3Twelf: A Tool for Reasoning About Programming Languages

Author: Key Shin
Advisor: Jonathan Aldrich
Date: May 15, 2007

1 Abstract

Programming language theory is a completely new concept to many students, and many times it is one of the disciplines that students struggle the most with during their introductory computer science curriculum. This is exacerbated by the fact that most of the work assigned in helping to learn the basic concepts usually involve tedious hand-written proofs that take a turnover time of at least a week before the grader can give adequate feedback. To help students in their academic endeavors, we have designed and developed the basis of a tool that will allow students to experiment with programming language theory by creating and modeling various language features as well eventually being able to define and prove theorems about these models. We based the interface of the tool on human notation taught in related classes and utilize the powerful mechanics of Twelf.

2 Introduction

In many cases, computer science students com into their undergraduate collegiate environment with at least some knowledge about programming as well as some coursework in advanced mathematics. A smaller, lucky amount may even have some coursework in areas such as discrete mathematics. When it comes to classes involved in programming language theory, however, students are many times exposed to this sort of material for the very first time, and thus are less likely to rapidly adapt and will have a more difficult time learning the material since it is such a radical departure from anything the student has experienced before. This is further exacerbated by the fact that traditionally in classes that teach one to reason about programs, or even more specifically when it comes to proving theorems about programs, the student must be do work by hand and thus the feedback process, which is also done by hand, is painfully slow. Because of this, the feedback process is competely guided by how quickly the teaching staff can grade or comment on said work. Realistically, this process usually takes at least a week after the assignment has been handed in.

The major problem with this sort of manual process is that concepts taught in programming language courses build upon each other throughout the course, therefore not having a firm grasp on past material when new ideas are introduced is extremely detrimental to the educational environment. If we examine courses that include a significant programming aspect we can notice that in many ways a student can somewhat teach himself when he is confused through error messages from the compilation and execution of his code. Furthermore, programming assignments allow for some sort of incremental development pattern in that students can write and test small parts of a program and be relatively assured that these "functions" more or less work according to specifications. In many classes, the teaching staff even include some sort of automated testing process for assignments that further guide students and help in the learning process. The hand-written assignments that are necessary for courses related to programming languages really have none of these benefits. One could even draw some sort of relationship between writing sub-proofs or lemmas as incremental development, however, because there is no form of affirming these lemmas as correct it does not receive the normal benefits of incremental development.

2.1 Related Work

There are a variety of tools that the logic and programming language community have created for a variety of needs. The core audience of these tools are a diverse group since some tools are specifically aimed as educational tools while others are aimed for career researchers and those who closely work with them. Two such tools are Tutch and Twelf. Tutch is primarily aimed at undergraduates while Twelf is aimed at an audience of a higher level.

Tutch was designed for solving problems related to constructive logic and is used to help students in the course 15-399: Constructive Logic at Carnegie-Mellon University [1]. Tutch also has support for protocols such as proof annotations of programs. The constructive logic course, in many ways, aims to teach students about programming language theory through the Curry-Howard Isomorphism which states informally that "proofs are programs" and utilizes Tutch as an important tool. An example of Tutch input for a constructive logic proof is shown in Figure 1 which proves the statement $A \rightarrow (B \rightarrow A)$. In more recent work Tutch has been extended to support other logical paradigms including linear logic and classical logic. Tutch allows users to input some sort of proof and then mechanizes the process checking the proof for correctness. It also attempts to give the user meaningful errors and through this the user can many times correct his errors and learn from the entire process. The major problem with Tutch is that the logics are hard coded into the system, and the user is not given an option to define his own logic or language as well as the semantics that would govern said logic or language. Because of this, the usefulness is restricted to pre-existing systems.

```
proof kcomb : A => (B => A) =

begin;
[ A;
  [ B;
    A];
  (B => A)];
A => (B => A);
end;
```

Figure 1: Example Input to Tutch

Twelf is a “language used to specify, implement, and prove properties of deductive systems such as programming languages and logics,” and is heavily modeled on logical frameworks [3]. Twelf can be used to solve and model a wide variety of proofs and program semantics [2]. Example syntax for Twelf is shown in Figure 2 which is a possible way to represent the typed λ-calculus. Unlike Tutch, Twelf can model a large variety of language and allows the user to model his own custom language as well as defining the semantics for it although there are probably types of systems in which Twelf can not model at all or particularly well. Twelf syntax, however, is complicated to learn in a short period of time and it has a great deal of extra functionality that can further confuse its users. This is because Twelf assumes that the users have a good understanding of complex type systems such as a fully dependently typed λ-calculus. It could take some, even with a general understanding of types, until one would truly be comfortable in using the tool for everyday usage. Because of these reasons, Twelf is not very helpful as it currently stands in the undergraduate academic environment which prefers tools that quickly adaptable and have immediate use without much training or practice and is at minimum designed for scientists who have completed at least one such undergraduate course.

