MPhil to PhD Upgrade Report

Fahad Ausaf
Department of Informatics
King's College London
Email: fahad.ausaf@kcl.ac.uk

November 9, 2015

Abstract

This research is about regular expression matching. The interest in this topic arose from a 2014 paper by Sulzmann and Lu [16]. This paper contains serious errors in the proofs, which I try to fix. Once the errors are corrected, I attempt to make the algorithm as fast as possible and verify its translation to machine code.

1 Introduction

This work focuses on regular expression matching. When a string is matched against a regular expression, we are interested in the *matching tree* containing the information on how different parts of the regular expression matched the input string. This information is useful, for example, when lexing, or tokenising, programs. In general there are more than one matching tree for a regular expression and a string. However, there are two commonly used disambiguation strategies to create a unique matching tree: one is called *greedy* matching [8] and the other is *POSIX* matching [10, 16]. For the latter there are two rough rules:

- The Longest Match Rule (or "maximal munch rule"):

  The longest initial substring matched by any regular expression is taken as next token.

- Rule Priority:

  For a particular longest initial substring, the first regular expression that can match determines the token.

In the context of lexing, POSIX is the more interesting disambiguation strategy as it produces longest matches, which is necessary for tokenising programs. For example the string *iffoo* should not match the keyword *if* and the rest, but

as one string *iffoo*, which might be a variable name in a program. As another example consider the string xy and the regular expression $(x + y + xy)^*$. Either the input string can be matched in two 'iterations' by the single letter-regular expressions x and y, or directly in one iteration by xy. The first case corresponds to greedy matching, which first matches with the left-most symbol and only matches the next symbol in case of a mismatch. The second case is POSIX matching, which prefers the longest match. In case more than one (longest) matches exist, only then it prefers the left-most match. While POSIX matching seems natural, it turns out to be much more subtle than greedy matching in terms of implementations and in terms of proving properties about it. If POSIX matching is implemented using automata, then one has to follow transitions (according to the input string) until one finds an accepting state, record this state and look for further transition which might lead to another accepting state that represents a longer input initial substring to be matched. Only if none can be found, the last accepting state is returned.

Sulzmann and Lu's paper [16] targets POSIX regular expression matching. They write that it is known to be to be a non-trivial problem and nearly all POSIX matching implementations are "buggy" [16, Page 203]. For this they cite a study by Kuklewicz [10]. My current work is about formalising the proofs in the paper by Sulzmann and Lu. Specifically, they propose in this paper a POSIX matching algorithm and give some details of a correctness proof for this algorithm inside the paper and some more details in an appendix. This correctness proof is unformalised, meaning it is just a "pencil-and-paper" proof, not done in a theorem prover. Though, the paper and presumably the proof have been peer-reviewed. Unfortunately their proof does not give enough details such that it can be straightforwardly implemented in a theorem prover, say Isabelle. In fact, the purported proof they outline does not work in central places. We discovered this when filling in many gaps and attempting to formalise the proof in Isabelle.

Although we pointed out a number of problems in their proof, Sulzmann and Lu seem to not overly interested in fixing them so that the proofs could be formalised in a theorem prover. In private communication, Sulzmann wrote about one flaw we pointed out

> *"How could I miss this? Well, I was rather careless when stating this Lemma…Great example [of] how formal machine checked proofs (and proof assistants) can help to spot flawed reasoning steps."*

but about another problem he wrote

> *"Well, I don't think there's any flaw. The issue is how to come up with a mechanical proof. In my world mathematical proof = mechanical proof doesn't necessarily hold."*

Since working with the Isabelle theorem prover, I clearly disagree with this last statement. I am trying to formalise a correctness proof (not necessarily the one by Sulzmann and Lu, because of the problems). Once finished we can be really sure their algorithm is correct. This work is currently in progress.

2 Background

Six years ago or so, Leroy and his coworkers verified a compiler, called *CompCert*, compiling a rather large subset of C to machine code [11]. The promise this compiler gives is that if a source program has a certain observable behaviour, then also the generated machine code will have this behaviour. This is a strong promise that is more often than not violated by conventional compilers. Bugs usually creep in when optimisations are applied, but not all necessary safety conditions are observed.

In 2011, Regehr et al [22] introduced a compiler testing tool, called CSmith. They tested C-compilers by generating in a clever way "random" C-programs. These random programs were compiled with GCC, LLVM and other commercial C-compilers. If one compiler miscompiled a program (just showed a different behaviour) then they know there must be something anomalous going on. They can then investigate whether or not there is a bug in that C-compiler. In this way they found, for example, more than 300 bugs in GCC and also many bugs in LLVM. In contrast, about CompCert, which was also one of their target C-compilers, Regehr et al wrote [22, Page 288]:

> *"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task."*

While this is an amazing result showing how trustworty the CompCert compiler is, Regehr et al did manage to find five bugs, which were in the unverified front-end part of CompCert. Although these bugs were not necessarily in the lexing/parsing phase of the CompCert compiler, our work on verifying a lexer would of course be needed if CompCert or other verified compilers extend the verification to include all parts of their front-ends.

