Proof Assistant Using Predicate Transformer Semantics

Pfalzgraf Martin

Contents

Introduction ........................................... 2

Previous Work ........................................ 3
Road map of paper .................................... 3

Software Design ...................................... 3

First Pass of the Parser ............................... 4
Internal AST Intermediate ............................ 6
Applying Weakest Precondition Rules ............ 6
Software Testing and Verification .................. 7

Major Libraries Used .................................. 7
SBV ....................................................... 8
Converting AST to SBV ............................... 9
Trifecta ............................................... 9
Lens .................................................... 9
Text-show ............................................. 10

Performance evaluation ............................... 10

Future work ........................................... 10

Acknowledgement ..................................... 11

Appendix ................................................ 11
Abstract

This paper is about a proof assistant using Hoare logic with the predicate transformer semantics, also known as the weakest precondition calculus, to write a pseudocode verification tool. The purpose is to create a tool that students can practice writing postconditions and loop invariants in without having either the complexity of a full programming language or needing to learn complex tools like Coq or Agda. The assistant automatically checks code blocks against a postcondition and loop invariants for partial correctness. It returns either a set of variable assignments that invalidate the postcondition or asserts the code is partial correct.

Introduction

Formal verification is typically considered a niche skill that is often irrelevant to most software projects. I believe this is due to how hard many of the tools for formal verification are to use and the dearth of simple tools to teach the basics of software verification. At its core software verification is about specifying what the programmer wants their code to do, so a tool can tell them where their implementation differs from what they want. In this sense the purpose of the tool is to help the programmer analyze their code, rather than constrain how they program.

As a programmer I constantly run into cases where I do not know what is wrong with my code, yet I use test suites, write in strongly typed languages and adhere to a purely functional paradigm. So what is missing? The issue is that these techniques are trying to solve two distinct problems at the same time. The description of what do you want your code to do and the implementation of how do you want your code to do it. These are fundamentally different questions. A test suite is only a snapshot of that description, static typing does not check the semantics of your program and pure functions, although easier to reason about, can still be unfathomably complex. We need a way to write down what something should do and verify the code meets that description. This is formal verification in a nutshell. Write something down about what your code should do and verify that it will always do just that.

However, verifying arbitrary code meets some conditions is undecidable in the general case. The crux of the issue lies with termination. For any code with loops there is the possibility that the code will not terminate. We can prove partial correctness automatically if the loop invariant is given, but total correctness requires proving inductively that for all inputs that the loop variant will exit the loop in a finite number of steps. Which is undecidable. On the contrary, I postulate that just checking for partial correctness is enough to give the programmer a good idea of what their code is doing wrong. The purpose here is to use a formal logic to analyze code, not formally verify code. The idea is to provide the programmer some insight into what their code is doing relative to what they think their code is doing, not necessarily give a complete proof.
There is another issue with using formal verification as a tool to analyze code, how does the programmer specify what their code does? They clearly cannot use the same language that they are programming with or they would just writing two different implementations for every function. The specification for a function need only capture the core essence of the function or some aspect that is critical for its use. First order logic is a sound system that fits this purpose. Unfortunately first order logic is not something most programmers are familiar with.

The purpose of this tool is just that. Provide a tool that abstracts away the programming language and lets the programmer focus on using first order logic to specify what some pseudocode does. To do this I use the weakest precondition calculus, so the programmer only needs to provide what they think the code should satisfy and what each loop should accomplish. As a future work I hope to provide this tool online as a teaching resource.

**Previous Work**

The solver is based on the weakest precondition calculus as developed by Dijkstra [1976]. It builds on my experience from writing a solver using Hoare logic rules that was not automated, but shares no code from my previous project. Unfortunately, I had to have a fundamentally different internal structure to that of my previous proof assistant.

