# Setty: Optimizing A Set Language

Adam Blank
Computer Science Department
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
adamblan@cs.cmu.edu

Andrey Kostov
Department of Electrical Engineering
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213
akostov@andrew.cmu.edu

***Abstract***

*Because of the overall low level of knowledge of discrete mathematics exhibited by incoming college freshman, we decided to design and implement a programming language called {Setɬy} that would help better convey basic mathematical concepts in an introductory level class. The language uses an LLVM back-end, but its intermediate representation is difficult to optimize. To offset this we decided to implement our own optimizations for our compiler that target optimizations that LLVM cannot already achieve. The optimizations were based on the way sets function. The optimizations involved variable life-cycle tracking and set re-use. Because the language has few side-effects, we also detect function purity. Our optimizations did not yield great results for complex test cases, but are never a tradeoff. We believe we have laid down a very solid foundation for future work on optimizing the language.*

## 1 Introduction

Set theory and logic are two of the most fundamental topics in mathematics. Unfortunately, we often treat these topics as completely theoretical with no way to experiment with the concepts to learn them. We engineered a novel set-based language called {Setɬy} which provides a computational environment for students new to mathematics to explore sets and logic. Using {Setɬy}, students can write code to evaluate expressions like

$$\{1, 2\} \cup \{2, 3\}$$

$$\forall (x \in [5]).x > 0$$

$$\{x | x \subseteq [2]\}$$

A language called SETL was previously developed by a team at NYU and was intended to be used in a similar way to {Setɬy}. One major difference between {Setɬy} and SETL is that SETL was designed as a feature-rich language where students could translate their ideas into SETL code, whereas {Setɬy}'s grammar is defined in an identical way to what students would learn in an introductory course. Another interesting difference is that all {Setɬy} programs halt whereas SETL programs might not.

Our novel contributions in this paper are: (1) the design and implementation of {Setɬy}, and (2) several set-based optimizations to make programs written with {Setɬy} use less memory and run faster.

## 2 The Setty Language

{Setɥy} has normal control structures: if statements, for loops, `for x in set` loops. It also has several special constructs for dealing with sets:

**Set Comprehensions.** {Setɥy} supports set comprehensions of the form `{expr1 | expr2 }` where it can bind all variables in `expr1` using `expr2`. For example,

```
{(x,y) | x in [5] and y in [6]
```

is a valid {Setɥy} expression. We discuss set comprehensions in much more detail later, as implementing generalizable set comprehensions was one of our major optimization opportunities.

**Quantified Boolean Statements.** {Setɥy} supports `forall` and `exists` statements bound to particular sets. For example,

```
forall (x in [10]). \exists (y in [5]). x*y > 10
```

is a valid {Setɥy} statement.

As one might guess from the name of the language, the only primitive type in {Setɥy} is a set. All sets in {Setɥy} can only contain other sets. It follows that every {Setɥy} set is finite, and together the sets that are computable in {Setɥy} are the hereditarily finite ones.

Lastly, {Setɥy} programs all halt by design. Limited recursion is allowed where {Setɥy} can verify that the function must halt.

The following {Setɥy} code sample contains example functions in {Setɥy} to calculate prime factors and test primality:

```
1  use "prelim.set"
2
3  factors(n) :=
4     factors := {}
5     for x in [n] minus {1}:
6        for y in [n] minus {1}:
7           if mult((x,y)) = n:
8              factors := factors union {x}
9     return factors
10
11 prime(n) :=
12    return |factors(n)|=0
13
14 print factors(5)
15 print factors(4)
16 print prime(5) not in 1
17 print prime(4) not in 1
```

## 3 Setty Implementation Details

{Setɥy} targets LLVM as a backend which provides us with many modern compiler optimizations for free. Since sets in {Setɥy} are common and constrained, we can optimize the output by taking advantage of the structure of the language. When {Setɥy} is translated to llvm, the resulting representation handles sets through a very small number of function calls that take sets as input and produce or modify sets as output. We can split this set interface into several categories:

**Set Creation.** Set creation functions always return fresh pointers.

- `_set_new` which creates an empty set.
- `_set_from_num(n)` which creates a set representing the number $n$ using the Von Neumann ordinal representation.