Traditional regular expression matchers are based on automata. The process of how to derive an automaton (potentially a non-deterministic automaton) from a regular expression is well-understood and widely researched.t There are many algorithms for this derivation (for example [18]). While the matching algorithms involving automata are usually very elegant and fast, their problem is that it is rather difficult to reason formally (in a theorem prover) about automata. This might be surprising, but can be explained by the fact that automata are essentially a restricted class of graphs—the states are the nodes and the transitions are the edges. There needs to be a transition going out from a node for every letter in the alphabet. This restriction does not lead to a simple definition for automata in theorem provers. Wu et al managed to formalise a large part of regular language theory in Isabelle by side-stepping automata completely [21]. They argue that regular expressions and Brozozwski's notion of regular expression derivatives are much more convenient for reasoning about regular language theory. We hope to also get milage out of the fact that

Sulzmann and Lu use regular expressions and their derivatives as central components in their POSIX matching algorithm. However there are also dissenting opinions: for example Paulson [15] managed to formally verify a number of results about automata in Isabelle using a construction based on hereditary finite sets.

While Brzozowski's regular expression derivatives [5] are a rather old idea, and they are well-known in the text algorithmics community, they seem to have been forgotten over time in the functional programming and theorem prover communities. According to [13, Page 173] they had been "lost in the sands of time, and few computer scientists are aware of them." But since [13], there has been a furry of papers using derivatives in these two communities, for example [6, 17], and specifically in the theorem prover community, for example, [1, 3, 7, 9, 14, 19, 20]. However, apart from Sulzmann and Lu's paper, we are not aware of any work on verified lexers based on the POSIX rules (neither automata-bases, nor regular expression-based). The closest related work to ours is the work by Nipkow [12] on a greedy lexer, and by Norrish and his student [4] about a formally verified parsing algorithm.

3 Terminologies

Since regular expression matching is an old field and terminology varies widely, let us first fix our basic terminology and definitions. We use *, $\cdot$, $+$, ϵ, ϕ for Kleene star, concatenation, alternative, empty string and empty language. Regular expression are therefore given by the grammar

$$r ::= \phi \mid \epsilon \mid c \mid r_1 + r_2 \mid r_1 \cdot r_2 \mid r^*$$

where c is a letter of the alphabet, which we leave unspecified. For brevity, we often omit the $\cdot$ in the concatenation. Importantly in our context, there is also an associated notion of *values*, which record *how* a regular expression has matched a string. These values are given by the grammar

$$v ::= Void \mid Char\ c \mid Left(v) \mid Right(v) \mid (v_1, v_2) \mid [] \mid v :: vs$$

The last two stand for the empty list and list-cons. There is no value for ϕ since it does not match anything. *Void* is for ϵ, *Char c* for the character c and so on. It is easy to extract the underlying string from a value by the following *flattening operation*:

$$
\begin{array}{lcl}
|Void| & \stackrel{\text{def}}{=} & [] \\
|Char\ c| & \stackrel{\text{def}}{=} & [c] \\
|Left(v)| & \stackrel{\text{def}}{=} & |v| \\
|Right(v)| & \stackrel{\text{def}}{=} & |v| \\
|(v_1, v_2)| & \stackrel{\text{def}}{=} & |v_1|\,|v_2| \\
|[]| & \stackrel{\text{def}}{=} & [] \\
|v :: vs| & \stackrel{\text{def}}{=} & |v|\,|vs|
\end{array}
$$

where $[c]$ is the string consisting of only the character c and $[]$ is the empty string. Both Sulzmann and Lu, and Frisch and Cardelli use a kind of *type inhabitation relation* for values and regular expressions. This relation is defined by the rules

$$
\frac{}{\vdash [] : r^*} \qquad \frac{\vdash v : r \quad \vdash vs : r^*}{\vdash (v :: vs) : r^*}
$$

$$
\frac{\vdash v_1 : r_1 \vdash \quad v_2 : r_2}{\vdash (v_1, v_2) : r_1\, r_2}
$$

$$
\frac{\vdash v_1 : r_1}{\vdash Left(v_1) : r_1 + r_2} \qquad \frac{\vdash v_2 : r_2}{\vdash Right(v_2) : r_1 + r_2}
$$

$$
\frac{}{\vdash Void : \epsilon} \qquad \frac{}{\vdash Char\ c : c}
$$

The main idea of this type-like relation is the following property about the language, L, of a regular expression:

$$L(r) = \{|v| \;.\; \vdash v : r\}$$

This means that if we take all values "inhabited" in a regular expression and extract the underlying strings (by flattening), we obtain the language matched by the regular expression. The point of the values is that they contain more structure by encoding how a regular expression has matched a string. In the example with the regular expression $(x + y + xy)^*$ we have the following two values

$$[Left(Char\ x), Right(Left(Char\ y))] \quad \text{and} \quad [Right(Right((Char\ x, Char\ y)))]$$

giving two different answers to how the regular expression matched the string xy (the left-value states that two iteration are needed with the star, while the value on the right-hand side states only one iteration is needed). There are of course more values inhabited in this regular expression (in fact infinitely many).

4 The POSIX Matching Algorithm by Sulzmann and Lu

Sulzmann and Lu presented their POSIX matching algorithm in 2014 at the FLOPS conference [16]. This algorithm consists of two phases: a matching phase and a value construction phase. The matching phase goes back to a quite old idea from 1964 by Brzozowski [5], called *derivatives of regular expressions*. They can be used for a simple matching The matching phase can only decide whether a string is matched by a regular expression or not. The novel idea of Sulzmann and Lu is to add a second phase that, given a string is actually matched by a regular expression, constructs a value giving an answer for *how* the regular expression matched the string.

