Colocolo

Optimizations on Ocelot, a Relational Logic Solver in Rosette

Thomas Bernardi, Altan Haan

March 2019

1 Introduction

Ocelot. Ocelot is an embedding of relational logic, including some of Alloy, in Rosette. It was originally aimed for use in MemSynth, a tool for reasoning about memory consistency: which is the problem of defining how parallel threads can observe their shared memory state [4]. As such, a couple of optimizations that could have been implemented were not because they would not have had much effect on the type of problems that MemSynth would present. In particular, Ocelot did not implement skolemization or reduction to CNF for SAT (it simply sends queries to Rosette's default solver, Z3).

Problem Statement. Can these two optimizations, skolemization and translation to CNF, bring significant performance improvements to Ocelot?

Our Approach Colocolo extends Ocelot to include these two optimizations. We build on top of Ocelot's existing engine and implement the optimizations based on reference implementations from Kodkod [7], the current backend for Alloy.

2 Skolemization

In this section we provide a high level description of the implementation details of skolemization. To distinguish between the syntax of quantifiers and their semantics we introduce two new terms:

Definition 2.1 (Weak quantifier)**.** A **weak quantifier** is one that *semantically* acts as an existential quantifier (i.e. $\exists$ and $\neg\forall$).
Definition 2.2 (Strong quantifier)**.** A **strong quantifier** is one that *semantically* acts as a universal quantifier (i.e. $\forall$ and $\neg\exists$).

Skolemization is a transformation that moves weak quantifiers to the highest syntactic level by replacing weakly quantified variables with functions (or relations in the case of alloy). Skolemization relies on a key second order equivalence:

$$\forall x \exists y \ . \ P(x, y) \Leftrightarrow \exists f \forall x \ . \ P(x, f(x)) \tag{1}$$

where $f(x)$ is a function (relation) that maps x to y. Intuitively this is translating between "forall x there exists a y such that $P(x, y)$" into "there exists a function $f : x \to y$ such that, forall x, $P(x, f(x))$." This is useful because the existence of an f that satisfies the subformula is semantically equivalent to the satisfiability of the subformula if f is a symbolic constant sent to the solver.

In the case of bounded model checking (in both Kodkod and Ocelot), this significantly reduces formula size. Suppose we have the universe $\{A_1, \ldots, A_n\}$. Considering the formula from our equivalence 1, $\forall x \exists y \ . \ P(x, y)$ would get translated as follows:

Without skolemization:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} P(A_i, A_j) \tag{2}$$

With skolemization:

$$\bigwedge_{i=1}^{n} P(A_i, f(A_i)) \tag{3}$$

2.1 The Algorithm

Definition 2.3 (Declaration)**.** A **declaration** allows us to define variables in terms of other expressions (and all declarations can eventually be traced back to relations in the universe). A declaration contains a **variable**, a **multiplicity**, and an **expression**.

$$\underset{\text{variable}}{x} \quad : \quad \underset{\text{multiplicity}}{\text{some}} \quad \underset{\text{expression}}{E}$$

An weak quantifier extends the current scope with new bindings as defined in its **declarations**. For each of these **declarations** $d := v \ : \ m \ R$, skolemization does the following:

Assume $S_\forall$ is a list of all strongly quantified variables that are currently in scope.

- Define the **skolem relation**, R_v of arity $\text{size}(S_\forall) + \text{arity}(v)$ (every element of $S_\forall$ should be unary).
- Define our **skolem expression** E_v which will replace every occurrence of v further down the AST.
- Approximate an **upper bound** for the **skolem relation**.
- Calculate **domain constraints** that ensure the "inputs" to R_v are constrained in the same way that their corresponding universally quantified variables are—store these, these will be conjuncted with the formula on the way up the AST once there are no more existential quantifiers. Specifically, these constraints encode multiplicity information that relational bounds cannot infer.
- Add **range constraints** to the formula to ensure that the "outputs" of S are constrained in the same way as v.