**Set Operations.** Set operation functions always return fresh pointers.

- `pblSetUnion(s1, s2)` which creates the set $s_1 \cup s_2$.
- `pblSetIntersection(s1, s2)` which creates the set $s_1 \cap s_2$.
- `pblSetDifference(s1, s2)` which creates the set $s_1 \setminus s_2$.
- `_set_unary_union(s)` which creates the set $\bigcup_{x \in s} x$.
- `_set_unary_intersection(s)` which creates the set $\bigcup_{x \in s} x$.

**Set Editors.** Set editors modify the pointer they are given. {Setɬy} has been designed so that set editors must be called in the same basic block as the initial pointer they modify in all circumstances. There is only one set editor: `_set_add(s, x)` adds $x$ to the set $s$.

Based on the current state of the language, we implemented several optimizations. The first one is universe inference, which is an advanced language feature. After that we implemented Set Tracking, which is the foundation for the rest of the optimizations, namely set re-use and function output memoization. The order in which optimizations should be applied is to first perform Set Tracking, followed by re-use and memoization. Ideally our own flavor of dead code elimination is performed routinely to clean up unnecessary instructions.

## 4 Generalized Set Comprehensions

Many languages (like `Python` and `Haskell`) provide "list comprehensions" which (in `Python` syntax) look like

```
[f(x) for x in L if cond]
```

A similar, but more general, notation is used to specify sets in mathematics. {Setɬy} attempts to emulate this generalized notation and allows users to enter sets like:

```
{(x,y) | x in [5] and x < y}

{x | subset(x, Y)}

{x | y in [2] and z in [3] and x = y*z}
```

To accomplish this, we implemented automatic free variable inference. Users can specify how their functions should be generalized in set comprehensions and {Setɬy} will automatically start understanding them. For instance, if we specify that `subset(x,Y)`, then we could infer the universe of $x$ as $\mathcal{P}(Y)$ (the powerset of $Y$). The {Setɬy} compiler attempts to infer free variables by using the built-in and user specified rules recursively. If there are multiple restrictions, {Setɬy} will take the most restrictive universe that it can. Additionally, the compiler uses a topological sort to ensure that the dependancies are actually well-defined.

We have implemented automatic free variable inference (which was the first subgoal of universe inference). The best way to describe what this goal actually means is via a few code examples.

In the following code example, the compiler can automatically infer the fact that the domain of $x$ is the set $N := [100]$, without this being explicitly specified.

```
1 N := [10]
2 print {x | y in N minus {0,1} and z in N minus {0,1} and x = y*z}
```

In this code sample, the compiler can infer the domain of $y$ and thus is it okay that $y$'s universe is not specified.

```
1 equivalence_classes(p) :=
2    S    := pi1(p)
3    dom  := pi1(pi2(p))
4    codom := pi2(pi2(p))
5    return { {y | (x,y) in S} | x in dom}
```

## 5 Definition of constant

Since all variables in {Setıy} are sets, which cannot be contained within a register, there are no primitive variables in the language that are constant in the typical sense of the word. Nevertheless, because of our understanding of the structure of the language we decided on a new definition of a constant set, which is one which contains only numbers (this includes sets that represent numbers). The reason for this decision is that it is easy to statically determine equivalence of such sets. Ideally this definition could be expanded to include more varied structure, but this is far more complex and not necessarily as common in the typical usage of the language.

## 6 Set Tracking

The most difficult part of the project was tracking sets. The usual flow of the llvm code generated by the compiler produces code that will create a set and then at some point stop mutating it and transition into using it for whatever purpose it would require it for. Initially, we believed that this pass would only be a very simple liveness analysis, but this turned out to be untrue for several reasons. First of all, the lifecycle of a set, even if we consider it a constant set, is a little more complex than that of a normal constant variable. To begin with, all sets are created by either $set\_new()$ or $set\_from\_num(number)$ and at that point all sets start out as constant sets. However, what really matters is whether a set is still constant once its definition is complete. Thus, not only the concept of liveness is necessary, but a concept of the portion of code where the set variable is fully defined as well. To perform such an analysis we decided that we would need to keep track of the birthplace, the final alteration and a list of all possible deaths of a function in the code. Thus a function is constant if it is constant after its final alteration and can be used freely in optimizations such as constant propagation and memoization.