4.1 The Matching Phase

Central to the matching phase are two functions called *nullable* and *der*. The former decides when a regular expression can match the empty string or not. It can be defined by recursion over regular expressions as follows:

Nullable Function

$$
\begin{array}{lcl}
nullable(\phi) & \stackrel{\text{def}}{=} & false \\
nullable(\epsilon) & \stackrel{\text{def}}{=} & true \\
nullable(c) & \stackrel{\text{def}}{=} & \text{false} \\
nullable(r_1 + r_2) & \stackrel{\text{def}}{=} & nullable(r_1) \vee nullable(r_2) \\
nullable(r_1\, r_2) & \stackrel{\text{def}}{=} & nullable(r_1) \wedge nullable(r_2) \\
nullable(r^*) & \stackrel{\text{def}}{=} & true
\end{array}
$$

The second function, called *der*, is quite subtle: it takes a character, say c, and a regular expression, say r, as input and returns the derivative regular expression. The idea of the the derivative regular expression is: if r can match strings of the form $c :: s$, what does the regular expression look like that can match the strings s (where the leading character c has been "chopped off")? Again this function can be defined by recursion on regular expressions.

Derivative Function

$$
\begin{array}{lcl}
der\ c\ \phi & \stackrel{\text{def}}{=} & \phi \\
der\ c\ \epsilon & \stackrel{\text{def}}{=} & \phi \\
der\ c\ d & \stackrel{\text{def}}{=} & \text{if } c = d \text{ then } \epsilon \text{ else } \phi \\
der\ c\ r_1 + r_2 & \stackrel{\text{def}}{=} & der\ c\ r_1 + der\ c\ r_2 \\
der\ c\ r_1 \cdot r_2 & \stackrel{\text{def}}{=} & \text{if } nullable\ r_1 \\
 & & \text{then } (der\ c\ r_1) \cdot r_2 + der\ c\ r_2 \\
 & & \text{else } (der\ c\ r_1) \cdot r_2 \\
der\ c\ r^* & \stackrel{\text{def}}{=} & (der\ c\ r) \cdot r^*
\end{array}
$$

The first two cases are straightforward: for this recall that *der* should calculate a regular expression so that given the "input" regular expression can match a string of the form $c :: s$, we want a regular expression for s. Since neither ϕ nor ϵ can match a string of the form $c :: s$, we return ϕ. In the character case we have to make a case-distinction: In case the regular expression is c, then clearly it can recognise a string of the form $c :: s$, just that s is the empty string. Therefore we return the ϵ-regular expression. In the other case we again return ϕ since no string of the $c :: s$ can be matched. Rationalising the recursive cases are a bit more involved. Fortunately, the $+$-case is still relatively simple: all strings of the form $c :: s$ are either matched by the regular expression r_1 or r_2. So we just have to recursively call *der* with these two regular expressions and compose the results again with $+$. The $\cdot$-case is more complicated: if $r_1 \cdot r_2$ matches a string of the form $c :: s$, then the first part must be matched by r_1. Consequently, it makes sense to construct the regular expression for s by calling *der* with r_1 and "appending" r_2. There is however one exception to this simple rule: if r_1 can match the empty string, then all of $c :: s$ is matched by r_2. Consequently in case r_1 is *nullable* (that is can match the empty string) we have to allow the choice $der\, c\, r_2$ for calculating the regular expression that can match s. This means we have to add the regular expression $der\, c\, r_2$ in the result. The $*$-case is again simple: if r^* matches a string of the form $c :: s$, then the first part must be "matched" by a single copy of r. Therefore we call recursively $der\, c\, r$ and "append" r^* in order to match the rest of s.

We can lift the derivative of regular expressions from single characters to strings as follows:

$$\begin{array}{lcl} ders\, []\, r & \stackrel{\text{def}}{=} & r \\ ders\, (c :: s)\, r & \stackrel{\text{def}}{=} & ders\, s\, (der\, c\, r) \end{array}$$

Let us look at some examples. Suppose the regular expression r_0 is $(a + ab)(b + \epsilon)$ and the input string is ab. Obviously r_0 can match the input string. Below are the intermediate steps for calculating *ders*.

$$\begin{array}{lcl} r_1 = der\ a\ r_0 & = & (\epsilon + \epsilon b)(b + \epsilon) \\ r_2 = der\ b\ r_1 & = & (\phi + \phi b + \epsilon)(b + \epsilon) + (\epsilon + \phi) \end{array}$$

The point of the derivative is that we can decide whether a string is matched by a regular expression, if the final derivative (after exhausting the string) can match the empty string, that is it is *nullable*. So a regular expression matcher can be defined as

$$matches\ s\ r \stackrel{\text{def}}{=} nullable(ders\ s\ r)$$

This is essentially Brzozowski's algorithm from 1964. Its main virtue is that the algorithm can be easily implemented as a functional program and easily be reasoned about in theorem provers: regular expressions and values are simple inductive datatypes, and *nullable*, *der* and *ders* are straightforward recursive

functions. Therefore the correctness proof for *matches*, that is establishing the property

$$matches\ s\ r \ \text{ if and only if } \ s \in L(r)$$

is a relatively simple exercise in a theorem prover (for example done in [14] using the HOL theorem prover, but also part of the *Archive of Formal Proofs* for Isabelle). Another virtue of this algorithm is that it can be easily extended to the not-regular expression and other practically useful regular expressions.