2.2 Goals

The problems with Twelf and Tutch highlight the major needs of a tool used in aiding students in reasoning about programs in courses related to programming languages. First, it should be easy to use and quickly understandable. Because of this requirement we believe that the input method should closely model the format in which the material is taught in the classroom and how it is presented on paper. An example of how syntax may be modeled in a classroom is shown in Figure 3. This is an example of defining the natural numbers where one can either have zero or the successor of some number n. The format in which the natural numbers have been defined is known as Backus-Naur form and it and its variants are used in defining a wide

```
e : type.
tau : type.

e1: tau -> (e -> e) -> e.
e2 : e -> e -> e.
e3 : e.
e4 : (e -> e -> e) -> (e -> e) -> e.

tau1 : tau.
tau2 : tau -> tau -> tau.
tau3 : (tau -> tau) -> tau.

value : e -> type.
val-unit : value e3.
val-fn : {T:tau}{E:e -> e} value (e1 T E).

reduce : e -> e -> type.
c-app-l : reduce E1 E1' -> reduce (e2 E1 E2) (e2 E1' E2)
c-app-r : reduce E2 E2' -> value E1 -> reduce (e2 E1 E2) (e2 E1 E2').
r-app : value E2 -> reduce (e2 (e1 T ([x:e] (E x))) E2).
```

Figure 2: Example Input to Twelf

$$\begin{array}{lcl} \mathsf{nat} & ::= & \mathsf{zero} \\ & | & \mathsf{succ(nat)} \end{array}$$

Figure 3: Syntax of natural numbers

variety of grammars and systems. The immediate benefit of using the syntax in which material is presented is that it reduces the additional information that a student must learn to use the tool. It is important to never forget that the first and foremost purpose of the tool is to help students reason about program semantics and not learn how to use the tool itself.

Because of this important requirement, it would seem that Twelf would in fact not be useful, but we must not forget that Twelf has a team of dedicated researchers that use it successfully as well as add to its functinality, therefore it would be a boon to the student if somehow we could harness its power for the average undergraduate student. Therefore, it makes sense that instead of proposing the development of a radical new “theorem prover” which we aim at a core audience of undergraduate students, we instead create some sort of user interface for an established tool with tried and true processes. This tool should somehow encode the established tool’s processes in a way accessible to students. Creating an extensible interface allows two great benefits. First, by creating an extensible interface for Twelf we can hopefully tap a great deal of Twelf's capabilities and allow its usage for the non experts. Second, if the student wishes to further explore the discipline of programming language theory, then it familiarizes the student with Twelf and makes the transition to Twelf easier.

There are two other major goals that need to be accomplished to make a productive tool for students. First, the feedback system must be developed in a way such that it is useful to students. This includes useful error messages along the lines of the error messages that one would hope to get from a compiler. Error messages can also include hints that may indicate possible corrections to what the user has input. Without feedback, the tool would not be very helpful as an educational tool since it is highly unlikely that students would receive a correct "compilation" on the first try. Finally, the relationship between programming language theory and the tool itself should be immediately evident because the reasoning for creating the tool is to assist in reasoning about program. The tool should accomplish this goal in a method which is worthwhile to the classroom setting.

2.3 Technical Contributions

We would like to accomplish a wide variety of deeds with the creation of the tool. First of all, we would like to make a tool that is so helpful that students will elect to use the tool even though their instructors may not specifically require them to. Furthermore, we hope to invigorate excitement and appreciation for formal specifications of languages by partially elimating the tedious aspects of learning the related material. We also believe that the method in which we have approached the tool leads to some novel methods as well as determining some of the required aspects of formally encoding a language or logic.

We would like the code to be immediately understandable to a student that is attempting use the tool as well as others who may view the tool's input. This causes some technical problems because the input syntax is not truly streamlined for mechanical processes. We had to figure out a way to model languages in a way that is condusive to both humans and machines. This requires us to encode a larger amount of data than may be required for tools that are designed for purely mechanical syntax. Because of this, we have had to split our architecture into multiple passes to capture all the required information as well as determine what parts of the syntax are not necessary for the system, but are necessary for human understanding. These methods and reasonings are explained more thoroughly in the design and implementation section.

2.4 Outline

We will first discuss the design and implementation of the tool which will include a overview of input format, the backend data structures, and the mechanisms which involve compiling the input code into Twelf code. Then we shall discuss the results which include the types of systems which we believe can be modeled with this tool as well as the systems which we explicitly modeled as test cases. Finally we will discuss some of the future work that we believe can be incorporated into the tool to improve its capabitilites as well as briefly discuss the possible methods in which one could model proofs within our system.

3 Design and Implementation

There are three distinct sections the development of the tool. The first is the creation of some sort of specification for the input format. The input format should closely resemble the concepts and procedures in which the related material is taught in the classroom. Second, we had to design data structures which will represent the inputted information within the tool itself. Complementary to the data structures is the design of the translation process which details the translation from the tool's input format to the eventual form of Twelf input syntax.

To more easily facilitate the explanation the design and implementation of the tool, we have divided this section into three parts corresponding to the major parts of development. In the first section we will discuss the construction of the input syntax and its supported capabilities. The second section will detail the technical design of the data structures which will lead to the third section which will detail the translation process.

3.1 Input Syntax

Input into the tool is currently limited to textual input, much in the way a program might be written. The input syntax is divided into three sections: terminals, syntax, and judgments. The terminal section allows the user to define certain keyworrds in the language being defined. In reality, even though these tokens are not be defined, the tool should be able to determine the difference between terminals, nonterminals, and variables in the language therefore the declaration of the section may seem somewhat redundant. The terminal section, however, does serve two purposes. First it allows the user to symbolize to himself and others the insignificant tokens such that it is more understandable from a human's standpoint. Second, when a token is declared the system does not have to determine by its own processes how to tag that token for later use. In theory, this should allow the translation process to execute somewhat faster.

3.1.1 Syntax Section

The syntax section signifies to the tool that all following input, until the judgment section is reached, is related to the specification of the language which the user wishes to model. This section is designed such that users can model a language's grammar in Backus-Naur form. As per the standard rules for grammars defined in Backus-Naur form, a nonterminal can be defined as well as used in the definitions of itself and other nonterminals. Terminals are strictly located in the right side of a derivation rule. As in Backus-Naur form, one can have multiple choices for each nonterminal and each of these choices can equally stand for their respective nontermianl.

