A Formal Model of Polyglot Programs

680062203

Declaration: I certify that all material in this dissertation which is not my own work has been identified.

Abstract

In this project, a formal model of polyglot programs is being developed. Polyglot programs are programs which are valid in multiple languages. Existing literature surrounding polyglot programs is reviewed, and key concepts surrounding the formalisation of programming languages are also detailed. This formalisation focuses on C/Python polyglots (polyglots valid in both C and Python). Firstly, a simplified model of programs valid in one programming language is created. This model is then used to define input programs and a template for C/Python polyglot payloads. By feeding the input programs into the template, valid C/Python polyglot payloads can be generated. This formalisation of polyglot payloads is the first of its kind.

Contents

1 Introduction

According to a CSIS report from 2018, the cost of cyber crime is estimated to be up to 0.8% (or up to $600B) of the global gross domestic product (GDP) [1]. Thus, it is unsurprising that the"National Cyber Security Strategy 2016-2021" of the UK Government emphasises the need for improving the resilience to cyber attacks. The technical cause for the majority of cyber attacks and cyber crimes are software vulnerabilities - in other words, software bugs that can be exploited by attackers [2].

In this project, contributions will be made to address this problem by developing a technique for finding complex security vulnerabilities that cannot be found by current state-of-the-art security testing approaches. The focus will be on multi-staged security vulnerabilities, i.e. vulnerabilities that require an attacker to circumvent several layers of protection. This requires the generation of polyglot payloads that can be "bootlegged" through a first level of protection to then exploit a vulnerability deep inside the attacked system.

In more detail, the objectives of the project are to

1. develop a formal model of C/Python polyglot payloads,
2. develop a systematic approach for the generation of C/Python polyglot payloads, and
3. evaluate the approach on a real system.

This report is organised as follows: Section 2 will provide the background knowledge needed to complete this project. Section 3 provides a summary of the project specification. A brief summary of the usage of Isabelle can be found in Section 4. Section 5 will explain the development methodology used in this project. Sections 6 to 10 will demonstrate the different development phases. A brief comparison between this project and other related works is found in Section 11. Subsequently, Sections 11 and 12 will be used to evaluate and critically assess this project. Finally, the project report is summarised in Section 13.

2 Summary Of The Literature Review

In the following review, existing research concerning polyglot payloads will be considered. Subsequently, as this research project concerns the formalisation of polyglot payloads, it is critical to review literature surrounding the formalisation of programming languages. The formalisation of polyglot payloads is concerned with the semantics and specification of programming languages, and is important for deductive and testing-based verification [3]. Additionally, it is important to consider the core features and logic behind programming languages before one develops a model of a polyglot payload. When one defines a polyglot program to be one that is valid in multiple languages, abstract features including comment and documentation syntax can allow or hinder the interleaving of code from multiple languages, such that the same file is valid in multiple languages.

2.1 Polyglot Payloads

The most notable and relevant article was written by Magazinius et al. in 2013 and is titled "Polyglots: Crossing Origins by Crossing Formats". In this research, the main focus was on polyglot payloads. They demonstrated how certain file formats, such as the PDF and GIF formats, are particularly useful when creating polyglot payloads. Furthermore, they demonstrated multiple possible attacks utilising these payloads, including content-smuggling attacks and PDF-based injection attacks. The vulnerabilities of PDF readers of multiple browser applications to these attacks were considered. Additionally, multiple cloud-based storage servers were also identified as vulnerable to content smuggling attacks. Though the article was published in 2013 and multiple PDF readers and browser applications have been updated so as to no longer be vulnerable to these attacks (or at least, less vulnerable), it is still possible for one to use a polyglot payload in an attack in other contexts. This article is particularly useful in that it provides one with a clear explanation of what polyglot payloads are and why they are dangerous, thus inspiring this research project [4].

2.2 The Significance Of Syntax And Of Formal Definitions

The syntax of a programming language determines the form and structure of programs, whether said programs are valid or invalid in said language, and the program's internal grouping structure. Thus, a precise definition of a language's syntax is needed before one can then determine its semantics. In this project, it is important to distinguish between concrete and abstract syntax.
The concrete syntax of a programming language is concerned with text and parsing, and determines which strings are valid programs. For those valid programs, a concrete syntax tree or a parse tree is then created, indicating the grouping of the program. It is recommended that the grammar for concrete syntax be of an unambiguous nature, so that each valid program has a unique concrete syntax tree. If the grammar is ambiguous, then precedence rules should also be provided, in order to provide valid programs with unique trees.
The abstract syntax of a programming language is concerned with the structure of programs, and is usually represented using trees. In abstract syntax trees, nodes represent certain constructors. Branches represent arguments of said constructors. Thus, a node has no branches represents a constructor with no arguments. Nodes can also include numbers, identifiers, truth-values and mathematical operations [5].

2.3 Concrete vs Abstract Syntax

Concrete syntax is usually more complex than abstract syntax, as with the former, one makes distinctions between certain concrete nonterminal symbols. This is not the case for abstract syntax, and the grouping together of similar nonterminal symbols allows abstract syntax trees to become more general than concrete syntax trees. As well as making abstract syntax more simple, this generalisation also allows abstract syntax to be used as an interface between syntax and semantics. For more information concerning the formalisation of programming languages, please see [6] and [5].

3 Summary Of Project Specification

Within this project, there are multiple objectives to be fulfilled. Following from the literature review, a formal model of C/Python polyglot payloads will be developed using Isabelle/HOL, a generic proof assistant [7]. This could potentially include a general model of a program and of programming languages to facilitate the creation of a model of a polyglot program, and include key features, such as executable code and code or text hidden within a comment or documentation section. Additional features may be taken into consideration within the model, depending on the findings surrounding the key features of certain programming languages, and how they interact with each other to facilitate or hinder the creation of a polyglot payload.

3.1 Functional and Non-Functional Requirements

Below, a list of the functional requirements is given:

- An abstract representation of programs
  - A basic structure for representing programming languages. This includes a model of the comment structure. It also includes a general structure to be used for defining the concrete syntax for the following features found in most significant programming languages, including block comments, line comments, string literals, and an identifier for the programming language being defined.
  - A model of a program block, which supports the features indicated within the general structure or record
  - A function for representing programs as strings, for code evaluations and debugging
  - Tests for program validity
  - Function that removes comments from a program
  - Proofs of functions
  - Examples to demonstrate the formalisation of programs
- Representation of polyglot programs as programs valid in multiple languages
  - Examples to represent polyglot programs
  - Proofs of important properties regarding the polyglot programs
  - Tests for polyglot validity
  - A function for representing polyglot programs as strings, for code evaluations and debugging
  - Tests for polyglot program validity in the languages provided
  - Proofs of important characteristics of polyglot programs and related functions
  - Examples to demonstrate the formalisation of polyglots
- The formalisation is to be used to generate polyglot payloads.

In terms of non-functional requirements, Isabelle/HOL is to be used to write the the formalisation and the proofs.

3.2 Evaluation Criteria

- Does the formalisation includes all of the necessary components to model programs?
- Are there mechanisms in place to determine to determine whether a program is valid in a given language?
- Are there mechanisms in place to determine which programs can create valid polyglot programs?
- Are the programs created using the formalisation valid in multiple given languages?
- Are there sufficient proofs to show that the formalisation will always result in valid polyglots?

