Semilinear Regular Expressions in Agda

Joey Eremondi
Utrecht University Capita Selecta
Supervisor: Wouter Swierstra
UU#: 4229924

July 22, 2015

1 Introduction

1.1 Semi-Linear Sets

Let $\mathbb{N}$ be the set of natural numbers including 0. We deal with vectors of natural numbers, with addition and scalar multiplication defined normally.

A linear set $L \subseteq \mathbb{N}^k$ is a set that can be written as $L = \{v_0 + c_1 \cdot v_1 + \ldots c_n v_n \mid n \geq 0, c_i \in \mathbb{N}\}$. A semi-linear set is a finite union of linear sets.

Semi-linear sets are closed under several operations, including finite union, addition, and infinite unions of sums.

1.2 Regular Languages

Given an finite alphabet Σ, a regular expression over Σ is defined as:

- The empty set, $\emptyset$
- The empty word, ϵ
- A symbol $c \in \Sigma$
- A union of two REs, $r_1 + r_2$
- The concatentation of two REs, $r_1 \cdot r_2$
- The Kleene-closure of two REs, r^*

The languages matched by a regular expression is defined intuitively for all cases, except the star case, where $L(r^*) = \bigcup_{i=1}^{\infty} r^k$.

1.3 Parikh Images

Suppose our alphabet $\Sigma = \{a_1, \ldots, a_k\}$.

Given $w = w_1 \cdots w_n$ with $w_j \in \Sigma$, the letter count of a_i in w is defined as $|w|_{a_i} = |\{w_j \mid w_j = a_i\}|$.

The Parikh vector of a word w is the vector $\Phi(w) \in \mathbb{N}^k$ such that the ith component of $\Phi(w)$ is equal to $|w|_{a_i}$. Essentially, it is the vector storing how many times each letter occurs in a word.

The Parikh image of a language $L \subseteq \Sigma^*$ is defined as $\Phi(L) = \{\Phi(w) \mid w \in L\}$.

1.4 Parikh's Theorem

Parikh's Theorem, proved by Rohit Parikh in 1966 [**?**], states the Parikh-image of every context-free langauge is semi-linear. Because regular lagnauges are a strict subset of the context-free languages, this property holds for all regular languages.

2 The Project

For this Capita Selecta, I implemented a constructive proof that the Parikh image of any regular language is semi-linear. The proof was implemented in Agda, a dependently-typed programming language which allows for formal verification of constructive proofs.

Some aspects of the proof, regarding the Star operator, were not completed, due to time constraints. They are left as Agda "holes". However, helpful lemmas towards filling these gaps have been provided, showing that it is feasible to complete eventually.

2.1 Applications

The theorems proved here can be used to show non-emptiness of regular expressions, to disprove equivalence of regular expressions, or to aid in analysis of the language accepted by a regular expression. However, there are existing, simpler algorithms for this.

The main application of these proofs is as a first step in building a verified proof of Parikh's Theorem for context-free languages. For example, there is no algorithm for proving that two context-free languages are unequal, but comparing their Parikh images can be performed algorithmically. This, in turn, lays a foundation for combination of formal-language theory and dependently-typed programming.

Finally, this Capita Selecta served as a tool for me to learn dependently-typed programming. The problem was simple enough to fit in a two-month course, but large enough that I was forced to explore Agda's features and libraries, and gain experience with the practical issues surrounding dependently typed programming.

2.2 Approach

The recursive structure of the definitions of regular expression makes it easy to perform proof by induction. However, the definition of star is poorly suited to Agda's recursion: since $r^* = \epsilon \cup (rr^*)$, normal recursion will not terminate. Thus, we annotate every regular expression with a field indicating whether it is nullable or not: that is, whether the empty string is matched by the expression. An expression of the form (r^*) is only allowed if r is not nullable. This means that every non-empty word that matches r^* can be broken into two strictly smaller parts which match r and r^* respectively, allowing for termination.

