May 2021

Refactoring SASyLF Proofs

Kayla Rodenbeck
University of Wisconsin-Milwaukee

REFACTORING SASYLF PROOFS

by

Kayla Rodenbeck

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the degree of

Master of Science
in Computer Science

at
The University of Wisconsin-Milwaukee
May 2021

ABSTRACT

REFACTORING SASYLF PROOFS

by

Kayla Rodenbeck

The University of Wisconsin-Milwaukee, 2021
Under the Supervision of Professor John Boyland

Refactoring is a common practice undertaken by software developers that is used to improve the quality of existing code. Originally done by hand, several automated refactorings have been introduced over the years, saving both time and effort expended by the developer. Proof engineering, on the other hand, is a more recent concept which has not advanced as quickly over the years, thus they do not have similar tools to be able to make similar changes automatically. Since proof assistants resemble programming languages in many regards, a similar practice may be applied.

Thus, the main idea behind this thesis is introduced. The topic of refactoring is discussed, and how it can be applied to proof assistants as well as for programming languages. The framework of the proof assistant SASyLF is also introduced, including a description of its different components and its syntax. Additionally, some related work regarding proof refactoring is mentioned.

Preface

In the fall of 2019, I had the opportunity to take CS 732: Type Systems for Programming Languages. Through it, I was introduced to SASyLF, and we used it for the entirety of the semester as a supplement to prove type theory. I became very accustomed to it, and over time came to enjoy using it, especially because it was made easier to use thanks to the Eclipse plugin that was developed to use it. Simultaneously, I was taking an advanced object-oriented programming class, and learned about the true power behind refactoring.

As the semester wore on, there were several instances where I had originally planned on a naming system for an assignment, but later wanted to go back and re-work it after spending some time actively working on it. I became acutely aware that if I wanted to do something simple such as renaming a theorem or making other similar small but necessary changes, I had to go back and manually fix every single instance of the old name that I had used. Because I am not a very patient person when it comes to such details, the process of changing a name quickly became frustrating bordering on infuriating, especially in situations where I had already used the name for other subsequent proofs of lemmas and theorems. The errors were easy enough to spot if I missed any changes before saving, but the process got tiresome very quickly. I yearned for a way to be able to do this without having to take care of everything myself.

It did not occur to me until the beginning of the next semester that there is a way to do it, for programming languages at least – we can use refactoring! I was already using it all the time in other mediums to make these kinds of changes for me, both to improve the quality of my code and to maintain its aesthetic. These refactorings were included in the

IDEs I was using. If I wanted to be able to apply a similar concept to SASyLF, I would need to implement it myself.

And thus, this is how my lack of patience and general laziness ended up leading me spending so much of my work and time to try and make other users' lives easier.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This day has been a long time coming, but I never would have been able to make it this far without the support I received along the way. Each and every person that I've worked with along the way played a role in ensuring that I would get here, but for brevity's sake I will keep the list short.

I cannot imagine how different my situations would be if I did not have the support of my family behind me. They were the ones who bore the brunt of my frustration, who calmed me when I was most anxious, who offered shoulders to lean on when I needed comfort, and who most of all encouraged me to follow and pursue my dreams. They are the wind in my sails that keeps me going even when the path ahead seems stormy, and I am so fortunate to have them for their unconditional love, for their moral (and financial) support, and for instilling in me the drive and stubbornness I needed to accomplish everything that I have been able to.

On the academic side, I need to thank my supervisor John Boyland. In 2016 I stepped foot on UWM's campus again with only a bus pass, a lot of anxiety, and a faint hope that maybe coming back to school would be the right decision. It ultimately was the right choice, and I don't think it would have been the case if it were not for him. He is an amazing and passionate teacher who unknowingly sparked the passion for learning in me that I didn't even know I had been looking for. He is a great wealth of knowledge and encouragement who always has the right advice and feedback to give, even when it would knock me down a few pegs in times where my ego started to swell too much. I can safely say that without him I would not have had the opportunities that I have experienced, and without his guidance I never would have made it this far in my academic journey, both undergraduate and graduate work alike. I am so incredibly grateful to have worked with him, and I'm not sure if I even

have the words to express how blessed I am to have done so, but hopefully this attempt will suffice. John, thank you.