4 Isabelle/HOL

Before one can create a model of polyglot programs, one must understand the key features of Isabelle. Firstly, to declare variables, one writes $variable_{name}$ :: $variable_{type}$, where $variable_{type}$ can be a base type, such as an int, or a user-defined datatype. An example of a user-defined datatype can be seen below:

```
datatype program = SKIP
                 | block string                        ("ℬ _")
                 | linecomment  string                 ("⟨ _ ⟩")
                 | blockcomment string program string ("⟪ _ _ _ ⟫" [60, 60] 60)
                 | semi program program                ("_ ; _"  [59, 60       ] 59)
```

Here, the datatype program can be a SKIP operation, a block of code, a line comment, a block comment string which contains another variable of type 'program', or multiple variables of type 'program' separated by a semi-colon. If a variable does not follow the format of a certain datatype, then an error message appears. An example of a valid instantiation of a variable of this type can be seen below:

```
definition hello_world_c where
  ‹hello_world_c = (⟨ ‹// Hello World in C› ⟩ ;
                    ℬ ‹#include<stdio.h>⏎
                        void main () {⏎
                          printf("Hello World!");⏎
                        }⏎›)›
```

Here, the variable hello_world_c demonstrates a simple Hello World program, which is valid in C. The usage of denotations to make this instantiation more concise is possible through the definitions of the denotations used here within the program datatype. At the bottom, one can see that this variable is indeed of type 'program':

```
consts
  hello_world_c :: "program"
```

The datatypes can also be used to create functions. One can see an example of a function below:

```
fun string_of_program :: "program ⇒ char list" where
  "string_of_program  SKIP = ‹›"
| "string_of_program (block b) = b"
| "string_of_program (blockcomment bcomment prog ecomment) = bcomment@(string_of_program prog)@ecomment"
| "string_of_program (linecomment comment) = comment"
| "string_of_program (semi p0 p1) = (string_of_program p0)@(string_of_program p1)"
```

This particular function allows one to represent a program as a string. Depending on the nature of the variable of type 'program', the function executes differently, to allow the program to be represented as a string. It is important to list both the input and output datatypes after the two colons, in order to ensure that the function is well-typed. One can then use functions to construct proofs. An example of a proof can be seen below:

```
lemma prove_rmskip: "rmskip (SKIP) = SKIP"
  by simp
```

Here, a lemma $prove_{rmskip}$ has been created, which aims to prove that applying a function $rmskip$ to a variable $SKIP$ results in $SKIP$. The statement to be proven is contained within quotation marks, and the methods of proving the statement are given underneath. In this case, as Isabelle can already infer that this will always be true regardless of other functions, only $simp$ is required to prove the lemma. Such simple lemmas are often required before one is able to successfully construct more complex proofs. More information on the syntax and usage of Isabelle can be found in Nipkow and Klein's "Concrete Semantics" [7].

5 Development Method

For this project, the most suitable development method is Test Driven Development (TDD). In TDD, one must write tests for some given code. If the test fails, then the code is rewritten or more code is written, until the test passes. Once all tests have passed, then the code is considered to be written completely [8].

Firstly, the functional requirements are converted to proof statements, in order to clarify how the model should be designed. Subsequently the required datatypes, models and definitions are designed. This code will then be tested using the 'value' keyword in Isabelle. This keyword will signal whether the correct output is produced or not. In the latter case, the code is rewritten until the correct output is produced. Resulting polyglots are then generated using the model, a set of programs, which fulfil the requirements as stated in the concrete syntax of certain given languages. Once the polyglot programs are shown to be valid in all given languages, as well as well-formed, the proof statements are then converted into lemmas. The functions are then tested and rewritten until all lemmas are proven to be true. Given that this final condition holds, one can consider polyglot programs to have been successfully formalised, and we have successfully proven that the model will always produce valid polyglot programs, given a set of programs which are valid in their respective languages.

6 Proof Statements

Now, we convert the requirements into proof statements. We do this before formalising polyglot payloads, to provide clarity into how the model should be designed, so that the proof statements hold.

6.1 Polyglot Programs Are Valid Only If They Have The Same String Representation And Are Valid In All Given Languages

As noted earlier, a polyglot program is a program that is valid in multiple languages. To be certain that this is the case, we design the following proof statement (which is specific to the context of this project):

"Given a valid C program and a valid Python program, the resulting composition will be valid in both C and Python".

For this statement to hold, we need to design a function that checks the validity of C and Python programs, and a function that checks the validity of the C/Python polyglot. Both functions should return true if the programs and polyglots are valid, and false if they are not. A function that is able to generate a valid polyglot given two valid programs must also be created. If either of the programs are invalid, then no polyglot should be returned.

The same polyglot program will not necessarily be interpreted or executed in the same way by each given programming language. In one language, certain parts may be interpreted as being part of a comment structure, whilst that same part may be executed in another language. Therefore, the representation of the syntax of a polyglot in each of its languages will vary. A function that provides a string representation for each of these depictions must be created. This function will be used within the function that validates a polyglot program.

This is the most important and overarching statement that needs to be proven. However, given the functional requirements of this project, there are other statements that should be proven as well. These are primarily concerned with functions that will be required later on in order to prove the statement in 6.1. One of these statements are provided below.

6.2 Removing comments from two successive statements is the equivalent to removing comments from each of the statements, before joining them and then removing the comments from the result.

To ensure that a polyglot payload can be produced from two programs written in two different languages, it is best to implement a function that removes comments from the input programs. Programs can be seen as - on a highly simplified level - a series of successive statements that need to be executed. Therefore, when removing comments from a program, the function needs to be able to remove comments from each statement of the program. The mathematical equivalent of statement 6.2 is: $(rmcomments\ (p1;\ p2)) = (rmcomments\ ((rmcomments\ p1);\ (rmcomments\ p2)))$. From this, we can use these proof statements to produce a clearer design for the model of a polyglot payload.

7 Model design

7.1 The Program Structure

We begin by designing the program model. Each 'program' can be seen as a section of code that can be sequenced together to produce a larger program.

```
datatype program = SKIP
                 | block string                        ("ℬ _")
                 | linecomment  string                 ("⟨ _ ⟩")
                 | blockcomment string program string ("⟪ _ _ _ ⟫" [60, 60] 60)
                 | semi program program                ("_ ; _"  [59, 60        ] 59)
```

We abstract away from most parts of the concrete syntax of programming languages. In this model, we only consider the most relevant traits for creating polyglot programs. A program can be a *block* of executable code, a line comment, a block comment, SKIP (this will be used to remove comments later on), or two programs joined together.

7.2 Language Record

To be able to validate any program, we need to design a model for the language the program is supposed to be valid in.

```
record pl = identifier::string
            block_comment:: "block_comment list"
            nested_blockcomments :: bool
            line_comment::"string list"
            string_literals::"string_literals list"
```

The *pl* record can be used to define a new programming language. Here, we do not consider every single aspect that can affect the validity of a program, such as whether variables are correctly instantiated, arithmetic errors, resource errors, and more. Only the most relevant factors needed to combine two programs to create a valid polyglot will be considered in this formalisation. These include the programming language's name (or identifier), the block and line comment notation used, and the string literal notation.

```
record block_comment =
        start ::string
        "end" ::string
        forbidden :: "string set"
```

Block comments often have differing strings at the beginning and ending of the comment. Certain (but not all) types of block comments, even within the same language, may forbid certain characters. If a language has multiple types of block comments, it is often forbidden to mix the start and end notations from the differing block comment types. With this record, we can define separate types of

block comments, as well as the characters that the different types may forbid.

