LANGUAGE LEVELS IN TEACHING AN INTRODUCTORY FORMAL METHODS COURSE

QIANFAN CHEN

Computer Science Honors Thesis
Brown University, May, 2022

Advisor
Tim Nelson
Brown University Computer Science

Reader
Ben Greenman
Brown University Computer Science
ABSTRACT

When teaching Alloy-like model finders to students without a strong background in discrete mathematics, a lot of time at the beginning of the course can be spent reviewing and teaching concepts of relations and set theory. This thesis explores the possibility of using epistemically closed language levels as stepping stones and starting the course with a beginning level language where relations are restricted to functions, in order to avoid this issue and teach students to model concrete problems earlier in the semester. The work in this thesis is implemented in the language used to teach CS1710 at Brown University, and deployed in the Spring 2022 semester version of the course.
I would like to thank Professor Tim Nelson for his excellent mentoring and unconditional support, Professor Shriram Krishnamurthi for his insightful guidance, and Doctor Ben Greenman for his tremendous help in discussions and suggestions as well as his help with Racket knowledge. I would like to thank Benjamin Ryjikov for his help in the early development and implementations of Forge language levels. I would like to thank my friends and family for their emotional support and their encouragement.
## CONTENTS

1 **INTRODUCTION**  

2 **BACKGROUND**  
   2.1 Model Finders and Forge  
   2.2 Teach Formal Methods with Model Finder  
   2.3 Language Levels  

3 **ISSUES TEACHING WITH FORGE AND SOLUTION**  
   3.1 Function vs. Relation for Teaching  
   3.2 Solution: Language Levels in Forge  
      3.2.1 Functional Language for Beginner Level  

4 **LANGUAGE LEVEL IMPLEMENTATION AND FROGLET**  
   4.1 Language Level Interface  
   4.2 Froglet the Beginning Student Language  
      4.2.1 Restrictions on Field Declaration  
      4.2.2 Restriction on Relational Operators  
      4.2.3 Force Singletons  
      4.2.4 Error Messages  
   4.3 Deparsing  
   4.4 Builtin Libraries Enhances Expressiveness  
      4.4.1 Mixed Language Levels  
   4.5 Mistakes Made and Fixes  

5 **EVALUATION**  
   5.1 Leaky Abstractions  
   5.2 Expressiveness of the Froglet Language  
   5.3 Teaching  
   5.4 Analysis of In-Class Quiz  
   5.5 Thoughts on Analysis  
      5.5.1 Quantification vs. relations:  
      5.5.2 Set vs. Relations  

iii
5.5.3 "Leaky Abstraction" from Students' Prior Knowledge

6 Contributions and Future Work

6.1 Contributions

6.2 Future Work

Bibliography
INTRODUCTION

For complicated systems with multiple states and large input space, and for programs where bugs could be disastrous, a large number of test cases is not enough to give us confidence in the correctness of the program. Formal methods help in situations like these by helping to model and analyze programs and systems using logic and mathematics. Model finders such as Alloy \cite{7} allow us to define the structure of a model, specify constraints using Boolean formulas, and ask the solver to give instances that satisfy certain constraints, verify user-specified properties, or give counterexamples.

At Brown University, we use Forge, our own model finder with syntax similar to that of Alloy, to teach CS1710, an introductory formal methods course. We aim to provide it as an inclusive course where students from all levels of background could learn to use formal tools to check models and verify properties, as well as to think formally about verifying any programs and systems they create.

However, there are also challenges for our goal of teaching an inclusive introductory formal methods course with Forge: in model finders like Alloy or Forge, everything is based on relations and set theory, and these concepts could be difficult for some students, especially those without a strong background in discrete mathematics. In previous years, the first two weeks of the course had to be spent mainly explaining sets and relations. We want this course to include students who might not be interested in theory at all, and let them dive right into modeling realistic systems without having to study or review discrete math first.

One of our proposed solutions to help with this problem is to use language levels, which aim to provide epistemic closures over just
enough syntax for each learning stage, and have been used in some introductory courses taught with Scheme or Racket [4]. A more detailed description of language levels could be found in Section 2.3. For our use in teaching CS1710, we designed a beginning student language level of Forge, which we call Froglet. The syntax of Froglet is essentially a small subset of the syntax of full Forge or Alloy, but through redesigned lectures and error messages, we explain the syntax differently from how we would explain the same syntax in Forge. Specifically, we want to use concepts from object-oriented programming rather than concepts from set theory and relations to explain definitions and expression operators. In this way, we hope to make it easier to understand and use model finders even for students with little background in mathematical theories.

This thesis shows that it is possible to implement the infrastructure for language levels in Forge, as well as to design and implement a functional language. Moreover, this functional language Froglet is expressive enough for the early teaching goals in our introductory formal methods course. I will present our design and implementation of language level infrastructures as well as the Froglet language in Forge. I will also try to address the questions of how our language level design helps students, where it is helpful and where it is not, and lastly, how it could be improved in the future, based on students’ projects, TA hour survey, and an in-class quiz.
2.1 MODEL FINDERS AND FORGE

Model finders such as Alloy and Forge allows users to define models, use logical formulas to add constraints, and ask the solver to verify theorems or automatically find models according to constraints. Model finders’ automation, speed, and the ability to produce instances or counterexamples [2] allows them to be used in various domains such as networking [9], security [1], and software engineering [10].

This section briefly introduces how model finders work, using small examples to demonstrate the basic concepts and some basic syntax. Forge has syntax inspired by Alloy, and is implemented using Racket. The motivation for implementing Forge and using it in CS1710 Logic for Systems rather than simply using Alloy in the course was so that the course staff could modify the language to modify or add features for education purpose. In this thesis, I will write code in Forge syntax. Readers who are interested in more detailed description of Alloy or Forge syntax are encouraged to read their respective documentations [7] [6].

Listing 1: Sig definition

#lang forge

sig Node{
    neighbor: set Node
}

The above code in Listing 1 defines a "signature", or "sig", called Node, and a field called neighbor which is actually a relation that, for each Node, points to a set of other Nodes.

Then we can add constraints in a "predicate", or "pred", and ask the solver to give us instances satisfying certain predicates.

