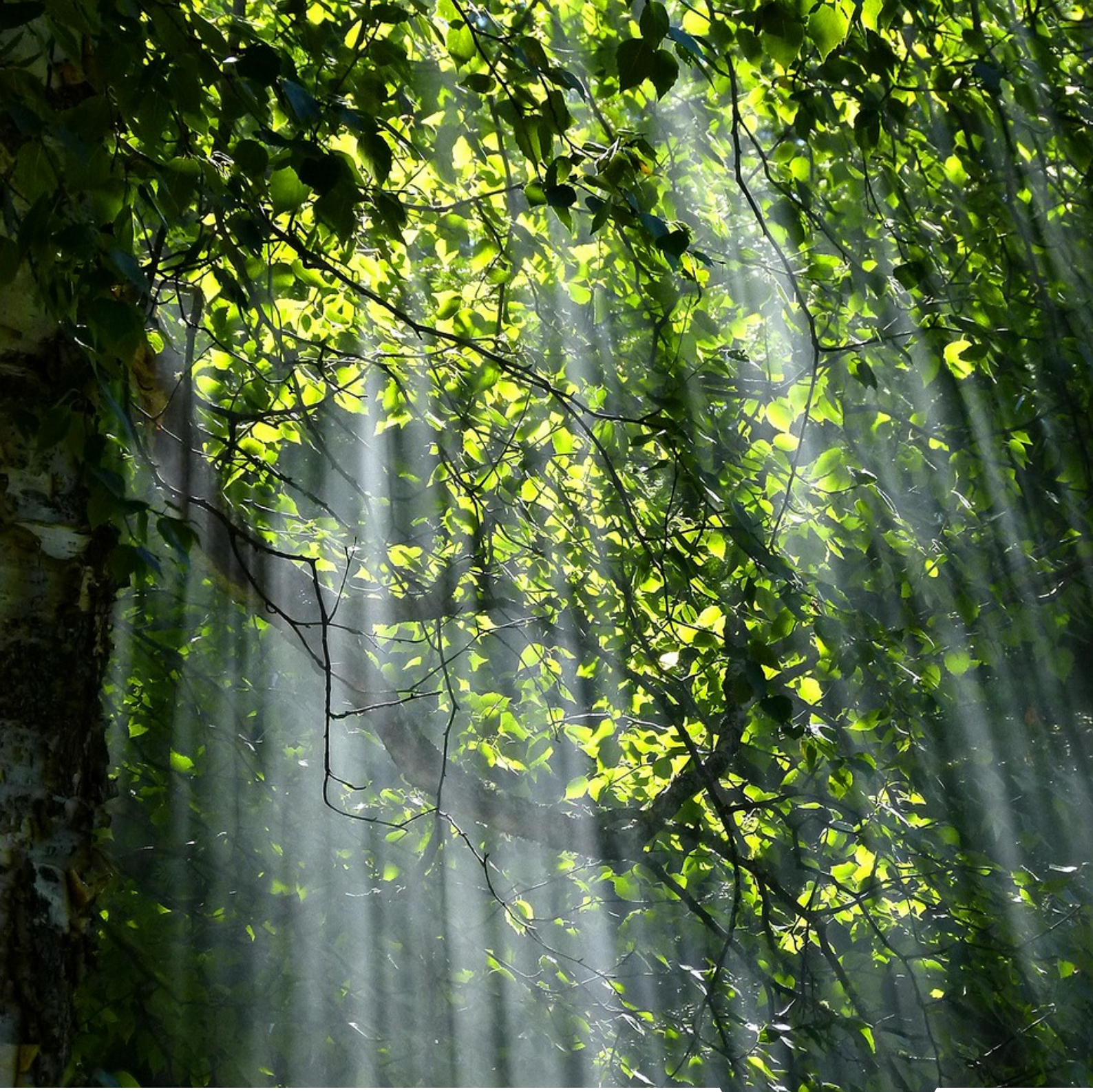


Regular Expression Refinement Types

WARWICK

Progress Report

Adam Williams

Supervisor: Michael Gale

Table of Contents

1 Introduction 2

2 Project State 2

3 Project Management 9

4 References 10

A Specification 11

1. Introduction

Information security vulnerabilities often arise due to improper input handling. From SQL injection to cross-site scripting, vulnerabilities of this class are found on a daily basis in production applications by security professionals. It is desirable to try to detect and report on potentially insecure code as early as possible in the development process in order to maximise the probability that the issue is resolved (Sadowski et al., 2018).