The main disadvantage is that the version of *matches* explained above behaves abysmal in terms of running time. The reason is that the *der*-function can grow the regular expression and an increased size of the regular expression means more work for the following stages, since the "next" *der* needs to traverse the resulting regular expression again. However, this problem can be solved easily by noticing that the derivative produces many instances of ϕ and ϵ, which can be simplified away. In this way the regular expressions can be made smaller and the algorithm runs quite fast. We omit here mentioning anything about the simplification step, because it is not relevant for the correctness proof.[1]

4.2 The Value Construction Phase

While the matching phase only calculates a yes or no answer for whether a regular expression matches a string, the novel idea of Sulzmann and Lu [16] is to append another phase to calculate a value. For this consider again the definition of regular expressions and values side-by-side:

Regular Expressions			**Values**		
r	$::=$	ϕ	v	$::=$	
	$\mid$	ϵ			$Void$
	$\mid$	c		$\mid$	$Char(c)$
	$\mid$	$r_1 \cdot r_2$		$\mid$	(v_1, v_2)
	$\mid$	$r_1 + r_2$		$\mid$	$Left(v)$
				$\mid$	$Right(v)$
	$\mid$	r^*		$\mid$	$[v_1, \ldots v_n]$

For each regular expression, there is a specific value that records how the regular expression matched the string. The exception is the ϕ-regular expression, because it cannot match anything and therefore does not need a corresponding value; and the two values for the alternative regular expression, which correspond to the two choices in the alternative.

The values need to be calculated in the second phase of the Sulzmann and Lu algorithm. Central to their algorithm are two functions called *mkeps* and

[1] The hard part is to prove that the slow version is correct, and then to show that the improved version still produces the same result. We guess, and this is confirmed by the comments in [16], that the second part is not difficult to establish.

inj (injection). If in the last step of the first phase, *nullable* returns true, then *mkeps* will generate a value that shows how the last derivative regular expression could have matched the empty string. The function *mkeps* can be defined recursively as follows:

***Mkeps* Function**

$$
\begin{array}{lcl}
mkeps(\epsilon) & \stackrel{\text{def}}{=} & Void \\
mkeps(r_1 + r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } Left(mkeps(r_1)) \\
 & & \text{else } Right(mkeps(r_2)) \\
mkeps(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & (mkeps(r_1), mkeps(r_2)) \\
mkeps(r^*) & \stackrel{\text{def}}{=} & []
\end{array}
$$

Notice how this function makes some subtle choices leading to a POSIX value at the end of the second phase: for example if the alternative, say $r_1 + r_2$, can match the empty string and furthermore r_1 can match the empty string, then we return a $Left$-value. The $Right$-value will only be returned if r_1 is not *nullable*. The four regular expressions in *mkeps* are the only cases we need to consider, since the other regular expressions cannot match the empty string. Below is the calculation of *mkeps* with the intermediate steps shown when applied to the regular expression obtained after the matching phase of the algorithm (see r_2 in Section 4.1)

$$
\begin{array}{ll}
 & mkeps((\phi + (\phi b + \epsilon))(b + \epsilon) + (\epsilon + \phi)) \\
= & Left(mkeps((\phi + (\phi b + \epsilon))(b + \epsilon))) \\
= & Left((mkeps(\phi + (\phi b + \epsilon)), mkeps(b + \epsilon))) \\
= & Left((Right(mkeps(\phi b + \epsilon)), Right(mkeps(\epsilon)))) \\
= & Left((Right(Right(mkeps(\epsilon))), Right(Void))) \\
= & Left((Right(Right(Void)), Right(Void))) = v_2
\end{array}
$$

Given the regular expression

$$(\phi + (\phi b + \underline{\epsilon}))(b + \underline{\epsilon}) + (\epsilon + \phi)$$

the value calculated by *mkeps* corresponds to the two underlined ϵs which are responsible (according to POSIX matching) for recognising the empty string.

The more interesting function in the second phase is called *injection* and written *inj*. Recall that the derivative essentially "chops off" a single character from a regular expression. The injection function undoes this "chopping off" by injecting back a character…just on the level of values, rather than regular expressions. The *inj* function can be defined recursively as follows:

Injection Function

$$
\begin{array}{lcl}
inj(c)\ c\ Void & \stackrel{\text{def}}{=} & Char\ c \\
inj(r_1 + r_2)\ c\ Left(v) & \stackrel{\text{def}}{=} & Left(inj\ r_1\ c\ v) \\
inj(r_1 + r_2)\ c\ Right(v) & \stackrel{\text{def}}{=} & Right(inj\ r_2\ c\ v) \\
inj(r_1 \cdot r_2)\ c\ (v_1, v_2) & \stackrel{\text{def}}{=} & (inj\ r_1\ c\ v_1, v_2) \\
inj(r_1 \cdot r_2)\ c\ Left((v_1, v_2)) & \stackrel{\text{def}}{=} & (inj\ r_1\ c\ v_1, v_2) \\
inj(r_1 \cdot r_2)\ c\ Right(v) & \stackrel{\text{def}}{=} & (mkeps(r_1), inj\ r_2\ c\ v) \\
inj(r^*)\ c\ (v, vs) & \stackrel{\text{def}}{=} & inj\ r\ c\ v :: vs
\end{array}
$$

A sanity property Sulzmann and Lu prove, and which we could formally verify is

$$\text{if}\ \ \vdash v : r\ \ \text{then}\ \ |inj\ r\ c\ v| = c :: |v|$$

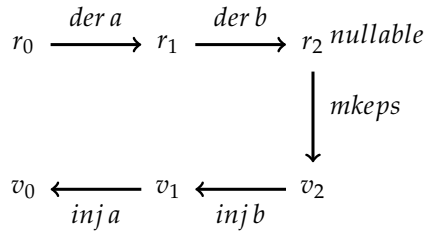
meaning the character c is really injected back into the value v—the underlying string is extended at the front by c. Below are the details of intermediate steps after applying the *inj* function with r_2, v_2 and b

$$
\begin{array}{l}
\quad inj((\epsilon + \epsilon b)(b + \epsilon))\ b\ Left((Right(Right(Empty)), Right(Empty))) \\
= (inj(\epsilon + \epsilon b) b Right(Right(Void)), Right(Void)) \\
= (Right(inj(\epsilon b) b Right(Void)), Right(Void)) \\
= (Right((mkeps(\epsilon), inj(b) b Void)), Right(Void)) \\
= (Right(Void, Char(b)), Right(Void))
\end{array}
$$

