Game, Setty, Match: Re-Implementing Setty - A Programming Language to Teach Discrete Mathematics - In Rosette and Racket

Vimala Jampala
Department of Computer Science and Engineering
University of Washington
29 April, 2015

***Abstract*—One of the first upper division classes students in the Computer Science & Engineering major take is a class on computing foundations that covers discrete math topics and their relation to computing. Many students taking the class have trouble connecting the concepts taught to the familiar world of programming. To that end, this paper presents a new implementation of a language called Setty, which was created to bridge the gap between programming and computer science. While Setty was initially implemented in Python, this paper presents a re-implementation using Rosette (a solver aided programming language) and Racket (a functional programming language). In addition to providing many of the features available in the previous implementation, this implementation contains new features that will be discussed in detail.**

I. Introduction

Foundations of Computing I (CSE 311) is one of the first classes incoming Computer Science & Engineering majors at the University of Washington take. It covers discrete mathematics topics and their relation to computer science and computer engineering. At that point, these students' only experience of Computer Science & Engineering is programming. Some of them may not even have taken a math class at the university level before. As a result, CSE 311 can be confusing to students taking the class. The class covers a broad range of topics, from propositional logic to undecidability, some of which are more closely connected to programming than others. While the course does connect these topics back to computer science, the abstract nature of these topics and the shift from programming assignments to written homework can leave students confused.

A. *Bridging the gap between mathematics and programming*

A programming language called Setty was created so that students can experiment with mathematics [1]. Its syntax and semantics resemble those used in discrete mathematics. Setty provides sets as primitive types, and supports operations such as set union and set intersection. It also supports functions, quantifier expressions, and other mathematical expressions.

B. *Re-implementing Setty*

The original implementation of Setty was used as part of the course "15-151: Mathematical Foundations of Computer Science" at Carnegie Mellon University. One of the results of using Setty in this course was that students had a hard time adjusting to a UNIX environment. This, along with the slow development cycle of compiling Setty code prevented them from receiving the full benefits of Setty.

Additionally, the original implementation of Setty had only one primitive type- sets. The main goal of this was to show students that a language could be built out of mathematical primitives. However, this concept ended up being difficult for the students, to the point where it became the main thing they took away from the language.

As a result, it was decided that a new version of Setty would be created. This version would be interpreted, instead of compiled, to allow students to iterate faster. Additionally, this version would support primitive types other than sets. Lastly, this version would use Rosette to support infinite sets and queries on predicates.

C. *Similarities between the implementations*

There are several similarities between the original and new implementations of Setty. Both support mathematical operations (add, subtract, multiply, divide) as well as Boolean operations (conjunction, disjunction, negation). Additionally, both support set operations such as union, intersection and difference. Both implementations also have functions, conditionals, loops, set comprehensions and predicates. Lastly, neither implementation supports mutation of expressions.

D. *Differences between the implementations*

In the original implementation of Setty, all values, including natural numbers and Booleans, represented sets. The new implementation of Setty supports four types - integers, Booleans, sets and tuples. Additionally, the new implementation of Setty introduces infinite sets, and allows users to perform operations on them. Lastly, the new implementation also introduces queries, which allow students to explore sets and truth tables using Rosette.

E. *Using Setty*

This implementation of Setty is interpreted, not compiled. We read input from the console, and print the results of evaluating that input. Users can access the interpreter either through the Racket REPL or the command line. For example, a user can issue commands as follows:

```
>> 3
3
```

F. Organization

Section 2 of this paper provides an overview of the type system in the new implementation of Setty. Sections 3 through 6 describe features present in the old implementation of Setty that have been re-implemented in the new implementation. Sections 7 and 8 present new additions to Setty - queries and operations on infinite sets. They are followed by a discussion of future work.

II. Types in Setty

A. Types in the original implementation

In the original implementation of Setty, all values, including natural numbers and Booleans, represented sets. When used as operands to boolean operations, the empty set represented false, with all other sets representing true. Natural numbers were shorthand for the von Neumann ordinals, which are defined as follows:
$0 := \emptyset$
$n := (n-1) \cup \{n-1\}$

In other words $n = \{0, 1, ..., n-1\}$. The numbers and their corresponding sets could be used interchangeably.

B. Types in the new implementation

Setty supports four primitive types- Booleans, integers, sets and tuples. Booleans in Setty have two primitive values - T (for true) and F (for false).

