Proposal

Thesis
FG Software Engineering (Prof. Dr. Reiner Hähnle)
Supervisor: Lukas Grätz
David Holland
June 10, 2021

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software
Engineering
Group

1 Motivation

To explain the necessity of the following elaborations, as well as to ground the motivation, I will use the following java class, which implements a demonstration of a high score table. The functionality of this class is that `scores` consists of the highest values added to the high score table by `addScore`.

```java
public class HighSoreTable {
    int[] scores = new int[10];
    /**
     * adds a new score to the high score list
     * @param newScore newScore to add
     * @return 0 when not added (since below 10 best)
     *         2 if new high score
     *         1 if otherwise
     */
    public int addScore(int newScore) {
        int min = 0;
        boolean maxScore = true;
        for (int i=0; i < 10; i++) {
            if (scores[i] < scores[min]) {
                min = 1;
            }
            if (newScore <= scores[i]) {
                maxScore = false;
            }
        }
        if (newScore <= scores[min]) {
            return 0;
        }
        scores[min] = newScore;
        return maxScore ?2 :1;
    }
}
```

If I now want to be able to verify its correctness using KeY, I have to annotate the class with JML first. I won't bother annotating this class to completion, which would include also specifying invariants for the `for` loop, as well as specifying invariants for the array modifications.

```java
public class HighSoreTable {
    int[] scores = new int[10];
    /**
     * adds a new score to the high score list
     * @param newScore newScore to add
     * @return 0 when not added (since below 10 best)
     *         2 if new high score
     *         1 if otherwise
     */
    /*@ requires newScore >= 0;
      @ ensures (\forall int i; 0<=i && i<10;
      @             newScore <= \old(scores[i])
      @         )
      @             ==> (\result == 0);
      @ ensures (\exists int i; 0<=i && i<10;
      @             newScore > \old(scores[i])
      @         )
      @      && (\exists int i; 0<=i && i<10;
      @             newScore <= \old(scores[i])
      @         )
      @             ==> (\result == 1);
```

```
      @ ensures (\forall int i; 0<=i && i<10;
      @             newScore > \old(scores[i])
      @         )
      @             ==> (\result == 2);
      @*/
    public int addScore(int newScore) {
        // [...]
    }
}
```

As one can see, while being functional, as well as delivering the intended result, this example definitely is very verbose and not very practical, as the lack of readability, as well as the big redundancy could introduce bugs, unintended behaviour, etc. during modification or similar.

To mitigate this issue, there are two main ways, one of which I will discuss in-depth as the topic of my thesis and the other one of which I will possibly still have to utilize to achieve the intended result.

1.1 Macros

I will start to shortly explain the solution first, which I won't discuss as a main topic within my thesis, namely the usage of *macros*. This method improves code reusage and in consequence also readability.

```
public class HighSoreTable {
    int[] scores = new int[10];
    /**
     * adds a new score to the high score list
     * @param newScore newScore to add
     * @return 0 when not added (since below 10 best)
     *         2 if new high score
     *         1 if otherwise
     */
    //@ requires newScore >= 0;
    /*@ def lowerThanMin(score) =
      @     (\forall int i; 0<=i && i<10;
      @         score <= \old(scores[i])
      @     );
      @*/
    //@ ensures lowerThanMin(newScore) ==> (\result == 0);
    // [...]
    public int addScore(int newScore) {
        // [...]
    }
}
```

Note hoewever that the notion of macros is currently neither implemented in JML, nor KeY, so this solution would be subject to an implementation from my side. For implementing this, I could use the concept of a preprocessor, which simply expands all occurrences of defined macros with their definition.

So for example

```
/*@ def lowerThanMin(score) =
  @     (\forall int i; 0<=i && i<10;
  @         score <= \old(scores[i])
  @     );
  @*/
//@ ensures lowerThanMin(newScore) ==> (\result == 0);
```

would get expanded to

```
//@ ensures (\forall int i; 0<=i && i<10; newScore <= \old(scores[i])); ==> (\result == 0);
```

However there are multiple issues I see with this approach.