This work builds off of the Haskell SBV SMT library that is an interface to several SAT solvers. I also wrote bindings to Zhe Hou’s solver, but did not use them in my final product. The weakest precondition rules are taken from (Martin 1983)

**Road map of paper**

- Software design
  - How each component fits into the solver
  - How testing was integrated in the project
  - First two stages of the Analyzer
- Discussion of libraries used in the project
  - Explains how SBV is integrated into the solver
  - Walks through the final stage of the Analyzer

**Software Design**

The Software is broken up into these sections:

- Lexing and Parsing into a intermediate structure is made up of
  - First order logic
- pseudocode
- Note: the library I use combines both the lexing and parsing, so when this papers refers to parsing something it means both lexing and parsing.

- Applying the Weakest precondition calculus to the intermediate structure
- Converting the intermediate structure to SAT (Boolean satisfiability problem)
  - This is done through SMT (Satisfiability modulo theories)
  - (this is covered in the Major Libraries Used section under SBV)

The overall idea is to create a “question” in SMT that is translated into SAT for analysis. This “question” is generated by passing a postcondition through the code by applying the weakest precondition calculus. The loop invariants are integrated into the “question” whenever a while loop is encountered. In this paper a query is the constructed “question” from the code.

**First Pass of the Parser**

First the code, postcondition and any loop invariant are read into an basic proof structure.

- This done by creating a `ProofSequent` which is made up of a list of `Sequent`, a `PostCondition` and a list of `loopInvariant`.
- A `Sequent` is a section of my pseudocode it is any of:
  - `IfThenElse Condition [Sequent] [Sequent]`
  - `While Condition [Sequent]`
  - `Assignment Name Expr`.

This internal structure maps directly to the pseudocode. Here are two examples of what the syntax of code block look like.

```plaintext
// {-@ forall n, a (sum = n * a) @@ sum = (j * a) @-}
sum := 0;
j := 0;
while (j /= n) {
    sum := sum + a;
j := j + 1;
}

// {-@ (m >= a) && (m >= b) @-}
if (a > b) {
    m := a;
} else {
    m := b;
}
```

This is covered in the Major Libraries Used section under SBV.)
The IfThenElse Condition [Sequent] [Sequent] is read as:

If (Condition) {
    Any number of Sequent
} else {
    Any number of Sequent
}

Where both branches of the If can be empty, same for while.

The While Condition [Sequent] is read as:

While (Condition) {
    Any number of Sequent
}

The Assistant Name Expr can be read as:

name := expr

Condition, Expr and loopInvariant are all parsed into Formulae, which is first order logic with implicit quantifiers, while the PostCondition is the only part parsed as complete first order logic. The weakest precondition rules do not care about existential quantifiers, so there is no reason to have them anywhere besides the PostCondition. Any variable that is not quantified over is considered free and therefore universal by the SMT solver.

For example, this is a PostCondition and loopInvariant that comes from the first example code.

// {-@ forall n, a (sum = n * a) @@ sum = (j * a) @-}

The forall n, a (sum = n * a) is the PostCondition and the sum = (j * a) is the loopInvariant. To add another loopInvariant, simply add the invariant separated by @@ from the previous invariant. The // stands for comment.

In the above example, j has no quantifier, but is still treated as universal is the query. This means that only the exists quantifier actually changes the contents of the query made to the SMT solver. In general this is what the programmer should want, since they want to make sure their code will do the same thing regardless of where it is in their code.

The ProofSequent also keeps track of all the variables in the code block. This was going to be used to determine the type of each variable so the solver could handle different types of numbers, but writing a type checker is beyond the scope of this project.
Internal AST Intermediate

The AST (Abstract Syntax Tree) is what makes up the Condition, Expr, loopInvariant, PostCondition and as a result make up Formulae. The AST does not store any information about where it came from, that is kept in the ProofSequent (the above structure).

The AST has three levels:

- The Quantifier of variables that holds either a another Quantifier or a Formulae
- The Formulae that contains both:
  - boolean connectives like: \&\& between Formulae
  - boolean connectives like: = between Expr
- The Expr that contains math operations like: + and / between other Expr

This structure enforces that no Quantifier is inside a Formulae and neither can be found inside an Expr. This is critical to writing code that operates over the AST, so I can make some assumptions about each level of processing. Once I am passed the Quantifier I know all new variables with be free variables. After Formulae I know the type of Number is now concrete (important for a type checker).