The main proof consists of 3 parts:

- A function mapping regular expressions to their parikh images
- A function which, given a regular expression, the (above defined) Parikh image for that RE, a word, and a proof that that word matches the RE, generates a proof that the Parikh vector of that word is in the Parikh image of the RE
- A function which takes a regular expression, the (above defined) Parikh image of that RE, a vector v, and a proof that that v is in the Parikh image, and generates a proof that there is some word w with $\Phi(w) = v$ and a proof that the RE matches w.

These together prove that the generated set is precisely the Parikh image of the regular language.

Many lemmas are proved, providing basic properties of linear and semi-linear sets, such as commutativity and associativity of vector addition, the proving the zero vector's role as an identity, proving that $v \in S \implies v \in S \cup S'$, and so forth. These lemmas are then used to take the results of applying an inductive hypothesis to our correctness and completeness proofs, and transforming it into a form that can be used to prove a case correct.

The main design decisions for the proof lie in the representation of different data. Regular expressions were defined using an custom data type, and words are assumed to be lists of Unicode characters (Agda's Char type).

Linear sets are represented by a triple: a "base" vector v_0, a number of vectors m, and a collection of m vectors which are combined. Witnesses that an vector v is in a linear set take the form of linear combinations: a set of coefficients $c_1, \ldots, c_m$, and a proof that $v_0 + c_1 \cdot v_1 + \cdots + c_n \cdot v_n = v$. We do not allow for empty linear sets: this may not align with the theory, but it is convenient for our representation.

A semi-linear set is simply represented as a list of linear sets. A witness of membership in a semi-linear set is either a linear combination of the vector and the head linear set, or a witness that the vector is a member of one of the tail linear sets.

Vectors of natural numbers were represented using Agda's Vec type, which was convenient for allowing type-level proofs that vectors were the same size. For calculating Parikh mappings, our proofs are all parameterized over mappings from characters to a finite-set of natural numbers, allowing us to use the Char type to represent any sized finite alphabet.

Equality proofs were completed using Agda's built-in homogeneous equality relation. However, for linear and semi-linear sets, we avoid any notion of equality, instead showing inclusion by mapping witness of membership in one set to witnesses of membership in another set. This allowed us to avoid issues arising from different representations of equal sets.

3 Code Structure and Notes

Here we highlight the key parts of each module. Full documentation on lemmas and helper functions can be found in the source code itself.

3.1 RETypes.agda

Types and some basic theorems surrounding regular expressions.

- `Null?`: basic enumeration for whether an RE can match the empty string or not
- `NullTop, NullBottom`: proof terms for combining `Null?` values. Top is nullable iff its inputs are both nullable, and bottom is nullable iff either of its inputs are nullable.
- `RE`: datatype representing regular expressions.
- `REMatch`: datatype representing a witness that a given string matches a regular expression. This forms the definition of matching.

3.2 SemiLin.agda

Definitions of types for vectors, linear, and semi-linear sets, as well as several useful lemmas surrounding them.

- `Parikh`: synonym for a vector of natural numbers
- $\cdot_s$: scalar multiplication
- `+v`: vector addition, element-wise
- `v0`, the vector containing all 0 elements, parameterized by length
- `LinSet`: representation of a linear set, combining a base vector, a number m, and m vectors to combine.
- `applyLinComb`: given a list of vectors, a base vector and a vector of coefficients, multiply each vector by its coefficient, then sum them and add the base.
- `LinComb:` witness that a vector is in a linear set. Pairs the coefficients for each vector combined with a proof that summing the linear set's vectors, weighted by the coefficients, gives the desired vector.
- `+l`: constructs the set $\{x + vy \mid x \in L_1, y \in L_2\}$. Adds the bases and concatenates the vector lists.
- `SemiLinSet`: alias for a list of linear sets, representing their union.
- `+s`: like `+l` but for semi-linear sets. Concatenates l+ for each pair of linear sets from the given semi-linear sets.