First off, it would make debugging the code with KeY magnitudes harder, as the developer would never get to see the expanded code (s.a.), but in contrast the parser, etc. would only see the expanded code. This means that if there is a type mismatch for example which would throw a compiler error, the error would complain about a code point which the developer never gets to see. One would need to manually search & replace the whole code, in other words expanding the macros manually, to see what KeY complains about, which would make this feature completely useless.

In the case of a type mismatch between macro parameter usage and parameter input by the caller, it could theoretically be mitigated by introducing typed macro parameters, which in turn makes the macro more of a named lambda, in which case however it wouldn't be feasible anymore to just use a "dumb" macro expansion.

```
/*@ def lowerThanMin(int score) =
  @     (\forall int i; 0<=i && i<10;
  @         score <= \old(scores[i])
  @     );
  @*/
//@ ensures lowerThanMin(newScore) ==> (\result == 0);
```

or alternatively, to serve the lambda notion more

```
//@ lowerThanMin = (int score) -> (\forall int i; 0<=i && i<10; score <= \old(scores[i]));
//@ ensures lowerThanMin(newScore) ==> (\result == 0);
```

Another problem not solved by this approach, especially not with using a simplistic macro expansion, would be the problem of infinite recursion, which is not really sensible in this example, but should be kept in mind anyways.

```
/*@ def lowerThanMin(int score) =
  @     (\forall int i; 0<=i && i<10;
  @         score <= \old(scores[i])
  @      && lowerThanMin(score-1)
  @     );
  @*/
//@ ensures lowerThanMin(newScore) ==> (\result == 0);
```

1.2 ε-operator

As mentioned above, there is another way to mitigate the problem of unreadable code. One of the problems of the original JML is that it can prove to be more complex to understand. One example of such a case is that even though $\neg\exists x \in M \neg(x < a)$ expresses the exact same thing as $\forall x \in M(x < a)$, the first term is magnitudes harder to understand immediately, which makes the second term more expressive. Of course the first term has the "advantage" of not needing a $\forall$, but sacrifices intuitive understanding for that.

An analogy can be made for our discussed example. Instead of defining a *method* on *how* it can be determined if a *given score* is lower than the minimal score of the high score table, we could define the *property* of a minimal score, which we could then compare our *given score* to. This property of a minimal score can then be reused and clearly states what it denotes, therefore improving readability and code reusage.

```
/*@ def minScore =
  @     (\some int score;
  @         (\exists int i; 0<=i && i<10; score == scores[i])
  @      && (\forall int i; 0<=i && i<10; score <= scores[i])
  @     );
  @*/
//@ ensures newScore <= \old(minScore) ==> (\result ==0);
```

This notion of a *property* definition, for which there exists `some` value that fulfills it, is called *Hilbert's ε-term*. Compared to the analogy given previously, the second example utilizes a new `\some` which improves the understanding of what the `minScore` *is* instead of just determining *a* lowest score. In the following paragraphs I'll investigate this method further.

This example introduces the keyword `\some`. Other possible keywords with a similar notion would be `\all`, as well as `\one`. The naming of those keywords makes it pretty obvious what intended their behaviour is.

`\one` selects *the one* value fulfilling the specified conditional property. This means there can only be one value doing so and respectively that the handling of multiple values fulfilling the condition is the same as if there were none. This is a so called *definite description* and denoted by ι, where $\iota_x A(x)$ denotes *the* object x with a property A.

`\some` selects *some* value non-deterministically, which fulfills a specified conditional property. At this point, it isn't of any interest *which* value exactly is selected, it is only of importance that it does fulfill the property definition. This is a so called *indefinite description* and denoted by ε, where $\varepsilon_x A(x)$ denotes *some* object x with a property A.

`\all` in turn selects *all* values fulfilling the specified conditional property. This means there can be *one* or *multiple* values doing so.

There already are theories used to solve the aforementioned scenarios, especially in the case of `\some`. The notion described by this keyword is called *Hilbert's ε-operator* and there are methods to check theories for consistency using this operator.

I can however see some challenges with the implementation of this concept. First of epsilon terms are nondeterministic. This means that *if* multiple values fulfill the conditional property, the selection of one of those values is non-deterministic. This makes it very hard to implement in a real-world application computational system, as our classical computational components can't really make real non-deterministic decisions. The nature of KeY might come to the rescue though, as the implementation would need to construct a *proof* using this epsilon-operator, rather than computing some real values. Checking a theory for consistency is possible in the epsilon-calulus and is called the *epsilon substitution method*.