Listing 2: Pred

```forge
#lang forge

sig Node{
    neighbor: set Node
}

pred complete{
    all u, v: Node | u != v => u in v.neighbor
}

run {
    complete
} for exactly 4 Node
```

In this program 2, the predicate complete defines the constraint that for any node u, any other node v must be in the set of u’s neighbors. Then, when we run the model on exactly 4 Node, with this pred enforced, the result would be a 4-complete graph. When this code actually gets executed, the default approach for Forge to display the result is through Sterling Visualizer [3], as shown in Figure 1. As we can see, it displays a 4-complete graph with Node0, Node1, Node2, Node3, and a neighbor relation between each pair of them. We can also ask Sterling to display in table view, as shown in Figure 2.

The dot "." operator in v.neighbor looks very similar to field access, but under the hood it is actually relational join. The fact that in situations like above, the same dot operator could be viewed either
Figure 1: Sterling Visualizer in graph view, for results from Listing 2

Figure 2: Sterling Visualizer in table view, for results from Listing 2

as field access or relational join [7] would be very important in our design and implementation of beginning student language level.

To better understand relational join in Forge, we need to understand how sigs and fields are represented internally in Forge. For Forge, all sigs and fields are relations. As can be seen from the table view of Sterling Visualizer in Figure 2, the field neighbor is a binary relation from Node to Node. The sig Node itself, on the other hand, is a singleton relation. As one can imagine, we can also define fields to be relations of high arity. For example, we can have a definition shown in 3, where flights is a field defined to be lone City->City->Plane,
under the sig Airline. Therefore, flights is a relation of arity 4, with the type being Airline->City->City->Plane.

Listing 3: Join Operator

```forge
#lang forge
sig City {}  
sig Plane {}  
sig Airline{  
    flights: set City->City->Plane
}
```

Listing 4: Join Operator

```forge
#lang forge
sig Node{  
    neighbor: set Node
}

one sig A, B extends Node{}

pred twoHop{  
    A->B in neighbor.neighbor
}

pred TransitiveClosure{  
    A->B in ^neighbor
}
```

To illustrate these concepts of relations and relational join more concretely, we can look at the next example 4. In this example, we have one sig A, B extend Node{}, where one sig means that the sigs A and B each have only one instance (while there could be multiple Node, there could be only one A), and A, B extend Node{} means that A and B inherit the definitions of Node (like subclasses in object ori-
ented programming; and of course, additional fields could be added in \{\} ). In the pred \texttt{twoHop}, the relational join \texttt{neighbor.neighbor} combines each pair of entries of \texttt{neighbor} where the right column of one is identical to the left column of the other, resulting in a new binary relation. The pred \texttt{twoHop} then adds the constraint that the relation \texttt{A->B} is a subset of the relation \texttt{neighbor.neighbor}, which means \texttt{B} is \texttt{A}'s neighbor's neighbor. Similarly, for any specific positive integer \( n \), one can build a relation indicating \texttt{B} reachable from \texttt{A} in \( n \) hops. If one wants to say that \texttt{B} is within reach from \texttt{A} by at most two hops, then one could use \texttt{A->B} in \( (\texttt{neighbor} + \texttt{neighbor.neighbor}) \). However, that only works up to any hard-coded number of hop. If one wants to say that \texttt{B} is within reach from \texttt{A} by any number of hops—that is—by repeatedly visiting neighbors starting from \texttt{A}, then one would use the pred \texttt{TransitiveClosure}. This pred adds the constraint \texttt{A->B} in \( ^{\text{neighbor}} \), with \( ^{\text{neighbor}} \) being the transitive closure of \texttt{neighbor}, which means \texttt{neighbor} + \texttt{neighbor.neighbor} + \texttt{neighbor.neighbor.neighbor} + \ldots.

### 2.2 Teach Formal Methods with Model Finder

Most parts of CS1710 Logic for Systems is taught with a model finder (namely Forge), as model finders best suits our vision of this course as an inclusive and practical introduction to formal methods. Compared to other tools taught in formal methods courses, such as proof assistants, model finders requires much less background knowledge in discrete math and proof theory. Moreover, especially for students who are not going to pursue a career or education in theory-related areas, model finders provide more versatility and practicality than proof assistants, and could help model practical problems or systems in real-life engineering applications.
In many introductory programming courses taught with Scheme or Racket, the language is divided into several levels, for example, Beginning Student, Intermediate Student, Advanced Student, and Full Language [4] [8], with each language level providing enough expressive power for teaching a certain set of concepts. The goal is that each language level acts like a stepping stone to help students learn new concepts gradually, without being exposed to too many new concepts at the same time. Without language levels, exposure to too many concepts at the same time might cause difficulties for students. For example, to use the full language, even just for trivial programs, many new concepts might need to be explained. In addition, students might unintentionally write in syntax not yet learned, resulting in unintentional results, silent failures, or error messages that they are not able to understand. A good language level design aims to solve these problems. Therefore, within one language level, irrelevant or incompatible
concepts and features from other language levels should not be allowed to interfere with the pedagogy of the current language level [5]. Using language levels rather than several separate languages to achieve this goal helps keep the languages more consistent throughout the course and also reduces the amount of work in developing the languages by reusing the same core language and a lot of shared syntax and semantics [8].

In this project, we applied the language level idea to Forge, and the Froglet team has built language levels around different interpretations of operations in Forge to divide the learning experience into separate stages.
3.1 Function vs. Relation for Teaching

One of the biggest issues in previous years of teaching CS1710 Logic for Systems was the fact that the Forge language relied too much on relations. Ideally, to understand relations, relational operators, and set theory, students need to have a background in at least an introductory discrete mathematics course. However, that does not align fully with our vision for the education offered in this course.

In CS1710, we want to equip students with the knowledge, the skills, and the insights to use formal methods in real life applications—and not just for the students who appreciate logic and formal science for the sake of their beauty within, but also the majority of CS students who will not take elective theory courses. We believe that any student, regardless of their level of background in discrete mathematics, could and should benefit from learning formal methods. Therefore, instead of spending too much time at the beginning of the course on mathematical prerequisites for learning Forge, we would prefer to use a redesigned version of Forge so that students can immediately start learning how to apply system-focused logical reasoning to concrete models.

