programm-flow reaches an assertion, the assertion must be true. Otherwise the program will terminate.

Strength of Assertions

Two assertions A and B can have different strengths. This happens for example if assertion A is more precise than assertion B.

For example, the assertion $A = 5$ is stronger than the assertion $B = 5 \vee 6$, because it is more specific. The assertion A only allows the value 5, while the assertion B allows the values 5 and 6.

This makes sense intuitively. But in order to use it in practice, we need to define what it means for an assertion to be stronger than another assertion.

Definition of Assertions-Strength

We say that an assertion A is stronger than an assertion B, if A implies B.

Using this definition, we can compare different assertions and determine if they are:

- **Equivalent**: $A \implies B$ and $B \implies A$
- **Ordered (eg. A is stronger)**: $A \implies B$
- **Uncomparable**: $A \not\implies B$ and $B \not\implies A$

Special Assertions

Remember that $true$ and $false$ are also valid assertions. They are called **tautologies** and **contradictions** respectively.

How do they fit into the strength definition?

- **Tautologies**: $A \implies true$ for all A
  - This means that every assertion is stronger than $true$ thereby making $true$ the weakest assertion.
- **Contradictions**: $false \implies A$ for all A
  - This means that $false$ is stronger than every assertion thereby making $false$ the strongest assertion.

In practice those assertions show up in the following cases:

- **Tautologies**: If you have no information about the variables at a specific time in the program, you can use $true$ as an assertion to express this.
- **Contradictions**: If you have a point that is **never** reached in the program, you can use $false$ as an assertion to express this. The only way for the program to meet all assertions is to never reach such a point.

Strongest Postconditions

The strongest postcondition of a statement s and a precondition A is the strongest assertion B that holds after the statement s has been executed.

Example

Consider the following program:

```
void main() {
    var i=2;
    var x=6;
```

```
    assert(x=3*i && i>=0);

    i=i+1;

    //state at this point:
    //i = 3
    //x = 6
    //since the i in the assertions refers to the old value of i, before the statement i=i+1 was exe

    //can we find a new assertion which explicitly computes the new value of x?
    assert(C);
}
```

What is the strongest postcondition of the statement `i=i+1` and the precondition `x==3*i && i>=0`? In other words what is the strongest assertion which we can insert in the second assertion?

This can be written as:

$$\mathbf{SP}[[\mathbf{i = i + 1}]](\mathbf{x = 3 * i \land i \geq 0})$$

To compute the assertion after the statement `i=i+1` we basically need to **undo** the statement `i=i+1` because the original assertion refered to the old value of i, before it was updated.

> Note: This only works for updates of variables. Other assignments might be alot more complicated.

We first compute the **undo** of the statement `i=i+1`:

$$\mathbf{Undo}[[\mathbf{i = i + 1}]] \equiv \mathbf{i = i - 1}$$

Then we replace the varibale i (which has already gotten updated) inside the assertion with the **undo-ed** statement:

$$\begin{aligned} B &:= x = 3 * i \land i \geq 0 \\ &\longrightarrow x = 3 * (i - 1) \land (i - 1) \geq 0 \\ &\equiv x = 3(i - 1) \land i \geq 1 \\ &=: C \end{aligned}$$

In total we have:

$$\begin{aligned} C &:= \mathbf{SP}[[\mathbf{i = i + 1}]](\mathbf{x = 3 * i \land i \geq 0}) \\ &\equiv x = 3 * (i - 1) \land i \geq 1 \end{aligned}$$

Week 2: Preconditions, Postconditions and Local Consistency

Weakest Preconditions

Weakest Preconditions are used calculate the minimum requirements, which need to hold bevore an assignment, so that a given Assertion after the assignment holds.

Its written as:

$$\mathbf{WP}[[\mathbf{s}]](\mathbf{e})$$

Where s is a statement and e is an assertion.

Example

Consider the following program:

```
void main() {
    var r = 5;
    assert(A);
    var t = 3*r;
    assert(t>=0);
}
```

We want to find the minimal requirements which need to hold at `assert(A)` so that `assert(t>=0)` holds after `var t = 3*r;`.