- `inSemiLin`: witness that a vector is in a semi-linear set. Just follows the list structure to show that the vector is in one of the linear sets of the union
- `linStar`: the vector equivalent of star. Represents the set $L^* = \left\{\Sigma_{i=1}^{k} v_i \mid k/geq0, v_i \in L\right\}$.
- `linStarExtend`: given $u \in L$ and $v \in L^*$, show thatt $u + vv$ is in L^*. This lemma would have been used proving the star cases correct in SemiLinRE.agda, but could not be used due to time constraints.
- `linStarDecomp`: show that every nonzero vector in L^* is the sum of a nonzero vector in L and some other vector in L^*. Similar to the extend case, this would have been used in the completeness proof of SemiLinRE.agda.

3.3 SemiLinRE.agda

Functions and theorems for Parikh images of regular languages.

- `wordParikh`: the parikh vector of a word, given a mapping from Char to a finite set.
- `reSemiLin`: the function mapping a regular expression to its Parikh image, a semi-linear set.
- `reParikhCorrect`: Proof that the Parikh vector of every word matched by an RE is in its Parikh image from `reSemiLin`
- `reParikhComplete`: Proof every vector in the set given by `reSemiLin` is the Parikh vector of some word matched by the given RE.

4 Challenges

4.1 Star and termination

As it is traditionally presented, the star operator provides some problems with termination. A word matching r^* is either empty, or matches $r \cdot r^*$. However, problems arise when r can be "nullable:" that is, when it matches the empty string. If to match w to $r*$ and r is nullable, we could match ϵ to r and w to r^*, but this would never terminate.

To work around this, we pair our regular expressions with proof objects, describing whether or not they accept the empty string. This ensures that termination will occur. However, proving this to the Agda termination-checker is often difficult. Many necessary lemmas, such as the relationship between nullable regular expressions and their Parikh vectors, were not proved due to time constraints. These are responsible for many of the holes left in the proof.

4.2 List and number representation

Because we represented semi-linear sets using lists, and vectors as Agda Vectors (lists parameterized by their size at type level) of Natural numbers, many of the proofs relied on the underlying properties of these data structures.

While Agda has a wealth of existing proofs of properties in its standard library, often these are embedded in more complicated structures, such as a SemiRing or Monoid instance.

Using these tools would have shortened many of the proofs, and would be advisable for a long-term project, but the extra complexity made it simpler to prove properties by hand for this project.

Because of this, many of the proofs consist of long equality-chains, manipulating lists and vectors using commutativity and associativity.

4.3 Implicits

As a beginner Agda programmer, one of the more challenging design problems I faced was knowing when to make parameters to functions implicit or not. Leaving parameters explicit resulted in verbose, hard to read proofs. However, when implicit parameters were used, often errors about unresolved metas would arise.

For the most part, I erred on the side of leaving variables explicit, with the main exception being the size of the alphabet used in regular expressions (which also was the length of Parikh vectors). Because we abstracted over this value, and never manipulate it within proofs, it could easily be left implicit.

If this project were to be continued, cleaning up the use of implicits to be consistent across proofs would be advisable..

5 Future work

The obvious continuation of this work is to fully prove Parikh's theorem for semi-linear languages. While the proof itself is more complicated, the properties of semi-linear languages performed for this proof will be incredibly valuable, providing much of the ground-work for a full proof.

As a "warm-up" exercise, a verified regular-expression continuation-passing-style matching algorithm was implemented. This was not fully completed, so a complete proof would complement this project nicely.

It is my hope that I can use this project as a starting point for a general library of formal-language-theoretic proofs in Agda. Formal language theory allows for many limited models of computation, which are weaker than Turing machines, but have more decidable properties. The various algorithms of formal-language theory could eventually be used to provde automatic proving procedures for some cases, reducing the amount of programmer work needed to create proofs in Agda.

References

[1] Rohit J. Parikh. 1966. On Context-Free Languages. *J. ACM* 13, 4 (October 1966), 570-581. http://doi.acm.org/10.1145/321356.321364