Skolem Expression

The **skolem expression**, E_v, is what actually replaces v. It is the result of joining each non-skolem $a_i \in S_\forall$ with the R_v, or $\mathsf{E}_v := f \ S_\forall \ \mathsf{R}_v$ where $S_\forall$ is interpreted as a list $[a_1, \ldots, a_n]$ sorted by order of declaration, and f is defined as follows:

$$f \ [] \ X = X$$
$$f \ a_i :: L \ X = f \ L \ a_i.X$$

Upper Bound

At the moment, Colocolo assumes all quantified variables will be quantified over sets. This way, the upper bound of the **skolem relation** is simply the cross product of the upper bound of each strongly quantified variable (in order of declaration), together with the upper bound of R in the last position.

Domain Constraints

Suppose R is of arity n. We call U the universe. Let $I_v = \mathsf{R}_v.\mathsf{U}\ldots\mathsf{U}$ where U is joined to R_v n times. Let the ith declaration $a_i \ : \ m_i \ R_i$ be the declaration that binds the ith variable, $a_i \in S_\forall$. Let $k = \text{length}(S_\forall)$. Our domain constraint is:

$$I_v \ \texttt{in} \ \{a_0 \ : \ m_0 \ R_0, \ldots, a_k \ : \ m_k \ R_k \mid \top\}.$$

Note the expression $\{a_0 \ : \ m_0 \ R_0, \ldots, a_k \ : \ m_k \ R_k \mid \top\}$ defines a k-ary relation where the ith element obeys the ith **declaration** with no other constraints (any relation satisfies $\top$).

The domain constraints for each subformula in the skolemized formula are lastly conjuncted together at the top level to form the final domain constraint for the whole formula.

Range Constraints

Our range constraint is simply $(\mathsf{E}_v \ \texttt{in} \ R) \wedge (m \ \mathsf{E}_v)$. Let $f' := \mathsf{skolemize}(f[\mathsf{E}_v/v])$ in $Qv \ : \ m \ R \mid f$ where $Q \in \{\exists, \neg\forall\}$. When our quantifier is $\exists$ the skolemized (sub)formula becomes $(\mathsf{E}_v \ \texttt{in} \ R) \wedge (m \ \mathsf{E}_v) \wedge f'$, otherwise when Q is $\neg\forall$ we skolemize to $[(\mathsf{E}_v \ \texttt{in} \ R) \wedge (m \ \mathsf{E}_v)] \rightarrow f'$.

2.2 An example

Let A and B be relations of arity n and m respectively, and let P be a predicate on $A \times B$. Suppose we wish to skolemize the formula

$$\forall x : \texttt{one} \ A. \ \exists y : \texttt{some} \ B \mid P(x, y).$$

We first traverse through the universal quantification of x, adding $x : \texttt{one} \ A$ to $S_\forall$. Next we reach the existentially quantified y. Following the above procedure, we define $\mathsf{R}_y :_{m+1} [\{\langle\rangle\}, U(A) \times U(B)]$, where $U(X)$ gives the upper bound on a relation X. We can then define the domain constraint

$$\mathsf{D}_y := \mathsf{R}_y \bullet_{i=1}^{m} \mathsf{U} \ \texttt{in} \ \{x : \texttt{one} \ A \mid \top\},$$

along with the skolem expression $\mathsf{E}_y := x.\mathsf{R}_y$. Note that $\mathsf{R}_y.\bullet_{i=1}^{m}$ means left-associatively joining U to R_y m times. Lastly, our range constraint is $(\mathsf{E}_y \ \texttt{in some} \ B)$. As the quantifier is just an $\exists$, our skolemized subformula becomes $(\mathsf{E}_y \ \texttt{in some} \ B) \wedge P(x, \mathsf{E}_y)$. Conjuncted with the top level domain constraints, we obtain the total skolemized formula

$$\mathsf{R}_y \bullet_{i=1}^{m} \mathsf{U} \ \texttt{in} \ \{x : \texttt{one} \ A \mid \top\} \wedge [(\mathsf{E}_y \ \texttt{in} \ B) \wedge (\texttt{some} \ \mathsf{E}_y)] \wedge P(x, \mathsf{E}_y).$$