An example of a sample syntax input is shown in Figure 4 that shows this use case. The example syntax has a definition for an e nonterminal as well as a tau nonterminal. Note the usage of tau in the first rule for e. This is visually more apparent in the eventual Twelf translation output in Figure 2 where we notice a tau type in a declaration for the Twelf typing in the type constructor $e1$.

3.1.2 Judgment Section

Each judgment is declared with a judgment keyword followed by the actual declaration of the judgment form. This means that rules that are declared within the judgment should match the judgment form. For example, if we refer to the sample input syntax we see that the judgment value has the form of e which means that all the rules associated with the value judgment should have conclusions that match a singular e expression. In the reduce example, we can see that the form must be matched to some e expression followed by an arrow operation and concluded with another e expression. Syntactically it is necessary that the user enclose e expressions for the rules on both sides of the arrow operator with parentheses. This can be evidenced in the the example input in Figure 4. Here we see that for the conclusion in `c-app-l` contains two application expressions, one on each side of however both of these are enclosed in in parentheses to indicate the ending of an expression related to the judgment declaration. This makes it easier for the system to find the matching expression further discussed in the transformation section.

These rules that are associated with the judgment are declared after the judgment form and name

```
terminals fn unit

syntax

e ::= fn x : tau => e[x]
    | x
    | f
    | e e
    | ()
    | letrec f(x) = e[f,x] in e[f]

tau ::= unit
      | t
      | tau -> tau
      | forall t.tau[t]

judgment value e

-- val-unit
()

------------------ val-fn
fn x : tau => e[x]

judgment reduce e -> e

e1 -> e1'
-------------------- c-app-l
(e1 e2) -> (e1' e2')

e2 -> e2'
value e1
--------------- c-app-r
(e1 e2) -> (e1 e2')

value e2
-------------------------------- r-app
((fn x : tau => e[x]) e2) -> e[e2]
```

Figure 4: Example Input

declaration. Rules represent the methods in which we can arrive at a specific type of judgment. For example, the only two value judgments we can have in the example syntax is either the unit value or a value derived from a function application. Rules are declared in a way such that they mimic inference rules that are common in logic and programming language theory courses. The premises of the rule are the properties that must be satisfied for the conclusion to hold true. In the input syntax, the premises are listed on separate lines followed by a dividing bar and then finally the conclusion that they support. Furthermore, rules are given a name on the same line as the bar dividing the premises and the conclusions.

3.2 Data Structures

In this section we will discuss the data structures used to represent the input syntax. In general there is an a program object that contains the subsections of the program, such as the syntax and judgment declarations. Each subsequent data structure makes it simpler for translating the input syntax into Twelf syntax. in the figures representing the syntax for data structures, a token that has been overlined represents a list of that type.

3.2.1 Internal Representation One

This data structure encodes the input from a user. The syntax of this structure is shown in Figure 5 along with its respective concrete syntax of the actual impelementation in ML. The typing rules for the the syntax are shown in Figure 6. Because of the limitations of the parser and an effort to make the transitions eaiser to to understand and modifty, the intial data structure mostly just describes the structure of the input syntax without any significant effort towards converting the input towards code that could be executed in Twelf. The top level program object comprises of a list of terminal objects, a list of syntax objects, and a list of judgment objects and wraps all of the data from the input. The terminal objects are the convenience tokens discussed in the input syntax section.

A singular syntax object is comprised of an object that defines the nonterminal of the syntax item definition and a list of clause objects that represent choices of expressions that match that particular nonterminal. The clause objects consist of a list of element objects that represent each token within a clause. These element objects are broken up into two types which simply differentiate between normal tokens which correspond to the variable type and tokens that contain some sort of binding information which corresponds to the binder type. In this version of the input data structure, there is no specific typing information for the

		Abstract Syntax	Concrete Syntax
str	$::=$	string	$\mathtt{string}$
e	$::=$	$\mathsf{variable}^1(str)$	$\mathtt{Variable}^1(\mathtt{str})$
	$\mid$	$\mathsf{binder}^1(str_1, \overline{str_2})$	$\mathtt{Binder}^1(\mathtt{str}_1, \overline{\mathtt{str}_2})$
j	$::=$	$\mathsf{judges}^1(\overline{j})$	$\mathtt{Judges}^1(\overline{\mathtt{j}})$
	$\mid$	$\mathsf{jvariable}^1(str)$	$\mathtt{JVar}^1(\mathtt{str})$
	$\mid$	$\mathsf{jbinder}^1(str_1, \overline{str_2})$	$\mathtt{JBinder}^1(\mathtt{str}_1, \overline{\mathtt{str}_2})$
t	$::=$	$\mathsf{terminal}^1(str)$	$\mathtt{Terminal}^1(\mathtt{str})$
c	$::=$	$\mathsf{clause}^1(\overline{e})$	$\mathtt{Clause}^1(\overline{\mathtt{e}})$
r	$::=$	$\mathsf{rule}^1(str, \overline{j_1}, j_2)$	$\mathtt{Rule}^1(\mathtt{str}, \overline{\mathtt{j}_1}, \mathtt{j}_2)$
s	$::=$	$\mathsf{syntax}^1(e, \overline{c})$	$\mathtt{Syntax}^1(\mathtt{e}, \overline{\mathtt{c}})$
ju	$::=$	$\mathsf{judgment}^1(str, c, \overline{r})$	$\mathtt{Judgment}^1(\mathtt{str}, \mathtt{c}, \overline{\mathtt{r}})$
p	$::=$	$\mathsf{program}^1(\overline{t}, \overline{s}, \overline{ju})$	$\mathtt{Program}^1(\overline{\mathtt{t}}, \overline{\mathtt{s}}, \overline{\mathtt{ju}})$

Figure 5: Syntax of Internal Representation 1

element objects.