To sum up, we started with the string ab and the regular expression $r_0 = (a + \underline{ab})(b + \underline{\epsilon})$. For this the algorithm returns the value

$$(Right((Char(a), Char(b))), Right(Void))$$

as answer, which means the underlined parts of r_0 contributed (according to POSIX rules) how the string ab is matched. Pictorially the whole process can be illustrated as follows:



where the upper line is the first phase calculating derivatives or regular expressions. At the end of the line we test whether the resulting regular expression is nullable; if yes, then we call *mkeps* in order to calculate the first value (v_3 in the

picture). Then we go back on the lower line (the second phase) to the beginning of the string. For this the *inj* function takes the regular expression r_{i-1} and v_i, as well as the character from the derivative, as input and calculates v_{i-1}.

Again the virtues of this algorithm is that it can be implemented with ease. Although we could verify the first phase already and also all facts about *mkeps* from the paper by Sulzmann and Lu, reasoning about the second phase and and ordering $\succcurlyeq^r_{PX}$ (explained below) is not so straightforward. We shall show this next, but before we shall outline the definitions from [8] about a greedy regular matching algorithm. This is where the inspiration for Sulzmann and Lu came from.

5 Greedy Ordering Rules

Sulzmann and Lu explicitly refer to the paper [8] by Frisch and Cardelli from where they have taken their main idea for their correctness proof of the POSIX algorithm. Frisch and Cardelli introduced an ordering, written $\succcurlyeq_{gr}$, for values and they show that their greedy matching algorithm always produces a maximal element according to this ordering (from all possible solutions). The ordering $\succcurlyeq_{gr}$ is defined by the following rules:

$$\frac{v_1 \succcurlyeq_{gr} v_1'}{(v_1, v_2) \succcurlyeq_{gr} (v_1', v_2')} \qquad \frac{v_2 \succcurlyeq_{gr} v_2'}{(v_1, v_2) \succcurlyeq_{gr} (v_1, v_2')}$$

$$\frac{v_1 \succcurlyeq_{gr} v_2}{Left\ v_1 \succcurlyeq_{gr} Left\ v_2} \qquad \frac{v_1 \succcurlyeq_{gr} v_2}{Right\ v_1 \succcurlyeq_{gr} Right\ v_2}$$

$$\frac{}{Left\ v_2 \succcurlyeq_{gr} Right\ v_1}$$

$$\frac{v_1 \succcurlyeq_{gr} v_2}{v_1 :: vs_1 \succcurlyeq_{gr} v_2 :: vs_2} \qquad \frac{vs_1 \succcurlyeq_{gr} vs_2}{v :: vs_1 \succcurlyeq_{gr} v :: vs_2}$$

$$\frac{}{v :: vs \succcurlyeq_{gr} []}$$

$$\frac{}{Char\ c \succcurlyeq_{gr} Char\ c} \qquad \frac{}{Void \succcurlyeq_{gr} Void}$$

That these rules realise a greedy ordering can be seen in the first rule in the third line where a $Left$-value is always bigger than (or preferred over) a *Right*-value. What is interesting for our purposes is that the properties reflexivity, totality and transitivity for this greedy ordering can be proved relatively easily by induction.[2] This is illustrated next:

Lemma 1 (Reflexivity)**.** *If* $\vdash v : r$ *then* $v \succcurlyeq_{gr} v$.

Proof. This is by a straightforward induction on the definition of $\vdash v : r$. □

[2] I have formalised these proofs in Isabelle.

Lemma 2 (Totality). *If* $\vdash v_1 : r$ *and* $\vdash v_2 : r$ *then* $v_1 \succcurlyeq_{gr} v_2$ *or* $v_2 \succcurlyeq_{gr} v_1$.

Proof. This is again by a straightforward induction on the definition of $\vdash v_1 : r$ and a case-analysis of $\vdash v_2 : r$. □

More interesting is the proof for transitivity which can be shown by induction on r.

Lemma 3 (Transitivity). *Suppose* $\vdash v_1 : r$, $\vdash v_2 : r$ *and* $\vdash v_3 : r$. *If* $v_1 \succcurlyeq_{gr} v_2$ *and* $v_2 \succcurlyeq_{gr} v_3$ *then* $v_1 \succcurlyeq_{gr} v_3$.

Proof. By induction on r analysing all cases of $\vdash v_1 : r$ and so on. The only interesting case is for sequences, where we can assume $v_1 = (v_{1l}, v_{1r})$, $v_2 = (v_{2l}, v_{2r})$, and $v_3 = (v_{3l}, v_{3r})$. We need to show that $(v_{1l}, v_{1r}) \succcurlyeq_{gr} (v_{3l}, v_{3r})$ under the assumptions that $(v_{1l}, v_{1r}) \succcurlyeq_{gr} (v_{2l}, v_{2r})$ and $(v_{2l}, v_{2r}) \succcurlyeq_{gr} (v_{3l}, v_{3r})$ hold. There are two rules which could have derived each assumption. For example $v_{1l} \succcurlyeq_{gr} v_{2l}$ and $v_{2l} \succcurlyeq_{gr} v_{3l}$. In this case we can apply the induction hypothesis and derive $v_{1l} \succcurlyeq_{gr} v_{3l}$ from this we obtain $(v_{1l}, v_{1r}) \succcurlyeq_{gr} (v_{3l}, v_{3r})$. The other three cases are similar (where in one case we need to appeal to the reflexivity property). □

It should not come as a surprise that if we make changes to the ordering rules, the proof ideas behind these proofs might not necessarily transfer to the modified rules. That is what we shall show in the next section about the POSIX ordering rules introduced by Sulzmann and Lu.