We think that a good starting point is to use an abstraction that relies on functions rather than relations. Coming from high school, students generally have a much better understanding of functions than that of relations. We think it is a good idea to build on the intuitions that they already have. In fact, the Alloy language (and Forge which has a very similar syntax) already allows a subset of the syntax, where relations are restricted to functions, to be interpreted using familiar
concepts such as field access instead of relations (the details of which will be discussed in later sections). However, it is not enough for us to simply only teach students that subset of syntax, and pretend that there are only functions and function application is just field access—if students unintentionally tries something beyond that subset, the unexpected outputs or resulting errors mentioning relational concepts would be really confusing. To prevent such disasters from happening, we need language levels to prevent them from being exposed to concepts beyond that functional subset of the language.

## 3.2 Solution: Language Levels in Forge

We think that language levels could be a helpful concept in solving the aforementioned issues. Instead of overwhelming students with relations and set theory at the same time that they are introduced to the topic of logic for systems, we could use language levels to create an epistemic closure so that they have limited exposure to new concepts at each stage in their learning. In this way, we hope to help students learn these new concepts gradually without being overwhelmed.

In particular, at the beginning of the course we want to use the functional subset of Forge syntax to avoid the need to explain relations before teaching anything substantial about formal methods. Language levels could help to prevent students’ exposure to nonfunctional concepts in the Forge language.

### 3.2.1 Functional Language for Beginner Level

We started carrying out our idea of language levels in Forge by designing and implementing a beginning student level language. As Forge language files have the file extension .frg, which could be pronounced "frog", we decided to call this beginning student level language Froglet, which means a younger frog (and could still retain
The key idea of this Froglet language is to restrict relations to functional or partial functional ones. A functional relation is one where for each value of the the left column (or columns, for relations with arity higher than 2), there is one and only one possible value for the right most column. In a partial function, for each value of the the left columns, there is one or zero value for the right most column. If we restrict relations to functions, then the relational joining of a sig with a binary relation (which is probably the most common case used in Forge) could simply be viewed as field access [7]. For example, in Listing 5, \texttt{Plane0.Model} could be seen as accessing the \texttt{Model} field of \texttt{Plane0}, an object of the class \texttt{Plane}, rather than the actual relational interpretation of joining \texttt{Plane0} to the relation \texttt{model}. This use of "." as field access is similar to what students have learned about object-oriented programming from introductory CS courses. Moreover, even relational join on relations with higher arity could be viewed as accessing the value corresponding to certain keys in a hash table, which will be discussed in more detail in section 4.2.2.

Listing 5: Functional Fields in Froglet

```forge
#lang forge/bsl
sig Model{}
sig Plane {
    model: one Model
}
```

When we restrict relations to functions in Froglet, we could first teach students basic Forge/Froglet syntax and the ideas of model finders without having to expose them to concepts of set theory and relations. Instead, they could learn to use model finders using familiar concepts from object-oriented programming.

In the Spring of 2022, course CS1710 was taught with Froglet for the first third of the semester, amounting to roughly half the time of
the model finder part of the semester. After two assignments and one self-proposed project using Froglet, the course would transition to teach with full Forge, explaining the concepts of set theory and relations in full Forge, giving students the ability to model more complex systems.
4

LANGUAGE LEVEL IMPLEMENTATION AND FROGLET

4.1 LANGUAGE LEVEL INTERFACE

According to our design for using language levels in Forge, the same piece of syntax written in different language levels should always compile to the same core language syntax. However, in different language levels the same piece of syntax could be interpreted differently in different language levels (for example, "." operator interpreted as field access in Froglet but relational join in full relational Forge). Since an essential idea of the language levels is to prevent students being unnecessarily exposed to advanced concepts, it is important that each language level forms its own epistemic closure, i.e. students should not be able to see higher level abstractions of a concept. Therefore, it is important to provide specifically designed error messages for each language level.

The key feature in our implementation of the language level infrastructure is different error checking functions for different language levels. This interface is implemented by choose-lang-specific.rkt, which sets several error-checking functions and symbols for the compiler to use. The reader.rkt of each language can then use this interface to set each of these error checking functions and symbols to its own choice, as shown in Figure 4.

The current interface includes ast-checker-hash, a set of functions to check syntax at AST-node creation time; checker-hash, a set of functions to check syntax and semantics used in last-checker.rkt, before information is sent to the solver; and check-lang, a symbol that indicates which language level is used.
Figure 4: Language-specific checks

Note that these language-specific checks do not aim to replace all error checking; rather, they are used in addition to the majority of error-checking that is shared among all languages. The point of
language-specific checks is to enforce restrictions on syntax that is allowed in the base language but prohibited in the specific language level, or to overwrite the error message thrown in order to accommodate different interpretations of the same syntax across different language levels.

4.2 FROGLET THE BEGINNING STUDENT LANGUAGE

As mentioned in Section 3.2.1, the main idea of Froglet the beginning student level language is to avoid concepts from relations and set theory, and to use only functional relations rather than all relations. Our implementation to achieve this goal involves several components: we restrict field definition so that all fields are functions or partial functions; we restrict the use of relational operators; we restrict the syntax so that every expression written explicitly by students must evaluate to an object (or Null), rather than a relation. All of these restrictions are accompanied by appropriate error messages to make sure that we provide a consistent interpretation of the syntax within one language level.

4.2.1 Restrictions on Field Declaration

In the original Forge language, fields could be defined to be of one of different multiplicities, namely set, one, lone, func, pfunc [7]. In Froglet, we prohibit the use of set field declarations.

Furthermore, we make sure to allow one and lone only for fields of a singleton type (in other words, fields that expand to a binary relation in the compiler), such as the field model in the code 6 (which expands into a binary relation of the type Plane->Model), while allowing func and pfunc only for fields of higher arity, such as the field flights. For students with a background in object-oriented programming, it is natural for them to think of model: one Model as a field of Plane
that points to another object of the type Model, not so much different conceptually from a field in, say, Java. For fields with higher arity such as flights: pfunc City->City->Plane, students are encouraged to think of them as a field containing a hash table or a dictionary. For example, flights: pfunc City->City->Plane can be viewed as a field of Airline, a field of hash table type, where the key is of the type (City, City). More precisely, for pfunc, or a partial function, the field is just like a hash table, while for func, or a function, the field is like a hash table with a value defined for each key in the domain.