7.3 Validity Of A Program

As programs can take different shapes (as seen by the *program* datatype), it is important to consider the validity for of each of these 'shapes'.
If the program takes the form of SKIP or a block of code, the program should always be true.
For both block and line comments, the comments can only be valid if the commenting notation used is allowed by that language. For a block comment, the allowed start and end notation should be at the start and end of the comment respectively. Also, no forbidden characters should be present within the block comment. For a line comment, the allowed start notation should be at the start of the comment.
If the program consists of two programs sequenced together, then this program can only be valid if the two 'sub-programs' are themselves valid.

7.4 Polyglot Program Definition

Now that we can define programs that are valid in one language, we can now design polyglot programs. Different approaches were considered when designing polyglots, but the simplest and most successful approach can be seen below:

```
type_synonym polyglot = "(pl × program) set"
```

In this project, we model polyglot programs as a set of programs associated with different programming languages. This will allow us to denote certain parts of the same polyglot as being a comment in one language, and a code block within another, for example. Therefore, not only is this method of modelling polyglot programs quite simple, but it also allows us to demonstrate the ways two different language interpreters may interpret the same program.

7.5 Validity Of Polyglot Programs

From this, we can stipulate that to create a valid polyglot, each input program must be valid in the given language. Also, it must be possible to combine programs in such a way that the polyglot itself is valid in all given languages. This would mean that each version of the polyglot in the set must be valid in its given language. The generation of certain types of polyglots (including C/Python polyglots) could be facilitated through the usage of a template wherein individual programs are inserted in the correct locations.

8 Development Of Functions And Definitions

We must now use functions and definitions to implement the designed functionality for the model of a program. The following functions can be found in program.thy. Below we can see the function validP, which has been created according to the requirements specified in section 7.3. Something that should be noted is the ⊨ notation seen in the first line in the figure below. This simply allows us to write that a given program is valid in a programming language. For example, "C ⊨ c_prog" means that c_prog is valid in C.

```
fun validP ("_ ⊨ _") where
  "validP _  SKIP = True"
| "validP _  (block _) = True"
| "validP pl  (blockcomment bcomment content ecomment) = (∃ bc ∈ set(block_comment pl).
                                                  (prefix (start bc) bcomment)
                                                ∧ (prefix (rev (end bc)) (rev ecomment))
                                                ∧ (¬ (∃ f_str ∈ (forbidden bc).
                                                    sublist (f_str)
                                               (rm_start_end_comment bc bcomment content ecomment))))"
| "validP pl (linecomment comment) = (∃ c ∈ set(line_comment pl). prefix c comment)"
| "validP pl (semi p0 p1)  =  ((validP pl p0) ∧ (validP pl p1))"
```

The definition below is a simple solution which replaces the ‘ symbol with a full-stop. This stops programs from being printed character by character, which can make it hard to interpret the output.

```
definition "hackish_non_printable_CHRs = (map (λ c . if c = CHR 0x22 then CHR ''.'' else c))"
```

To be able to prove statement 6.2, we must write a function that removes comments from a program. To do this, we first create a function that converts comments to SKIP. Blocks of code are not converted. If two programs are sequenced together, then *comments2skip* must be performed on each program, before they can be joined together using semi.

```
fun comments2skip::"program ⇒ program" where
  "comments2skip  SKIP = SKIP"
| "comments2skip (block b) = block b"
| "comments2skip (blockcomment _ _ _) = SKIP"
| "comments2skip (linecomment _) = SKIP"
| "comments2skip (semi p0 p1 ) = semi (comments2skip p0) (comments2skip p1)"
```

The function below then removes unnecessary SKIP operations. There are two cases: the case where the program consists of two programs sequenced together using semi, and the base case where this is not the case. In the former case, if p0 is SKIP, then *rmskip* p1 is returned. Likewise, if p1 is SKIP, then *rmskip* p0 is returned. For the base case, the program itself is returned.

```
fun rmskip::"program ⇒ program" where
   "rmskip (semi p0 p1 )   = (let lh = rmskip p0 in
                              (let rh = rmskip p1 in
                                  (if lh = SKIP
                                  then rh
                                  else (if rh = SKIP
                                       then lh
                                       else semi lh rh))))"
 | "rmskip s                = s"
```

The function below uses the two previously-described functions to remove comments in a program. It does this by first using *comments2skip* to convert all comments to SKIPs, which are then removed using *rmskip*.

```
definition rmcomments::"program ⇒ program" where
  "rmcomments = rmskip o comments2skip"
```

Now, we can use functions and definitions to implement functionality in the polyglot model. The following functions and definitions can be found in polyglot.thy. As polyglot programs are being modelled as multiple programs (each associated with a programming language), it is important to ensure that these programs have the same string representation - in other words, that we are concerned with the same payload.

```
fun string_of_program where
  "string_of_program  SKIP = ‹›"
| "string_of_program (block b) = b"
| "string_of_program (blockcomment bcomment prog ecomment) =
                                                  bcomment@(string_of_program prog)@ecomment"
| "string_of_program (linecomment comment) = comment@‹␤›"
| "string_of_program (semi p0 p1) = (string_of_program p0)@(string_of_program p1)"
```

This function is later used in the *valid_poly* definition, as seen below. This function allows us to answer whether a given polyglot is valid or not. This check ensures that: each program in the set of programs is valid in its individual language, that the string representation of all programs in the set is syntactically identical, and that the set of programs contains a program for each programming language specified in P.

```
definition "valid_poly pls P = (
                                ( ∀ p ∈ P.((fst p) ⊨ (snd p))
                               ∧ (∀ p' ∈ P.(string_of_program (snd p) =
                                 string_of_program (snd p'))))
                               ∧ pls ⊆ fst `  P

                               )"
```

In this project, due to time constraints, we will be focusing on creating valid C/Python polyglots. For this task, we will create two templates, which can be used to generate these polyglots. One template will model the Python version of the polyglot, and the other will model the C version of the polyglot. Below is the Python version of the template. To ensure that the polyglot will be valid, any comments within the input programs *python_prog* and *c_prog* will be removed using *rmcomments*.

```
definition compose_python_poly :: "program ⇒ program ⇒ program" where
‹compose_python_poly python_prog c_prog = linecomment ‹#if 0›;
                                    rmcomments python_prog;
                                    linecomment ‹#endif›;
                                    linecomment ‹#if 0›;
                                    blockcomment
                                    ‹""" "›
                                    (linecomment ‹#endif›;
                                    rmcomments c_prog;
                                    linecomment ‹#if 0›)
                                    ‹" """›;
                                    linecomment ‹#endif›
                                    ›
```

Below is the C version of the template.

```
definition compose_c_poly :: "program ⇒ program ⇒ program" where
‹compose_c_poly python_prog c_prog = blockcomment
                            ‹#if 0⏎›
                              (rmcomments python_prog)
                            ‹#endif⏎›;
                          blockcomment
                            ‹#if 0⏎›
                              (block ‹""" "›)
                            ‹#endif⏎›;
                          rmcomments c_prog;
                          blockcomment
                            ‹#if 0⏎›
                              (block ‹" """›)
                            ‹#endif⏎›
                        ›
```

These two templates are then used within the *mk_c_python_poly* definition to create the C/Python polyglot. The definition first checks that *c_prog* is a valid C program and that *python_prog* is a valid Python program. If the programs are not valid, then we can declare that no valid polyglot program exists for these two input programs. However, if they are valid, then the definition creates a polyglot wherein the Python and C versions of the polyglot are associated with Python and C respectively.

```
definition mk_c_python_poly:: "program ⇒ program ⇒ polyglot" where
"mk_c_python_poly c_prog python_prog = (if (C ⊨ c_prog) ∧ (Python ⊨ python_prog)
                                    then {
                                          (Python, compose_python_poly python_prog c_prog),
                                          (C, compose_c_poly python_prog c_prog)
                                         }
                                    else {})"
```

Now that the functionality has been implemented, we can begin the testing phase.