Judgment objects are made up of a string that identifies the name of the judgment, a clause object that stores the form of the judgment, and finally a list of rules that are associated with this judgment. Each rule consists of a name, a list of premises which consist of judge objects, and finally a conclusion judge object. The possible types of judge objects mirror that of a element objects in the syntax declaration by having both a variable and binder subtypes, however there is also a subtype that consists of a list of judges.

3.2.2 Internal Representation Two

This data structure represents the first effort towards associating type with the input syntax. An important change here is that we no longer store the list of terminals that were part of hte input syntax since they were only necessary as part of the actual typing process that occurs during the translation mechanism. translation mechanism between the first and second representation is further discussed in the translation section further along. The important changes in syntax and typing rules are presented in Figure 6 and Figure 8, while some parts have been omitted it is because they remain the same as the first representation.

.

The main change in the actual syntax datastructure is that the first parameter now consists of an element object, or specifically the nonterminal option of thelement object. The interesting sections that are changed

$$\frac{\overline{j} : \mathsf{judge}^1\ \mathsf{list}}{\mathsf{judges}^1(\overline{j}) : \mathsf{judge}^1} \qquad \frac{s : \mathsf{string}}{\mathsf{jvariable}^1(s) : \mathsf{judge}^1} \qquad \frac{s_1 : \mathsf{string} \quad \overline{s_2} : \mathsf{string\ list}}{\mathsf{jbinder}^1(s_1, \overline{s_2}) : \mathsf{judge}^1}$$

$$\frac{s : \mathsf{string}}{\mathsf{variable}^1(s) : \mathsf{element}^1} \qquad \frac{s_1 : \mathsf{string} \quad \overline{s_2} : \mathsf{string\ list}}{\mathsf{binder}^1(s_1, \overline{s_2}) : \mathsf{element}^1}$$

$$\frac{s : \mathsf{string}}{\mathsf{terminal}(s) : \mathsf{terminal}^1} \qquad \frac{\overline{e} : \mathsf{element}^1\ \mathsf{list}}{\mathsf{clause}(\overline{e}) : \mathsf{clause}^1} \qquad \frac{s_1 : \mathsf{string} \quad \overline{s_2} : \mathsf{clause}^1\ \mathsf{list}}{\mathsf{syntax}^1(s_1, \overline{s_2}) : \mathsf{syntax}^1}$$

$$\frac{s : \mathsf{string} \quad \overline{j_1} : \mathsf{judge}^1\ \mathsf{list} \quad j_2 : \mathsf{judge}^1}{\mathsf{rule}^1(s, \overline{j_1}, j_2) : \mathsf{rule}^1} \qquad \frac{s_1 : \mathsf{string} \quad s_2 : \mathsf{clause}^1 \quad \overline{s_3} : \mathsf{rule}^1\ \mathsf{list}}{\mathsf{judgment}^1(s_1, s_2, \overline{s_3}) : \mathsf{judgment}^1}$$

$$\frac{\overline{s_1} : \mathsf{terminal}^1\ \mathsf{list} \quad \overline{s_2} : \mathsf{syntax}^1\ \mathsf{list} \quad \overline{s_3} : \mathsf{judgment}^1\ \mathsf{list}}{\mathsf{program}^1(\overline{s_1}, \overline{s_2}, \overline{s_3}) : \mathsf{program}^1}$$

Figure 6: Typing Internal Representation One

		Abstract Syntax	Concrete Syntax
str	$::=$	string	$\mathtt{string}$
e	$::=$	$\mathsf{variable}^2(e, str)$	$\mathtt{Var}^2(\mathtt{e}, \mathtt{str})$
	$\mid$	$\mathsf{nonterminal}^2(str)$	$\mathtt{NonTerm}^2(\mathtt{str})$
	$\mid$	$\mathsf{terminal}^2(str)$	$\mathtt{Term}^2(\mathtt{str})$
	$\mid$	$\mathsf{binder}^2(e_1, \overline{e_2})$	$\mathtt{Binder}^2(\mathtt{e}_1, \overline{\mathtt{e}_2})$
j	$::=$	$\mathsf{judges}^2(\overline{j})$	$\mathtt{Judges}^2(\overline{\mathtt{j}})$
	$\mid$	$\mathsf{jvariable}^2(j, str)$	$\mathtt{JVar}^2(\mathtt{j}, \mathtt{str})$
	$\mid$	$\mathsf{jnonterminal}^2(str)$	$\mathtt{JNonTerm}(\mathtt{str})$
	$\mid$	$\mathsf{jterminal}^2(str)$	$\mathtt{JTerm}(\mathtt{str})$
	$\mid$	$\mathsf{jbinder}^2(j_1, \overline{j_2})$	$\mathtt{JBinder}(\mathtt{j}_1, \overline{\mathtt{j}_2})$
r	$::=$	$\mathsf{rule}^2(str, \overline{j_1}, j_2)$	$\mathtt{Rule}(\mathtt{str}, \overline{\mathtt{j}_1}, \mathtt{j}_2)$
c	$::=$	$\mathsf{clause}^2(\overline{e})$	$\mathtt{Clause}^2(\overline{\mathtt{e}})$
ju	$::=$	$\mathsf{syntax}^2(e, \overline{c})$	$\mathtt{Syntax}^2(\mathtt{e}, \overline{\mathtt{c}})$
p	$::=$	$\mathsf{program}^2(\overline{s}, \overline{ju})$	$\mathtt{Program}^2(\overline{\mathtt{s}}, \overline{\mathtt{ju}})$