We can calculate this using the following formula:

$$\begin{aligned} &\mathbf{WP}[[\mathbf{t} = \mathbf{3} * \mathbf{r}]](\mathbf{t} \geq \mathbf{0}) \\ \equiv\ & 3 * r \geq 0 \\ \equiv\ & r \geq 0 \ =: A \end{aligned}$$

Now we know, that for the assertion `t>=0` to hold after `var t = 3*r;`, the assertion `r>=0` needs to hold before `var t = 3*r;`.

Local Consistency

Two assertions A and B are locally consistent, if A is **stronger** than the **weakest precondition** of B. This is written as:

$$A \implies \mathbf{WP}[[\mathbf{s}]](\mathbf{B})$$

Note that it is not required that $A = \mathbf{WP}[[\mathbf{s}]](\mathbf{B})$. Because a stronger assertion than required is also fine.

Local consistency is important: It mathematically proves that whenever the assertion A holds, then the assertion B holds after the statement s. This can be used to prove that a program actually computes what it is supposed to compute.

Example Local Consistency

Consider the following program:

```
void main() {
    var x = 30;
    assert(x>25); //A
    x=x+5;
    assert(x!=0); //B
}
```

$$
\begin{aligned}
A &\equiv x > 25 \\
B &\equiv x \neq 0 \\
s &\equiv x = x + 5
\end{aligned}
$$

At the moment all the Assertions are arbitrary, and there is no garantee that they actually hold during the execution of the program.

To prove them, we need to:

1. Show that all the assertions are locally consistent
2. We arrive at *true* at the start of the program

Local Consistency of A and B

We can calculate the weakest precondition of B and s as follows:

$$
\begin{aligned}
&\mathbf{WP}[[\mathbf{s}]](\mathbf{B}) \\
\equiv\ &\mathbf{WP}[[\mathbf{x = x + 5}]](\mathbf{x \neq 0}) \\
\equiv\ &x + 5 \neq 0 \\
\equiv\ &x \neq -5 \ \ =: B'
\end{aligned}
$$

We can check the local consistency of A and B by checking if $A \implies B'$ holds.

This is the case, because:

$$
\begin{aligned}
&A \implies B' \\
\equiv\ &x > 25 \implies x \neq -5 \\
\equiv\ &\mathit{true}
\end{aligned}
$$

So we proved that A and B are locally consistent. This means that whenever A holds, then B holds after the statement s.

Weakest Precondition of A

If we compute the weakest precondition of A and `x=30;` we get:

$$
\begin{aligned}
&\mathbf{WP}[[\mathbf{x = 30}]](\mathbf{x > 25}) \\
\equiv\ &30 > 25 \\
\equiv\ &\mathit{true} =: A'
\end{aligned}
$$

This is obviously also locally consistent, because $\mathit{true} \implies A' \equiv \mathit{true} \implies \mathit{true} \equiv \mathit{true}$.

Since we arrived at $true$, we know that the whole chain of assertions from the start to the end of the program holds and is locally consistent.

This means that we proved that when the assertion at the start (aka. $true$) holds,then the assertion A and consequently Assertion B holds.

In this case, we proved that in all instances of the program, the variable x cannot be 0 at the end.

Week 3: Loop Invariants

What is a loop invariant?

A loop invariant is a assertion that holds in each iteration of a loop. Finding such a loop invariant is needed to calculate weakest preconditions for programs with loops, because the **normal** way of finding the preconditions does not work for loops.

The Problem with Loops

Lets say you are trying to calculate the weakest precondition for the following program:

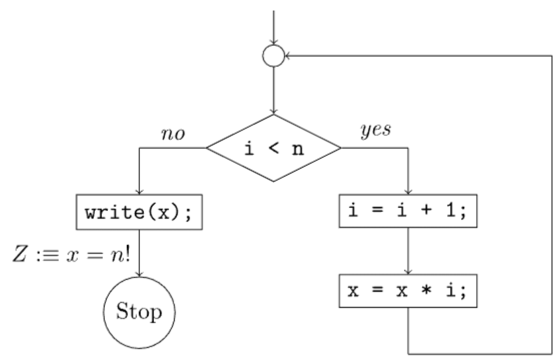

Abbildung 2: Program with loop

The normal way of finding the weakest precondition would be to start at the end of the program and work your way backwards.