9 Testing

Throughout the development of the proof statements into lemmas, the functions will be thoroughly tested. If the lemmas prove true, then we can be certain that the programs (including the polyglot payloads) have been suitably tested to be valid. Before this, we carry out unit and integration tests in Isabelle, and use the templates to generate polyglot payloads. This allows us to see if any of the functions and definitions developed do not give the correct output and thus need to be redeveloped.

9.1 Unit Testing

Before any tests are performed, we must first establish our languages and input programs. Below, we have two (highly-simplified) grammars for C and Python. As discussed earlier, the focus of these models is on the comment notation used by the two languages. Other details, such as whether any other syntax errors are present in the program, are assumed to be taken care of by the developer. As such, they are not considered in this abstraction.

```
definition ‹C = ⦇
         identifier = ‹C›,
         block_comment= [
           ⦇ start =  ‹/*›, end=  ‹*/›, forbidden={} ⦈,
           ⦇ start =  ‹#if 0⏎›, end=  ‹#endif⏎›, forbidden={‹#›} ⦈
         ],
         nested_blockcomments = True,
         line_comment = [(‹//›)],
         string_literals=[⦇ str_start =  ‹"›, str_end=  ‹"›, escape_sequences={‹\›} ⦈]
       ⦈›
definition ‹Python = ⦇
         identifier = ‹Python›,
         block_comment= [ ⦇ start =  ‹""" "›, end=  ‹" """›, forbidden={‹"""›}⦈ ],
         nested_blockcomments = True,
         line_comment = [( ‹#›)],
         string_literals=[
            ⦇ str_start =  ‹"›, str_end=  ‹"›, escape_sequences={‹\›} ⦈,
            ⦇ str_start =  ‹'›, str_end=  ‹'›, escape_sequences={‹\›} ⦈,
            ⦇ str_start = ‹r"›, str_end=  ‹"›, escape_sequences={} ⦈,
            ⦇ str_start = ‹r'›, str_end=  ‹'›, escape_sequences={} ⦈
         ]
       ⦈›
```

The model below represents a program that outputs “Hello in Python” when interpreted by a Python interpreter.

```
definition hello_world_python where
  ‹hello_world_python = (⟨ ‹# Hello World in Python› ⟩ ;
                         ℬ  ‹print("Hello in Python")⏎› )›
```

Similarly, the model below represents a program that outputs “Hello in C” when compiled by a C compiler.

```
definition hello_world_c where
  ‹hello_world_c = (⟨ ‹// Hello World in C›⟩ ;
                    ℬ ‹#include<stdio.h>⏎
                        void main () {⏎
                          printf("Hello in C");⏎
                        }⏎›)›
```

Here, we demonstrate the unit test for one of the functions, namely the *validP* function. Using value, we can check whether *validP* is correctly validating programs. *hello_world_python* and *hello_world_c* should only be valid in Python and C respectively. If the program is valid, then the result is ‘true’. If not, then the result is ‘false’. Below each value statement, the output is displayed.

```
value "Python ⊨ hello_world_python"
"True"
  :: "bool"
value "C ⊨ hello_world_python"
"False"
  :: "bool"
value "C ⊨ hello_world_c"
"True"
  :: "bool"
value "Python ⊨ hello_world_c"
"False"
  :: "bool"
```

As seen above, each program is validated correctly. Thus, it is clear that *validP* is working correctly, and no redevelopment is required for this function. Once all functions and definitions have been tested and redeveloped, we can proceed with the integration testing.

9.2 Integration Testing

We begin by testing that each function correctly gives the correct output in the unit tests. For example, we ensure that each program created for the unit tests is correctly validated. After this, we create a polyglot using the *mk_c_python_poly*. If this function always produces a valid polyglot payload, we can be sure that all of the functions have been successfully integrated together.

When creating the original template to be used in *compose_python_poly*, there were certain small errors that needed to be corrected. For example, the original template accidentally added too many new-line characters. This meant that the string representation of the C and Python versions of the polyglot were unequal (thus going against statement 6.1). This also prevented the creation of valid polyglots when using *mk_c_python_poly*. To remedy this, the new-line characters had to be removed from the template. The old version of the template can be seen below:

```
definition compose_python_poly :: "program ⇒ program ⇒ program" where
‹compose_python_poly python_prog c_prog = linecomment ‹#if 0⏎›;
                                          rmcomments python_prog;
                                          linecomment ‹#endif⏎›;
                                          linecomment ‹#if 0⏎›;
                                          blockcomment
                                          ‹""" "›
                                          (linecomment ‹#endif⏎›;
                                          rmcomments c_prog;
                                          linecomment ‹#if 0⏎›)
                                          ‹" """›;
                                          linecomment ‹#endif⏎›
                                          ›
```