Figure 7: Syntax of Internal Representation Two

$$\frac{e : \textsf{nonterminal}^2 \quad s : \textsf{string}}{\textsf{variable}^2(e, s) : \textsf{element}^2} \qquad \frac{s : \textsf{string}}{\textsf{nonterminal}^2(s) : \textsf{element}^2}$$

$$\frac{s : \textsf{string}}{\textsf{terminal}^2(s) : \textsf{element}^2} \qquad \frac{e_1 : \textsf{nonterminal}^2 \quad \overline{2} : \textsf{element}^2\ \textsf{list}}{\textsf{binder}^2(e_1, \overline{e_2}) : \textsf{element}^2}$$

$$\frac{\overline{e} : \textsf{clause}}{\textsf{clause}^2(\overline{e}) : \textsf{clause}^2} \qquad \frac{e : \textsf{nonterminal}^2 \quad \overline{s} : \textsf{clause}^2\ \textsf{list}}{\textsf{syntax}^2(e, \overline{s}) : \textsf{syntax}^2} \qquad \frac{\overline{s_1} : \textsf{syntax}^2\ \textsf{list} \quad \overline{s_2} : \textsf{judgment}^2\ \textsf{list}}{\textsf{program}^2(\overline{s_1}, \overline{s_2}) : \textsf{program}^2}$$

Figure 8: Partial Typing for Internal Representation Two

the most between the preceding data structure and this one is that of what occurs to the element object and the judge object. If we refer back to the figure which reflects derivation rules for second representation and observe the grammar for the element object and compare with the definitions for internal representation one.

The variable structure that exists in the first data structure simply stood for all tokens that were not of binder format. All tokens of type variable are converted into either nonterminals, terminals, or the variable structure of the second data structure. The variable type in the second datastructure differs in that it keeps both the nonterminal to which the variable belongs as well as the name of the variable itself. The changes in the judge type are exactly the same in the nature, except we keep the judge choice which allows us to declare a list of judges as a judge object. The declaration of the judge object has been left out of the figure because of its similarity to the declaration of the element object

3.2.3 Internal Representation Three

This data structure is the final stage before the final conversion to Twelf. The input is represented in such a way where typing for the syntax definitions are easily derived for both the syntax and the judgment declarations. The partial abstract and conrete syntax for the third representation are shown in Figure 9. We strip throw away data that is unnecessary to output the Twelf in the main datastructure, however we keep the extraneous data that describes the syntactical nature of the input in an outside data structure. We keep this information around in case we would like to output the original input sequences.

The actual construction of the data structure is very similar to both the second and first representations. The critical difference is that instead of the clause object be constructed out of a list of elements, it is made up of a tagged name, a list of terms, and a list of forms. The form list is the extraneous data described earlier

		Abstract Syntax	Concrete Syntax
str	$::=$	string	$\texttt{string}$
f	$::=$	$\mathsf{variableform}^3(f, str)$	$\texttt{VarForm}(\texttt{f}, \texttt{str})$
	$\mid$	$\mathsf{nonterminalform}^3(str)$	$\texttt{NonTermForm}(\texttt{str})$
	$\mid$	$\mathsf{terminalform}^3(str)$	$\texttt{TermForm}(\texttt{str})$
	$\mid$	$\mathsf{binderform}^3(f_1, \overline{f_2})$	$\texttt{BinderForm}(\texttt{f}_1, \overline{\texttt{f}_2})$
t	$::=$	$\mathsf{const}^3(s)$	$\texttt{Const}^3(\texttt{s})$
	$\mid$	$\mathsf{abst}^3(t_1, t_2)$	$\texttt{Abst}^3(\texttt{t}_1, \texttt{t}_2)$
j	$::=$	$\mathsf{judges}^3(str, \overline{j})$	$\texttt{Judges}^3(\texttt{str}, \overline{\texttt{j}})$
	$\mid$	$\mathsf{jnonterminal}^3(str)$	$\texttt{JNonTerm}^3(\texttt{str})$
	$\mid$	$\mathsf{jterminal}^3(str)$	$\texttt{JTerm}^3(\texttt{str})$
	$\mid$	$\mathsf{jbinder}^3(j_1, \overline{j_2})$	$\texttt{JBinder}^3(\texttt{j}_1, \overline{\texttt{j}_2})$
c	$::=$	$\mathsf{clause}^3(s, \overline{t}, \overline{f})$	$\texttt{Clause}^3(\texttt{s}, \overline{\texttt{t}}, \overline{\texttt{f}})$
r	$::=$	$\mathsf{rule}^3(str, \overline{j_1}, j_2)$	$\texttt{Rule}^3(\texttt{str}, \overline{\texttt{j}_1}, \texttt{j}_2)$
s	$::=$	$\mathsf{syntax}^3(t, \overline{c})$	$\texttt{Syntax}^3(\texttt{t}, \overline{\texttt{c}})$
ju	$::=$	$\mathsf{judgment}^3(str, t, \overline{r})$	$\texttt{Judgment}(\texttt{str}, \texttt{t}, \overline{\texttt{r}})$
p	$::=$	$\mathsf{program}^3(\overline{s}, \overline{j})$	$\texttt{Program}(\overline{\texttt{s}}, \overline{\texttt{j}})$