Tooling exists to perform static analysis using *taint tracking*, which involves monitoring data flow of user input and generating alerts for unsafe interactions with potentially dangerous functions or *sinks* (Jovanovic et al., 2006). Taint tracking is often unable to evaluate the effectiveness of input validation code that may already be in place. As a result, these tools lend themselves to generating false positives. For example, a naive taint tracking tool which does not consider validation measures may determine the code below which interpolates user input into a SQL query string to be vulnerable. However, as the user input is filtered first such that it matches a restrictive regular expression, such a vulnerability report would be a false positive and would not be exploitable.

```php
<?php
preg_match("/[A-Za-z]+/", $_GET['name'], $matches); // sanitises name
$name = $matches[0]; // name is guaranteed to only consist of A-Z and a-z
mysql_query("select * from users where first_name = '$name'"); // not exploitable
```

Within type systems, refinement types allow for predicate-based constraints to be applied to types in order to restrict the domain of elements which belong to the type (Pierce, 2002). A local variable used to store natural numbers could be constrained via a refinement type such as $\{n : \mathbb{N} \mid n \leq 5\}$; only $\{0, 1, 2, 3, 4, 5\}$ would be permitted for values of n under this constraint.

This project applies refinement types to strings which enforce membership of $L(R)$ for some regular expression R. For example, $\{s : \Sigma^* \mid s \in L(ab+)\}$ [1] would allow `"ab"` to be assigned as a value of some variable s, but not `"a"`. These refinement types can be used within local variable types, as well as function return types and can be syntactically included after the type keyword:

```
function GetString(): string[/N+/] { // return type is a string which matches the N+ regex
    return "November"
}
```

We design a proof-of-concept language with syntax for expressing such refinement types and implement a type checker in order to evaluate the feasibility of compile time regular expression checks. Vulnerabilities such as local file inclusion, directory traversal, SQL/LDAP injection can all arise due to unsafe user input handling. The purpose of this language is to enable security-critical functionality involving user input to be implemented/wrapped and then invoked from an existing codebase. The efficacy and false positive rate of this type checker can then be compared with existing static analysis tools designed to detect these kind of vulnerabilities.

Per the specification included in Appendix A, Weeks 1-9 of the project were scheduled to involve implementing and testing a prototype parser and type checker for the language.

2. Project State

The project is currently ahead of schedule. By week 7, both phases 1 and 2 have been completed, resulting in a functional prototype.

Early on in the project lifetime we decided to begin prototyping work with a simplified goal, in order to gain experience working with refinement types and type checkers in general. Once complete, work would continue in a 2nd phase which introduced regular expressions.

[1] For some regular expression R, $L(R)$ is used to denote the language that it accepts

Phase 1

Prototype work began with simple refinement types which could be applied to natural numbers. The final syntax was similar to the example presented in the specification, with type constraints specified in square brackets [] after the type keyword:

```
function Main(id: uint): uint[< 4] {
    // A simple function declaration, returns an int less than 4
    return 0
}
function StackOne(): uint[> 5] {
    // This function must return an unsigned int greater than 5
    return Main(1)
}
```

As the `Main()` function returns an integer less than 4, a violation is present within `StackOne()` where it is returned. This violation arises because `StackOne`'s return type is constrained to be greater than 5.

Running this simple program with the prototype tool yields the following output:

```
➜ PocLang (git:master) cat input.txt | ./gradlew run
> Task :run
Reading program from stdin (use Ctrl+D when finished)...
Violation via value 0
L7:4 Return type uint [(< x 4)] of function Main didn't satisfy uint [(> x 5)]
```

The prototype tool has found a valid value (here, 0) which satisfies $x < 4$ but violates $x > 5$. If integrated into a "real" programming language, this would be identified at compile time and prevent successful compilation.

This initial version of the prototype which operated exclusively on the natural numbers was built using the following tools, libraries and languages:

JavaSMT Library which provides an abstraction layer over a variety of different SMT solvers (such as Z3 and MathSAT) and provides added type safety (Karpenkov et al., 2016).

ANTLR 4 An $LL(*)$ lexer and parser generator which can target languages such as Java, Go and C# (Parr and Fisher, 2011). In the initial prototype, ANTLR was used to generate a combined lexer-parser using a composite grammar in which both non-terminals and terminal symbols (tokens) are specified.

JUnit A unit testing framework for Java. Used to build a collection of unit and integration tests to support refactoring and a test-driven development approach.

Gradle Industry-standard build automation system which co-ordinates the grammar generation, Java class compilation, construction of a JAR file with shaded dependencies and running the unit tests.

IntelliJ IDEA An advanced Java IDE with leading code inspection and analysis capabilities (Wedyan et al., 2009) (Jemerov, 2008). Used in conjunction with the official ANTLR plugin which provides grammar syntax highlighting and analysis.

The general steps involved in the process are described below.