Now that the function has been redeveloped, we can produce a valid C/Python polyglot using *hello_world_python* and *hello_world*¸ Below, we can see the correct output of this function.

```
value "mk_c_python_poly hello_world_c hello_world_python"
"{((|identifier = ''Python'',
      block_comment =
        [(|start = [CHR 0x22, CHR 0x22, CHR 0x22, CHR '' '', CHR 0x22],
            end = [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22],
            forbidden = {[CHR 0x22, CHR 0x22, CHR 0x22, CHR 0x22]}|)],
      nested_blockcomments = True, line_comment = [''#''],
      string_literals =
        [(|str_start = [CHR 0x22], str_end = [CHR 0x22], escape_sequences = {[CHR 0x5C]}|),
         (|str_start = [CHR 0x27], str_end = [CHR 0x27], escape_sequences = {[CHR 0x5C]}|),
         (|str_start = [CHR ''r'', CHR 0x22], str_end = [CHR 0x22], escape_sequences = {}|),
         (|str_start = [CHR ''r'', CHR 0x27], str_end = [CHR 0x27], escape_sequences = {}|)]|),
  ⟨ ''#if 0'' ⟩ ; ℬ [CHR ''p'', CHR ''r'', CHR ''i'', CHR ''n'', CHR ''t'', CHR ''('', CHR 0x22, CHR ''H'', CHR ''e'',
                     CHR ''l'', CHR ''l'', CHR ''o'', CHR '' '', CHR ''i'', CHR ''n'', CHR '' '', CHR ''P'', CHR ''y'',
                     CHR ''t'', CHR ''h'', CHR ''o'', CHR ''n'', CHR 0x22, CHR '')'',
                     CHR ''⏎''] ; ⟨ ''#endif'' ⟩ ; ⟨ ''#if 0'' ⟩ ; ⟪ [CHR 0x22, CHR 0x22, CHR 0x22, CHR '' '',
        CHR 0x22] (⟨ ''#endif'' ⟩ ; ℬ [CHR ''#'', CHR ''i'', CHR ''n'', CHR ''c'', CHR ''l'', CHR ''u'', CHR ''d'',
                                       CHR ''e'', CHR ''<'', CHR ''s'', CHR ''t'', CHR ''d'', CHR ''i'', CHR ''o'',
                                       CHR ''.'', CHR ''h'', CHR ''>'', CHR ''⏎'', CHR ''⏎'', CHR '' '', CHR '' '',
                                       CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                       CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                       CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                       CHR '' '', CHR '' '', CHR ''v'', CHR ''o'', CHR ''i'', CHR ''d'', CHR '' '', CHR ''m'',
                                       CHR ''a'', CHR ''i'', CHR ''n'', CHR '' '', CHR ''('', CHR '')'', CHR '' '',
                                       CHR ''{'', CHR ''⏎'', CHR ''⏎'', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                       CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                       CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                       CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
```

```
                                    CHR '' '', CHR ''p'', CHR ''r'', CHR ''i'', CHR ''n'', CHR ''t'', CHR ''f'',
                                    CHR ''('', CHR 0x22, CHR ''H'', CHR ''e'', CHR ''l'', CHR ''l'', CHR ''o'',
                                    CHR '' '', CHR ''i'', CHR ''n'', CHR '' '', CHR ''C'', CHR 0x22, CHR '')'',
                                    CHR '';'', CHR ''⏎'', CHR '' '', CHR '' '', CHR ''⏎'', CHR '' '', CHR '' '',
                                    CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                    CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                    CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                                    CHR '' '', CHR ''}'',
                                    CHR ''⏎''] ; ⟨ ''#if 0'' ⟩) [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22,
        CHR 0x22] ⟫ ; ⟨ ''#endif'' ⟩),
 ((|identifier = ''C'',
    block_comment =
      [(|start = ''/*'', end = ''*/'', forbidden = {}|), (|start = ''#if 0⏎'', end = ''#endif⏎'', forbidden = {''#''}|)],
    nested_blockcomments = True, line_comment = [''//''],
    string_literals = [(|str_start = [CHR 0x22], str_end = [CHR 0x22], escape_sequences = {[CHR 0x5C]}|)]|),
 ⟪ ''#if 0⏎'' ℬ [CHR ''p'', CHR ''r'', CHR ''i'', CHR ''n'', CHR ''t'', CHR ''('', CHR 0x22, CHR ''H'', CHR ''e'',
                 CHR ''l'', CHR ''l'', CHR ''o'', CHR '' '', CHR ''i'', CHR ''n'', CHR '' '', CHR ''P'', CHR ''y'',
                 CHR ''t'', CHR ''h'', CHR ''o'', CHR ''n'', CHR 0x22, CHR '')'',
                 CHR ''⏎''] ''#endif⏎'' ⟫ ; ⟪ ''#if 0⏎'' ℬ [CHR 0x22, CHR 0x22, CHR 0x22, CHR '' '',
 CHR 0x22] ''#endif⏎'' ⟫ ; ℬ [CHR ''#'', CHR ''i'', CHR ''n'', CHR ''c'', CHR ''l'', CHR ''u'', CHR ''d'', CHR ''e'',
                              CHR ''<'', CHR ''s'', CHR ''t'', CHR ''d'', CHR ''i'', CHR ''o'', CHR ''.'', CHR ''h'',
                              CHR ''>'', CHR ''⏎'', CHR ''⏎'', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR ''v'', CHR ''o'', CHR ''i'', CHR ''d'', CHR '' '',
                              CHR ''m'', CHR ''a'', CHR ''i'', CHR ''n'', CHR '' '', CHR ''('', CHR '')'', CHR '' '',
                              CHR ''{'', CHR ''⏎'', CHR ''⏎'', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR ''p'', CHR ''r'', CHR ''i'',
                              CHR ''n'', CHR ''t'', CHR ''f'', CHR ''('', CHR 0x22, CHR ''H'', CHR ''e'', CHR ''l'',
                              CHR ''l'', CHR ''o'', CHR '' '', CHR ''i'', CHR ''n'', CHR '' '', CHR ''C'', CHR 0x22,
                              CHR '')'', CHR '';'', CHR ''⏎'', CHR '' '', CHR '' '', CHR ''⏎'', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '',
                              CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR '' '', CHR ''}'',
                              CHR ''⏎''] ; ⟪ ''#if 0⏎'' ℬ [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22,
 CHR 0x22] ''#endif⏎'' ⟫)}"
 :: "(pl × program) set"
```

Now that the functions have been redeveloped so a valid polyglot is always produced by *mk_c_python_poly*, we can export and test this code in the command line.

9.3 Code Generation

With the previously-created models, we now have a C/Python polyglot. The last test that needs to be performed to ensure the validity of the polyglot and the models is to run the program using both a Python interpreter and a C compiler. To do this, the outputs of *compose_c_poly* and *compose_python_poly* can be processed using *string_of_program* to provide the String representation. For readability, *hackish_non_printable_chars* is then used to replace quotation marks with full-stops. This prevents the string representation from being printed character-by-character by Isabelle. Now, after the generated polyglot has been properly formatted, it can be tested.

```
#if 0
#Python
print("Hello in Python")
#endif

#if 0
""" "
#endif
#include <stdio.h>
int main(){
  printf("Hello in C\n");
}
#if 0
" """
#endif
```

Below, we can see the output (with names censored out). We can see that the polyglot that was generated results in no errors or warnings when run using a Python 3 interpreter and a C compiler. When run using the Python interpreter, the interpreter only executes the print(“Hello in Python”) instruction. Similarly, when compiled by GCC, the resulting executable file only outputs print(“Hello in C \n”). This is the desired output. As a result, we can verify that the program is a valid C/Python polyglot.

```
[                :~       $ python hello_world_c_python.py
Hello Python
[                :~       $ gcc -Wall -o p1 hello_world_c_python.c
[                :~       $ ./p1
Hello in C
```

Now that we have generated and tested the polyglot payload properly, the proofs statements can now be transformed into lemmas. Each lemma must hold in order to verify that the formalisation has been rigorously proven to be valid.

10 Development of Proofs

10.1 Proving The Validity Of A Polyglot Program

The main statement to be proven is:
"Given a valid C program and a valid Python program, the resulting composition will be valid in both C and Python".

To do this, we first need the following lemma. This simple lemma states that if the sets of x and y are not equal, then x and y are unequal.

```
lemma set_next: "set x ≠ set y ⟹ x ≠ y"
  by(auto)
```

The following proof shows that the C version of the polyglot is valid in C. Below, we can see the overall structure of the proof, which is a proof by induction over the C program. Overall, there are five different cases, each representing the different forms the C program can take. Within each of these main five cases is another proof by induction, but this time over the Python program. For each main case, there are five sub-cases, to represent the different forms the Python program can take.

```
lemma valid_poly_compose_c_poly: assumes "C ⊨ c_prog"
          and "Python ⊨ python_prog"
          and "python_valid_ifdef_c python_prog"
      shows
     "valid_poly {C} {(C, compose_c_poly python_prog c_prog)}"
using assms proof (induction "c_prog")
  case SKIP
  then show ?case  proof (induction "python_prog") [23 lines]
next
  case (block x)
  then show ?case proof (induction python_prog) [39 lines]
next
  case (linecomment x)
  then show ?case proof (induction python_prog) [24 lines]
next
  case (blockcomment x1 c_prog x3a)
  then show ?case  proof (induction python_prog) [23 lines]
next
  case (semi c_prog1 c_prog2)
  then show ?case  proof (induction python_prog) [16 lines]
qed
```