Figure 9: Summary of Internal Representation Three

$$\frac{e \notin \theta \quad \beta \propto \mathsf{true}}{\gamma; \mathsf{clause}(e) \rhd \mathsf{var}(\gamma, e)} \qquad \frac{}{\gamma; \mathsf{clause}(\overline{e}) \rhd \cdot}$$

$$\frac{\cdot; e \rhd \mathsf{syntax}(e) \quad e; \overline{c} \rhd \Sigma_c \quad \Sigma = \Sigma_c, \mathsf{syntax}(e)}{\cdot; \mathsf{syn}(e, \overline{c}) \rhd \Sigma}$$

$$\frac{\cdot; \overline{s} \rhd \Sigma}{\cdot; \mathsf{prog}(\overline{t}, \overline{s}) \rhd \Sigma}$$

Figure 10: Sigma Construction Rules

while the name is a dynamically generated name that can be used in the judgment section to refer back to to the syntax. In fact the form type represents how the clause was structured in the second representation and resembles the `element`^{2} structure. The reasoning for this is discussed in more detail in the transformation section. The actual change of interest is the creation of the term type. Terms are represented as either an abstracton or as constants and merely encode the eventual Twelf typing declarations. The abstraction allows us to derive an arrow type as required by Twelf type definitions while the const data structures makes up the basic individual elements of a term. For example if we examine the example Twelf code in Figure 2 `e2` would exist in the third representation as `Abst(Const(e),Abst(Const(e),Const(e)))` and `e3` consists of `Const(e)`.

3.3 Translation

This section will discuss the design and implementation of the transformation method that converts the input of the tool into the Twelf code format.

3.3.1 Sigma Construction

The rules for sigma construction are defined in Figure 10. The data structure definition has been omitted, however they can be simply be derived by observing the rule structure. The point of sigma construction is to determine what tokens defined in clauses are important in determining the correct Twelf representation and those that are purely syntactical sugar. To this end we build up a list of information which is basically a mapping of important tokens to a type. By observing the rules, the only tokens that need to be classified and stored are those of each nonterminal which is evident in the first premise of the sigma construction rule for syntax and associating variables with their respective nonterminal.

```
nat : type.
z : nat.
s : nat -> nat.
```

Figure 11: Twelf Representation of Natural Numbers

Once we have obtained a nonterminal type, then we search for variables defined in the associated derivation rules. These are defined as clauses which contain only one token, and we also store the nonterminal in which they occur for typing purposes later on in the translation process. An example of a variable associated with a nonterminal definition is that of x and f in the definitions for e that apperas in the sample input syntax. During the translation of the sample syntax e would be recognized and added to sigma, and both x and f would be added to sigma with the information of being binded to e. It is important to notice that the rule for associating variables with a nonterminal requires that the variable in question is not defined as a terminal in the terminal section of the input but also that it is binded in one of the derivation rules for the nonterminal. In the sigma construction rules θ represent the list of terminals and β is a list of variables with information declaring whether the variable was bound or not.

We can see this small difference used when we examine our example input versus the definition of natural numbers. The desired Twelf representation for natural numbers is defined in Figure 11 with the input syntax shown in Figure 3. In the natural numbers case, zero is understood to be a terminal, however in the example syntax input x satisfies the rules of sigma construction since it is bound to the lambda derivation rule and is not declared previously in the list of terminals. It is also important to notice that our tool does not attempt to correct mistakes made by the user where the user may mistakenly declare a variable as a terminal while using it in a manner befitting an associated variable in the rest of the input.

3.3.2 Translation Mechanism

The initial translation takes the sigma judgment which stores the typing for the elements of note, and then traverses through the program data structure replacing occurences with their necessary object in relation to the second data structure. An example of this is the declaration of each nonterminal itself. In the initial data structure these are represented as strings, however after the sigma construction, we can use the data entailed by sigma to change the first parameters to the syntax object with an more refined element type rather than simply a string. The rules for translation from the first representation to the second representation are shown in Figure 12. The translation of the judgment statements are much in the same vein

$$\frac{}{\Sigma, \mathsf{syntax}(s) \vdash \mathsf{variable}^1(s) \Rightarrow \mathsf{nonterminal}^2(s)} \quad \frac{}{\Sigma, \mathsf{var}(\gamma, s) \vdash \mathsf{variable}^1(s) \Rightarrow \mathsf{variable}^2(\gamma, e)}$$

$$\frac{\Sigma \vdash s_1 \Rightarrow s_1' \quad \Sigma \vdash s_2 \Rightarrow s_2'}{\Sigma \vdash \mathsf{binder}^1(s_1, s_2) \Rightarrow \mathsf{binder}^2(s_1', s_2')} \quad \frac{\Sigma \vdash s_1 \Rightarrow s_1' \quad \Sigma \vdash s_2 \Rightarrow s_2'}{\Sigma \vdash \mathsf{syntax}^1(s_1, s_2) \Rightarrow \mathsf{syntax}^2(s_1', s_2')}$$

$$\frac{\Sigma \vdash s_2 \Rightarrow s_2' \quad \Sigma \vdash s_3 \Rightarrow s_3'}{\Sigma \vdash \mathsf{program}^2(s_1, s_2, s_3) \Rightarrow \mathsf{program}^2(s_2', s_3')}$$

Figure 12: Partial Translation Rules 1 to 2

Data Structure Representation	Twelf Output
`Abst(Const(e), Abst(Const(e), Const(e)))`	`e -> e -> e.`
`Abst(Abst(Const(e), Const(e)), Const(e))`	`(e -> e) -> e.`

Figure 13: Demonstration of usage of Abstraction

as can be observed by the translation rules.