Chapter 1

Introduction

1.1 What is Refactoring?

Refactoring refers to the process of changing a software artifact that will improve its internal structure without changing any of the external behaviors [FBB+99]. Its main purpose is to rework the code to something that is more flexible, or perhaps simpler to understand. To be a true refactoring, the changes that are made must be done in a way that no observable difference in behavior is noted – it should give the same results regardless of whether the refactoring was performed or not. Many different types of refactoring exist – Martin Fowler et al. [FBB+99] published an entire book dedicated to all different kinds of refactoring, as well as describing the motivation behind performing each refactoring in detail.

Refactoring has evolved into such a common and necessary practice in programming that automated refactoring tools have been built into IDEs such as Eclipse for Java developers. With the click of a button, common refactorings such as renaming variables or methods become nearly instantaneous, and are more efficient and less error-prone than a refactoring performed manually. Local variables can be extracted, and entire methods can be inlined or even moved to another class with little to no effort on the programmer's part.

In this work, the specific refactorings focused on are the movement of a file from one package to another, renaming a file, and renaming of a "theorem".

1.2 What is SASyLF?

SASyLF, a.k.a. **S**econd-order **A**bstract **Sy**ntax **L**ogic **F**ramework, is a proof assistant designed for education of programming language theory [ASS08]. It is basically as the name describes: a LF-based [HHP93] logical framework with built in variable binding while restricting it to second-order abstract syntax. This lends to a proof that is easy to read and mimics what can be written on paper. Its primary use is for graduate-level courses for proving theorems in type theory.

The components of SASyLF's meta-logic is described in greater detail in the following subsections.

1.2.1 Syntax

The start of a SASyLF proof defines the general syntax that will be used. Identifiers that will be used as terminals in the grammar are declared as such using the `terminals` keyword declaration; characters such as +, -, =, and similar symbols are already presumed to be terminals and do not need to specified.

The `syntax` keyword indicates the start of the block that will define the grammar productions. The left side of the production contains the nonterminal identifier, while the right hand side defines the rules of application, separating all possible rules with the "|" symbol. The left and right sides of the production are themselves separated by "::=". An example of this can be seen in Figure 1.1.

```
terminals S

syntax

n ::= 0 | S n
```

Figure 1.1: Example of terminals and syntax declaration in SASyLF.

1.2.2 Judgments

Once the grammar has been defined, SASyLF further uses judgments to define the operational semantics. Each judgment has is given a name as an identifier followed by its syntactic form. Once a judgment has been declared, inference rules are defined for it using a horizontal line made by "-" symbols, and are given a name as well. Above each "line" are all the premises, and below it is the conclusion. Figure 1.2 shows an example of what this looks like.

```
judgment equal: n = n

----- eq
n = n


judgment notequal: n ≠ n

------- zero-ne-succ
0 ≠ S n

------- succ-ne-zero
S n ≠ 0

n1 ≠ n2
---------- ne-rec
S n1 ≠ S n2
```

Figure 1.2: Example of judgment and rule definitions in SASyLF.

1.2.3 Theorems

Theorems can be defined as either a lemma or a theorem, but are considered essentially the same in SASyLF (for ease of explanation, I will hereafter refer to both as theorems). In SASyLF, these theorems can be invoked in proofs in the same manner as functions or procedures in a program. Each theorem has the form "`forall` ⟨metavariables and judgments⟩ `exists` ⟨judgment⟩". The theorem contains a list of judgments and the justification used for each judgment. These "justifications" include induction, induction hypothesis, case analysis, substitution, or contradiction. Figures 1.3 and 1.4 show proofs by induction and by contradiction, respectively.