The first case is where the C program is a SKIP operation. The sub-cases where the Python program is a SKIP operation, a line comment, or a block comment, are relatively simple to prove. Here, these intermediate statements are easily simplified using code_simp. To prove the block sub-case, Isabelle adds the definitions of *valid_poly*, *compose_c_poly*, *rmcomments*, *C*, *rm_start_end_comment*, *sublist*, *Python*, and *python_valid_ifdef_c* to its simplifier to infer that, under these conditions, these definitions will always return True. For the semi sub-case, by using the built-in *sublist_append*, this intermediate statement will also hold. In doing this, there are no further sub goals and the proof for this case complete.

```
  case SKIP
  then show ?case  proof (induction "python_prog")
    case SKIP
    then show ?case by code_simp
  next
    case (block x) note * = this
         have "(∀ps ss. [CHR 0x22, CHR 0x22, CHR 0x22, CHR '' '', CHR 0x22] ≠ ps @ CHR ''#'' # ss)
             ∧ (∀ps ss. [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22] ≠ ps @ CHR ''#'' # ss)"
       by(rule conjI, (rule allI, rule allI, rule set_next, (auto)[1])+)
     then show ?case using *
      by(simp  add: valid_poly_def compose_c_poly_def rmcomments_def
                    C_def rm_start_end_comment_def sublist_def Python_def python_valid_ifdef_c_def)
  next
    case (linecomment x)
    then show ?case by(code_simp)
  next
    case (blockcomment x1 python_prog x3a)
    then show ?case by(code_simp)
  next
    case (semi python_prog1 python_prog2)
    then show ?case
      apply(code_simp, auto)
      using sublist_append apply blast
      using sublist_append by blast
  qed
```

The second case is where the C program is a block of code. For all but one of the sub-cases, Isabelle once again adds the definitions of *valid_poly*, *compose_c_poly*, *rmcomments*, *C*, *rm_start_end_comment*, *sublist*, *Python*, and *python_valid_ifdef_c* to its simplifier to infer that, under these conditions, these definitions will always return True. The proof for the semi sub-case is the same as before.

```
  case (block x)
  then show ?case proof (induction python_prog)
    case SKIP note * = this
         have "(∀ps ss. [CHR 0x22, CHR 0x22, CHR 0x22, CHR '' '', CHR 0x22] ≠ ps @ CHR ''#'' # ss)
             ∧ (∀ps ss. [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22] ≠ ps @ CHR ''#'' # ss)"
       by(rule conjI, (rule allI, rule allI, rule set_next, (auto)[1])+)
     then show ?case using *
      by(simp  add: valid_poly_def compose_c_poly_def rmcomments_def
                    C_def rm_start_end_comment_def sublist_def Python_def python_valid_ifdef_c_def)
    next
    case (block x) note * = this
         have "(∀ps ss. [CHR 0x22, CHR 0x22, CHR 0x22, CHR '' '', CHR 0x22] ≠ ps @ CHR ''#'' # ss)
             ∧ (∀ps ss. [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22] ≠ ps @ CHR ''#'' # ss)"
       by(rule conjI, (rule allI, rule allI, rule set_next, (auto)[1])+)
     then show ?case using *
      by(simp  add: valid_poly_def compose_c_poly_def rmcomments_def
                     C_def rm_start_end_comment_def sublist_def Python_def python_valid_ifdef_c_def)
  next
    case (linecomment x) note * = this
           have "(∀ps ss. [CHR 0x22, CHR 0x22,CHR 0x22,CHR '' '', CHR 0x22] ≠ ps @ CHR ''#'' # ss)
             ∧ (∀ps ss. [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22] ≠ ps @ CHR ''#'' # ss)"
       by(rule conjI, (rule allI, rule allI, rule set_next, (auto)[1])+)
     then show ?case using *
      by(simp  add: valid_poly_def compose_c_poly_def rmcomments_def
                    C_def rm_start_end_comment_def sublist_def Python_def python_valid_ifdef_c_def)
  next
    case (blockcomment x1 python_prog x3a)
     note * = this
           have "(∀ps ss. [CHR 0x22,CHR 0x22,CHR 0x22, CHR '' '', CHR 0x22] ≠ ps @ CHR ''#'' # ss)
             ∧ (∀ps ss. [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22] ≠ ps @ CHR ''#'' # ss)"
       by(rule conjI, (rule allI, rule allI, rule set_next, (auto)[1])+)
     then show ?case using *
      by(simp  add: valid_poly_def compose_c_poly_def rmcomments_def
                    C_def rm_start_end_comment_def sublist_def Python_def python_valid_ifdef_c_def)
  next
    case (semi python_prog1 python_prog2)
    then show ?case
      apply(code_simp, auto)
      using sublist_append apply blast
      using sublist_append by blast
  qed
```

Next, the case where the program is a line comment. Proving the sub-cases where the Python program is a SKIP operation, a line comment, or a block comment once again only requires the usage

of code_simp. The proofs for the block and semi sub-cases are the same as was seen in previous proofs for these sub-cases.

```
  case (linecomment x)
  then show ?case proof (induction python_prog)
    case SKIP
    then show ?case by (code_simp)
  next
    case (block x)
     note * = this
            have "(∀ps ss. [CHR 0x22,CHR 0x22,CHR 0x22, CHR '' '', CHR 0x22] ≠ ps @ CHR ''#'' # ss)
             ∧ (∀ps ss. [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22] ≠ ps @ CHR ''#'' # ss)"
       by(rule conjI, (rule allI, rule allI, rule set_next, (auto)[1])+)
     then show ?case using *
      by(simp  add: valid_poly_def compose_c_poly_def rmcomments_def
                    C_def rm_start_end_comment_def sublist_def Python_def python_valid_ifdef_c_def)
  next
    case (linecomment x)
    then show ?case by (code_simp)
  next
    case (blockcomment x1 python_prog x3a)
    then show ?case by (code_simp)
  next
    case (semi python_prog1 python_prog2)
    then show ?case
      apply(code_simp, auto)
      using sublist_append apply blast
      using sublist_append by blast
  qed
```

Next, we have the case where the C program is a block comment. The only difference between the proof for this case and the previous one is that *sublist_append* is only used once for the semi sub-case.

```
next
  case (blockcomment x1 c_prog x3a)
  then show ?case  proof (induction python_prog)
    case SKIP
    then show ?case by (code_simp)
  next
    case (block x)
    note * = this
            have "(∀ps ss. [CHR 0x22,CHR 0x22,CHR 0x22, CHR '' '', CHR 0x22] ≠ ps @ CHR ''#'' # ss)
             ∧ (∀ps ss. [CHR 0x22, CHR '' '', CHR 0x22, CHR 0x22, CHR 0x22] ≠ ps @ CHR ''#'' # ss)"
       by(rule conjI, (rule allI, rule allI, rule set_next, (auto)[1])+)
     then show ?case using *
      by(simp  add: valid_poly_def compose_c_poly_def rmcomments_def
                    C_def rm_start_end_comment_def sublist_def Python_def python_valid_ifdef_c_def)
  next
    case (linecomment x)
    then show ?case by (code_simp)
  next
    case (blockcomment x1 python_prog x3a)
    then show ?case by (code_simp)
  next
    case (semi python_prog1 python_prog2)
    then show ?case
      apply(code_simp, simp)
      using sublist_append by blast
  qed
```

Finally, we have the simplest case where two C programs are joined together using the semi operation. The proof for each sub-case is exactly the same, and merely uses *code_simp* and *simp*.

```
next
  case (semi c_prog1 c_prog2)
  then show ?case  proof (induction python_prog)
    case SKIP
    then show ?case
      by(code_simp, simp)
  next
    case (block x)
    then show ?case by (code_simp, simp)
  next
    case (linecomment x)
    then show ?case by (code_simp, simp)
  next
    case (blockcomment x1 python_prog x3a)
    then show ?case by (code_simp, simp)
  next
    case (semi python_prog1 python_prog2)
    then show ?case by (code_simp, simp)
  qed
qed
```

With this, all subgoals have been solved. We have proven that the output of *compose_c_poly* (the C representation of the polyglot) is valid in the C language.