In Setty, sets are an unordered collection of values of the same type. For example, $\{1, -7\}$ and $\{T, F\}$ are valid sets, while $\{1, F\}$ is not. On the other hand, tuples are ordered collections of values whose elements may be of the same or of different types. $(1, 2, 3)$ is a valid tuple, as is $(1, F, 7)$. Accessing an element can be done by using the array access notation supported by many languages. For example, to get the second element of the tuple $(T, \{3\}, 1)$, a user would write $(T, \{3\}, 1)[1]$. Tuple indices are zero-based. Tuples are the only way to create an ordered collection of elements, as well as a collection of elements of multiple types. They can also be used by students to learn more about ordered pairs and cartesian products.

Setty also supports infinite sets. Some infinite sets are built-in. For example, Setty supports N (the set of natural numbers) and Z (the set of integers). Other infinite sets can be constructed using set comprehensions. For example, the set comprehension $\{x \mid x \in Z \wedge x < 5\}$ constructs an infinite set of integers less than five. Infinite sets can be a difficult topic for many students to grasp, due to their abstract nature. Supporting infinite sets in Setty allows students to learn more about these sets by performing operations on them.

Setty throws a type error when a user enters input that doesn't typecheck. The following inputs all result in type errors:
$>>\ \{1, T\}$
$Type\ error:\ Set\ contains\ elements\ of\ different\ types$
$>>\ 3 + \{1\}$
$Type\ error:\ Addition\ applied\ to\ non-number$

Currently, the type errors do not provide a lot of details. We plan to improve this in the next iteration of Setty. Since Setty is intended for use by students, useful and descriptive error messages are an important feature to have.

III. Primitive Operations

Setty supports basic arithmetic operations such as addition, subtraction, multiplication and division. Additionally, boolean algebra is supported as well, with conjunction, disjunction, negation, xor, implication, and bi-implication built in. Setty also supports operations to compare integers, as well as an equals operator that can be used to compare expressions of the same type. Lastly, Setty supports basic set operations such as union, intersection and difference, as well as the set containment and subset operations.

Below are some examples of primitive operations Setty supports, as well as their results:
$>>\ 2 + 4$
6
$>>\ not\ (T\ implies\ (\{1\}\ subset\ \{2, 3, 4\}))$
T
$>>\ \{1, 2, 3\}\ intersect\ \{3\}$
$\{3\}$

Users have two options when specifying boolean and set operations. The first is to use the keyword for each operand, as demonstrated above. The second is to use the actual symbol for each command. The following examples demonstrate this:
$>>\ \neg\ (T\ \oplus\ F)$
F
$>>\ \{1, 2, 3\}\ \subseteq\ \{1, 2, 3, 4, 5\}$
T
$>>\ \{1, 2, 3\}\ \cap\ (\{1\}\ \cup\ \{2\})$
$\{1, 2\}$

IV. Variables and Functions

A. Variables

Variables are declared using the syntax $v := exp$. Here, v corresponds to the variable name, while exp is an expression. In Setty, expressions are immutable. For example, declaring a variable in Setty looks like the following:
$>>\ x := T$

Declaring another variable called x will result in the previous variable being shadowed.
$>>\ x$
T
$>>\ x := (1, 3)$
$>>\ x$
$(1, 3)$

B. Functions

All functions in Setty take a single argument. Setty supports two types of function definitions - standard multi-line functions as seen in most programming languages, and one line functions similar to mathematical functions.

1) Multi-line functions: Multi-line functions in Setty resemble functions seen in programming languages. They have the following syntax:

⟨*block*⟩ ::= `{` `\n` ⟨*statement-list*⟩ `}`

$\langle$*func-def-block*$\rangle$ ::= $\langle$*function-name*$\rangle$
`(`$\langle$*argument-name*$\rangle$`)` `:=` $\langle$*block*$\rangle$

For example, the following function prints out the elements of a set:

```
print-set (set) := {
  for x in set:
    print x;
  return 0;
}
```

In Setty, all multi-line functions must return an expression. Any multi-line function that has a path that does not end in a return statement is invalid.

2) Single-line functions: Single line functions in Setty take the following form:

$\langle$*func-def-inline*$\rangle$ ::= $\langle$*function-name*$\rangle$
`(`$\langle$*argument-name*$\rangle$`)` `:=` $\langle$*expr*$\rangle$ `\n`

They are intended to be used for simple function definitions such as $f(x) = x^2$.