We started restricting one and lone to fields of a singleton type, and func and pfunc to fields of higher arity when designing Froglet. However, we found this to be a reasonable restriction in general, not just for beginning students, so we plan to move this restriction to the base Forge language.

Listing 6: Field Multiplicity in Froglet

```bash
#lang forge/bsl
sig City {}
sig Model{}
sig Plane {
    model: one Model
}
sig Airline{
    flights: pfunc City->City->Plane
}
```

4.2.2 Restriction on Relational Operators

We prohibit the use of most relational operators, namely the ones shown in table 1, with the exception that set comprehension {} is allowed when directly after a cardinality operator, namely that the use of #{} is allowed, while {} by itself is prohibited, as in Froglet,
<table>
<thead>
<tr>
<th>PROHIBITED OPERATOR</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>Subset of</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Cross Product</td>
</tr>
<tr>
<td>+</td>
<td>Set Union</td>
</tr>
<tr>
<td>-</td>
<td>Set Difference</td>
</tr>
<tr>
<td>{}</td>
<td>Set Comprehension</td>
</tr>
<tr>
<td>&amp;</td>
<td>Set Intersection</td>
</tr>
<tr>
<td>~</td>
<td>Transpose</td>
</tr>
<tr>
<td>^</td>
<td>Transitive Closure</td>
</tr>
<tr>
<td>*</td>
<td>Reflexive Transitive Closure</td>
</tr>
</tbody>
</table>

Table 1: Prohibited Relational Operators

#() is interpreted as "counting the number of instances that satisfies the specified condition", without appealing to set theoretical concepts. We implement these restrictions in bsl-ast-checker-hash and bsl-checker-hash so that whenever they encounter an AST-node with the corresponding operators, an error message is thrown saying "Froglet cannot recognize [...] operator". Moreover, in the cases of '+' and '-', we also point out in the error message that if the user wants addition and subtraction of integers, they should look at add and subtract instead.

The only "relational operator" we would like students to use in Froglet is the relational join operator "." and its syntactic sugar box join operator "[]", with B[A] simply meaning A.B.

In Froglet, the dot join operator "." is viewed as "field access". For example, under the definitions in Listing 6, if we let Plane0 be an instance of Plane, then Plane0.model would mean joining the atom Plane0 with the relation model; in Froglet, on the other hand, it would
be interpreted as accessing the `model` field of the object `Plane0`. Of course, both abstractions would lead to the same result.

Previously when students were taught in full Forge, they were told that the box join "[ ]" is a syntactic sugar of the dot join ".". In Froglet, however, it is viewed as getting the value corresponding to a key in a hash table. For example, `Airline0.flights[City0][City1]` would mean "first get the `flights` field of `Airline0`, which is a hash table with keys being two cities, then access the value corresponding to the key `(City0, City1)`. Under the hood, this expression gets desugared to `City1.(City0.(Airline0.flights))`, which leads to the same result. However, students using Froglet are not informed of the relation between the dot join operator "." and the box join operator "[ ]".

To be more consistent with this interpretation, we also made sure to include the syntactic sugar `Airline0.flights[City0, City1]` which desugars to `Airline0.flights[City0][City1]`, for those who are more comfortable with this style of using a tuple as the key.

4.2.3 Force Singletons

In Froglet, we do not want students to be exposed to concepts of set theory and relations. It is not enough to simply prohibit the definition of non-functional relations and the use of relational operators. Students might be (possibly accidentally) using a relation (or, in Froglet terms, using a field without accessing it from an object) in a formula; for example, one might write `no Airline0.flights[City0]` when trying to write `no Airline0.flights[City0][City1]`. There are two serious problems if students are allowed to write formulas like this:

1. It might lead to the formula being an unintended constraint, and it would be very difficult for students to debug.

2. More importantly, when student ask why this formula works (or why it doesn’t work), we would not be able to answer the question under the concepts of Froglet.
To address these problems, we decide that expressions like \( \text{no} \ \text{Airline0}.\text{flights}[\text{City0}] \), which evaluate to a relation with arity higher than 1, should not be allowed.

The above design choice prevents some exposure to non-functional relations. Similar problems arise with set theory. For example, \( \text{Plane}.\text{model} \) would evaluate to a set of \( \text{Model} \) objects; and even the expression \( \text{Plane} \) by itself evaluates to a set of \( \text{Plane} \) objects. To address such issues, we decided that expressions evaluating to sets should not be allowed either.

Combining the two design choices above, we only allow expressions that evaluate to Null or singletons, which just means a single object in Froglet abstractions. Of course, we cannot simply prohibit all AST-nodes of non-singleton expressions, as singleton expressions such as \( \text{Plane0}.\text{model} \) also contain the sub-expression \( \text{model} \), which is a non-singleton expression. Instead, we enforce a singleton check at the top-level expressions—that is, on either side of the equality operator “\( = \)”, or following a multiplicity claim such as \( \text{no}, \text{one} \) or \( \text{some} \). In full Forge, there would be scenarios for top-level expressions, such as on either side of \( \text{in} \), but those scenarios would not appear in Froglet as most relational operators are prohibited. Notice that the only way to build expressions in Froglet is through relational join, either using the dot join or the box join operator. Thus, we implement the singleton check by checking the current join result and recursively checking the left hand side of the join (if the left hand side of the join is a non-singleton, such as \( \text{Plane} \) or \( \text{model} \), then the result of the join cannot be a singleton either).