1.3 Refactoring Proofs

It has been observed that the process used in proof assistants is similar to that of programming languages. However, while software engineering has adapted to create automated revisions for both large and small scale changes, proof engineering still requires changes of such a nature to be done by hand. Given this observation, it is not too far of a leap to suggest that automated refactoring could also be adapted to apply to proof assistants. The main goal of refactoring would stay the same – the internal structure of the proof is altered, but still gives a working proof after the changes have been made. Thus, any changes that need to be made should maintain the same semantics as before the change was applied. SASyLF is a good candidate for refactoring because it is already supported by the Eclipse IDE.

1.4 Related Work

At the time of writing, there has not been a lot of work done regarding refactoring of proofs. However, the idea has at least been posited by other academics. A specific example of this was proposed in a paper by Iain Johnston Whiteside [WADG11]. Here, a similar resemblance is pointed out – since programming languages and proof languages have similar properties, refactoring ought to be able to be applied to proofs as well. Furthermore, demonstration of proof refactoring is applied to Hiproof, an invented representation of a proof based on Hitac [ADL10].

Whiteside takes things a step further in his doctoral thesis [Whi13]. He introduces Hiscript, a proof framework language consisting of a procedural tactic language, a declarative proof language, and a modular theory language, based on Hiproof mentioned above. Using Hiscript as his example, he categorizes many possible refactorings such as renaming, move, and subproof flattening, among others. He goes on to systematically prove the correctness of each refactoring, verifying that after a proof has been refactored, the proof still remains valid even though some of the inner structure has changed.

1.5 Roadmap to Remainder of Thesis

The next sections of this thesis detail the work that was put into making refactoring work with SASyLF. Chapter two describes the process taken to implement refactoring in SASyLF. First, some background work needed to be completed to get some already-implemented refactorings working properly in SASyLF: Rename Proof Module and Move Proof File. These changes were necessary due to changes in SASyLF that allowed modules to be referenced. Once this task was completed, a brand new refactoring, Rename Theorem, was created, with

each component described in detail. Chapter three then outlines some difficulties encountered during the implementation process, and how they were ultimately overcome. Finally, this thesis is wrapped up by some concluding remarks and some desired future work to be undertaken.

```
theorem eq-or-ne:
  forall n1
  forall n2
  exists n1 = n2 or n1 ≠ n2.
  proof by induction on n1:
    case 0 is
      proof by case analysis on n2:
        case 0 is
          e: 0 = 0 by rule eq
          proof by e
        end case
        case S n2' is
          d: 0 ≠ S n2' by rule zero-ne-succ
          proof by d
        end case
      end case analysis
    end case
    case S n1' is
      proof by case analysis on n2:
        case 0 is
          d: S n1' ≠ 0 by rule succ-ne-zero
          proof by d
        end case
        case S n2' is
          d1: n1' = n2' or n1' ≠ n2' by induction hypothesis on n1', n2'
          proof by case analysis on d1:
            case or e: n1' = n2' is
              e': S n1' = S n2' by lemma succ-preserves-eq on e
              proof by e'
            end case
            case or d2: n1' ≠ n2' is
              d3: S n1' ≠ S n2' by rule ne-rec on d2
              proof by d3
            end case
          end case analysis
        end case
      end case analysis
    end case
  end induction
end theorem
```

Figure 1.3: Example of a theorem proved by induction in SASyLF.

```
theorem ne-anti-reflexive:
  forall d: n ≠ n
  exists contradiction.
  proof by induction on d:
    case rule
      d': n' ≠ n'
      ------------- ne-rec
      _: S n' ≠ S n'
      where n := S n'
    is
      proof by induction hypothesis on d'
    end case
  end induction
end theorem
```

Figure 1.4: Example of a theorem proved by contradiction in SASyLF.

Chapter 2

Implementation

2.1 Structure of SASyLF Projects

SASyLF projects are structured the same as projects in Java using Eclipse, in that proof files are contained within packages. Additionally, other proof modules may be referenced so that other theorems and syntax can be used without needing to duplicate the exact syntax in multiple different proof files, a feature that was added in SASyLF version 1.4. This can be done in one of two ways:

1. appending the package(s) and proof module to the front of a referenced theorem using '.' symbols, e.g. *PackageName*.*ProofModuleName*.*TheoremName*; or

2. defining a new name for the module in a separate syntax and appending it to the front of a referenced theorem, e.g. *Name*.*TheoremName*.