10.2 Proving Statement 6.2

Before we prove this statement, we must prove a number of other lemmas first. The proof structure of the first lemma can be seen below. Similarly to the proof in section 10.1, the proof is divided into five cases.

```
lemma rmcomments_2x: "rmcomments (rmcomments P) = rmcomments P"
proof (induction "P")
case SKIP
then show ?case by(code_simp)
next
  case (block x)
  then show ?case by(code_simp)
next
case (linecomment x)
then show ?case by(code_simp)
next
  case (blockcomment x1 P x3a)
  then show ?case by(code_simp)
next
case (semi P1 P2)
  then show ?case [24 lines]
qed
```

Other than the semi case, all cases are resolved using *code_simp*. A number of statements are needed to resolve the semi case.

The statement in f1 means "For all p in pa, if $rmskip(p)$ is a SKIP operation, then $rmskip(p; pa)$ is equal to $rmskip(pa)$. However, if $rmskip(pa)$ is a SKIP operation, then $rmskip(p; pa)$ is equal to $rmskip(p)$. Otherwise, $rmskip(p; pa)$ is equal to $rmskip(p)$;$rmskip(pa)$."

The statement in f2 means "If $rmskip(comments2skip\ P1; comments2skip\ P2)$ is equal to $rmskip(comments2skip\ P2)$, then $rmskip(rmcomments(P1; P2))$ is equal to $rmcomments(P1; P2)$".

Statements f3 and the following statement are the same, except that the former applies to program $P1$, and the latter applies to $P2$.

The next statement means "if $rmskip(comments2skip\ P2)$ is a SKIP operation, then $rmskip(comments2skip\ P1;\ comments2skip\ P2)$ is equal to $rmskip(comments2skip\ P1)$. Otherwise, $rmskip(comments2skip\ P1;\ comments2skip\ P2)$ is equal to $rmskip(comments2skip\ P1)$; $rmskip(comments2skip\ P2)$. From this, we can deduce that $rmcomments(rmcomments(P1; P2))$

is $rmcomments(P1; P2)$ or we can deduce that $rmskip(comments2skip\ P1;\ comments2skip\ P2)$ is equal to $rmskip(comments2skip\ P2)$."

```
case (semi P1 P2)
  then show ?case
  proof -
    have f1: "∀p pa. if rmskip p = SKIP then rmskip (p ; pa) =
              rmskip pa else if rmskip pa = SKIP then rmskip (p ; pa) =
              rmskip p else rmskip (p ; pa) = rmskip p ; rmskip pa"
      using rmskip.simps(1) by presburger
    have f2: "rmskip (comments2skip P1 ; comments2skip P2) =
              rmskip (comments2skip P2) ⟶ rmcomments (rmcomments (P1 ; P2)) =
              rmcomments (P1 ; P2)"
      by (metis (no_types) comments2skip.simps(5) comp_apply rmcomments_def semi.IH(2))
    have f3: "rmskip (comments2skip (rmcomments P2)) = rmcomments P2"
      by (metis comp_apply rmcomments_def semi.IH(2))
    have "rmskip (comments2skip (rmcomments P1)) = rmcomments P1"
      by (metis comp_apply rmcomments_def semi.IH(1))
    then have "(if rmskip (comments2skip P2) =
               SKIP then rmskip (comments2skip P1 ; comments2skip P2) =
               rmskip (comments2skip P1) else rmskip (comments2skip P1 ; comments2skip P2) =
               rmskip (comments2skip P1) ; rmskip (comments2skip P2)) ⟶
               rmcomments (rmcomments (P1 ; P2)) =
               rmcomments (P1 ; P2) ∨ rmskip (comments2skip P1 ; comments2skip P2) =
               rmskip (comments2skip P2)"
      using f3 by (simp add: rmcomments_def)
    then show ?thesis
      using f2 f1 by meson
  qed
qed
```

Finally, statement 6.2 can be written as a lemma and proven in Isabelle.

```
lemma rmcomments_p1_p2_general_2x:
"(rmcomments (p1;p2)) = (rmcomments ((rmcomments p1);(rmcomments p2)))"
```

The proof for this lemma is divided into two main cases: one where program $p1$ is a SKIP operation, and one where it is not. Within each of these cases, there are two sub-cases, one where $p2$ is a SKIP operation, and one where it is not. The sub-case where both programs are SKIP operations is resolved by adding $rmcomments\_def$ to Isabelle's simplifier. The sub-case where only $p1$ is a SKIP operation is resolved using Isabelle's metis proof method, the functions $comments2skip$ and $rmskip$, $rmcomments\_def$, the built-in lemma $comp\_apply$, and our previously-defined lemma $rmcomments\_2x$.

```
proof(cases "rmcomments p1 = SKIP")
  case True
  then show ?thesis
  proof(cases "rmcomments p2 = SKIP")
    case True
    from ‹rmcomments p1 = SKIP› and ‹rmcomments p2 = SKIP›
    show ?thesis by (simp add: rmcomments_def)
  next
    case False
    from ‹rmcomments p1 = SKIP› and ‹rmcomments p2 ≠ SKIP›
    show ?thesis by (metis comments2skip.simps(5) comp_apply rmcomments_def rmcomments_2x
                     rmskip.simps(1))
  qed
```

The aforementioned method is also used to prove the other two sub-cases, where only $p2$ is a SKIP operation, and where neither are SKIP operations.

```
next
  case False
  then show ?thesis
  proof(cases "rmcomments p2 = SKIP")
    case True
    from ‹rmcomments p1 ≠ SKIP› and ‹rmcomments p2 = SKIP›
    show ?thesis by (metis comments2skip.simps(5) comp_apply rmcomments_def rmcomments_2x
                     rmskip.simps(1))
  next
    case False
    from ‹rmcomments p1 ≠ SKIP› and ‹rmcomments p2 ≠ SKIP›
    show ?thesis by (metis comments2skip.simps(5) comp_apply rmcomments_def rmcomments_2x
                     rmskip.simps(1))
  qed
qed
```

By doing this, we have no further subgoals and our proof is complete.

11 Related Work

As of now, this is the very first formalisation of polyglot payloads. As such, we do not have any existing research with which we can directly compare our formalisation or its results with. However, there exist multiple tools, which can be used to aid in the generation of polyglot payloads. One of these examples is Truepolyglot, a polyglot file generator which can be installed and used on the command line. This tool is primarily concerned with the generation of PDF/ZIP polyglots, but it is also possible to use it to generate PDF/JAR, PDF/HTML, and PDF/PE polyglots. Truepolyglot is mainly written in Python and shell script, whilst our formalisation is written using Isabelle. Unlike Truepolyglot, our formalisation is used for the generation of C/Python polyglots [9].
Another piece of related work is the polyglot payload research conducted by Magazinius et al. in 2013 (as previously described in section 2.1). Though this article does not feature tools used for the generation of polyglot payloads, it demonstrates how polyglot payloads can be used for PDF-based injection and content smuggling attacks [4]. It also provides the original inspiration for this project.