In the translation from the first internal representation to the second, we use sigma that was constructed according to the sigma construction rules. Using sigma we convert all of theh tokens such that all instances of a nonterminal becomes retyped from a nondescript variable type to a nonterminal type. Furthermore variables that are explicitly bound to some nonterminal become a variable type of the second representation which contain information on which nonterminal the variable belongs to. Furthermore these changes are for the most part mirrored in the judgment section. One change that is of notice in the judgment section, however, is where we have declarations of $e1$, $e2$, ... In these cases we keep the syntactical description of the variable as well determine the nonterminal of which they resolve to. For instance, the aforementioned tokens would resolve to nonterminals of e. To prevent ambiguity between syntax definitions and judgment definitions, we explicitly restrict numbered nonterminal usage in the syntax and reserve its usage in the judgment section. Nonterminals can also only be defined with standard alphabet characters such that to determine the nonterminal one only must observe the token until they reach the end of standard alphabet characters.

There are two main functions of the translation from the second representation to the third. The first takes part in the syntax. Since we have now correctly typed all the tokens with their relevant information in the syntax (as well as the judgments) we can now decide what tokens are relevant in construction for Twelf

$$\frac{}{\textsf{variable}^2(e, str) \sim \textsf{const}^3(e)} \quad \frac{}{\textsf{nonterminal}^2(str) \sim \textsf{const}^3(str)}$$

$$\frac{\overline{e_2} \models \textsf{nil}}{\textsf{binder}^2(e_1, \overline{e_2}) \sim \textsf{const}^3(n)} \quad \frac{\overline{e_2} \models (eh :: et) \quad eh : \textsf{variable}^3(e, x) \quad e \sim e' \quad et \sim et'}{\textsf{binder}^2(e_1, \overline{e_2}) \sim \textsf{abst}^3(e', eh')}$$

$$\frac{\overline{e} \models (eh :: et) \quad eh : \textsf{variable}^2(e_1, x) \quad et \sim et'}{\textsf{clause}^2(\overline{e}) \sim et'} \quad \frac{\overline{e} \models (eh :: et) \quad eh : \textsf{nonterminal}^2(x) \quad eh \sim eh' \quad et \sim et'}{\textsf{clause}^2(\overline{e}) \sim \textsf{abst}^3(eh', et')}$$

$$\frac{\overline{e} \models (eh :: et) \quad eh : \textsf{terminal}^2(x) \quad et \sim et'}{\textsf{clause}^2(\overline{e}) \sim et'} \quad \frac{\overline{e} \models (eh :: et) \quad eh : \textsf{binder}^2(e_1, \overline{e_2}) \quad \overline{e_2} \sim \overline{e_2}' \quad et \sim et'}{\textsf{clause}^2(\overline{e}) \sim \textsf{abst}^3(\overline{e_2'}, et')}$$

Figure 14: Rules for Constructing Terms from Clause^2

typing of each clause. What really happens here is we iterate through each element in the list of elements representing a clause and then strip out the terminals from the list. We in fact keep the entire element list from the second internal representation in our form list which consists of the third parameter of clause type. After this preparation has been done to the syntax, the element list with the terminals stripped is converted into a term. The rules for these are shown in Figure 14. In the rules the $\models$ symbol is used to define the breakup of lists which we split into the first element, the head, and the rest of the list. We use the ML notation to represent lists. The $\sim$ operator represents the conversion from clauses of the second representation to the term representation. Twelf is right associative, therefore the order mechanics of the parameters in `Abst` are detailed in Figure 13. If we wish to explicitly evaluate some sort of precedence than Twelf requires that we surround this precedence in parentheses, otherwise it will always assume right hand precedence.

Since judgments are now typed as well, the system must now figure out what expressions each set of tokens belong to. For example in the reduce judgment `e -> e` we know that the rule statement must have an expression of type `e` on either side of the the arrow operator, which the system interprets as a terminal token, to be a correctly defined reduce judgment. Recall that the new form of the clause type for the third representation now have a name binded to it. We can now have this since the syntax has already been translated by the time the system attempts to translate the judgment section. This is where we utilize the forms parameter of the clause type. We use both the forms to attempt to find the matching clause for the rule we are currently translating. The most important methods to note is that the typing should match some clause in the syntax, however this is not the only requirement since two expressions could in theory have the same Twelf typing. For example, if we defined two derivation rules such that one defined a right subtree and one defined a left subtree, these would have the same typing. The only way that our tool can tell the difference is by also comparing the nonterminals and making sure that they are string equivalent.

The final translation to Twelf output is simple after the data is represented by the third form of the data structures. We have already shown how the twelf translation takes place for the syntax with the usage of terms in the third data structure representation. Judgments are somewhat more complicated but still relatively simple after the conversion to the third data structure is complete. The important information has constructed which is finding the matching clause in the syntax. This corresponds to `e2` in `reduce (e2 E1 E2)` where reduce is the name of the rule itself. Since rules are constructed such that there is a list of rules functioning as premises and a second rule function as the conclusion, the output to Twelf is the traversal of the list of premises by first outputting the name of the rule followed by the matching clauses as expressed above. The conlclusion is stated last with each premise and conclusion separated by an Twelf arrow operator.

3.4 Technical Information

We used parser and lexer generators for creating the first data structure from the input syntax. Since the entire tool was written in ML, the parser and lexer generators of choice were ML-Yacc and ML-Lex respectively. The entire project was developed in the linux environment and compiled with Standard ML v110.59.