### 4.2.4 Error Messages

In order to prevent students being exposed to concepts of set theory and relations, we need to carefully design error messages for Froglet language-specific checks, as well as to overwrite some of the
error messages in Forge. The former is relatively easier to do, and we mostly work on thinking how to phrase messages such as "this operator is not recognized in Frogllet" or "this expression is not a singleton" to make them more helpful to students’ understanding and debugging. For the latter, we looked at error messages thrown in Forge, and considered whether they could leak unwanted abstractions into Frogllet level. For example, in Forge, when the compiler knows that a relational join always results in an empty set, it would raise an error saying "join always results in empty", which is not acceptable in Frogllet, as users of Frogllet are not supposed to know what "join" is. Instead, in this situation, Frogllet adds additional checks to determine different cases, and either throws an error saying "Sig ... does not have such a field ..." or saying "Field access on ... which is not a singleton object".

4.3 Deparsing

Previously, when Forge throws an error for a particular syntax, the error message would cite the syntax that raised the error, but in racket parenthetical syntax. For example, `Plane0.flights` would cause an empty join error, and in the case the error message would cite `(join Plane0 flights)`. The error messages behaved this way because, based on our current implementation of Forge, it was very difficult to preserve the original user input syntax at the time of error, and it was much easier for the compiler to just throw an error using the internal representation of the piece of syntax. It worked previously, as users could read the parenthetical version of the syntax. In Frogllet, however, students’ understanding of the syntax is now different from the internal parenthetical representation of that same piece of syntax. For example, students would understand `Plane0.flights` as trying to access the `flights` field of the object `Plane0`, and `(join Plane0 flights)` would no longer make sense to them.
To partially address this problem, we implemented a deparser, which takes in the internal racket parenthetical representation of any expression or formula, and outputs the same expression or formula in surface Forge syntax. This method is not perfect, as the deparsed syntax might be different from the original user input (for example, $a \iff b$ desugars into $a \implies b$ and $b \implies a$, and the deparser outputs the latter), but it provides a piece of surface syntax close enough for students to understand.

Ideally, we could have just used the syntax location and opened the original file to retrieve the original user-input syntax. However, there were several challenges for us:

1. Preservation of syntax locations was not implemented perfectly (for example in partial instances), so sometimes the syntax location simply doesn’t work.

2. Some constraints are not directly generated by user input (for example, we would constrain that children of a parent sig are disjoint) and thus do not have syntax location.

The first issue should eventually be fixed, and the best solution of the second issue is to redesign the error messages related to non-user-generated constraints. However, due to time constraints at that moment, a deparser was the most efficient option for us.

After implementing the deparser, we decided to add it also to the base Forge language, as any user could benefit from error messages represented in surface syntax, which is more intuitive and easier to read.

4.4 BUILTIN LIBRARIES ENHANCES EXPRESSIVENESS

Admittedly, the restriction to functions would limit the expressiveness of the language. Specifically, the concept of transitive closure, or reachability, is crucial to many models, but is not expressible in the
4.4 BUILTIN LIBRARIES ENHANCES EXPRESSIVENESS

Froglet language. Moreover, it is very difficult in Froglet to express a collection of data, which is necessary in many models. To allow students to model more complex problems before learning the full relational Forge language, we created several builtin library functions, that allow students to model constraints inexpressible in Froglet.

We included a sequence library that allows students to enforce a field to be a sequence, and several helper functions, such as returning the last index in the sequence and checking whether there are duplicates in the sequence. This library allows students to model systems with sequential states or inputs whose length might vary in different instances (so that the states cannot be hard coded as "State0", "State1", etc.).

We also included a helper function reachable, with \texttt{reachable[A, B, r1, r2, \ldots, rn]} meaning "A is reachable from B by one or more applications of accessing one of the fields r1, r2, \ldots, rn". In implementation, \texttt{reachable[A, B, r1, r2, \ldots, rn]} actually desugars to \texttt{B in A.^(r1 + r2 + \ldots + rn)}, where ^ means transitive closure. This helper function allows students to use transitive closure without having to explain transitive closure in terms of binary relations.

4.4.1 Mixed Language Levels

One great challenge in introducing library functions was that now we have mixed language levels in one program (for example, the main program is written in Froglet, but calls library functions, which are implemented in full Forge), which makes it a difficult question how to correctly execute language-specific checks. Our solution was to add a symbol \texttt{check-lang}, which indicates which language level each piece of syntax comes from, as mentioned in Section 4.1. The syntax expander macros then pass the \texttt{check-lang} to every AST-node constructed from user written syntax. For example, a \texttt{iff b} desugars into \texttt{a implies b} and \texttt{b implies a}, so if the \texttt{iff} constructor macro
sees a `bsl check-lang` symbol when constructing a `iff b`, it will pass this symbol to the `and` node as well as the two `implies` nodes that it constructs. The AST-nodes constructed by library functions, on the other hand, do not contain that `check-lang` symbol (technically, we let those nodes contain another `check-lang` symbol, indicating that there is no need to perform a language-specific check). Then, the language-specific error-checking functions are only enforced on an AST-node if the `check-lang` symbol for that node indicates that it should be checked with that language’s language-specific error checking.

### 4.5 Mistakes Made and Fixes

In our implementation we initially made some mistakes, leading to bugs or leaky abstractions:

1. Forge is not traditional "programming language" and it could produce different kinds of "runtime" errors, which could be thrown at different stages. One problem was that certain errors in Forge are thrown before language-specific checks are run. For example, when constructing AST-nodes, it is possible to have arity mismatch (basically type mismatch) errors for certain relational operators. Even though these relational operators are not in Froglet, these error messages could still be thrown as they are run earlier than language-specific checks. These error messages could be confusing to students as they are not exposed to the concepts of arity. We solved this problem by moving the execution of many of the language-specific checks from `last-checker.rkt` to `ast.rkt`, to make sure they are run before the arity mismatch checks, in order for language-specific errors to be thrown before arity mismatch checks are applied.