Parsing

In the **Parsing** step, the ANTLR parser is used to generate a parse tree from the input program. For example, consider the following simple function declaration below. The parse tree generated for this function definition is shown in figure 1.

```
function Main(id: uint[> 1]): uint { // accepts one argument id, which must be greater than 1
    return 1*1
}
```

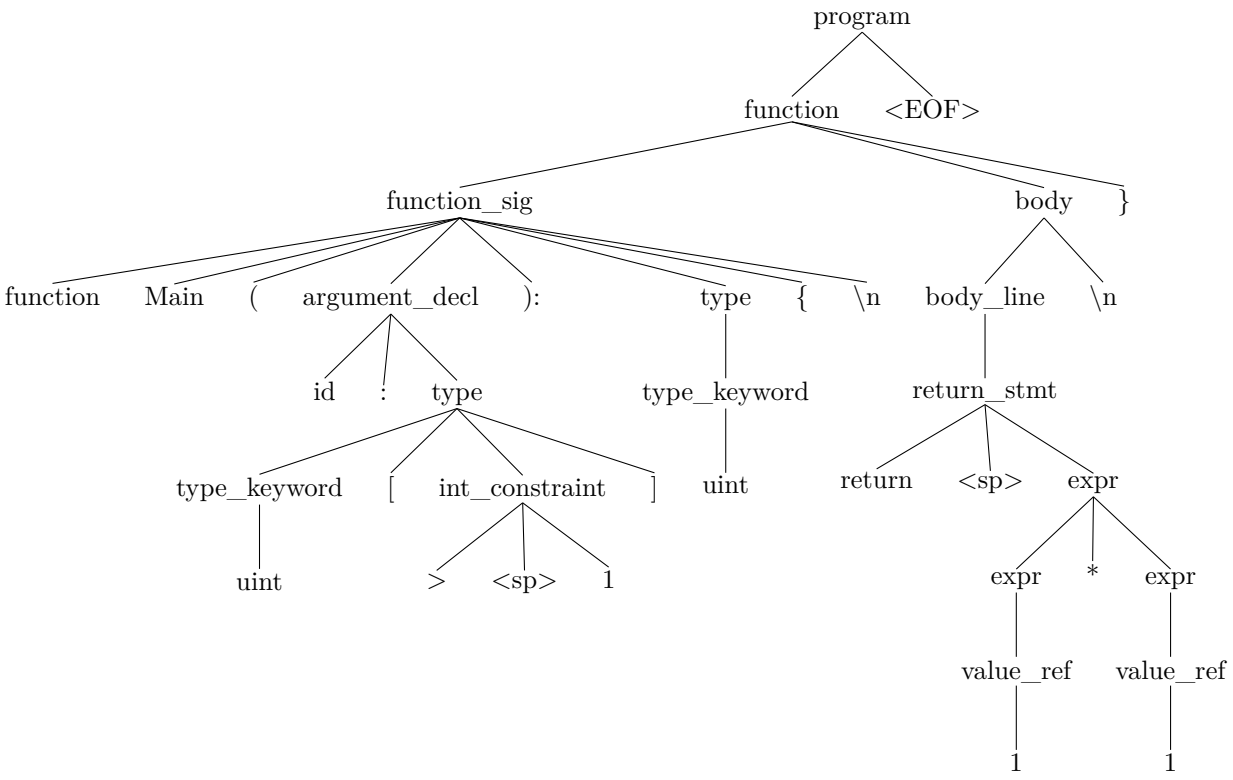


Figure 1: Parse tree for the `Main` function.

For the phase 1 prototype, the lexer and parser are both generated from the same grammar file. Any *syntactical* errors are found during this step, these must be fixed before the type checking will be carried out. The result from this step is a `ParseTree` object generated by ANTLR – methods are available to traverse the tree and retrieve information about non-terminal and terminal nodes.

Initial Walk

The language designed for this prototype allows for functions to be called from within functions that may occur anywhere within the program code. As a result, it is necessary to perform one initial pass through the parse tree in order to collect function names to store in the function table. This table stores the identifier (`Main` in the example used previously) and the return type.

ANTLR provides a `BaseListener` class which can be extended in order to create a listener which is called for each non-terminal encountered when walking the parse tree.

The proof-of-concept language implementation uses one listener which is parametrised with a `VisitorPhase` value to specify the stage at which the listener should operate. This is set to `COLLECTING_FUNCTIONS` for the **Initial walk** stage.

It is within this stage that function re-declarations can be discovered and reported. Errors are stored centrally and associated with the input token closest to the origin of the problem.

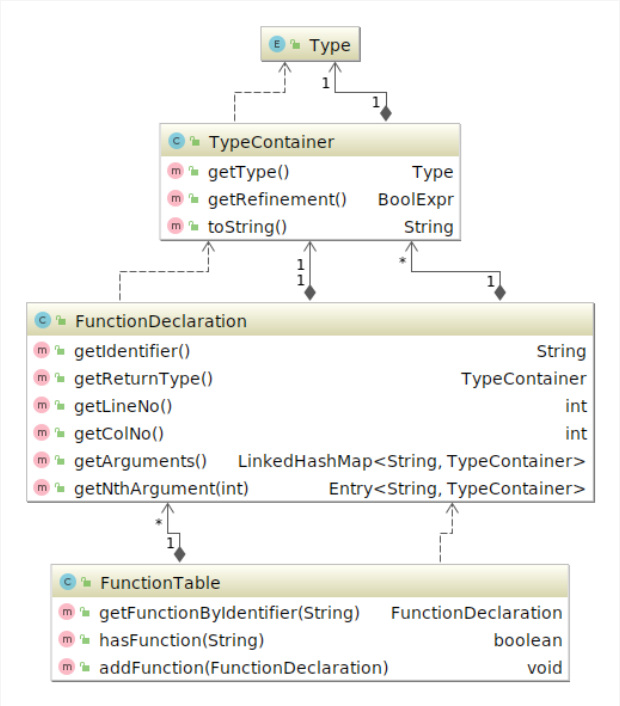


Figure 2: Dependency diagram showing how the functions declarations are stored and the data stored alongside them.