6 POSIX Ordering Rules and Correctness Proofs

There are a number of proofs in [16] that suffer from a problem where the induction hypothesis is not strong enough. This problem arises from how the POSIX ordering rules are defined and what properties they satisfy. Recall that the correctness proof of the POSIX matching algorithm in [16] needs to establish that a maximal value is returned amongst all possible values. That requires for example a proof that there is a unique maximal element (otherwise there might be several "correct" answers, which would be counter intuitive).

As mentioned before, the rules by Sulzmann and Lu are a variant of the greedy rules by Cardelli and Frisch. The difference is that Sulzmann and Lu's ordering, written $\succcurlyeq^r_{PX}$, also includes a regular expression.[3] The rules are as follows:

[3]Though, we have already shown that the regular expression in the $\succcurlyeq^r_{PX}$-ordering is redundant—it does not contribute to what is defined.

$$\frac{v_1 \succcurlyeq^{r_1}_{PX} v_1'}{(v_1, v_2) \succcurlyeq^{r_1 r_2}_{PX} (v_1', v_2')} \qquad \frac{v_2 \succcurlyeq^{r_2}_{PX} v_2'}{(v_1, v_2) \succcurlyeq^{r_1 r_2}_{PX} (v_1, v_2')}$$

$$\frac{len|v_2| > len|v_1|}{Right\ v_2 \succcurlyeq^{r_1+r_2}_{PX} Left\ v_1} \qquad \frac{len|v_1| \geq len|v_2|}{Left\ v_1 \succcurlyeq^{r_1+r_2}_{PX} Right\ v_2}\,(P2)$$

$$\frac{v_2 \succcurlyeq^{r_2}_{PX} v_2'}{Right\ v_2 \succcurlyeq^{r_1+r_2}_{PX} Right\ v_2'} \qquad \frac{v_1 \succcurlyeq^{r_1}_{PX} v_1'}{Left\ v_1 \succcurlyeq^{r_1+r_2}_{PX} Left\ v_1'}$$

$$\frac{|v : vs| = []}{[] \succcurlyeq^{r^*}_{PX} v : vs} \qquad \frac{|v : vs| \neq []}{v : vs \succcurlyeq^{r^*}_{PX} []}$$

$$\frac{v_1 \succcurlyeq^{r}_{PX} v_2}{v_1 :: vs_1 \succcurlyeq^{r^*}_{PX} v_2 :: vs_2} \qquad \frac{vs_1 \succcurlyeq^{r^*}_{PX} vs_2}{v :: vs_1 \succcurlyeq^{r^*}_{PX} v :: vs_2}$$

$$\frac{}{Char\ c \succcurlyeq^{c}_{PX} Char\ c} \qquad \frac{}{Void \succcurlyeq^{\epsilon}_{PX} Void}$$

The interesting rules are in the second line. For this remember that the greedy ordering always prefers a $Left$-value over a $Right$-value. This is different in POSIX rules above: there a $Right$-value is preferred when it can match a longer string (the rule on the left); a $Left$-value is only preferred when it can match a longer or equal string than the $Right$-value. Perhaps surprisingly, but perhaps not, this "small" change has drastic consequences for the proofs.

To start with, transitivity does not hold anymore in the "normal" formulation, that is:

Property 1. *Suppose* $\vdash v_1 : r$, $\vdash v_2 : r$ *and* $\vdash v_3 : r$. *If* $v_1 \succcurlyeq^{r}_{PX} v_2$ *and* $v_2 \succcurlyeq^{r}_{PX} v_3$ *then* $v_1 \succcurlyeq^{r}_{PX} v_3$.

If formulated like this, then there are various counter examples: Suppose r is $a + ((a + a)(a + \epsilon))$ then the v_1, v_2 and v_3 below are values of r:

$$\begin{array}{lcl} v_1 & = & Left(Char\ a) \\ v_2 & = & Right((Left(Char\ a), Right(Void))) \\ v_3 & = & Right((Right(Char\ a), Left(Char\ a))) \end{array}$$

Moreover $v_1 \succcurlyeq^{r}_{PX} v_2$ and $v_2 \succcurlyeq^{r}_{PX} v_3$, but *not* $v_1 \succcurlyeq^{r}_{PX} v_3$! The reason is that although v_3 is a $Right$-value, it can match a longer string, namely $|v_3| = aa$, while $|v_1|$ (and $|v_2|$) matches only a. So transitivity in this formulation does not hold—in this example actually $v_3 \succcurlyeq^{r}_{PX} v_1$!

Sulzmann and Lu "fix" this problem by weakening the transitivity property. They require in addition that the underlying strings are of the same length. This excludes the counter example above and any counter-example we could find with our implementation. Thus the transitivity lemma in [16] is:

Property 2. *Suppose* $\vdash v_1 : r$, $\vdash v_2 : r$ *and* $\vdash v_3 : r$, *and also* $|v_1| = |v_2| = |v_3|$. *If* $v_1 \succcurlyeq^{r}_{PX} v_2$ *and* $v_2 \succcurlyeq^{r}_{PX} v_3$ *then* $v_1 \succcurlyeq^{r}_{PX} v_3$.