Should the latter form be used, the new name is defined as *module Name = PackageName*.*ProofModuleName*, so that *Name* can be used as a shortened version of it throughout the document. Thus, when looking for uses, it is important to look for both types of references as either are sufficient, and to make sure that they are updated correctly.

2.2 Document Dependencies

Because SASyLF allows for proofs to reference other proofs, it is important to make sure that all possible references are being checked and changed where necessary. Two approaches were considered:

1. a brute-force approach where all proof files in a project are examined; or
2. utilizing a system to keep track of all proof dependencies.

The first approach has an obvious flaw to it: it is inefficient. For instance, if a refactoring is applied to a proof file located in a project containing hundreds of other proofs, but is referenced by only a few proofs (or worse, none at all) then it traverses through many more proof files than are strictly necessary, a waste of both time and effort.

It was decided to go with the second approach: maintaining coarse dependencies between proof files so all uses can be found and accessed immediately. This added a layer of complexity as these dependencies needed to be computed at compile time, and also needed to be verified that all dependencies were added properly. A system to do this was already partially in place so that reference modules could be accessed, but did not fully work as intended, and so some modification was needed to get the all correct dependencies to be mapped for each proof.

2.3 Existing Refactorings

Once dependencies were able to be established reliably between linked documents, the next logical step was to tackle some pre-existing refactorings in SASyLF. These were already previously implemented, but did not fully work due to the lack of document dependencies. The refactorings worked on the document itself, but never looked outside of the originating

document for any referenced uses to files because the refactoring was added before version 1.4 of SASyLF. Because of this, the referencing proofs were never re-checked after the change was made to the original file.

2.3.1 Rename Proof Module

This refactoring renames the module and the file containing the proof module. This refactoring was already partially working in that it would successfully rename the file in question, but did not locate any references to the file itself that may have needed to be changed. Despite this, no errors were initially detected despite the proof module names no longer matching the expected reference file.

To start, code was added to reanalyze the proof files to make sure that errors were actually detected once the refactoring was executed for a proof module referenced in other files. Once confirmation of the error was noted, the process of manually changing all of the necessary references was needed. In each dependent file, a collection of all references to the file name were gathered, filtering out all names all name references that either:

1. did not match the old name of the changed file, or
2. matched the old name of the changed file, but referred to a different module or document.

It is obvious that only names that match the previous name (prior to the refactored change) should be included. The second criterion is necessary in the event that different packages include proof modules of the same naming scheme, and also happen to be linked in the same proof file. While this may not be a common occurrence or recommended practice,

the distinction is important to make in the event that files that should not be changed are not altered erroneously.

Once all necessary name references are collected for each file, the changes are able to be applied. To preserve the integrity of the file, the changes are performed in reverse order; that is, the affected names located further down the file are changed first, working upwards. Because all changes made are in reference to specific text regions, any changes to text regions means that regions further down in the document will no longer reference the correct areas that need to be changed. Doing so in this fashion prevents changes made earlier in the file to affect the text regions later in the file, preventing any additional corrections that may have been necessary.

2.3.2 Move Proof File

This refactoring moves the file containing the proof module to a different package. Similarly to the Rename Proof Module refactor, this was already partially working – the intended refactoring worked as expected, but did not make any changes to any other files that referenced it. The changes are applied the same way as for the Rename Proof Module refactoring mentioned in the section above.

2.4 Rename Theorem

The new refactoring I introduce to SASyLF is the ability to rename a theorem. It can be applied by highlighting the name of the theorem in its declaration and right-clicking and selecting *Rename Theorem* from a newly created *Refactor* menu. When the refactoring is applied, the original name of the theorem is changed, and the changes are also applied to

all references both within the originating document and in any proof modules dependent on the document.

Unlike for the Rename Proof Module and Move Proof File refactorings, there are no existing Eclipse classes that provide most of this work, so a number of original classes were created to handle the refactoring.