12 Evaluation Of The Final Product

Here, the project will be assessed based upon the evaluation criteria outlined in section 4.2, keeping in mind that the focus of the project is on C/Python polyglots.

- Does the formalisation includes all of the necessary components to model programs?

Yes, and these components can be found in the program.thy file. These components include the *block_comment* record, the *pl* record, and the *program* datatype.

- Are there mechanisms in place to determine to determine whether a program is valid in a given language?

Yes, and these mechanisms can be found in the form of the function *validP* and the auxiliary function *rm_start_end_comment*. *validP* has been tested and correctly classifies valid programs. For the purposes of this formalisation, not all aspects that can affect the validity of a program (such as whether variables have been correctly instantiated) are not covered in this formalisation - only the necessary features needed to create C/Python polyglots have been implemented. In the future, other aspects which may be needed for the formalisation of other types of polyglot payloads will need to be modeled. However, due to time constraints, this was not currently possible.

- Are there mechanisms in place to determine which programs can create valid polyglot programs?

Yes. C/Python polyglot programs are created using *compose_python_poly*, *compose_c_poly*, and *mk_c_python_poly*. The latter function ensures that only valid polyglots can be created, if *c_prog* and *python_prog* are valid in C and Python respectively. If either of the input programs are invalid, then a valid polyglot featuring these programs cannot be created, and the empty set is returned.

- Are the programs created using the formalisation valid in multiple given languages?

Yes. The usage of *compose_c_poly*, *compose_python_poly*, and *string_of_program* provide us with polyglot payloads we can test on the command line (once slight modifications have been made to the syntax of the program). An example of a C/Python polyglot has been demonstrated in section 9.3 to be free of errors or warnings.

- Are there sufficient proofs to show that the formalisation will always result in valid polyglots?

Yes. The function *valid_poly* validates polyglot payloads, and a number of lemmas proving that this function provides the correct output (including *valid_poly_compose_c_poly*) can be found in polyglot.thy.

Given this evaluation, we can determine that the formalisation, polyglot payloads and proofs produced are of a high quality. This shows that the project has provided a good solution to the problem as stated in the set criteria. The only departures from the original requirements outlined in Section 3 have also been well justified as being left out from the project, given the time constraints.

13 Critical Assessment Of The Project As A Whole

The aim of formalising and proving the validity a polyglot program has been approached well. As this is the first formalisation of a polyglot payload, not much literature exists with which we can use to compare our formalisation against. A test-driven development methodology has been used to develop and prove the polyglot model. Functions and definitions have been redeveloped, allowing for the formalisation and proving of C/Python polyglots. This has been done to the point of fulfilling the project requirements.

The final product does successfully formalise and prove the validity of a C/Python polyglot. However, the process of reaching this goal has been difficult. The two main reasons for this are the time constraints and my previous inexperience with Isabelle. Before this research project, I had never used Isabelle, let alone any other proof assistants. Consequently, the first obstacle to overcome was to learn and understand the syntax. With the amount of time I had to familiarise myself with Isabelle, a high-quality end result has been produced. The language Isabelle is very different to the most common programming languages, so learning the language was a very interesting and exciting experience.

Given the (mainly) functional style of the formalisation, one of the main focuses was to ensure that the resulting project was functionally correct, rather than efficient. Hence, no baselines can be seen in the results. As this is a formalisation, no user testing or customer feedback was required. The end results (the proofs) have been evaluated by the theorem prover, rather than by any number of people.

If I had more time, the model of a polyglot program would have been more general, to allow other types of polyglot payloads to be formalised (such as PDF/ZIP polyglots, as seen in the Truepolyglot tool).

14 Conclusion

The aim of this project was to use the theorem prover Isabelle to formalise and prove the validity of a simple C/Python polyglot program. To do this a simplified model of programs valid in one programming language was created. This model was then used to define input programs and a template for C/Python polyglot payloads. By feeding the input programs into the template, valid C/Python polyglot payloads were generated. This formalisation of polyglot payloads is the first of its kind.

A test-driven modelling method was used to complete this project. In this case, the proofs statements were the test cases. From this, we were able to create a model design to provide us with clarity of what the desired output of the formalisation would be. The functionality for the program and polyglot models (required for the proof statements to hold) were then designed and developed. Each function was tested through the use of value statements to ensure the desired output was produced. If not, then the functions were redeveloped until they did produce the desired output. Subsequently, the functions were tested to ensure that they worked together properly, and only ever generated a valid polyglot. The generated polyglot payload was then tested on the command line to verify the formalisation in two standard programming languages. The proofs statements were then converted to lemmas. Functions were once again redeveloped until each lemma held.

All of the functionality was developed. All tests have passed and all lemmas hold. Hence, the project has successfully been formalised and proven to only produce valid C/Python polyglots.

Due to time constraints a more generalised formalisation of polyglot payloads could not be created. Thus, given the differing behaviours of other programming languages, not all possible factors that can allow or prevent programs from different programming languages from being combined to produce valid polyglot payloads have been considered. However, the existence of this formalisation opens up opportunities for further research to explore the formalisation of other types of polyglot payloads. As such, this project can still be classified as a success.

14.1 Future Work

The current formalisation focuses on primarily C/Python polyglot programs. As such, the main extension that can be made to this project is to create a more general formalisation of polyglot programs. For this to be possible, the model could be altered to account for other factors that can aid in the creation of polyglot programs, including spacing of characters, the ability to define functions from one language in another (such as in the C-preprocessor), nested block comments, and more. Once these functionalities has been added, due to the significance of the work, the project can be published into Isabelle's Archive of Formal Proofs [10].

14.2 Acknowledgements

First of all, I would like to thank my project supervisor, Professor Achim Brucker for the incredible support and motivation he has provided me with during the development of a formalisation of polyglot payloads.

Also, I would like to thank the Computer Science and Mathematics departments for providing me with the rich knowledge basis needed to complete this final year project.

Finally, I would like to thank the University of Exeter for providing me with the ability to further my education with their extra-curricular opportunities.

References

[1] J. A. Lewis, "Economic impact of cybercrime," Feb 2018. [Online]. Available: https://www.csis.org/analysis/economic-impact-cybercrime

[2] C. Office, "National cyber security strategy 2016 to 2021," Sep 2017. [Online]. Available: https://www.gov.uk/government/publications/national-cyber-security-strategy-2016-to-2021

[3] V. Benzaken, S. Boldo, E. Contejean, J.-H. Jourdan, G. Melquiond, and B. Wolff, 2014. [Online]. Available: https://vals.lri.fr/languages.html

[4] J. Magazinius, B. K. Rios, and A. Sabelfeld, "Polyglots: crossing origins by crossing formats," *Proceedings of the 2013 ACM SIGSAC conference on Computer amp; communications security - CCS '13*, Nov 2013.

[5] P. D. Mosses, "Formal semantics of programming languages: â an overview â," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 41–73, 2006, proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1571066106000429

[6] Goguen and Moriconi, "Formalization in programming environments," *Computer*, vol. 20, no. 11, pp. 55–64, 1987.

[7] T. Nipkow and G. Klein, "Concrete semantics with isabelle/hol," Mar 2021.

[8] A. Koutifaris, Jul 2018. [Online]. Available: https://www.freecodecamp.org/news/test-driven-development-what-it-is-and-what-it-is-not-41fa6bca02a2/

[9] Ansemjo, "ansemjo/truepolyglot," Jul 2019. [Online]. Available: https://github.com/ansemjo/truepolyglot

[10] [Online]. Available: https://www.isa-afp.org/