Type Checking

Parsing | Initial walk | **Type checking** | Reporting

Once the function definitions have been collected, the main type checking phase can begin. This covers validating function calls, variable assignments and return statements.

This is mainly achieved by visiting function bodies and initialising a stack for scoped storage of local variables and function arguments as is standard in most programming languages (Watt, 2004, p. 88). Function calls are checked to ensure that the function identifier has been encountered before and exists in the function table. Variable assignments are checked to ensure that the type of the right-hand side expression satisfies the type declaration for the variable.

For types with a provided constraint, the JavaSMT library is used to check the inverted *expected* type against the *actual* type.

For example, consider the code below:

```
function Main(): uint[> 4] {
    var u: uint[< 10]
    return u
}
```

In this case, the return value of `Main` is constrained as $\{x : \mathbb{N} \mid x > 4\}$. When the `return u` statement is checked, the *expected* type is $\{x : \mathbb{N} \mid x > 4\}$ and the *actual* type is $\{x : \mathbb{N} \mid x < 10\}$. It should be apparent that u could be a value such as 2 (since $2 < 10$) which would violate the return type constraint.

To find this, the *expected* type is negated to give $\{x : \mathbb{N} \mid x \leq 4\}$ and this constraint, along with the constraint defined by the *actual* type are passed to the SMT solver to find a model. This model will include a value for x that satisfies the conjunction of the two constraints $x \leq 4 \wedge x < 10$. The code below shows how the JavaSMT API allows these problems to be configured and solved.

We first use the formula manager to set up a variable to represent our unknown:

```
IntegerFormulaManager imgr = context.getFormulaManager().getIntegerFormulaManager();
IntegerFormula x = imgr.makeVariable("x");
```

Once this is complete, we can build formulae involving the variable as well as other operations. These map to the constraints discussed previously:

```
BooleanFormula expectedNegated = imgr.lessOrEquals(x, x.makeNumber(4)); // x <= 4
BooleanFormula actual = imgr.lessThan(x, x.makeNumber(10)); // x < 10
```

Finally, we create a prover environment with our two constraints added. We specify that we would like models to be generated alongside the satisfiability result. The model, if found, will include an example value for x that satisfies all of the constraints.

We check the result of the operation with the `isUnsat` method. If the SMT solver is unable to find a satisfying model, this method will return `true`.

```
try (ProverEnvironment prover = context.newProverEnvironment(ProverOptions.GENERATE_MODELS)) {
    prover.addConstraint(expectedNegated);
    prover.addConstraint(actual); // both constraints must hold
    boolean isUnsat = prover.isUnsat(); // isUnsat = false here.
    try (Model model = prover.getModel()) {
        // Prints the value of x found for which the two constraints hold
        // (i.e. a violation of the refinement type has been found)
        System.out.printf("Violation found with: ", model.evaluate(x));
    }
}
```

If the SMT theorem is satisfiable, we have found a violation – it is possible for the refinement type constraints to not hold during execution. These are reported via the central error reporting mechanism.

Reporting

During the 4th and final step of execution, the prototype application reports the errors discovered previously. Each of these includes a line number and message to describe the problem.

These are simply printed to the standard error stream at the command line, but a real language could show these violations in an IDE to highlight problems inline.

Phase 2

Once the initial prototype for a language with refinement types for natural numbers was completed and a set of passing test cases had been written, the focus shifted to adding `string` support. A new type keyword was added, and the grammar was modified to accept regular expression constraints as shown in the example below.

```
function LookupUserById(): string[/g+/] {
    return "f"
}
```

Adding support for these constraints required being able to parse regular expressions. This was more challenging than anticipated – although BNF descriptions of regular expressions are readily available, they are indirectly left-recursive. ANTLR, along with all LL parser generators, requires that such production rules are removed before a working parser can be generated (Dick and Ceriel, 1990).

It is reasonably straightforward to remove left-recursion by introducing new production rules, however another issue arised with the programming language grammar's tokens:

```
IDENTIFIER : [A-Za-z_] [A-Za-z_0-9]* ;
CHARACTER  : ~('\n'|'\r'|'.'|'('|')'|'['|']'|'*'|'+'|'/'|'|') | ESCAPED_META;
```

The `IDENTIFIER` token was used for function and variable names. The `CHARACTER` token was used for single characters within a regular expression. As the two token definitions overlapped, only one would be identified by the lexer which had the effect of breaking the parsing of either the regular expressions, or the function definitions.