Replacing E_y with $x.\mathsf{R}_y$, we get our final skolemized formula

$$\mathsf{R}_y \bullet_{i=1}^{m} \mathsf{U} \ \texttt{in} \ \{x : \texttt{one} \ A \mid \top\} \wedge [(x.\mathsf{R}_y \ \texttt{in} \ B) \wedge (\texttt{some} \ x.\mathsf{R}_y)] \wedge P(x, x.\mathsf{R}_y).$$

3 Translation to SAT

For Colocolo, we attempted to implement an optimized translation from arbitrary (quantifier-free) first-order boolean formulas to a CNF-SAT instance. In particular, we relied on Rosette's internal (symbolic) boolean AST representation, which we descended recursively and applied the following transformation rules:

$$\bigwedge_{i=1}^{n} F_i \rightsquigarrow \left(\bigwedge_{i=1}^{n} [\tilde{F}_i \vee \neg o]\right) \wedge \left(o \vee \bigvee_{i=1}^{n} \neg \tilde{F}_i\right), \tag{4}$$

$$\bigvee_{i=1}^{n} F_i \rightsquigarrow \left(\bigwedge_{i=1}^{n} [\neg \tilde{F}_i \vee o]\right) \wedge \left(\neg o \vee \bigvee_{i=1}^{n} \tilde{F}_i\right). \tag{5}$$

In these translations, o is a fresh auxiliary variable that is true if and only if the original n-ary formula is true, and $\tilde{F}_i$ represents the corresponding auxiliary variable produced by recursively applying the rule to F_i. Note that both (4) and (5) are translation rules used by Kodkod internally.

In the process of translating, we also encode each original boolean variable and new auxiliary variable as an integer (with negative literals represented as the negation of the integer representation of the variable). Lastly, the instance was translated into DIMACS format and sent to Lingeling, a fast SAT solver [5].

4 Results

4.1 Implementation Challenges

Skolemization. While skolemization seems like it should be a simple algorithm, it contains many nuances in the context of relational logic. The largest challenge was translating these details from Kodkod's implementation to work with Ocelot's AST. We found that the best way to understand Kodkod's implementation was to create a mathematical representation as described in Section 2. Once we had this, implementing it in Ocelot was relatively straightforward.

CNF-SAT Translation. During our implementation of the optimized CNF-SAT translation scheme, we encountered some performance difficulties due to the usage of Racket lists, which are linked lists with poor asymptotics. We fixed this slightly by using smarter list merging functions, although more improvements are definitely possible. A non-negligible bottleneck with our performance under this optimization can probably be alleviated using smarter data structures.

4.2 Benchmarks

Since we implemented a few optimizations on top of Ocelot, we are mainly interested in what sort of performance improvement we may have gotten. We are interested in both the overall improvement (time to solve an Ocelot problem), as well as performance of our final SMT/SAT query. To evaluate our optimizations, we translated a supported subset of the standard benchmarks/examples bundled with Kodkod, which are written in Java and use the Kodkod internal representation for problems. We ran the benchmarks over all combinations of optimizations, and recorded the total time, solver time, and whether or not the SAT solver returned within a 30-second time limit. We had difficulty implementing a time limit for Rosette's internal solver, so we did not impose one.

4.2.1 Meow: Compiling Kodkod to Ocelot

In order to translate Kodkod benchmarks into Ocelot/Colocolo, we wrote *Meow* in Java, a compiler from Kodkod ASTs into equivalent problem instantiations in Ocelot/Colocolo.

Graph structure. Internally, Kodkod represents its formulas and expressions as nodes in a DAG, rather than a tree. This node sharing is crucial to Kodkod's internal notion of equality between relations (and nodes in general), and additionally improves memory performance. This means the translation into Colocolo cannot simply inline each node. To address this, we cached the Kodkod nodes during the graph traversal, and declared them in the compiled output in a topologically sorted order (where incoming edges represent syntactic children).

Ocelot limitations. During our benchmark compilation, we discovered some limitations of Ocelot that went beyond the scope of our project. In particular,

- Ocelot currently has no support for integers, which Kodkod supports through extensive bit-blasting.
- Ocelot has difficulties with higher-arity declarations in quantified formulas. In particular, Ocelot assumes all declarations to be unary, with `one` multiplicity.
- Some relational operators are missing from Ocelot, such as relational override and if-then-else.