When trying to compute the birthplace of a set, it is necessary to consider the origins of all the sets that were used in the creation of the set (including sets that are subsets of it). Finding the end of all alterations of a set was also not too difficult and thus finding

out which sets are constant was obtained. The real issue came when trying to find all the deaths of an instruction. In a normal liveness analysis this would be simple since the last use of a set is the end of its liveness range, but with sets, a set might become a subset of another set in which case its liveness range is extended to the liveness range of the set it is a subset of and this continues on. Another difficulty came from an additional goal of the pass, which was to identify a point of convergence that was dominated by all the deaths of a set. The usage of this point would be that that is where the sets memory can finally be freed. Another difficulty were phi nodes in the intermediate representation. The problem with them is that they can have multiple origins. When tracking back the origin of a set, we usually stop as soon as we hit a phi node, because the set can no longer be constant. This is one of the design decisions that we intend to rethink in the future, as it stops os from having full life-cycle information on all sets.

## 7 Dead Code Elimination

Standard dead code elimination is very difficuly in {Setɻy}, since most llvm IR instructions are either function calls, phi functions or memory instructions. However, our Set tracking pass provides us with enough information to perform an enhanced version of Dead code elimination. To perform this pass we need to iteratively remove all functions that have only one point at which their lifecycle ends and this point coincides with their final alteration. If this is the case, then we know that the variable is not used at a later time and since all functions with side effects cannot alter non-global sets, we know that the set must be dead code. Removing such sets, however, eliminates some of the uses of sets that were involved in the creation of the set we removed and can thus yield further optimization. By removing them iteratively, the only sets that remain will be ones that cause a side effect, namely either be used by or help construct a set that is used by a print function or be used in the alteration of a global variable.

## 8 Set Re-use

Set re-use is meant to mimic constant propagation in typical compilation models. It can only be performed after the Set tracking pass has generated the necessary information about sets to be able to determine the liveness and usage of all sets used in the program. The basic principle is that if a built-in set operation is performed on two sets that we can statically compute, then we can statically compute the output of this operation.

This pass is split into two phases. In the first phase we create a worklist of all sets in the function that we are iterating over and attempt to compute all constant sets. We can pre-compute all constant sets statically and can thus determine their equality during compilation. After we are done with the pre-computation of sets we are ready to colapse all constant set variables that are the same constant set. This part was surprisingly challenging, since when merging two variables you must determine which came earlier. As explained during set tracking, this can become complicated and might even involve pulling definitions of variables to a dominator of all equal sets. A potential future optimization will most likely involve converting all constant sets to global variables. We did not perform it at this time due to time limitations.

The final result of the pass as that for each function, all constant sets within the function are only represented by a single variable. This results in an often large imrpovement in both memory usage and execution time (the savings on time are due to less memory operations)

## 9 Function Purity and Memoization

Another interesting optimization is allowed by the fact that user defined functions only take one set as an argument and always return a set. This allows us to sometimes make the recomputation of a function unnecessary. This can be done in the case where the function was already computed on the same value and the function is pure.

A function is pure if given the same input it always produces the same result. Additionally we adhere to making functions impure if they have side effects. This means that if a function prints, then it is impure. Additionally if it uses a global variable in the computation of its result or modifies a global variable, it is also impure. All other functions are considered pure. Note that if we have function calls inside of other functions, we already know for all inner functions if they are pure since by design in {Set↿y} you cannot use functions you have not yet defined. Also, recursive functions are considered pure until proven otherwise. Their computation can never be memoised, however, since if their input does not depend on the output of a previous call, then there is no descent and the function does not compile.