We fix this issue with lexical modes: when the start of a string constraint is detected, the lexer should switch into a `REGEX` mode and match arbitrary characters with the `CHARACTER` token. At the end of the regular expression the mode should return to the default. Unfortunately, lexical modes are not supported in ANTLR's composite grammars and it was therefore necessary to split the parser and lexer into separate grammar files.

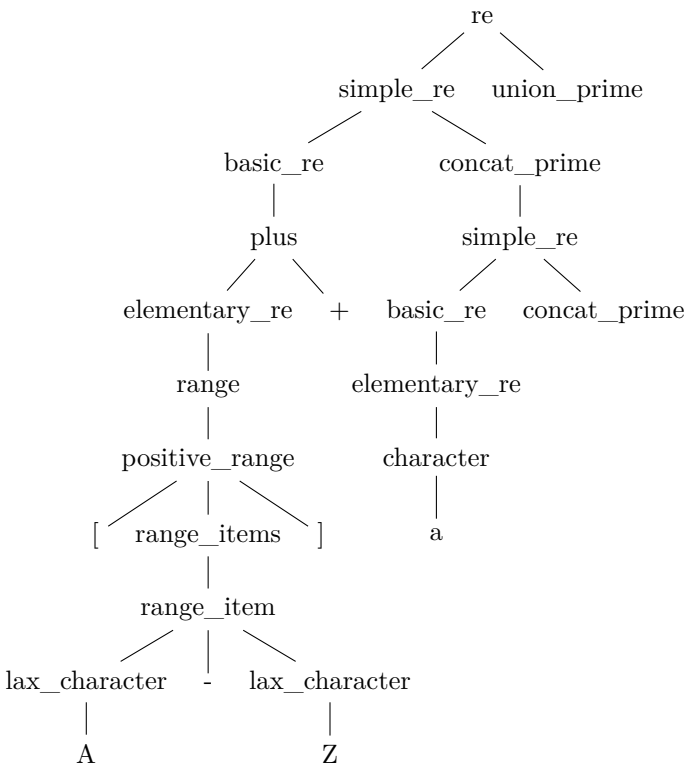


Figure 3: Parse tree for regular expression `[A-Z]+`.

With the parser for this modified language complete, it should have simply been a matter of taking advantage of the SMT solver's regular language membership support to find violations in much the same way as was performed on the integers. String (and regular expression support) is not universal amongst SMT solvers, however – and JavaSMT did not provide an abstraction layer over such functionality.

As a result, we decided to build Z3 from source and use its Java bindings instead of the JavaSMT library. Z3, a C++ SMT solver, includes the Z3str3 component which is able to efficiently support solving string equations and predicates involving regular language membership (Berzish et al., 2017). The original code which used the JavaSMT APIs to solve constraint problems with integers was then modified to use Z3. An adapter was written to construct a Z3 regular expression object from the parse tree.

The example Java below shows what is involved in formulating and solving string constraint problems using Z3str3's API.

We first must build the regular expressions which we want to use as constraints. In the same way that regular expressions can be defined inductively, Z3 offers an API that allows us to build up complex `ReExpr` objects by repeatedly applying operations to them:

```
// /[a-z!]+/
ReExpr matchAnyChar = ctx.mkPlus(
```

```
    ctx.mkUnion(
        ctx.mkRange(ctx.mkString("a"), ctx.mkString("z")), // [a-z]
        ctx.mkToRe(ctx.mkString("!")) // !
    )
);

// /foo[a-z!]+/, uses result above
ReExpr exprOne = ctx.mkConcat(ctx.mkToRe(ctx.mkString("foo")), matchAnyChar);
// /[a-z!]+bar/
ReExpr exprTwo = ctx.mkConcat(matchAnyChar, ctx.mkToRe(ctx.mkString("bar")));
```

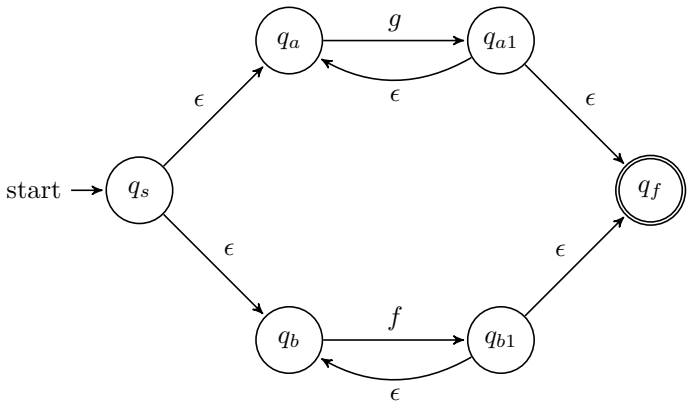
With our regular expression objects built, we can now create the string equation. This requires an unknown variable x of string type/sort and a `Solver` instance, which we can inform of the constraints.

```
// our unknown string
SeqExpr x = (SeqExpr) ctx.mkConst(ctx.mkSymbol("x"), ctx.getStringSort());

Solver s = ctx.mkSolver();
// must satisfy both expressions - x must belong to the intersection
s.add(ctx.mkInRe(x, ctx.mkIntersect(exprOne, exprTwo)));
```