To deal with these restrictions, we only used benchmarks that did not utilize any of the mentioned features.

4.2.2 Benchmark Results

Figure 1 is enough to get an idea of the effects of each optimization on performance but we included a detailed graph in Figure 2 to show that that there there are some cases where certain optimizations perform contrary to the averages shown in Figure 1.

Our best (and really only) performance improvement is when we use skolemization, where we get a 2.7× speedup in total runtime and 4.1× speedup in solver runtime. As discussed above, this is most likely due to the significant decrease in formula size we get when we apply skolemization. Note that the SAT solver time includes the time spent translating to CNF in Rosette.

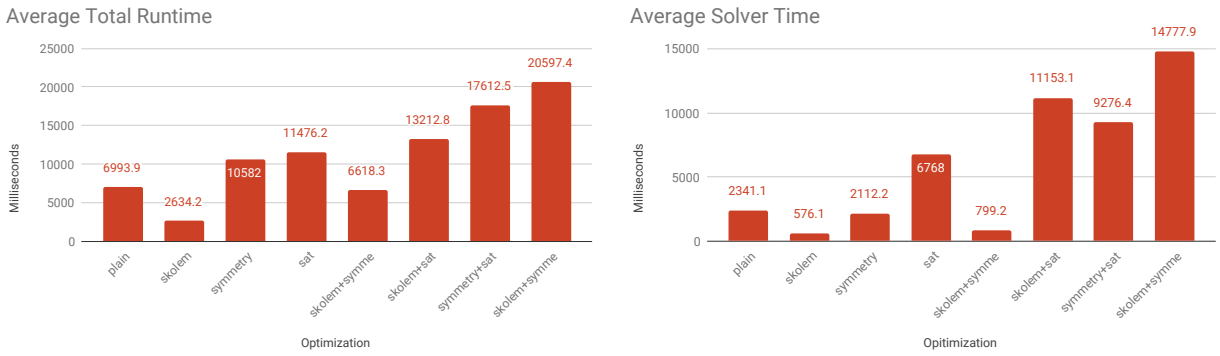

Figure 1: Averages.

4.3 Project Potential

We see significant improvement in performance with the application of skolemization. Further investigation is required to see why symmetry breaking and translation to CNF do not improve results. In particular, we conjecture that the CNF translation works better in Kodkod due to its conjunction with circuit compaction. Answering these questions and addressing our findings would be a significant step towards a conference paper.

We believe that if we managed to get the performance improvement we hoped for from these two optimizations, implemented a compact boolean circuit representation within Ocelot, and addressed limitations in its expressiveness, then Colocolo would become a robust and expressive tool. In particular, if we could achieve the same performance as Kodkod, Colocolo would provide a much more versatile embedding of Alloy than is possible in Java, which would open up many new possibilities for exploration and experimentation in relational logic.