When we determine a function to be pure, we attempt to detect if the function is ever called with the same input. If it is, then we defer the death of the output set until after all uses of the result of the latest call to the function on this same input and simply re-use the old output (since it will be the same). This optimization is simple, but can yield good results, especially if for example a pure function is called on the same input inside of a loop (this is something the llvm compiler would not be able to optimize because set functions involve memory modifications, which are considered side-effects).

The process of determining whether the input to two different calls of the same function is the same was the most difficult part of this optimization. If the inputs to both are constant sets, then the Set re-use pass will already have coalesced them into the same variable, thus making this case simple. The only difficulty then is when we detect that two calls to the same function take the same instruction as their argument to determine if they are both in the same state. To do this we made sure that the input was fully defined at both call sites to the function.

Unfortunately, due to time limitations, we were unable to fully finish function memoization, despite producing a very powerful purity checker.

## 10 Evaluation

We evaluated our passes using two metrics: (1) we ran each piece of code 100 times and timed the total time and (2) we used valgrind to get the number of bytes allocated during the course of the program execution.

The commandline invocation for the cells marked “no opts” was

```
opt -S -globalopt -mem2reg -O2 -mergereturn -dce
```

The commandline invocation for the cells marked "with opts" was

```
opt -S -load SetTracking.so -load SetDCE.so
       -load SetReuse.so -globalopt -mem2reg
       -mergereturn -postdomtree -targetlibinfo
       -O2 -set-dce -set-reuse -dce -set-dce
```

The tests we used were four pieces of real-world {Setɥy} code (the prime checker from earlier in the paper, a list library, a relations library, and a recursive function) and one contrived test case that had many dead sets ("test8").

You can see below that our optimizations generally decreased memory usage and marginally increased performance. We suspect that we could use similar passes to get much larger gains by increasing the scope of what we consider to be a constant.

| | primes.set | | list.set | | rel.set | |
|---|---|---|---|---|---|---|
| | No Opts | With Opts | No Opts | With Opts | No Opts | With Opts |
| Time | 0m0.487s | 0m0.488s | 0m2.736s | 0m2.639s | 0m2.714s | 0m2.649s |
| Space | 882320 | 880864 | 9306088 | 9305984 | 8046264 | 7960240 |

| | test8.set | | rec.set | | |
|---|---|---|---|---|---|
| | No Opts | With Opts | No Opts | With Opts | |
| Time | 0m0.129s | 0m0.128s | 0m0.187s | 0m0.187s | |
| Space | 2424 | 752 | 138600 | 138480 | |

## 11 Future work

One interesting idea for a future optimization is to expand the definition of constant sets to any sets that can be statically computed by the llvm compiler. The actual problem is statically detirmining if two sets are equal at some point in the program. If we could do this, then this would make our set re-use and our memoization techniques slightly more powerful. As of now we do not think that statically determining the equality of a larger set of sets than what we currently consider constant sets will be in any way easy.

Another future optimization is to improve function purity and memoization by dynamically remembering the print statement associated with a function that is pure without the function and considering it pure with it. This way we can extend the set of functions that are considered pure.

## 12 Conclusion

{Setɥy} is a toy language, created for the purpose of teaching fundamental mathematical skills to computer science freshman. It is unique in the fact that it has only one primitive variable, which is a set. While this complicates the initial compilation of code to the llvm intermediate representation, it allows for a few very neat optimizations based on the

simplicity the set of set manipulations available to the programmer. Due to a limited set of side effects and good design decisions in the design of the language, we have been able to make an effective set of optimizations that cannot be done without specific knowledge of the language and how it functions. Unfortunately, the extent to which we implemented these optimizations does not yet yield large optimizations for most non-trivial examples, but the optimizations we provided also provide a lot of information that will be useful when further improving the language before it is put to use next semester.

## 13 Credit Distribution

We went beyond our project's 100% goal and the work should be distributed in a 70/30 ratio in favor of Adam, because he designed and implemented {Setɥy} and lead the project in terms of the ideas behind the project and the design.