While we agree with Sulzmann and Lu that this property probably holds, proving it seems not so straightforward. Sulzmann and Lu do not give an explicit proof of the transitivity property, but give a closely related property about the existence of maximal elements. They state that this can be verified by an induction on r. We disagree with this as we shall show next in case of transitivity.

The case where the reasoning breaks down is the sequence case, say $r_1\, r_2$. The induction hypotheses in this case are

$$\begin{array}{lll} \text{IH } r_1\text{:} & & \text{IH } r_2\text{:} \\ \forall v_1, v_2, v_3. & \vdash v_1 : r_1 \wedge & \forall v_1, v_2, v_3. \quad \vdash v_1 : r_2 \wedge \\ & \vdash v_2 : r_1 \wedge & \vdash v_2 : r_2 \wedge \\ & \vdash v_3 : r_1 \wedge & \vdash v_3 : r_2 \wedge \\ & |v_1| = |v_2| = |v_3| \wedge & |v_1| = |v_2| = |v_3| \wedge \\ & v_1 \succcurlyeq^{r_1}_{PX} v_2 \ \wedge \ v_2 \succcurlyeq^{r_1}_{PX} v_3 & v_1 \succcurlyeq^{r_2}_{PX} v_2 \ \wedge \ v_2 \succcurlyeq^{r_2}_{PX} v_3 \\ & \Rightarrow v_1 \succcurlyeq^{r_1}_{PX} v_3 & \Rightarrow v_1 \succcurlyeq^{r_2}_{PX} v_3 \end{array}$$

We can assume that

$$(v_{1l}, v_{1r}) \succcurlyeq^{r_1 r_2}_{PX} (v_{2l}, v_{2r}) \qquad \text{and} \qquad (v_{2l}, v_{2r}) \succcurlyeq^{r_1 r_2}_{PX} (v_{3l}, v_{3r}) \tag{1}$$

hold, and furthermore that the values have equal length, namely:

$$|(v_{1l}, v_{1r})| = |(v_{2l}, v_{2r})| \qquad \text{and} \qquad |(v_{2l}, v_{2r})| = |(v_{3l}, v_{3r})| \tag{2}$$

We need to show that

$$(v_{1l}, v_{1r}) \succcurlyeq^{r_1 r_2}_{PX} (v_{3l}, v_{3r})$$

holds. We can proceed by analysing how the assumptions in (1) have arisen. There are four cases. Let us assume we are in the case where we know

$$v_{1l} \succcurlyeq^{r_1}_{PX} v_{2l} \qquad \text{and} \qquad v_{2l} \succcurlyeq^{r_1}_{PX} v_{3l}$$

and also know the corresponding typing judgements. This is exactly a case where we would like to apply the induction hypothesis IH r_1. But we cannot! We still need to show that $|v_{1l}| = |v_{2l}|$ and $|v_{2l}| = |v_{3l}|$. We know from (2) that the lengths of the sequence values are equal, but from this we cannot infer anything about the lengths of the component values. Indeed in general they will be unequal, that is

$$|v_{1l}| \neq |v_{2l}| \qquad \text{and} \qquad |v_{1r}| \neq |v_{2r}|$$

but still (2) will hold. Now we are stuck, since the IH does not apply. Sulzmann and Lu overlook this fact and just apply the IHs. Obviously nothing which a theorem prover allows us to do.

The immediate effect is that the existence of a unique maximal value cannot be inferred. We know totality of $\succcurlyeq^{r}_{PX}$ and know that for every regular expression there are only a finite number of (proper) values. But without transitivity

it seems hard to establish that given a regular expression and given a string, there exists always a unique maximal value…which the algorithm is supposed to calculate. Without this basic property, the whole correctness proofs already collapses.

But let us assume we could somehow establish this properties by other reasoning steps. Surprisingly *all* main proofs in [16] are "infected" from this bogus reasoning, not just transitivity. There is a proof in [16] that the inj-function preserves maximal elements. This follows similar reasoning and therefore breaks down when subvalues of a sequence are required to have equal length.

The advantage of having such proofs formalised in Isabelle, say, is that all assumptions are clearly spelled out—the restriction about the equal lengths is somewhat opaque in the paper by Sulzmann and Lu. It seems Sulzmann and Lu are actually not really aware of them, judging from their reply shown in the Introduction. To sum up, the weakening of the properties by requiring that values need to have equal length seems to make the properties to hold, but destroys all inductive properties in the sequence case.

7 Conclusion and Future Work

At the moment I am working on fixing the correctness proof of Sulzmann and Lu. This should yield a publication at a venue like ITP, where such work about formalisations is valued (especially since it fixes a problem that has been overlooked by "pencil-and-paper" reasoning). I have now the necessary theorem proving skills to do such a proof in the theorem prover Isabelle. I was already able to verify a number of results from the Sulzmann and Lu paper, like all properties about $mkeps$. What is missing are the proofs about the existence of unique maximal elements and proofs about inj. While attempting to prove the former, I found that the regular expression r in the POSIX-ordering $\succcurlyeq^r_{PX}$ is redundant in the definition by Sulzmann and Lu. That is, I could show that

$$v_1 \succcurlyeq^r_{PX} v_2 \qquad \text{holds if and only if} \qquad v_1 \succcurlyeq_{PX} v_2$$

After completing the proof, the plan is to make the algorithm as fast as possible. This includes the simplification steps which we omitted in this report. I do not foresee any problems with this in terms of proving the correctness of the improved version of the POSIX matching algorithm. However it is clear that this will not produce an implementation for a lexer that is on par with automata based lexers. These lexers might be fast, but of course they are not formally proven to be correct and it seems difficult to do so (since it is hard to reason formally about automata). We therefore want to achieve speed via a different route: there is already very promising work on a verified compiler for the SML language. This work is done in HOL, not in the Isabelle theorem prover. The algorithm by Sulzmann and Lu only requires a very simple functional language, much simpler than SML. We therefore aim at implementing a very small functional programming language inside the Isabelle theorem prover and then

implement a verified compiler for it. Since our language will be very small, it should be feasible to achieve this given the time constraints. This will build on my work I did during my MSc where I already implemented a compiler for a small language—of course at the time I did not verify the correctness of the compiler. Once this is done, I expect another publication containing the verified compiler and the case study about the POSIX matching algorithm.

