FPV-Tutorial - SS23

Materialien für Manuel's FPV-Tutorium im Sommersemester 2023

Manuel Lerchner

Zuletzt aktualisiert: 6. 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

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:

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

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

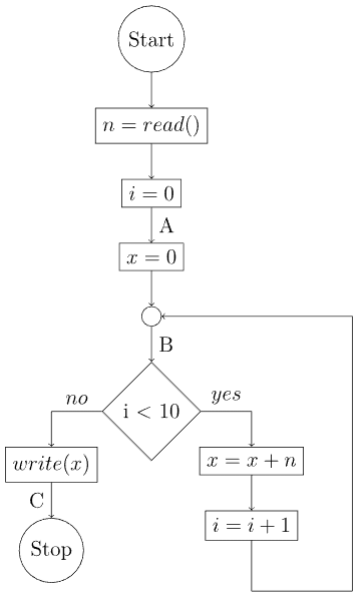

Abbildung 1: Flow Diagram

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:

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

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`:

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

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

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

In total we have:

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

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:

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

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:

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

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.

Debugging OCaml

Different ways to debug OCaml

Using the `#use` command in `utop`

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`

   ```
   dune utop
   ```

3. Load the file you want to debug via the `#use` command, e.g.

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

   - All the content of the file will be loaded into the `utop`-environment
   - Variables and functions are now available in the `utop`-environment
   - To reload the file, use the previous command again. This will shadow the previous definitions of the variables and functions