3) Piecewise functions: Setty allows users to define piecewise functions. For example, a function to compute the factorial of a number can be defined as follows:
$f(0) := 1$
$f(1) := 1$
$f(n) := n * f(n-1)$

Each function may be either a one-line or multi-line function. These functions do not have to accept every possible argument. If the function is called with an argument that it has not been defined for, Setty will throw a run-time error. For example, given the following function definitions, the function call $negate(1)$ would result in a run-time error.
$negate(T) := F$
$negate(F) := T$

In our implementation, when a function is defined, a mapping from the function name to a closure representing the function is added to the environment. When a user creates a new function with the same name as an existing function, this mapping is amended. Instead of pointing to a closure, it is updated to point to a list of pairs, with one pair per function. The first value in the pair is the parameter taken by the function, and the second value in the pair is the function's closure. The pairs are stored in chronological order.

For example, after a user enters $f(0) := 1$, the environment looks as follows:
f $\Rightarrow$ closure($f(0) := 1$, env).
After this, when the user enters $f(1) := 1$, the environment is updated to look like:
f $\Rightarrow$ ((0, closure($f(0) := 1$, env)), (1, closure($f(0) := 1$, env)))

When a user calls the function f, we check every pair in the list mapped to the function f. If the parameter stored in the pair is compatible with the argument passed in by the user, the corresponding function definition is used to evaluate the function call. Otherwise, we proceed down the list, until we either find a compatible parameter or reach the end of the list. If we reach the end of the list, a run-time error is thrown, as mentioned above.

If a parameter is a variable, then any argument is considered a match for it. Otherwise, a parameter and argument are considered to match only when they are equal.

V. Conditionals and Loops

A. Loops

There are two types of for loops in Setty. The first type of loop iterates over a set. It has the following syntax:

$\langle$*loop-stmt*$\rangle$ ::= `for` **name** `in` $\langle$*expr*$\rangle$ `:` $\langle$*block*$\rangle$

where expr denotes a set. Since sets are unordered, Setty guarantees that each element will be iterated over, but does not guarantee that each element will be iterated over in the same order each time.

The following example demonstrates the output of a for loop over the set {1, 2, 3}:

```
>> for x in {1, 2, 3}: {
...  print x;
...}
3
1
2
```

The second type of loop starts at zero and iterates until the upper bound. It has the following syntax:

$\langle$*loop-stmt*$\rangle$ ::= `for` **name** `to` $\langle$*expr*$\rangle$ `:` $\langle$*block*$\rangle$

where expr denotes a numerical expression. For example, if the numerical expression is n, the for loop will have exactly n iterations.

Setty does not support continue or break statements.

B. Conditionals

Conditionals in Setty have the following syntax:

$\langle$*conditional-stmt*$\rangle$ ::= `if` $\langle$*expr*$\rangle$ `:` $\langle$*block*$\rangle$

The following example demonstrates what an if statement in Setty code looks like:

```
if x in {1, 3, 4}: {
  f(x)
}
```

Currently, Setty does not support if/else statements. We hope to add them to the next iteration of Setty.

VI. Set Comprehensions and Quantifier Expressions

A. Set comprehensions

Many languages support list comprehensions as a way to create a list using existing lists. Setty provides a similar construct called "set comprehensions", which allow users to specify sets by performing operations on other sets. For example, $\{x \mid x < 5\}$ specifies the set of integers x where x is less than five. Similarly, $\{x \in \{1, 2, 3\} \mid x + 1\}$ would evaluate to $\{2, 3, 4\}$.

Set comprehensions have the following syntax:

⟨*set-comprehension*⟩ ::= `{`**name** `in` ⟨*set-expression*⟩ `|` ⟨*expr*⟩ `}`
  | `{` ⟨*expr*⟩ `|` ⟨*expr*⟩ `}`

For example, the following are valid uses of set comprehensions in Setty:
$>>\ \{x \in \{1, 2, 3\} \mid x + 1\}$
$\{2, 3, 4\}$
$>>\ \{x \mid x < 3\ \wedge\ x > 5\}$
$\{\}$
$>>\ \{x \mid x < 3\ \wedge\ x\ \in\ \{0, 1, 2, 3\}\}$
$\{0, 2, 3\}$

B. Quantifier Expressions

Setty supports first-order logic expressions such as $\forall(x \in \{1, 2, 3\}).F(x)$, and treats them as expressions that evaluate to a boolean. They have the following syntax:

⟨*quantifier*⟩ ::= ∀ | `forall` | ∃ | `exists`

⟨*quantified-expression*⟩ ::= ⟨*quantifier*⟩ `(`**name in** ⟨*set-expression*⟩ `)` `.` ⟨*expr*⟩

Setty supports expressions with multiple quantifiers as well. For example, $\forall(x \in \{1, 2, 3\}).\ \exists(y \in \{8, 9, 10\}).\ y > x$ is a valid quantifier expression.