Of course such a result relies on me being able to fix the proof of Sulzmann and Lu. If I am unable to fix the proof, then there are a number of options. One is to change the algorithm to a version that is easier to verify. For example there is a slight variant of Brzozowski's notion of regular expression derivatives, called *partial derivatives* [2]. Perhaps these partial derivatives help with establishing the correctness of a POSIX matcher. However this is a bit speculative, since partial derivatives have never been used in the context of lexing. Another option is to just focus on the compilation part. This part is less speculative, as there is some previous work. The novelt of our work is to start from a very small functional programming language and generate for example JVM code (for which there exists the necessary formal model in Isabelle). If this all fails, ther might be a possibility to continue work I like to do as intern at FireEye. They use the Coq theorem prover in order to verify parts of an operating system. I am not yet hired by them, but have already advanced to the second stage of their selection process and have been invited to come to Dresden, Germany, in order to talk to their lead developers.

To sum up, the main goal for my PhD is a fix for the correctness proof given by Sulzmann and Lu, and then generating machine code for the algorithm that is provably correct with respect to the fixed, high-level, correctness proof.

References

[1] J. B. Almeida, N. Moriera, D. Pereira, and S. M. de Sousa. Partial Derivative Automata Formalized in Coq. In *Proc. of the 15th International Conference on Implementation and Application of Automata (CIAA)*, volume 6482 of *LNCS*, pages 59–68, 2010.

[2] V. Antimirov. Partial Derivatives of Regular Expressions and Finite Automata Constructions. *Theoretical Computer Science*, 155:291–319, 1995.

[3] A. Asperti. A Compact Proof of Decidability for Regular Expression Equivalence. In *Proc. of the 3rd International Conference on Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 283–298, 2012.

[4] A. Barthwal and M. Norrish. A Mechanisation of Some Context-Free Language Theory in HOL4. *Journal of Computer and System Sciences*, 80(2):346–362, 2014.

[5] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[6] H. Chen and S. Yu. Derivatives of Regular Expressions and an Application. In *Proc. in the International Workshop on Theoretical Computer Science (WTCS)*, volume 7160 of *LNCS*, pages 343–356, 2012.

[7] T. Coquand and V. Siles. A Decision Procedure for Regular Expression Equivalence in Type Theory. In *Proc. of the 1st Conference on Certified Programs and Proofs (CPP)*, volume 7086 of *LNCS*, pages 119–134, 2011.

[8] A. Frisch and L. Cardelli. Greedy Regular Expression Matching. In *Proc. of the 31st International Conference on Automata, Languages and Programming (ICALP)*, volume 3142 of *LNCS*, pages 618–629, 2004.

[9] A. Krauss and T. Nipkow. Proof Pearl: Regular Expression Equivalence and Relation Algebra. *Journal of Automated Reasoning*, 49:95–106, 2012.

[10] C. Kuklewicz. Regex Posix. `https://wiki.haskell.org/Regex_Posix`.

[11] X. Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[12] T. Nipkow. Verified Lexical Analysis. In *Proc. of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1479 of *LNCS*, pages 1–15, 1998.

[13] S. Owens, J. H. Reppy, and A. Turon. Regular-Expression Derivatives Re-Examined. *Journal of Functinal Programming*, 19(2):173–190, 2009.

[14] S. Owens and K. Slind. Adapting Functional Programs to Higher Order Logic. *Higher-Order and Symbolic Computation*, 21(4):377–409, 2008.

[15] L. C. Paulson. A Formalisation of Finite Automata Using Hereditarily Finite Sets. In *Proc. of the 25th International Conference on Automated Deduction (CADE)*, volume 9195 of *LNAI*, pages 231–245, 2015.

[16] M. Sulzmann and K. Lu. POSIX Regular Expression Parsing with Derivatives. In *Proc. of the 12th International Conference on Functional and Logic Programming (FLOPS)*, volume 8475 of *LNCS*, pages 203–220, 2014.

[17] M. Sulzmann and P. Thiemann. Derivatives for Regular Shuffle Expressions. In *Proc. of the 9th International Conference on Language and Automata Theory and Applications (LATA)*, volume 8977 of *LNCS*, pages 275–286, 2015.

[18] K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[19] D. Traytel. A Coalgebraic Decision Procedure for WS1S. In *Proc. of the 24th Annual Conference on Computer Science Logic (CSL)*, volume 41 of *LIPIcs*, pages 487–503, 2015.

[20] D. Traytel and T. Nipkow. A Verified Decision Procedure for MSO on Words Based on Derivatives of Regular Expressions. In *Proc. of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 3–12, 2013.

[21] C. Wu, X. Zhang, and C. Urban. A Formalisation of the Myhill-Nerode Theorem based on Regular Expressions. *Journal of Automatic Reasoning*, 52(4):451–480, 2014.

[22] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, 2011.