2. Our initial implementation of singleton checks on relational join was not complete. At first we only prohibited any expression that is not a chain of field access from a singleton object,
such as quantified sig or one sig. For example, in the model described by code 6, \texttt{flights[City0]} is not a chain of field access; 
\texttt{Airline.flights[City0]} is not field access from a singleton object. However, we overlooked the possibility of a chained field access from a singleton object of a relation resulting in a relation of arity higher than 1, such as the expression \texttt{Airline0.flights[City0]}, which in relational explanation is of the type \texttt{City -> Plane}. This could potentially lead to confusion if students wonder what \texttt{Airline0.flights[City0]} means, as we do not have the concept of a relation in Froglet. We later realized this problem and fixed it using our implementation described in 4.2.3.
5.1 Leaky Abstractions

In order for language levels to work perfectly, each level should create an epistemic closure, and different abstractions from other language levels should not leak in through error messages or through unintentionally allowed or prohibited syntax. In practice, when CS1710 Logic for Systems was taught in 2022 Spring, we discovered several places where the concepts from full Forge leaks through. Fortunately, we were able to resolve most of those problems of leaky abstractions through the semester:

1. Some leaky abstractions happened are described in 4.5 and fixed.

2. Library functions could introduce unexpected error messages from Forge. For example, `reachable[A, B, r]` desugars to `B in A.^[r]`. It would result in "empty join" error if `A.r` is an empty set. As `B in A.^[r]` is produced by a library function, not user input syntax, the error message is thrown by Forge, saying "join always result in an empty relation", instead of the desired error message thrown by Froglet, which would say "Sig A has no such field r". We tried to partially resolve this by giving the AST-node `A.r` the `check-lang` symbol of Froglet—that is, even syntax produced by library functions, should be checked by user’s language level, if possible. However, this does not solve this problem entirely, as we cannot apply Froglet language-specific checks to, for example, `in` and `^` in `B in A.^r`, as they use operations prohibited in Froglet. During semester, we did not have the time and resource to modify the language level infrastruc-
tures to solve this problem, but it would be worth exploring for future iterations.

3. Instance syntax: Definition of partial instances have syntax shown in Listing 7. We can see that to define a sig, we have to use "+" to union all the instances of that sig; even worse, definition of fields (relations) uses the relation builder "->". Limited to partial instances, these operator do not contradict our functional abstraction of Froglet, but might unexpectedly lead students to misunderstandings.

Listing 7: Quiz Program

```plaintext
inst MyInst {
  sig Person {} 
  sig Room {} 
  sig Course {
    instructor: one Person, 
    TAs: set Person, 
    classroom: lone Room, 
    prereqs: set Course 
  }
  one sig CS1, CS2, CS3, MATH1 extends Course {} 
  CS1 = 'CS1 
  CS2 = 'CS2 
  CS3 = 'CS3 
  MATH1 = 'MATH1 
  Course = 'CS1 + 'CS2 + 'CS3 + 'MATH1 
  prereqs = 'CS1->'CS3 + 'CS1->'MATH1 + 'CS2->'CS1 
}
```

4. Sterling Visualization and Evaluator: right now the Sterling visualizer has a graph view and a table view, both of which were designed for relation based models, and is not extremely helpful for models written in the function based Froglet language.
Moreover, the evaluator of Sterling is currently similar to the full relational Forge language, without Froglet restrictions, which it should eventually adopt.

### 5.2 Expressiveness of the Froglet Language

Perhaps the greatest worry of using the Froglet Language is that because of the expressive limitations of functions, not many interesting models could be written with Froglet. Fortunately, through our use of Froglet this semester, we found this was not too much a problem.

As mentioned in Section 3, the Froglet part of the course included two homework assignments and a self-proposed midterm project. In the assignments, students were able to model systems like family trees, stacks with push and pop actions, and various river crossing puzzles. During this part of the semester, Froglet was expressive enough for these assignments, which helped to teach students the basic syntax, logic and conventions used in model finders.

Equipped with these knowledge learned in Froglet assignments, students were able to do a midterm project, working in pairs modeling a system of their own choice. We were glad to find that Froglet was expressive enough for modeling almost all groups’ self-proposed projects. The following table shows a list of midterm projects modeled by students. The vast majority of student groups were able to do their projects in Froglet, fully modeling their proposed systems or at least making significant progress.

<table>
<thead>
<tr>
<th>Group</th>
<th>Project</th>
<th>Pure Froglet?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2048 Game</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>Fast Food Orders</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>SET Game</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>Connect 4 Game</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>Game</td>
<td>Y/N</td>
</tr>
<tr>
<td>---</td>
<td>-------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>5</td>
<td>007 Game</td>
<td>Y</td>
</tr>
<tr>
<td>6</td>
<td>Battleship Game</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>Minecraft Villager Traders (Simplified)</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>UNO Game</td>
<td>Y</td>
</tr>
<tr>
<td>9</td>
<td>Operations of a Bakery</td>
<td>Y</td>
</tr>
<tr>
<td>10</td>
<td>Valid BST</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>Connect 4 Game</td>
<td>Y</td>
</tr>
<tr>
<td>12</td>
<td>Five Hand Poker</td>
<td>Y</td>
</tr>
<tr>
<td>13</td>
<td>Chopsticks Game</td>
<td>Y</td>
</tr>
<tr>
<td>14</td>
<td>Life Cycle of a Thread</td>
<td>Y</td>
</tr>
<tr>
<td>15</td>
<td>Connect 4 Game</td>
<td>Y</td>
</tr>
<tr>
<td>16</td>
<td>Valid Moves in 2048 Game</td>
<td>Y</td>
</tr>
<tr>
<td>17</td>
<td>Connect 4 Game</td>
<td>Y</td>
</tr>
<tr>
<td>18</td>
<td>Connect 4 Game</td>
<td>Y</td>
</tr>
<tr>
<td>19</td>
<td>Dijkstra’s Algorithm</td>
<td>Full Forge</td>
</tr>
<tr>
<td>20</td>
<td>Project Partner Matching</td>
<td>Z3</td>
</tr>
<tr>
<td>21</td>
<td>&quot;Literature&quot; Card Game</td>
<td>Y</td>
</tr>
<tr>
<td>22</td>
<td>BST and Operations</td>
<td>Full Forge</td>
</tr>
<tr>
<td>23</td>
<td>Lambda Calculus</td>
<td>Full Forge</td>
</tr>
<tr>
<td>24</td>
<td>NBA Trade</td>
<td>Y</td>
</tr>
<tr>
<td>25</td>
<td>Count to 21</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>3 X 3 KenKen Puzzle</td>
<td>Y</td>
</tr>
<tr>
<td>27</td>
<td>BS Card Game</td>
<td>Full Forge</td>
</tr>
</tbody>
</table>
A few groups asked for permission to use full relational Forge (as full relational Forge was introduced to students during their midterm project working time), but they used it just to "unlock" certain functionalities related to sets, while still mostly using Froglet abstractions of dot operation as field access functions instead of relations. The projects done with full Forge involved modeling Dijkstra algorithm, BST, Lambda calculus, BS game and Othello Board game, though an-