VII. Queries

One feature we added to the new implementation of Setty was the ability to investigate sets and truth tables using queries. We implemented this feature using Rosette [2], a solver aided programming language built using Racket. Rosette relies on satisfiability solvers to provide program synthesis, verification and debugging.

A. About Rosette

Rosette adds several solver-aided tools to Racket. These tools provide three building blocks - symbolic values, assertions, and queries - that allow programmers to ask a constraint solver questions about program behaviors.

The first building block, symbolic values, refer to variables that are a placeholder for a concrete constant of the same type. Once a solver is called, it will assign a value to all symbolic constants in a query depending on the query asked. For example, to create two different symbolic numbers, a programmer could do the following:
$>>\ (define - symbolic * a\ number?)$
$>>\ a$
$a\$0$
$>>\ (define - symbolic * b\ number?)$
$>>\ b$
$b\$0$

The second building block, assertions, allows users to express important properties or assertions with a truth value, as follows:
$>>\ (assert\ (equal?\ 1\ 2))$
$assert : failed$
$>>\ (assert\ (equal?\ 1\ 1))$
$(no\ output)$

If one of these assertions contains a symbolic value, it does not pass or fail immediately. Instead, whether or not it passes or fails is determined when it is passed to a solver.
$>>\ (assert\ (equal?\ a\ b))$
$(no\ output)$
$>>\ (assert\ (equal?\ a\ a))$
$(no\ output)$

The third building block, queries, allows users to query solvers to receive answers. The only query used in this implementation of Setty was the verify query. The verify(assertion) query queries the solver for a concrete value for each symbolic constant in the assertion, such that the assertion fails when these symbolic constants are replaced with the concrete values. In other words, verify asks the solver if the assertion can be made false, and not if the assertion is always true. This mapping from symbolic constants to concrete values that make an assertion false is called a counterexample. If a counterexample cannot be found, Rosette throws an error. For example, the following code verifies the assertions in the previous example:
$>>\ (verify\ (assert\ (equal?\ a\ b)))$
$(model\ [a\$0\ \ -2]\ [b\$0\ 1])$
$>>\ (verify\ (assert\ (equal?\ a\ a)))$
$verify : no\ counterexample found$

We wrote a macro around verify that ensured that it returned true instead when a counterexample could not be found. This makes verify behave a little more intuitively to novice Rosette users- verify returns true if the assertion is true, and a counterexample if it can be made false.

B. The is-equal query

The is-equal query takes two Boolean formulae as arguments. It returns true if the two formulae are equivalent. Otherwise, it returns a counterexample. For example, the following query evaluates to true:
$>> is - equal\ \neg(x \wedge y), (\neg x \vee \neg y)$
T

However, this query returns a counterexample:
$>> is - equal\ \neg(x \wedge y), (\neg x \vee y)$
F
$Counterexample : x = T,\ y = T$

This was implemented by first translating both the Setty expressions into their Rosette equivalents. The translation was simple- $x \wedge y$ became $x\&\&y$, and so on. Once each expression was translated, we used Rosette to verify that they were equal.

C. The is-negation query

The is-negation query takes two Boolean formulae as arguments. It returns true if one argument is the negation of the other argument. Otherwise, it returns a counterexample. For example, the following query returns true:
$>> is-negation\ x, \neg x$
T

The following query has a counterexample:
$>> is-negation\ x, x$
F
$Counterexample : x = T$

This query is syntactic sugar for the is-equal query. is-negation $x1, x2$ is equivalent to is-equal $x1, \neg x2$.

D. The is-subset query

The is-subset query takes two sets as arguments. It returns true if the first set is a subset of the second set. Otherwise, it returns a counterexample. For example, the following query returns true:
$>> is-subset\ \{x \mid x < 3\}, \{x \mid x < 4\}$
T

The following query has a counterexample:
$>> is-subset\ \{x \mid x < 4\}, \{x \mid x < 3\}$
F
$Counterexample : x = 3$

This query was implemented similarly to the is-equal query. First, the arguments were translated into their Rosette equivalents. Then, Rosette's verify command was used to find counter-examples.