This AST also enforces brackets to remove any ambiguity that the parser gives the programmer. The analyzer prints the query made to the SMT library, so the programmer can check for an incorrect interpretation.

The purpose behind reading things into the AST is to put the data in format that is easy to work with in the next stages of the analyzer.

Applying Weakest Precondition Rules

The second stage is to walk through all of the weakest precondition calculus rules to prepare for conversion to SBV. This works by taking the postcondition, starting at the end of the AST and walking the code blocks backwards.

The Analyzer accomplishes this by flipping between two functions. One handles the case where a single Sequent is involved and the other applies the sequence rule over a list of Sequent.

I chose the weakest precondition calculus over the traditional Hoare rules because it is nearly impossible to convert code into Hoare triples in a single pass. Even if multiple intermediates are used the Hoare logic rules are problematic to automate. The issue comes from the fact that there is always the possibility of weakening the precondition or strengthening the postcondition to stitch two Hoare triples together. So as a result there is always the possibility that the constraints are too tight because some of the preconditions or postconditions could be changed. The weakest precondition calculus on the other hand gives me the weakest precondition for each code block and I can feed that the postcondition to the next code block.
I take the postcondition from the **ProofSequent** and generate a new constraint in first order logic, using the weakest precondition rules. See Appendix for rules. This single query is then handed off to be converted into SMT.

### Software Testing and Verification

Testing was an integral part of development and design. I used two testing frameworks: docktest and hspec. Initially I used LiquidHaskell as well, but was dropped to reduce the scope of the project. These frameworks use several different libraries, each with their own distinct methodology, to check my program for correctness:

- QuickCheck provides automatic testing of programs
- HUnit provides is a unit testing library/framework

Docktest allows me to write both normal tests and QuickCheck properties in the source files that both act as guide for development and documentation for the code. Hspec is a testing suit that is setup for more verbose and extensive testing than docktest. It supports QuickCheck and HUnit. LiquidHaskell would have allowed me to add annotations to my types with predicates drawn from decidable logics (Merz and Vanzetto 2014).

What these frameworks amount to is a methodology where I design my code through tests and propositions similar to test driven development. Docktest is useful in this regard for defining single examples of how to use a function in the source. This also provided me a way to plan what each function should do by writing the tests for the functions that had yet been written. However, more than two tests per function start to clutter up the code, making Hspec the place to put the real tests for the project.

QuickCheck also did not make it into this version. I was going to use it to check the parser by running the pretty printer on an arbitrary AST, then parsing the output and seeing if the AST was still the same. All the instances are there, just couldn’t get the types to line up.

### Major Libraries Used

There are far too many libraries my code relies on to talk about each, but there are three notable ones that I spent much of the project learning, SBV, Trifecta and Lens. Each of the libraries covered a key part of the project and provided their own unique challenges in both understanding them and using them. I had no prior experience with these libraries before the project. I also mention Show-Text, since I ended up modifying the source to fit by purpose.

There is some Haskell terminology I use that you may not be familiar with:

- typeclass
- defines a set of functions that instances of the typeclass must implement
• Example: `Num`, the number typeclass that has `+`, which all instances of `Num` must implement. That way we can call `+` all off types of `Num`.
• `kind`
• Describes the number of types a constructor can take before it is a concrete type.
• Example: `Maybe a` is of kind `* -> *`, while `Maybe Int` is of kind `*`, which is concrete.

**SBV**

SBV stands for SMT based verification and is designed to facilitate proofs of Haskell code. It provides two main functions that call the underlying SAT solver, `prove` and `sat`. Where `prove` returns Q.E.D if and only if there is no assignment of variables such that the query is falsified and `sat` returns the set of variables that satisfy the query or returns `unsatisfiable`. Effectively they are inverses of each other. I only care about the `prove` function that provides the variable assignments that falsify the query. With it I can provide why a given piece of code does not satisfy its postcondition.