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Frog Hop Game</td>
<td>Y</td>
</tr>
<tr>
<td>29</td>
<td>Sealed-Bid Second-Price Auction</td>
<td>Y</td>
</tr>
<tr>
<td>30</td>
<td>Bray-Liebhafsky Chemical Reaction</td>
<td>Y</td>
</tr>
<tr>
<td>31</td>
<td>Wordle</td>
<td>Y</td>
</tr>
<tr>
<td>32</td>
<td>Connect 4 Game</td>
<td>Y</td>
</tr>
<tr>
<td>33</td>
<td>Group and Homomorphism</td>
<td>Y</td>
</tr>
<tr>
<td>34</td>
<td>Conway’s Game of Life</td>
<td>Y</td>
</tr>
<tr>
<td>35</td>
<td>Mafia Game</td>
<td>Y</td>
</tr>
<tr>
<td>36</td>
<td>BST</td>
<td>Y</td>
</tr>
<tr>
<td>37</td>
<td>Get Home Game</td>
<td>Y</td>
</tr>
<tr>
<td>38</td>
<td>Mafia Game</td>
<td>Y</td>
</tr>
<tr>
<td>39</td>
<td>Iterative Deepening DFS</td>
<td>Y</td>
</tr>
<tr>
<td>40</td>
<td>Traffic Lights in a Network</td>
<td>Y</td>
</tr>
<tr>
<td>41</td>
<td>Othello Board Game</td>
<td>Full Forge</td>
</tr>
<tr>
<td>42</td>
<td>Elevator Operation</td>
<td>Y</td>
</tr>
<tr>
<td>43</td>
<td>Baseball Game</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 2: Students’ Midterm Projects
other group successfully used Froglet to modeled BST structure with fewer operations.

Students were able to use Froglet to model some impressive and complicated models such as Conway’s Game of Life, Bray-Liebhafsky Chemical Reaction, Fast food restaurant ordering systems, group theory theorems, and Kenken puzzles. Many of the groups behind these projects found ways to deal with collections of data despite using the functional Froglet language. For example, a group modeling Kenken puzzle were able to model the cages of cells as a sequence of cells, thus without having to use sets.

From homework assignments and midterm self-proposed models, we can see that the Froglet language was expressive enough for teaching the basics of formal methods and model finders, and for students to explore most of their own interested topics. Moreover, in the few exceptions where Froglet was not expressive enough for student’s projects, the limitation lies more in the need for sets rather than relations.

5.3 Teaching

With the help of Froglet, this semester of CS1710 was able to move at a much faster pace at the beginning, allowing students to experience more realistic modeling even in shopping period. In previous years, the first assignment was almost exclusively on binary relations (as shown in Figure 5). In contrast, this year during the first assignment students were able to model the stack data structure (as shown in Figure 6). For the second assignment, students were able to model several river-crossing puzzles (for example, how to take 3 wolves and 3 goats to the other side of the river), which was identical to an assignment from last year that came much later in the semester. Not only were students able to do the river crossing puzzles earlier in the semester, their code in this assignment was also more readable to TAs,
thanks to the use of quantification instead of relational expressions, as described in Section 5.5.1.

5.4  ANALYSIS OF IN-CLASS QUIZ

Right before we transitioned to full Forge in course, a quiz was given in class, where we briefly described how some set operators work in full Forge, gave students model definitions in full Froge shown in 8, and several run commands with multiple potential outcomes. Then we asked students to indicate whether they expected and liked each outcome. This is class quiz was intended to study how students expect Forge syntax to work given their knowledge in Froglet and their intuitions, and to later explain to them how it actually works.

We are glad to find out that after half a semester with Froglet and a brief description of sets at the beginning of the quiz, most students...
were able to figure out some Forge questions on the quiz on their own. However, they also disliked or feel surprised by some other full relational Forge syntax. Here are some interesting questions and results from the quiz:

1. For the questions shown in figures 7 and 8, many students found it surprising and unlikable that an element could be equated with a set. Admittedly, we did not tell students prior to the quiz that a single element is treated as a singleton set in Forge, partly in order to surprise them and emphasize this point. However, students’ answers still do demonstrate that this set theoretical abstraction is not intuitive for students.

2. For the questions shown in figures 9 and 10, though there is still the previous problem of comparing a set to a singleton, from
Listing 8: Quiz Program

```plaintext
sig Person {}
sig Room {}
sig Course {
    instructor: one Person,
    TAs: set Person,
    classroom: lone Room,
    prereqs: set Course
}
one sig CS1, CS2, CS3, MATH1 extends Course {}

inst MyInst {
    CS1 = 'CS1
    CS2 = 'CS2
    CS3 = 'CS3
    MATH1 = 'MATH1
    Course = 'CS1 + 'CS2 + 'CS3 + 'MATH1
    prereqs = 'CS1->'CS3 + 'CS1->'MATH1 + 'CS2->'CS1
}
```

Figure 7: Quiz Question and Response

their choices and comments we can see most students have a correct understanding of non-functional dot join in this case, that CS1.prereqs evaluates to 'CS3 + 'MATH1, despite the fact that
3. For the question shown in 11, most students found the expression `prereqs.CS1` (which should evaluate to the set of courses which have CS1 as their prereqs, namely CS2) confusing, with
some students commenting that the expression should be invalid, and some other commenting that it might be equivalent to \( CS1.prereqs \). This shows that relational join that looks different from field access is still difficult for students to comprehend.

![Figure 11: Quiz Question and Response 5](image)

The conceptual difficulties for students presented in these quizzes demonstrate that, by avoiding teaching or reviewing all the set theoretical and relational concepts at the beginning of the semester, Froglet helps to enable a faster-paced and more in depth education of formal methods at the first month of the course. However, we also need to keep in mind that we need to explain these abstractions carefully and clearly when transitioning to full relational Forge.