$$X = \mathbf{WP}[\mathbf{write(x)}](\mathbf{x} = \mathbf{n}!) \qquad = x = n!$$

$$\begin{aligned} I &= \mathbf{WP}[\mathbf{i} < \mathbf{n}](\mathbf{X}, \mathbf{K}) && = (i < n \wedge K) \vee (i \geq n \wedge X) \\ K &= \mathbf{WP}[\mathbf{i} := \mathbf{i} + \mathbf{1}](\mathbf{J}) && = J[(i+1)/i] \\ J &= \mathbf{WP}[\mathbf{x} := \mathbf{x} * \mathbf{i}](\mathbf{I}) && = I[x * i/x] \end{aligned}$$

So we came to a conclusion that in order for us to compute the weakest precondition I we need to calculate I, K and J. But J itself depends on I so we have a circular dependency and therefore we cannot calculate I directly.

Finding a Loop Invariant

Since we cannot directly compute loop invariants we need to find a way to come up with them indirectly.

We can do this by guessing a loop invariant I that is **strong enough** to prove the correctness of the program. We do this by checking that our assertions (which are constructed using our guessed loop invariant) are locally consistent.

If we have shown local consistency we just need to check if the starting point is annotated with *true*. Then we have successfully proven the correctness of the assertion at the end of the program.

Evaluating different Loop Invariants

For the program above a suitable loop invariant would be: $I := x = i! \land 0 < i \leq n$. But how do we come up with this loop invariant?

For this we look at some other loop invariants and evaluate them:

Example Loop Invariants

1. $I := x \geq 0$:
   - This loop invariant is **not strong enough** to prove the correctness of the program. Since it it fails the local consistency check. ($I \not\Longrightarrow \mathbf{WP}[\mathbf{i} < \mathbf{n}](\mathbf{X}, \mathbf{K})$)
   - It was obvious that this loop invariant fails, because it does not contain any precise information about the value of x, which is needed to prove $x = n!$.
2. $I := i = 0 \land x = 1 \land n = 0$:
   - This loop invariant is way to strong, and is overall a bad choice because it would fail for any $n \neq 0$.
3. $I := x = i! \land 0 < i \leq n$:
   - This loop invariant is **strong enough** to prove the correctness of the program. Since it passes the local consistency check. ($I \Longrightarrow \mathbf{WP}[\mathbf{i} < \mathbf{n}](\mathbf{X}, \mathbf{K})$)
   - Using this loop invariant we can prove that *true* holds at the start of the program, which means that the program and its assertions is correct.
   - Why does this loop invariant work?
     - It encapsulates all "relevant" information about the variables which change in the loop (x and i).
     - Combined with the false-branch of the if-statement it follows that $i \leq n \land i \geq n \implies i = n$. Which is exactly what is needed to prove $x = n!$ after we exit the loop.
     - It is weak enough to not disturb the proving of *true* at the start of the program.

Tips for Finding a Loop Invariant

There exist some old videos from the lecture "EIDI2" from the year 2017 that explain how to find loop invariants. The video is in german and is not relevant for this years course, but it still contains some useful tips for finding loop invariants.

- https://ttt.in.tum.de/recordings/Info2_2017_11_24-1/Info2_2017_11_24-1.mp4 [Nico Hartmann 2017]

Week 4: Termination Proofs

Why are Termination Proofs Necessary?

Every program containing a loop is potentially dangerous. Under the right circumstances, a loop can run forever, causing the program to hang. This is called an **infinite loop**.

In general, programs which don't eventually halt are of little use and can be considered as faulty.

For example, the following program contains an infinite loop and will never print "Finished":

```
let i = 17;
let j = 5;
while (i > j){
   i += 2;
   j += 1;
}
console.log("Finished");
```

On the other hand, the following program will always halt:

```
let i = 17;
let j = 5;
while (i > j){
   i += 1;
   j += 2;
}
console.log("Finished");
```

But how can we be sure that a program will always halt? In some cases it is not so obvious as in the examples above.