The other challenge is in the nature of the epsilon calculus. The epsilon calculus can express first-order-logic (FOL), but not the other way around. This means that the epsilon calculus *extends* the FOL calulus. This is validated by looking at the epsilon substitution method, where it is mandatory that among other preocedures, the theory to be checked is embedded in an epsilon calculus and all quatified theorems are replaced by epsilon operations.

The easiest thing, in terms of complexity of the challenges, etc., to implement from scratch in a normal software development setting, would probably be the `one` keyword, as that is the notion of the *definite description*. In this case I neither have to deal with the epsilon calculus, nor the non-deterministic nature. *However* in terms of constructing proofs (read implementing it for KeY) I would rate the complexity *much* higher compared to `\some`! Proving that any value out of a defined set is fulfilling a condition is in my understanding way easier than proving that either `\all` or even exactly `\one` does so.

It is also worth noting that usability extends far beyond the showcased example. It would be thinkable to use those keywords outside of a reusable and non-redundant code design and just use them "in-line". This also means that an introduction of a "named property" would still be required to achieve the goals showcased throghout this section. That's why I mentioned that the need to have some kind of "dumb" macro, which would make it possible to assign a name to this epsilon-term, might still arise. I have to investigate how I could utilize the already implemented (within KeY) notion of *ghost variables*.

Luckily an already implemented function of KeY could be used, modified, or at least used as a foundation for the proposed functionality. The name of the class is `IfExThenElse` which one can use using the JML operator `\ifEx`. If we reference the documentation for this class[1], we can see some description on how to use it

$$\text{\ifEx } i;(\phi) \text{ \then } (t_1) \text{ \else } (t2)$$

This conditional operator `\ifEx` will check an integer logic variable i, which occurs in bound form within a formula ϕ and a term t_1, and proceed with term t_1 or t_2 respective of whether the bound variable can fulfull the formula ϕ for *some* value.

An implementation could therefore be possible using something like this pseudo code snippet:

```
/*@ def minScore =
  @     (\ifEx int score;
  @         (\exists int i; 0<=i && i<10; score == scores[i])
  @      && (\forall int i; 0<=i && i<10; score <= scores[i])
  @             \then
  @                 return score
  @             \else
  @                 return None
  @     );
  @*/
//@ ensures newScore <= \old(minScore) ==> (\result == 0);
```

Note that in this example some value gets "returned". This is probably not something you would want to achieve in KeY. We wanted to describe the *property* of a minimal score, not have a concrete value fulfilling this property. This could also be something some kind of ghosting variable could achieve, but I would need to investigate this further.

For the time being, one could inline this whole concept and possibly have a working example, though this would not necessarily improve either readability or code reusage, but definitely increase versatility and expressiveness.

```
/*@ ensures
  @     (\ifEx int minScore;
```

[1] http://i12www.ira.uka.de/ key/download/nightly/api/

```
  @         (\exists int i; 0<=i && i<10; minScore == scores[i])
  @      && (\forall int i; 0<=i && i<10; minScore <= scores[i])
  @             \then
  @                 newScore <= minScore ==> (\result == 0)
  @             \else
  @                 \result != 0
  @     );
  @*/
```

Note however that I noticed the `\else` branch is not really doing what it should be. The else part doesn't handle the case of the `newScore` being higher than the minScore! Instead it is handling the case when *no* `minScore`, with the property of being an element in the `scores` array, as well as being the smallest element in this array, exists! So in this case this would mean that our high score table is empty. So this branch should either throw some kind of error, or rather make sure the ensures doesn't evaluate to true in that case, if this behaviour is not allowed, or if this is allowed, handle this case differently. In case it isn't allowed to have an empty high score table, the example should be like follows

```
/*@ ensures
  @     (\ifEx int minScore;
  @         (\exists int i; 0<=i && i<10; minScore == scores[i])
  @         &&
  @         (\forall int i; 0<=i && i<10; minScore <= scores[i])
  @             \then
  @                 newScore <= minScore ==> (\result == 0)
  @             \else
  @                 false
  @     );
  @*/
```

The main difficulty of this approach is the handling of the bound variable, the way of returning the notion of the fulfilling value to the caller, as in the notion of a value matching the property description, as well as handling the case of no value fulfilling the formula.