2.4.1 RenameTheoremHandler

This class is what executes the start of the refactoring. It collects all of the information necessary for the RefactoringContext object, and then passes it to the RenameTheoremWizard class to kick off the work of performing the refactoring.

2.4.2 RefactoringContext

This class is a container for required information to perform the refactoring. It is given the original name of the theorem, the proof file containing the theorem, and the proof itself as information from the RenameTheoremHandler. It then obtains the proof document associated with the file and finds the associated theorem in the AST.

2.4.3 RenameTheoremWizardPage

This class is mainly used by RenameTheoremWizard to collect the new name of the theorem from the user. It autofills the old name into a text field, and the user can type in the new desired name. Before performing the refactoring, several checks are performed to make sure that the name is still syntactically valid: the field isn't blank, doesn't contain any invalid characters. It also checks that the new theorem name doesn't already exist in the document,

and that the theorem we're trying to rename is the original definition and not a reference anywhere else in the proof system.

2.4.4 RenameTheoremWizard

This is the class that does the grunt of the work of the refactoring. After the new theorem name has been validated and accepted by the RenameTheoremWizardPage, the refactoring can now officially begin.

First, the originating proof document is modified. The declaration of the theorem is given the new name, and then the document is searched for all text regions that contain references to the old name. This extra step was not necessary for the Rename Proof Module or Move Proof File refactorings since the modules did not contain multiple references to itself, but theorems can be referenced multiple times in the same document.

After the proof containing the declaration has been changed, the process can begin applying the same process to all document dependencies. Each dependent file linked must be checked for any references to the old theorem name, and if any are found, the text region replaces the old theorem name with the new user-defined name. This application continues until all linked documents have been checked for references to the original theorem definition.

Chapter 3

Obstructions

As is the case with most major projects, problems were typically encountered along the way. This chapter breaks down the two most common issues that required the most attention to deal with.

3.1 Eclipse

The vast majority of the difficulties in this project has originated from exploring (and more commonly, working around) Eclipse's plugin framework. It is notoriously not widely documented, and the woefully little that can be found online tends to be outdated by several years. Working with such a framework required deciphering general explanations of code as well as splicing together snippets of code scattered across multiple different web forums.

One example of this was getting the new refactoring to show up in the first place. Adding a new "Refactor" menu to appear on right-click was necessary so that the refactoring could be accessed and initiated, a very small but necessary task. However, the code needed to add it was not as straightforward as it would have appeared to be. This was complicated by the fact that adding this menu wasn't necessary for the Rename Proof Module and Move Proof File refactorings – they seemingly added the necessary menu items on their own once they were implemented. After scouring numerous forums for queries on similar issues, it finally became apparent that the previously-mentioned refactorings have support backed by Eclipse

that the new refactoring did not, so a different approach needed to be taken for it. Once this was determined, the code could be added and the menu was able to accessed as desired; a lot of time wasted over such a tiny detail.

3.2 Bugs

An additional complication was encountering bugs in SASyLF that only surfaced as work progressed further. While rather minor, they tended to be roadblocks to making progress on more than a few occasions. As such, they were also rather difficult to debug and fix.

The most frustrating occurrence of this involved obtaining the necessary documents that needed to be altered. The proof dependencies themselves were already being kept track of, but to actually make the changes the text of the proof file itself needed to be changed. For unclear reasons, occasionally when attempting to retrieve the proof document it would return null rather than the document, causing null pointer exceptions to occur and prevent the refactoring from executing fully. The most confusing aspect of this was that it was rather inconsistent on when this would happen – it seemed to depend on what theorem I was trying to test it on. In some cases, a referencing document would return null for one theorem, but when testing on another theorem it would be supplied just fine, but then *another* referencing file would return null instead. Code-wise, this was probably the biggest hinderance on the implementation of the new refactoring that did not show up during the amendments to the other previously-existing refactorings.