Translating the arguments of this query was a little more complicated than translating the arguments of the is-equal query. Since the arguments to this query were sets that could be of any type, the translate method had to support Boolean, integer, and set operations. Additionally, each argument could use different variables to represent the same underlying set. For example, one argument might be $\{x \mid x < 3\}$, and the other argument might be $\{y \mid y < 3\}$. Both of these expressions represent the same set, though it may not be immediately obvious. Therefore, before translating each argument, the variables in each argument were replaced with placeholder variables. The first variable was replaced by "v0", the second with "v1" and so on until all the variables were replaced. In the example outlined above, x and y would both be replaced by $v0$, and both arguments would be $\{v0 \mid v0 < 3\}$.

E. The assume-then-evaluate query

The assume-then-evaluate query takes two arguments. The first argument is an assumption, while the second is a boolean expression. It evaluates the second argument under the assumption that the first argument is true. It returns true if the second argument evaluates to true, and returns a counterexample otherwise. For example, the following query evaluates to true:
$>> assume\ (N \cap Z) = N\ evaluate\ (N \cap Z) \subseteq N$
T

On the other hand, this query has a counterexample:
$>> assume\ (N \cap Z) = \{-1\}\ evaluate\ (N \cap Z) \subseteq N$
F
$Counterexample : x = -1.$

The query does not throw an error even though the assumption passed in is an invalid assumption. For example, in this case, the assumption $N \cap Z = \{-1\}$ is false.

The implementation of this query was similar to the implementation of the is-subset query. Each argument was translated into its Rosette equivalent. Rosette was then used to find counterexamples to the query. Rosette's verify command allows an optional assumption to be specified. For this query, we used the first argument as the the optional assumption of the verify command, and called verify on the second argument.

F. The evaluate query

The evaluate query takes a boolean expression as its only argument and evaluates it. If the expression evaluates to true, it returns true. Otherwise, it returns a counterexample. For example, the following query returns true:
$>> evaluate\ N \subseteq Z$
T

However, this query has a counterexample:
$>> evaluate\ Z \subseteq N$
F
$Counterexample : x = -16$

The implementation of this query was similar to the implementation of the assume-then-evaluate query. The only difference is that the optional assumption parameter of the verify command was not used.

G. The get-matching-expression query

The get-matching-expression query takes a truth table as an argument. Here, a truth table is defined as a tuple containing tuples that satisfies the following conditions:

- Each tuple in the truth table is a sequence of k boolean values
- The first $k-1$ values in each tuple are the inputs to the expression that matches the truth table. The last value in each tuple is the output given that input.
- There are 2^{k-1} tuples in the truth table.

For example, this truth table represents the OR function: $((T, T, T), (T, F, T), (F, T, T), (F, F, F))$.

The get-matching-expression query returns a Boolean expression in disjunctive normal form that matches the truth table. This query does not use Rosette. Instead, it filters every row $(p_1, ..., p_n)$ in the truth table with truth value T. Then it creates a conjunction of literals C = $(v_1 \wedge ... \wedge v_n)$ such that v_i is p_i if p_i is true, and $\neg p_i$ otherwise. Lastly, it returns the disjunction $(C_1 \vee ... \vee C_k)$ of the above clauses.

For example, using the get-matching-expression query on the truth table for OR above would result in the following

output:
$>> get - matching - expression$
... $((T,T,T),(T,F,T),(F,T,T),(F,F,F))$
$(v0 \wedge v1) \vee (v0 \wedge \neg v1) \vee (\neg v0 \wedge v1)$

An alternative way to implement this would be to count the number of variables k in the expression represented by the truth table and then generate boolean expressions with k variables in disjunctive normal form. Then, we could just return the boolean expression which matches the truth table.

There are two reasons we preferred the first implementation over the second:

1) The first implementation mirrors the way students in CSE 311 are taught to convert truth tables into boolean expressions. This allows them to convert truth tables into boolean expressions in dnf form themselves and compare these expressions to Setty's output.
2) The second implementation requires that the truth table of each generated expression is compared to the original truth table, which is inefficient.

The downside of using the first implementation is that the resulting boolean expression may contain redundant clauses.

H. The is-cnf query

The is-cnf query takes a boolean expression as an argument. It returns true if the expression is in conjunctive normal form, and false otherwise. A boolean expression is in conjunctive normal form if it is an AND of one or more clauses, where each clause is an OR of literals or their negation.

is-cnf can be called as follows:
$>> is - cnf\ \neg A \wedge (B \vee C)$
T
$>> is - cnf\ (A \wedge B) \vee C$
F

We implemented this query by first checking that each clause only used ORs to combine literals. We then checked that the clauses were only combined using ANDs.