5.5 **Thoughts on Analysis**

During the Froglet part of the course, we also asked our TAs to collect interesting questions from the students. From the TA survey, students’ questions online on EdStem, and the in-class quiz, we have several interesting thoughts.

5.5.1 **Quantification vs. relations:**

As relational operators are not allowed in Froglet, to express the same constraints, in general, students now need to use more quantifica-
tions. For example, in the Listing 9, both predicates injective-rel and injective-quant express logically equivalent constraints, namely that the relation \( r \) is injective. To avoid the relational operators in pred injective-rel, we can see that in injective-quant, the formula has to be much longer and using three layers of nested quantification.

Listing 9: Expression using quantification vs. relation

```plaintext
#lang forge

sig Atom {
  r: set Atom
}

pred injective-rel{
  r.\~r in iden
}

pred injective-quant{
  all a, b: Atom | a != b implies (no c: Atom | c in a.r and c in b.r)
}
```

Quantifiers such as no, some, and all are much easier to explain to students than relational operators and are also mostly consistent with their intuition of those words. Moreover, the quantification version could be much easier to read compared to the relational version for TAs in case students make a minor mistake. However, in practice, for students without a strong background in theory and logic, quantifications could also be difficult to write and debug, especially with the conjunction of constraints in nested quantification. For example, in the Listing 10 from the 8-queen chess puzzle, which enforces the constraint that queens cannot attack each other, there are nested quantifications with other constraints between the nested layers. In practice, some students had a difficult time writing this correct con-
straint, and often could not easily debug the problem if one quantifier was incorrect. Therefore, we think that in the future design of course and assignment, it is important to think carefully about what kinds of quantification students would need to write certain constraints.

Listing 10: Nested quantification

```plaintext
pred notAttacking {
    // Enforcing uniqueness is a stumbling point
    all q1row, q1col, q2row, q2col : Int |
    // distinct queens
    Board.position[q1row][q1col] != Board.position[q2row][q2col]
    =>
    some Board.position[q1row][q1col] and some
    Board.position[q2row][q2col] => {
        q1row != q2row
        q1col != q2col
        absDifference[q1row, q2row] != absDifference[q1col, q2col]
    }
}
```

5.5.2 Set vs. Relations

In this year’s development of Froglet for CS1710, we have treated sets and relations roughly equally. We prohibited both of them at the Froglet language level, and allowed both of them at the next level, namely full relational Forge. However, in our surveys and quizzes, it seems that it is generally easier for students to intuitively understand sets and set operators such as +, -, and &,, while understanding relational operators requires more training. It is worth considering strategically adding sets to Froglet or adding a new intermediate language level that allows sets. This is not a trivial task: while performing a relational join on a set of objects makes total sense, performing "field
access” on a set of objects could be very confusing, so it requires careful design of language and pedagogy if one were to allow sets without relations.

5.5.3 “Leaky Abstraction” from Students’ Prior Knowledge

When using Froglet, we found that a small number of students were able to use skills not yet taught, sometimes to understand leaky abstractions. When asked why they were able to do so, they said that they had shopped CS1710 in previous years so that they knew that they were actually dealing with relations, or that they had a background in discrete math and relational database that allowed them to recognize the relational nature of Froglet. For example, when discussing how the library function reachable behaves, one of the students said that knowing it was transitive closure from last year’s class shopping helped them understand this library function easier. We were glad that most of them said that they still liked Froglet and preferred it to the previous year’s teaching. Even for them, who already know relations, starting with Froglet did a better job at presenting key concepts and motivations of formal methods clearly, compared to starting with full relational Forge, where the first assignment focused a lot on understanding relations rather than actual models.

However, the fact that some students were able to figure out the relational nature of Froglet still reminds us that some students have a strong background in discrete math and even have a deep understanding of relations. While we try to be as inclusive as possible for students without such a background, when we design the course, we should also try to improve the experience of students who do have such a background, and consider the risk and positive or negative consequences of such students figuring out the relational nature of Forge earlier than intended.
CONTRIBUTIONS AND FUTURE WORK

6.1 CONTRIBUTIONS

This work explores the possibility of using language levels in a model finder in teaching an introductory course on formal methods. We have implemented the infrastructure for language levels in Forge. We also designed and implemented Frogl, a beginning student language level where relations are restricted to functions, and relational joins are interpreted as field access. We have proved that it is possible to design and implement language levels and a functional language level for model finders with the Alloy language syntax.

The course CS1710 is taught with the new Forge with language levels, using Frogl for the first third of semester. Frogl has been shown to be expressive enough for the teaching goals during that period, and even for most of students’ self-proposed midterm projects.

We have investigated the effects and difficulties using Frogl to start the course. With Frogl, the course was able to move quicker to modeling more concrete and practical systems, and to direct more of the students’ learning to modeling and logical reasoning in systems rather than math and syntax. Restricting the relations to functions also required more use of quantification, which is more intuitive than relations and more readable to TAs, but is itself a difficult topic for students.

I believe that this work has shown that using language levels in model finders is worth considering and further exploring for teaching introductory formal methods courses.
6.2 FUTURE WORK

Further research could be done to investigate the possibility of separating set-theoretical concepts and relational concepts, enabling the possibility of language levels that have sets but not general relations. It is also worth investigating how to best strategically deploy such language levels in teaching and then design courses accordingly.

Specifically for the implementation of Forge and Froglet associated in this work, further effort could be put into better reducing leaky abstraction across language levels. More attention could be paid to error messages and syntax restriction in places with separate pieces of syntax, such as instance block or Sterling evaluator. Froglet syntax restrictions should also be imposed on the Sterling evaluator language. It would be a good idea to explore the possibilities of redesigning a partial instance syntax for Froglet, as well as redesigning a mode of Sterling visualizer for the function-based Froglet language. It is also worth investigating how to better design library functions to prevent leaky abstractions resulting from incorrect input types. Furthermore, eventually the deparser could be replaced by actual quoting actual input syntax, which would require fixing all preservation of syntax location, and designing error message thrown from non-user input constraints.
BIBLIOGRAPHY