Chapter 4

Concluding Remarks

4.1 Conclusion

Refactoring proofs may not be a commonly-done practice both in practice and research-wise, but is nevertheless a very useful tool that can be utilized in specific circumstances. Low-level refactorings can be rather easily modified to be used by proof assistants, especially if said refactoring only affects one proof document. Once some initial refactoring can be implemented and applied, adapting other similar refactorings should follow a similar pattern.

Maintaining the document dependencies played an important factor in ensuring that linked proof modules are able to be modified along with the original document. Its main benefit was in cutting down the work needed to examine each file so that only the linked proofs needed further examination. Additionally, this allowed for the ability to access the proof documents in question without much effort so that the necessary files could be changed quickly.

Proof refactoring also largely depends on the framework of the software supporting said proofs. In SASyLF's case, the AST based on each proof was determined by analyzing each proof file, so the changes needed to be applied to the document itself rather than the AST in order for it to be updated correctly. A key factor in doing so for text documents is ensuring that the changes are made in the correct order, as modifications earlier in a text document ripple down to affect all remaining text left in the document. Doing so in a non-optimal way

creates errors further along in the process if the affected text regions occurring later in the document are not updated accordingly.

4.2 Future Work

Now that the general groundwork for refactoring has been laid, I'd love to continue with implementations of additional refactorings in SASyLF. A very similar approach could be applied to the renaming of judgments or module names, where linked proof modules will need to be changed and reparsed. Perhaps even definitions within lemmas and theorems themselves could be done on a much smaller scale; rather than tracking uses across different proof documents, only the lemma or theorem in which it was defined would need to be checked.

4.2.1 Switch Lemma or Theorem

A feature I had been hoping to introduce as part of my thesis was the ability to switch a definition from either a lemma to a theorem, or vice versa (from a theorem to a lemma). Lemmas and theorems are technically the same construct in SASyLF – regardless of whether they are a lemma or a theorem, they are defined the same way, so they differ in kind only. After the proof has been parsed, this information is located in the abstract syntax tree.

Treating lemmas and theorems the same makes refactoring in some ways easier, but also provides more complications that simple theorem renaming did not encounter. As far as making things easier, when locating the lemma or theorem in question, we only need to search through all theorem declarations located within the proof module. For the rename refactoring, the name of the theorem was highlighted, giving a starting point for locating the

name of the desired theorem to change in the document. If we want to go a step further and swap it from a lemma or a theorem, we not only need to know where in the document the definition is, but also where in the document the keyword "lemma" or "theorem" is located so we can replace it properly. Thus, we would need to first locate the location of the name, and then backtrack so that we can find the keyword and replace it. This process would not only need to be done for the original name of the lemma/theorem, but also for every reference to it within the proof document and in all references in other proof documents.

Aside from the extra work needed to find and replace the lemma or theorem keywords, it may call into question how to initiate this refactoring as well. As mentioned above, the rename refactoring requires the user to highlight the name of the defined lemma/theorem. If we wanted to switch to a lemma or theorem, should we still highlight the name? Would we need to include the lemma/theorem keyword as well? The desire behind refactoring is make these kinds of changes easier and quicker for the user, so ideally the initiation of the refactoring process should be as intuitive as possible. However, this may mean that the work required on the back-end becomes more complex so they can be sorted out properly.

Bibliography

[ADL10] David Aspinall, Ewen Denney, and Christoph Lüth. Tactics for hierarchical proof. In *Mathematics in Computer Science*, volume 3, page 309–330, 2010.

[ASS08] Jonathan Aldrich, Robert J. Simmons, and Key Shin. SASyLF: An educational proof assistant for language theory. In *Functional and Declarative Programming in Education*, 2008.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Longman, Reading, Massachussetts, USA, 1999.

[HHP93] Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. In *Journal of the ACM*, volume 40, page 143–184, 1993.

[WADG11] Iain Whiteside, David Aspinall, Lucas Dixon, and Gudmund Grov. Towards formal proof script refactoring. In *Lecture Notes in Computer Science*, volume 6824, page 260–275, 2011.

[Whi13] Iain Johnston Whiteside. *Refactoring Proofs*. PhD thesis, University of Edinburgh, 2013.