I. The is-dnf query

Like the is-cnf query, the is-dnf query takes a boolean expression as an argument. It returns true if the expression is in disjunctive normal form, and false otherwise. A boolean expression is in disjunctive normal form if it is an OR of one or more clauses, where each clause is an AND of literals or their negation.
$>> is - dnf\ (A \wedge B) \vee C$
T
$>> is - dnf\ \neg(A \vee B)$
F

We implemented this query by first checking that each clause only used ANDs to combine literals. We then checked that the clauses were only combined using ORs.

VIII. Operations on Infinite Sets

While Setty supports both finite and infinite sets, they are both evaluated differently. When this implementation of Setty evaluates a set, it first checks to see whether it is finite. If Setty is sure the set is finite, it evaluates the set and stores it as a Racket set. For example, Setty would eagerly evaluate the set $\{1,2,3\}$ and store it as a Racket set. However, if Setty is not sure the set is finite, it does not evaluate it immediately. Instead, it stores the abstract syntax node representing the set for later use.

Therefore, Setty immediately evaluates expressions that contain finite sets and returns the result. However, if an expression contains an infinite set, it simply returns "This set may be infinite." instead of evaluating the potentially infinite set. The set in question may not actually be infinite. For example, the operation $N \cup Z$ would evaluate to "This set may be infinite.", as would the operation $N - Z$. However, the former does result in an infinite set, while the latter does not.

Users that want the result of a set expression involving an infinite set must use the take-at-most-N query. The take-at-most-N query takes two arguments. The first is a set expression, while the second is a non-negative number n. The take-at-most-N query returns at most N elements from the set expression passed in. If the timeout (which has a default of ten seconds) runs out before it finds n elements, it returns the number of elements it has found, and informs the user that the operation timed out.

In order to process a take-at-most-N($expr$, n) query, Setty creates two threads. The first thread manually enumerates each element in $expr$ until either n elements are enumerated, no more elements are left to enumerate, or the timer runs out - whichever comes first. The second thread queries Rosette for n or fewer elements in $expr$. As soon as one of the threads returns, Setty kills the other thread and returns the solution to the user.

One interesting side-effect of this query is that in some cases, it could be used to see if a set expression evaluates to a finite set. For example, if take-at-most-N($expr$, 10) and take-at-most-N($expr$, 5000) return the same set without a timeout occurring, it is likely that $expr$ is a finite set.

IX. Future Work

A. Extending the language

Currently, Setty supports many of the features supported by the previous implementation of Setty. However, there are several features not yet implemented which would be useful, such as if/else statements. Additionally, more built-in functions (for example, a function that returns the power set of a set) would be useful as well.

Lastly, we hope to add more queries to Setty. In order for students to fully explore mathematical constructs, they must be able to examine them and ask questions about them. We hope that by increasing the number of queries offered by Setty, we will be increasing the number and variety of questions students pose about otherwise abstract concepts.

B. Building a web environment for Setty

Currently, users of Setty have the option of using the Racket REPL or a command line interface when using Setty. The former requires installing and setting up Racket, while the latter can be daunting for a novice programmer. Therefore, having a web environment would make using Setty easier for students. Both the introductory programming classes at the University of Washington make use of Practice-It!, an online web application to practice programming, resulting in students being comfortable with such environments. Consequently, compiling Setty to JavaScript would allow students to use Setty in a familiar environment.

X. Conclusion

This paper presents a new implementation of Setty, a programming language that allows allows students to program mathematical and logical concepts using sets. We first provided an overview of the type system in this new implementation, followed by features present in the old implementation that were re-implemented in this new implementation. Finally, it provided an overview of new additions to Setty - queries and operations on infinite sets. This implementation of Setty solves the two major problems of the previous implementation - sets being the only primitive type and compiling leading to a slow development cycle. It is our hope that in the future, this language makes learning discrete mathematics easier and more interesting for programmers.

The code behind Setty can be downloaded from https://gitlab.cs.washington.edu/setty/setty.

References

[1] A. Blank, *Technological and Pedagogical Innovations for Teaching Introductory Discrete Mathematics to Computer Science Students*. Carnegie Mellon University, 2014.

[2] E. Torlak and R. Bodik, *A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages*. University of California: Berkeley, 2014.