Finally, with the constraints set, we can begin searching for a satisfying model. Z3 will attempt to find a string for x which matches both expressions:

```
// try and find model satisfying requirements
Status res = s.check();

// prints SATISFIABLE
System.out.println(res);
// print out value from the model, e.g. "fooc!bar"
System.out.println(s.getModel().getConstInterp(x));
```

Internally, Z3str3 uses Thompson's construction algorithm to generate an NFA representation for a regular expression (Berzish et al., 2015). These representations are used to find matching strings. For example, `g+|f+` could be represented as shown below:



The final prototype uses the complement of the *expected* type as a constraint along with the *actual* type. A model which satisfies these constraints indicates a string could be assigned or returned which violates the regular expression constraints. The prototype allows us to write a program like the one below:

```
function SecondaryFunction(): string[/g+|f+/] { // must return one or more 'g's OR one or more 'f's
    return "f"
}
function Main(): string[/f+/] { // must return one or more 'f's
    return SecondaryFunction()
}
```

The type checker will correctly determine that the union of `g+` in the constraint of `SecondaryFunction`'s return type is incompatible with `Main`'s return type. Note that the constraint used in the return type contract is used, despite the fact that `SecondaryFunction` in its current implementation returns a string which matches `g+|f+`.

```
➜ PocLang (git:master) cat input.txt | ./gradlew run
> Task :run
Reading program from stdin (use Ctrl+D when finished)...
Violation via value "gggg"
L5:4 Return type string [(str.in.re x (re.union (re.+ (str.to.re "g")) (re.+ (str.to.re
"f"))))] of function SecondaryFunction didn't satisfy string [(str.in.re x (re.+ (str.to.re
"f")))]
```

The string which the SMT solver has found, "gggg", matches `/g+|f+/` (and so is a valid return value for `SecondaryFunction`) – however, it does not match `/f+/`. In this way, it would be possible for the `Main` function at runtime to violate the contract constraining the values it can return.

3. Project Management

The following objectives were described in the project specification (Appendix A):

#	Objective	Status
1.	Formalise a type system that supports types predicated with a regular expression pattern that elements of the refined type will satisfy (be matched by).	✔
1. (a)	Explore the consequences of typical string operations (e.g. concatenation) and define the type of their return value when applied to elements of the regular expression type.	✖
1. (b)	At minimum, this should allow for simple functions to be declared that can safely accept/return a particular regular expression input.	✔
1. (c)	Evaluate the rate of false positives when compared to existing static analysis techniques.	✖
2.	Implement such a type system that can guarantee type safety, built against a simplified proof-of-concept language.	✔
2. (a)	Test the implementation against a variety of test cases. The testing strategy should make use of automated unit tests, and manual system testing considering both general expected input as well as any relevant "edge-cases" that need to be handled.	✔

Table 1: Objective progress to date.

Objective 2 and the sub-objective related to testing has been completed as of the time of writing. 26 individual passing JUnit test methods have been defined which cover the broad areas described below:

Parsing Tests for simple syntactically valid and invalid programs, as well as parser regression tests for functions that were incorrectly parsed during development and subsequently fixed.

Simple type checks Such as returning a string from a function marked as returning a string.

Other checks Including function/variable re-declaration.

Integer refinement types As described above, constraint violations for integer inequalities.

Regexp refinement types As described above, constraint violations for regular language membership.

```
➜ PocLang (git:master) ./gradlew test --rerun-tasks


BUILD SUCCESSFUL in 4s
4 actionable tasks: 4 executed
```

Manual testing has taken place throughout and has been used to influence development of new unit tests.

Objective 1. (b) has been completed, in addition to local variable type constraints. Objectives 1. (a)[2] and (c) have not yet been completed.

This level of progress is in line with the original schedule, which allocated false positive rate evaluation to occur in early February of 2019 (with implementation to be finished in late November and testing to take place from December until January).

There were some setbacks due to the initial use of JavaSMT. In hindsight, evaluating the effectiveness of the JavaSMT abstraction layer for use with strings *prior* to creating the prototype with unsigned integers would have saved time. It is likely we would have used Z3 from the start.

The immediate next steps once these objectives are complete include making the prototype language more expressive by introducing constructs to control execution flow. If time permits, the *stretch objectives* will also be considered. There does not appear to be a need to make changes to the schedule at this time.

4. References

Berzish, M., Zheng, Y. and Ganesh, V. (2015), 'PR 1562: automata-based regex engine for Z3str3'.
URL: *https://github.com/Z3Prover/z3/pull/1562*

Berzish, M., Zheng, Y. and Ganesh, V. (2017), 'Z3str3: A string solver with theory-aware branching', *arXiv preprint arXiv:1704.07935* .

Dick, G. and Ceriel, H. (1990), Parsing techniques, a practical guide, Technical report, Technical Report, Tech. Rep.