SBV’s `prove` really only works with `SBool` which is short for `SBV Bool`; which is just a type alias for `Bool`, Haskell’s internal boolean type. The reason for this is `prove` only works over types that are an instance of `provable`. The only instances for `provable` provided by SBV are `SBool`, `Symbolic SBool` and functions that take in some number of types of the `SymWord` typeclass and return something that is already an instance of `provable`. The `SymWord` typeclass is only a thin wrapper that provides a way to declare existential or universal variables of that type and the `Symbolic` type is a wrapper that SBV uses internally to keep track of intermediate results. So effectively, SBV can only prove things that are phrased in its internal boolean type, `SBool`.

To construct an `SBool` from an arbitrary type we have three typeclasses, `EqSymbolic`, `OrdSymbolic` and `SymWord`. There is a `Boolean` typeclass, but it only lets users stitch `SBool` together. `EqSymbolic` is defined for `Bool`, `SBV a` and some containers. The `SBV a` is just SBV’s way of creating type aliases for Haskell’s built-in types, like the `SBV Bool`, `SBV Integer`, `SBV Word8`, etc. `OrdSymbolic` has the same predefined instances as `EqSymbolic` and `SymWord` only lets you map Haskell datatypes to quantified variables, so the queries must be made up of `SBV a` and `Bool`.

The problem is `SBV a` is of kind `* -> *`, it is not concrete. Haskell does not know which instance of `EqSymbolic` or `OrdSymbolic` to use to create the `SBool`. Somehow I would have to take the type information from where the variable is defined to set the correct type. However, Haskell is statically typed, not dynamically typed. I have to know the types at compile time. This means I cannot write a function that reads in a variable of type `Integer`, returns a `SBV Integer`, reads `Word8` and returns a `SBV Word8`. Not only that, but I also need to make sure all the variable types are the same for a given section. In order to handle different number types I need a fully fledged type checker that gives me some level of abstraction to work from. This is why the analyzer only handles Integers.

However, all of this is still relevant to the last phase of the analyzer, converting the AST.
Converting AST to SBV

At this point the AST now contains all the information I need to turn it into a Symbolic SBool. The problem is I have to look through the AST to find out what quantifiers are over each variable and what each variable is called. So instead writing two traversals to get the Symbolic SBool, I return a function that takes a hash map of variables to quantifier and returns a Symbolic SBool. That way I don’t have to traverse the AST twice, instead leaving placeholders for each variable that get filled in by the hash map.

However, there is a problem with this approach. Symbolic is a monad that wraps all SBV’s types so it can have an abstraction from the actual variable names. This means when I put a Symbolic SInteger in a placeholder, instead up putting a named variable of type Integer I am putting an arbitrarily named symbolic variable. This meant that “a” in one part of the query would not be the same as all the other variables named “a”. It turns out the only workaround is to put SInteger in each placeholder, not the Symbolic version. However, you cannot get SInteger from the Symbolic contents. Getting the integer value of an arbitrary variable does not make sense. Still, I can access the internal name of the Symbolic computation while inside the monad. This meant I ended up wrapping the whole function call inside the Symbolic monad so I could set the placeholders to the their correct names.

Haskell is the only language where I would try to write a recursive function that returns a function, which fills in the blanks I did not know on the first pass.

This concludes the long journey the pseudocode takes through the analyzer.

Trifecta

It is a library for writing parsers. Trifecta combines the process of parsing and lexing to give more meaningful error messages. The error messages it produces are very clear and can be easily modified to give better feedback to the user. Plus it has a nice way to define expressions with some level of precedence and associativity. The confusing part is its heavy reliance on Haskell typeclasses. Effectively I needed to understand the typeclasses: MonadPlus, Alternative, Monad, Functor, Applicative, Monoid and Semigroup before I could use the library. I use Trifecta to parse in the first order logic and pseudo code.