This is where **termination proofs** come in.

What is a Termination Proof?

In an assertion proof, you generally try to prove that a certain variable only takes on positive values inside a loop. Furthermore, you try to prove that the variable is decreased by at least one in each iteration of the loop.

This means that the variable will eventually reach zero (or less) and the cannot be entered again. Because this would violate the Assertion we defined.

But just coming up with arbitrary assertions and then claiming that they prove termination is not enough. We also need to show that those assertions are **localy consistent**.

How to do a Termination Proof?

Bevore we can perform a termination proof, it is necessary to understand what the loop actually does.

In the second example above our intuition tells us that the loop will eventually terminate. Because the variable `j` is increased by two in each iteration, while `i` is only increased by one. This means that `j` will eventually overtake `i` and the loop will terminate.

With this understanding we can define an auxiliary variable `r` which represents this intuition.

Since we only want to prove that the loop terminates, we just need to prove *true* at the end of the program.

In general, we need to insert the following assertions / statements

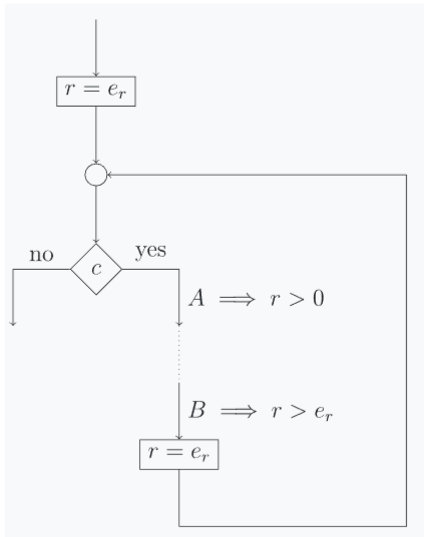

Abbildung 3: Flowchart with auxiliary variable

Notice that we need both these assertions to prove termination:

- $r > 0$ at the beginning of the loop
- $r > r_e$ at the end of the loop, right bevore $r = r_e$

Now the task is to show the following, we have proven that the loop terminates:

- local consistency of all assertions
- arrived at *true* at the start of the program
- The special assertions $r \geq 0$ and $r > r_e$ are also localy consistent

OCaml

Basic Syntax

Comments

```ocaml
(* This is a comment *)
```

Variables

```ocaml
let x = 1;;
```

Functions

```ocaml
let f x = x + 1;;
let succ: int -> int = fun x -> x + 1;;
```

Tuples

```ocaml
let x = (1, true);;
```

Lists

```ocaml
let x = [1; 2; 3];;
let y = 1 :: 2 :: 3 :: [];;
```

Records

```ocaml
type person = { name: string; age: int };;
let x = { name = "John"; age = 42 };;
let name = x.name;;
```

If-then-else

```ocaml
let x = if 3 < 6 then "1" else "2";;
```

Pattern Matching

```ocaml
let x = match 3 with
  | 1 -> "1"
  | 2 -> "2"
  | _ -> "else";;

let y = match "up" with
    | "up" -> (0, 1)
    | "down" -> (0, -1)
    | "left" -> (-1, 0)
    | "right" -> (1, 0)
```

Example Programs

Advanced Hello World

```
let welcome_string = "lmaCO ot emocleW ,!dlroW olleH"

let rec char_iterator sentence =
  if sentence = "" then ()
  else
    let head = String.get sentence 0 in
    let tail = String.sub sentence 1 (String.length sentence - 1) in
    char_iterator tail;
    print_char head

(* Call Function: Result will be visible in the console when loaded via utop *)
let _ = char_iterator welcome_string
```

Debuging OCaml

Using the `#use` command in `utop`

This method works best for single files containing the code you want to debug. The `#use` command works by just copy-pasting the entire content of the file into the `utop`-environment. It also shadows previous definitions of variables and functions.

1. Enter into the `root`-directory of the project and run `dune build` to initially build the project.

   ```
   dune build
   ```

2. Open `utop` via `dune utop src`

   ```
   dune utop src
   ```

   In all projects you will encounter, dune is configured in a way that will allow this command to work.