In my opinion one could either use some sort of `null` value, or some sort of an `option` type. With the `null` value, the "caller" would be in need of some kind of checking if such a `null` value has been returned. With the `option` type could lift this burden from the user and put it on the compiler/runtime/etc. The option type is a commonly used type in the programming language rust. In this context the generic option type, initialized with a concrete type, either holds an instance of the specified type, or it holds the *nothing* value. Therefore one calls a mehtod on the option type either returning the instance of the specified type, or it raises and exception. It would be interesting to investigate if this kind of behaviour would benefit this cause.

1.3 Conclusion

I think it might well be necessary to employ both proposed solutions together. As I mentioned, defining and implementing some kind of epsilon-terms is useful in itsefl as it would greatly increase KeY, JavaDL, etc. in terms of versatility and expressiveness. It would however not improve the original points of critique, namely redundancy and low readability. To do that one would have to be able to *name* a property, so it can be reused at multiple code points and upon finding an error, or edge case within a property definition, being able to change it in one centralized place and being sure that no other code point is in need of modification. This would therefore raise the need of some kind of naming, which could be either solved by some kind of ghosting variable, or some simplistic macro handling.

2 Work Packages

I think the work lying ahead of me is contained withing the following work subjects.

1. Research how `\ifEx` is implemented within JavaDL and/or KeY, in order to be able to implement something smilar myself. This should teach me all I need in terms of which source files are linked together or are standing in relation to one another, as well as how the different parts play together to create something that KeY can use for constructing a proof. Try implementing or extending some placeholder functionality in order to get a feel for it.
2. Research how I can implement the aforementioned epsilon substitution method. Possibly research how such a notion is implemented in Java in general, or other programming languages that serve the intent of proof construction or proving code correctness more, like e.g. functional programming languages like Haskell.
3. Research how ghosting variables could be used to achieve the wanted effect. If this is not feasible, I would have to think about employing some kind of macros.
4. Research how other programming languages implement macro handling, as well as to what extent. This would greatly improve the understanding of whether this task, even if it only serves the purpose of naming the properties, is as easy as employing a preprocessor, or if some more intricate handling is necessary.
5. Research which parts are implemented within KeY itself and which are implemented within JavaDL. This will greatly help my understanding of the underlying working mechanisms of KeY, how they work together to construct proofs, etc.
6. Possibly implement a minimal working example using `\ifEx` to see in which cases it is lacking, or if it is already sufficient compared to the expected behaviour of a (blackbox implementation) `\some`. If not, I would have to manually implement the aforementioned keywords. Therefore the goal of this task would be to determine if it would be feasible to embed the `\ifEx` into a `\some` keywoard or if a groud-up approach is necessary.
7. Determine (using this minimal working example) what the true goal to be achieved through this thesis is, to have a concrete expectation horizon.
8. Implement `some` with a minimal scope of functionality, i.e. a minimal working example. According to how easy/difficult this task is, it is determined if implementation of `one` and/or `all` is workable within the given 6 months timeframe.
9. Implement some kind of naming mechanism for the property definition.
10. Construct multiple usage examples to demo and test the new functionality.
11. Write extensive tests and also theoretically ground the approach and implementation to ensure issue-free usage.
12. Document the implementation as well as the whole process to maintain reproducability.
13. Write the thesis itself.

3 Schedule

I would estimate the amount of work needed to take (almost) all of these points to completion to be workable within the given timeframe of the thesis, namely 6 months. I will expect to write the documentation as well as snippets of sections of the final thesis during the whole duration. I will also expect to have the finalization of the thesis as well as the presentation take 6 weeks or more by themselves. Apart from that I will mainly focus on finishing the rough research part as fast as possible, in order for me to determine wheter a complete implementation from scratch is needed or not. Note however that this is subject to great fluctuation as well as change, as I can't really estimate the amount of work, further research, implementation and documentation that needs to be done. Also note that for some tasks a non-sequential completion can be possible.