Lens

This library is not necessary for writing the project, but it makes heavily nested types nice to work with. Haskell record types are great for expressively describing your type, but at the cost of making your types hard to use. Among the many things the Lens library provides I used the lenses and traversals. Lenses allow you to view deeply nested data. Traversals allow you to run traverse types within lenses, think a list within a record type.
Text-show

Text-show provides a typeclass `TextShow` for defining `showb` which constructs a `Builder` for types. It is yet another attempt at fixing Haskell’s issue with `String`. In short `String` is a linked list, which is really slow for string manipulation. `Builder` is an abstract type that lets you build up a string’s parts lazily and turn it into `Text`, `Bytestring` or `String` in one go. Unfortunately their instance for `Text` add “” around the text and that breaks Zhe Hou’s solver. You cannot have multiple instances for any type, so I forked the project and fixed it in the source code.

Performance evaluation

These are four tests of the parser that reads in the code and generates the `ProofSequent`. For each test I used random input where each line contained one code block.

- 2108 lines = (0.14 secs, 245,197,056 bytes)
- 4214 lines = (0.28 secs, 487,808,768 bytes)
- 8427 lines = (0.58 secs, 973,451,176 bytes)
- 16863 lines = (1.13 secs, 1,944,259,128 bytes)

It appears to be growing linearly.

Testing the other parts of the program get tricky. Because of lazy evaluation I would need to force the parser to finish before running the other major functions. This requires me to use the Haskell profiler, which is a bit beyond me.

Regardless, as long as my code is polynomial, the speed does not matter. Since every query has to be sent to a SAT solver, the bottleneck is the SAT solver.

Once I go beyond 10,000 lines the SAT solver takes too long for me to feasibly wait for its termination, while my code still generates the query under a second.

Future work

I created what I set out to do, with the caveat of not creating an online interface for the analyzer. There still are a number of things that did not make this release:

- Does not handle separation logic
  - The reason for this is due to how Zhe Hou’s solver works. If something is provable, then it generates a proof, but if not it returns that it did not find a proof. This contradicts the purpose of this project to give the programmer a meaningful response when their code has an error.
The Separation rules can be implemented in the SBV framework to provide the needed feedback

- Does not have functions
  - Would need minor changes to syntax parser
  - Functions can be represented as Haskell code and easily turned into SBV

- Lack of extensive testing
  - QuickCheck: expectingly for the parser
  - LiquidHaskell: To better constrain my functions

- Not online
  - I wrote an interface for the solver that is built off of the Ace editor is Javascript
  - I also created the API to serve the analyzer
  - What is missing is a conversion to and from JSON

- Carrying the location of where things came from through the code
  - The Parser points out exactly where any errors are and lists what could have been there
  - What is missing is the internal AST does not keep the positions of where each element was parsed from

Only the online portion is necessary for this tool to become a teaching resource, but the others are still meaningful improvements.

**Acknowledgement**

I wrote every line that appears in the project, besides the lines I generated with derive. It generates some of the Haskell instances for you.

Special thanks to Zhe for walking me through some of my logic and for moral support.

As much as SBV was difficult to work with I could not have written my project without it.

**Appendix**

- Assignment Rule:
  \[
  wp(x := E, R) \equiv \forall y, (y = E) \implies R[x \leftarrow y]
  \]

- Sequence Rule:
  \[
  wp(S_1; S_2, R) \equiv wp(S_1, wp(S_2, R))
  \]
• Conditional Rule:

\[ wp(\text{if } E \text{ then } S_1 \text{ else } S_2 \text{ end }, R) \equiv (E \implies wp(S_1, R)) \land (\neg E \implies wp(S_2, R)) \]

• While Rule (Partial Correctness):

\[ wlp(\text{while } E \text{ do } S \text{ done }, R) \equiv I \land \forall y, ((E \land I) \implies wlp(S, I))[x \leftarrow y] \land \forall y, ((\neg E \land I) \implies R)[x \leftarrow y] \]

The \([x \leftarrow y]\) means replace all instances of \(x\) with \(y\)

References