4 Analysis

For our major test case, we tested the lambda calculus for which the input is available as Figure 4. In this test case we were also able to define a judgment that determined when a given expression was able to evaluate to a value, and a reduce judgment that would allow for progress in the defined program semantics. If a text file with this exact input is ran in our tool, the exact output is shown in Figure 2. This exact output when run in Twelf will result in a successful Twelf compilation. We have shown that we can at least model one basic derivation of program semantics in our tool, however it should be possible to model a wide variety of simple and possibly more complicated languages. Further more, it is interesting to note that the tool in some way is self-describing. It should be possible theoretically that the tool would be able to recursively model itself and prove theorems about itself by following the syntax and semantic definitions that are summarized in this document.

Originally we were aiming to include the ability to prove theorems about the program semantics that one could model in our tool, however, we realized that this aspect would take just as much time as all the work we had done before, therefore we aimed to make the modeling aspect as solid as possible. Since the project was also a way to test if a tool based on Twelf that was easier to use coul be developed, we have laid the foundations for future work in such endeavors. Hopefully, in the future our project can be extended with work that is detailed int he future work section. Regardless, as a prototype and excursion project, we believe we have learned a a great deal in what is necessary in creating a good education tool that will allow rigorous reasoning about programs in relation to program semantics.

As stated in the introduction, we had three main goals that we wanted to accomplish with the tool. First, we wanted to make sure that the tool was easy to use. This requires that the input be simple to learn for the target user. Since we have modeled the input syntax on class room taught syntax, this part should be easily adaptable for students. Furthermore, the judgment section is modeled such that it resembles inference rule form. This inference rule form is how the rules for static and dynamic semantics are taught in class at Carnegie-Mellon University. We believe that the input language we designed successfully accomplishes this goal since it is modeled on familiar concepts.

Second, we believe that this tool satisfies the need to be immediately recognizable and relevant to students of programming language theory. The tool so far allows users to model languages and then check to see if the syntax and semantics of the language are compatible with each other. We do run into a problem in achieving our final goal of the system: giving good feedback to the user. Currently the error messages a user could possibly get is hard coded into the conversion process when a user attempts to convert his input syntax into Twelf. Ideally errors should both be related to finding errors in the syntax as well as possibly suggesting ways to fix the error. Meaningful error messages are a problem in perhaps every software package known to man and this is a very difficult problem to solve. Currently the system will only give internal typing errors which violate the semantics that we declared in the data structure specification section however we have no implemented any mechanism for catching errors and reporting them to users at the Twelf level, thus although the translation process may be complete, the user would not know that there is an error until attempted to execute the outputted code in Twelf. Error messages in Twelf are further discussed in the future work section.

5 Future Work

First of all, a solid system for impelementing features such that the user can input proofs would be the first step towards improving the tool. This is really a necessary aspect of the tool before it would truly be useful to an education setting. As it stands currently, it may be useful to users to learn how programming semantics may be defined, however, without the ability to prove and show how the programming semantics could execute, for instance proving theorems about the progress and preservation of the language, we are not able to assist the student in a large aspect of the introductory curriculum to programming language theory.

The proof system should simply allow for a natural way to encode the necessary information that Twelf requires for a proof in a natural language or simple list type. A good structure of how a proof maybe laid out may be a branching method such as how Tutch lays out its proofs for logic. For example, proofs such as progress which can have multiple cases for a rule could then be organized in a branching structure that resembles a tree, with each case representing a branch. The actual proof for each case could be the list of steps required. You could tell the input that you use apply the inducton hypothesis, then by the inversion lemma, etc etc... One can visualize each mini-proof such that it is a chain of requirements that lead to the proof.

A critical feature that should be addressed after a system for proofs has been implemented is that of better error messages from Twelf. Currently, error messages for Twelf are somewhat mystic even for the most experienced Twelf sages. This extension would probably require actually working on Twelf itself such that the error message system for when Twelf fails to interpret inputte Twelf code is more understandable. The tool should eventually be able to execute Twelf in the background without requiring the user to manually input what he has into Twelf himself, and also retrieve error messages internally from Twelf seamlessly.

Another feature that would be beneficial would be a visual editor that could support drag and drop interfaces for declaring input. In actuality, we realize that it maybe cumbersome and somewhat unnatural, especially for rules, to actually input rules simply by hand. Having a visual input application would in theory, at least in a interaction standpoint, allow for more efficient usage of the tool. Furthermore, with a visual editor we could expand the project to tutorial applications where one might envision preseting a student with partial derivations of a proof where then a student could enter the missing segments. The error message aspect would help the tutorial aspect out in cases where the user could be wrong in what he believes should go into the missing sections of the proof, and with the help of the error messages, the tool could perhaps suggest ways in which the user could correct his proof.

6 Conclusion

We have developed the basis for a tool of which we believe will help students in the future better understand and learn programming language theory. We have discussed the technical design and implementation including particularly troublesome aspects of creating such a tool as well as discussing our solutions to these problems. Although all the intial goals of the project were not fully accomplished, a good foundation for extending the work has been created and hopefully that future work that will truly make this tool useful to the undergraduate education environment

7 References

1. A. Abel, B.-Y. E. Chang, and P. Pfenning. Human-readable, machine-verifiable proofs for teaching constructive logic. In *Proceedings of the Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*, Siena, Italy, June 2001.

2. F. Pfenning and C. Schurmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202-206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

3. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. In *A Journal of the Association for Computing Machinery (40-1)*, pages 143-184, January 1993.