   This will open `utop` with the project's `src`-directory as the current working directory.

3. Reload the files

   - Instead of closing and reopening `utop` every time you change something in the files, you can use the `#use` command to load the files again.
   - But you need to be careful, because older definitions may still be around after reloading the file.

   ```
   #use "src/main.ml";;
   ```

Ocaml Exercises

List Module

Lists in OCaml

Lists in ocaml are basically linked lists. They are defined as follows:

```
let my_list = [1; 2; 3; 4; 5]
```

Since they are linked lists, you don't have direct access to the elements in the list. You can only access the head and the tail of the list. The head is the first element in the list and the tail is the rest of the list. The head is an element and the tail is a list. The tail can be an empty list.

```
let my_list = [1; 2; 3; 4; 5]

let head = List.hd my_list (* head = 1 *)
let tail = List.tl my_list (* tail = [2; 3; 4; 5] *)
```

Since Linked lists are recursive in nature, most of the functions that operate on lists are recursive. For example, the length function is defined as follows:

Example Length

```
let rec length l =
  match l with
  | [] -> 0
  | _ :: xs -> 1 + length xs
```

- This basically means that the length of list $l = \underbrace{[a_1, a_2, \ldots, a_n]}_{\text{list}}$ can be recursively defined as
  - $\text{length}(\underbrace{[a_1, a_2, \ldots, a_n]}_{\text{l}}) = 1 + \text{length}(\underbrace{[a_2, \ldots, a_n]}_{\text{tail l}})$.
- Since we need a base case for the recursion, we define the length of an empty list to be 0.
  - $\text{length}([]) = 0$.

Many other functions on lists are defined recursively. For example, the reverse function is defined as follows:

Binary Search Tree

Binary Search Tree in OCaml

A binary search tree is a binary tree where the value of the left child is less than the value of the parent and the value of the right child is greater than the value of the parent. The following is an example of a binary search tree:

```
type tree =
  | Empty
  | Node of int * tree * tree

let my_tree =
```

```
Node(6,
  Node(3,
    Node(1, Empty, Empty),
    Node(4, Empty, Empty)
  ),
  Node(8,
    Node(7, Empty, Empty),
    Node(9, Empty, Empty)
  )
)
```

The Tree type is defined as follows:

```
type tree =
  | Empty
  | Node of  int * tree * tree
```

This means that a tree is either an empty node or a node with a left subtree, a value and a right subtree. The left and right subtrees are also trees. The value is an integer.

The syntax `Node of int * tree * tree` means that the constructor `Node` takes three arguments. The first argument is an integer and the second and third arguments are trees. The result of the constructor is a value of type tree.

Since Binary Search trees are again recursive in nature, most of the functions that operate on them are recursive. For example, the function `insert` is defined as follows:

Example Insert

```
let rec insert v t =
  match t with
  | Empty -> Node(v, Empty, Empty)
  | Node(y, l, r) ->
    if v < y then
      Node(y, insert v l, r)
    else if v > y then
      Node(y, l, insert v r)
    else
      t
```

It takes a value `v` and a tree `t` and returns a new tree with the value `v` inserted into the tree `t`. If the value `v` is already present in the tree `t`, then the same tree `t` is returned.

It works as follows:

- If the tree is empty, then the value `v` is inserted into the tree as the root node.
- If the tree is not empty:
  - If the value `v` is less than the value of the root node, then the value `v` is inserted into the left subtree.
  - If the value `v` is greater than the value of the root node, then the value `v` is inserted into the right subtree.
  - Else nothing is done and the same tree is returned.

As you can see from this example, it is not possible to change the value of a node in a tree. This is because the tree is *immutable*.

If you want to change the value of a node, you have to create a new tree with the new value.

This means, that if we need to insert a value in the right subtree of a node, we have to create a completely new tree with the new value inserted, and then replace the right subtree of the node with the new tree.

At first this seams unnecessary and inefficient, but in practise it is not that big of a deal. This is because the compilers of functional programs are in general very good at optimizing the code, since they can make more assumptions about the code due to the functional nature of the language. This means that the compiler can do a lot of optimizations that are not possible in imperative languages.