Jemerov, D. (2008), Implementing refactorings in intellij idea, *in* 'Proceedings of the 2Nd Workshop on Refactoring Tools', WRT '08, ACM, New York, NY, USA, pp. 13:1–13:2.
URL: *http://doi.acm.org/10.1145/1636642.1636655*

Jovanovic, N., Kruegel, C. and Kirda, E. (2006), *Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)*, IEEE.

Karpenkov, E. G., Friedberger, K. and Beyer, D. (2016), Javasmt: A unified interface for smt solvers in java, *in* 'Working Conference on Verified Software: Theories, Tools, and Experiments', Springer, pp. 139–148.

Parr, T. and Fisher, K. (2011), 'LL (*): the foundation of the ANTLR parser generator', *ACM Sigplan Notices* **46**(6), 425–436.

Pierce, B. (2002), *Types and Programming Languages*, The MIT Press.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L. and Jaspan, C. (2018), 'Lessons from building static analysis tools at google', *Commun. ACM* **61**(4), 58–66.
URL: *http://doi.acm.org/10.1145/3188720*

Watt, D. A. (2004), *Programming language design concepts*, John Wiley & Sons.

Wedyan, F., Alrmuny, D. and Bieman, J. M. (2009), The effectiveness of automated static analysis tools for fault detection and refactoring prediction, *in* '2009 International Conference on Software Testing Verification and Validation', pp. 141–150.

[2]It should be reasonably trivial to define the plus operation for strings and infer the type constraint to be the concatenation of all involved regular expressions.

Appendix A – Specification

The original specification as submitted at the start of Term 1 is included overleaf.

Introduction

Within information security, entire classes of application vulnerabilities arise due to problematic user input handling (Christey and Martin, 2007). This covers cross-site scripting (XSS), injection (SQL, LDAP, etc), insecure deserialisation and file inclusion vulnerabilities – all of which are regularly discovered in major software products.

There are many existing products that aim to detect problems within an application. Sadowski et al. (2018) describe the benefits of static analysis tooling deployed within Google which allow checks for common issues to be performed as part of the compilation process. The authors explain one of the main challenges with developer adoption as *trustworthiness* "users do not trust to results due to, say, false positives" and highlight the importance of reporting issues early: "survey participants deemed 74% of the issues flagged at compile time as real problems, compared to 21% of those found in checked-in code".

In the context of application security, tooling can be broadly categorised into the two broad areas of DAST (dynamic application security testing) and SAST (static application security testing) tooling. SAST tooling can analyse a codebase at rest and lends itself to generating much more immediate results that can take the entire codebase into account. DAST tooling allows for assessment from a black-box perspective but is much more limited. Research into the effectiveness of DAST-based scanning tools by Doupé et al. (2010) drew the conclusion that commonly available tools of this kind often failed to crawl more complex parts of an application which resulted in decreased coverage from a security perspective.

Existing static analysis tools can provide useful warnings that help prevent introduction of vulnerabilities. In the .NET ecosystem, tools such as *Roslyn Security Guard* can detect e.g. injection vulnerabilities by tainting user input and then using information flow analysis – first described in Denning and Denning (1977) – to determine when unsafe input is passed to a dangerous *sink* function (Arteau, 2016). Of course, this approach is limited. All user input is deemed unsafe by virtue of being user input and prior validation (as is commonly performed using regular expressions) is not taken into account when deciding if an alert needs to be raised or not. This leads to false positive reports where a program is secure by virtue of already performing validation to prevent a vulnerability.

The main objective of this project is to explore the use of a type system which uses regular-expression based refinement types applied to user input. A refinement type in the context of a type system is a type that is subject to a particular predicate (Pierce, 2002, p. 207). By considering flow of data that belongs to a refinement type for a particular pattern, it will be possible to make inferences about vulnerabilities that may be present in a codebase based on declared safe argument types. Use of refinement types in this way provides for a more informed evaluation of a particular risk than base types alone and should therefore enable reporting of fewer false positive issues.

```
function LookupName(userId: /[A-Za-z0-9]+/): string {
    var name: string = GetNameByUserId(userId); // safe
    GetNameByUserId("'; DROP TABLE users;"); // type error
    return name;
}

function GetNameByUserId(query: /[^`"']+/): /[A-Za-z ]+/ { /* database lookup.. */ }
```

Listing 1: Example code illustrating a potential syntax. `userId` and `query` use the refinement type.

Objectives

Primary Objectives

- Formalise a type system that supports types predicated with a regular expression pattern that elements of the refined type will satisfy (be matched by).
    - Explore the consequences of typical string operations (e.g. concatenation) and define the type of their return value when applied to elements of the regular expression type.

- At minimum, this should allow for simple functions to be declared that can safely accept/return a particular regular expression input[1].
- Evaluate the rate of false positives when compared to existing static analysis

- Implement such a type system that can guarantee type safety, built against a simplified proof-of-concept language.
  - Test the implementation against a variety of test cases. The testing strategy should make use of automated unit tests, and manual system testing considering both general expected input as well as any relevant "edge-cases" that need to be handled.