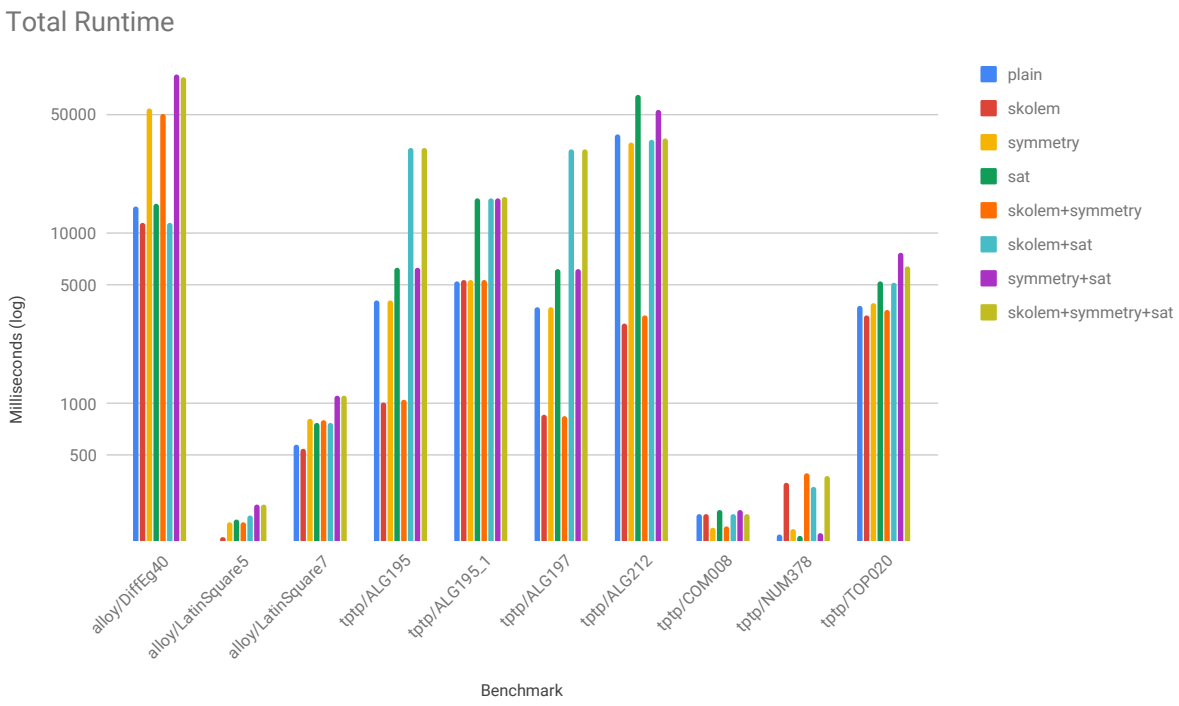

Figure 2: Total Times.

5 Member Contributions

Thomas took the lead on understanding Kodkod's implementation of skolemization and translating it into an equivalent algorithm over Colocolo's AST. Altan translated Kodkod's algorithm for reducing a boolean formula to CNF and created a translator from Kodkod's AST to that of Colocolo. Both Thomas and Altan spent time testing and fixing bugs in all parts of the project as well as running benchmarks.

6 Course Topics

Colocolo touched on most of the core topics that were covered in the course (though less from the applications seen towards the second half). The most important topics we drew from were **Bounded Model Checking** and **Relational Logic** in **Alloy** and of course **Rosette**. The project itself is a **Bounded Model Finder** written on top of **Rosette**—about half the course involved gaining proficiency in Rosette and an entire lecture was spent talking about **Bounded Model Finding**. Additionally, we applied knowledge gained about **SAT Solvers** and **SMT Solvers** in our translation to CNF and comparison to **Z3**.

The only topic we did not talk about in detail was **Skolemization**. We briefly touched on it but did not talk about any sort of implementation. Of course, our project involved a much deeper dive into the aforementioned topics than we had in class but this is to be expected. The biggest example were the details of **Alloy's Relational Logic**, which were only briefly discussed in lecture and clear, extensive documentation was difficult to find. Again, this is to be expected, the point of the course is to have a starting point to jump off from. Overall, the course provided us with the skills we needed to be able to get started and have a good idea of what direction we needed to go.

References

[1] Abdulla, P.A., Bjesse, P and Een, N.: *Symbolic reachability analysis based on sat-solvers* (TACAS'00).

[2] Aloul, Fadi A. and Sakallah, Karem A. and Markov, Igor L.: *Efficient Symmetry Breaking for Boolean Satisfiability* Proceedings of the 18th International Joint Conference on Artificial Intelligence, (IJCAI'03).

[3] Andersen, H.R. and Hulgaard, H.: *Boolean expression diagrams* (LICS'97).

[4] Bornholdt, James and Torlak, Emina: *Synthesizing Memory Models from Framework Sketches and Litmus Tests*, Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'17).

[5] Biere, Armin: *CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017*, Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions.

[6] Claessen, Koen and Sörensson: *New Techniques that Improve MACE-style Model Finding* Proceedings of Workshop on Model Computation (MODEL'03).

[7] Torlak, Emina and Jackson, Daniel: *Kodkod: A Relational Model Finder*, Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07).