Additional Objectives

- Apply the theory explored in the primary phase of the project to produce a type analysis tool which works against type annotations applied to a commonly-used language such as C# or Scala. This tooling could be integrated into an IDE or CI pipeline.

ℹ Primary objectives are expected to be completed during the lifetime of the project. Additional objectives are identified as potential goals to pursue beyond the original scope of the project, if time permits.

Schedule

Time Window	Work
October 1st – October 14th	Specification completion, research into prior related works. Study of elementary programming language and type system theory (e.g. simply typed λ-calculus, SLam).
October 15th – October 28th	Begin writing background for report, work on formalisation of regular expression refinement type. **Deadline**: CS353 presentation, 24th October
October 29th – November 11th	Explore and document properties of type system. Begin implementation of ideas to produce a concrete proof-of-concept.
November 12th – November 25th	Completion of progress report, continued implementation work.
November 26th – December 9th	Testing of implemented proof-of-concept. **Deadline**: CS915 coursework, 26th November
December 10th – January 6th	Slack time (to use if behind schedule, else to make a start on year scheduled in 2019).
January 7th – January 20th	Finalise testing of implementation, write-up test cases. **Deadline**: CS324 coursework
January 21st – February 3rd	Evaluate false positive rates against existing systems based exclusively on taint tracking.
February 4th – February 17th	Report work, project presentation preparation
February 18th – March 3rd	Project presentation preparation, report work
March 4th – March 17th	Project presentation delivery, report finalisation.

Table 1: Projected work by time period. Deadlines for other modules included where known.

[1] As a simplified example, an `unsafe_shell_exec` function might safely be able to accept any input that matches `^[^`]$`

Table 1 provides a breakdown of the project time into periods for each fortnight, along with the expected work to be completed. A meeting will be scheduled for each week to discuss progress and any road-blocks that arise with the project supervisor

Methodology

Software Engineering

This project includes an element of software engineering. Namely, the design and implementation of a proof of concept language which supports a type system incorporating regular expression refinement types.

This implementation work will be carried out in an Agile fashion to fit in with the short timescales inherent to the project and allow for greater flexibility. Time periods in the schedule which involve development work will be treated as a number of week-long development sprints with priorities formalised prior to the commencement of each period. Progress will be reviewed in weekly supervision meetings.

Testing will be automated via the use of unit testing to ensure that specific components function according to their specification in isolation. Where appropriate, integration and system testing can be used to test the solution as a whole (for example, an entire program as a test case would fit into this part of the testing process).

Evaluation

Towards the end of the implementation phase, the false positive rate of the proof of concept tool should be compared with that of existing static analysis tooling based on taint tracking alone.

Logically equivalent test cases should be built for each tool under evaluation in the necessary programming language. Both safe and unsafe function invocations should be included in each test case for completeness.

Resources and risks

The project is reliant on a number of resources. Use of these resources is subject to the risks outlined in table 2. These risks should be evaluated and managed to minimise any potential impact on the project.

Resource	Applicable risk(s)	Impact
VCS hosting: GitHub Storing and tracking code and report changes	Loss of availability due to outage	Minimal, *git* is decentralised so copy of files always available locally and at off-site backup
Report authoring: LaTeX Writing and compiling the report, tracking bibliography	Obsolescence	Unlikely, TeX tooling has been used for decades. Even if particular packages ceased working, the bulk of the content would still be accessible as plain text.
C# analysis: Roslyn libary Fulfilling the additional objective by analysing C# code	Loss of availability due to license change	Minimal. Even if Roslyn's OSS status changes, there is no requirement to integrate with C#, other languages would illustrate the potential just as well. This would also not impact a primary project objective.
Self Project work	Illness, coursework deadlines	Minimised by scheduled slack time and identification of applicable coursework deadlines.

Table 2: Resources and associated risks.

Legal, social, ethical and professional issues

As a project with some security motivation, it is possible that legal, social, ethical and professional issues will arise. In particular, evaluation of existing static analysis tooling must be performed with care to ensure that use of any particular external test cases is permitted by the *Copyright, Designs and Patents Act 1988* within the UK.

Additionally, in order to comply with the *Computer Misuse Act 1988*, any static analysis of external code should only be conducted with permission. Discovered issues should be disclosed responsibly.

References

Arteau, P. (2016), 'Roslyn Security Guard'.

Christey, S. and Martin, R. A. (2007), 'Vulnerability type distributions in CVE'.

Denning, D. E. and Denning, P. J. (1977), 'Certification of programs for secure information flow', *Communications of the ACM* **20**(7), 504–513.

Doupé, A., Cova, M. and Vigna, G. (2010), Why Johnny can't pentest: An analysis of black-box web vulnerability scanners, *in* 'International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment', Springer, pp. 111–131.

Pierce, B. (2002), *Types and Programming Languages*, The MIT Press.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L. and Jaspan, C. (2018), 'Lessons from building static analysis tools at google', *Commun. ACM* **61**(4), 58–66.
URL: *http://doi.acm.org/10.1145/3188720*