JaLoF: A Development Environment for Deduction Systems

Roderick Moten
Colgate Univesity

Extended Abstract

1 Introduction

We may implement an automated deduction system for a logic using a general purpose programming language or a logical framework. With a general purpose programming language, such as Lisp or Standard ML, we may create an automated deduction system as a collection of components. The primary component is responsible for performing reasoning within the logic. The other components, such as editors, pretty printers, and tables for storing theorems, provide services that make the system user–friendly. Alternatively, we may implement an automated deduction system for a logic by encoding it as an object logic of a logical framework, such as Twelf [10, 9] or Isabelle [8]. With this approach, we implement a deduction system by inherting all the components of the logical framework. Clearly, implementing an automated deduction system using a logical framework is easier than implementing it using a general purpose language. However, an automated deduction system implemented with a general purpose language may be easier to use than one implemented with a logical framework. The former only requires the user to know how to reason within the object logic. The latter requires the user to know how to reason within the base logic and to understand the encoding of the object logic in the base logic. As a result, we decided to create a program development environment, called the Java Logical Framework (JaLoF), that supports rapid development of deduction systems in which users reason directly within the object logic. JaLoF is currently under development, and in this extended abstract, we describe is current status.

JaLoF consists of an abstract syntax for representing constructs of an object language, a language for specifying the structured operational semantics of an object language, and a language for specifying a proof system for reasoning about entities of the object language. JaLoF is implemented in Java as an extendable command line interpreter. An object logic is encoded into JaLoF by defining a collection of classes that implement methods for constructing terms and assertions of the object logic, evaluating terms of the object logic, and performing inference on assertions of the object logic.

Our motivation for creating JaLoF comes from discovering that the Nuprl Proof Development System (Nuprl PDS) [2] is implemented as a program development environment for building reasoning system for the Nuprl type theory [1]. With the Nuprl PDS, we can develop reasoning systems for logics without encoding them into the Nuprl type theory. We discovered this feature of the Nuprl PDS when re-implementing Nuprl to exploit parallelism on a shared memory multiprocessor [7, 6]. Jason Hickey has recently developed an architecture for the Nuprl PDS, MetaPRL [4], to exploit the independence of the Nuprl PDS from the Nuprl Type Theory.

The Nuprl PDS contains an untyped abstract syntax, the Nuprl term language, that is used to encode the Nuprl type theory. Although the Nuprl PDS only defines operational semantics rules for terms of the Nuprl type theory, we can extend the semantics by defining new rules for terms independently of existing rules. We can encode an object language by extending the semantics of the Nuprl term language to include Nuprl terms representing constructs of an object language. Unlike HOAS [3], this approach supports evaluation for an object language using structural induction. However, the Nuprl term language contains features that we believe make it inadequate for our needs.

Nuprl defines two distinct functions to perform first order substitution and second order substitution. First order substitution replaces a first order variable with a Nuprl term. Second order substitution replaces a second order variable instance with a non-Nuprl term representing an abstraction. A second order variable instance in Nuprl corresponds to a λ-term in which a variable is applied to a term, for example $(f\, a)$. In Nuprl, replacing a second order variable instance consists of replacing the variable with an abstraction and then performing β-reduction on the resulting term. Using Nuprl's second order substitution in the λ-calculus, for example, to replace f with $\lambda x.(s\, x)$ in $(f\, a)$ produces $(s\, a)$. To define a single function for substitution, we change the binding structure of the Nuprl term language. In Nuprl, bindings are placed on subterms. For instance, $\lambda x.t$ is represented as the Nuprl term $\mathsf{lambda}(x.t)$. This term specifies that x only binds t because it appears as a subterm of $\mathsf{lambda}(x.t)$. In other words, $\mathsf{lambda}(x.t)$ is obtained by placing t in the hole in $\mathsf{lambda}(x.\Box)$.

The Nuprl term language designates a specific class of terms as variables. Substitution is defined with respect to this class of terms. Therefore, an object language must encode its variables as Nuprl's variables to use Nuprl's substitution. This requirement eliminates encoding object languages whose variables cannot be represented as Nuprl variables, for example variables that are tagged with a type. Encoding these languages into the Nuprl term language requires defining substitution for each of them.

JaLoF overcomes these problems of the Nuprl term language by defining the JaLoF abstract syntax (JAS). We created JAS by modifing the bindings structure of the Nuprl term language to support abstractions. In other words, we changed the binding structure so that the bindings of a term occur over the entire term and not only the subterms. Therefore, we can define a single function for substitution of first order and higher order variables. Futhermore, JAS has no terms designated as variables, but defines substitution to operate

on any class of terms designated as variables. Therefore, an object language can use JAS's substitution regardless of the class terms it uses as variables. To define the operational semantics of terms, JaLoF contains a language dedicated for specifying operational semantics. In addition, we are currently designing a language for specifying inference rules of an object logic. This language resembles the language for specifying the operational semantics except it attempts to model natural language specifications of inference rules. This language allows users to specify inference rules intended for constructing either top-down or bottom-up proofs.

We organize this extended abstract as follows. In Section 2 we define JAS. In Section 3 we describe the language for specifying evaluation rules by encoding the typed λ-calculus into JAS and then encoding its type system as evaluation rules. In Section 4 we give a brief overview of the language for specifying inference rules. In Section 5, we briefly describe the current status of the development of JaLoF.

2 JAS: The JaLoF Abstract Syntax

The JaLoF abstract syntax, JAS, is an untyped language and supports higher order substitution and a restricted form of higher order matching. We begin defining JAS by assuming that we have a set of countably infinite identifiers $\mathcal{I}$. We use identifiers to create *operators*. An operator is an identifier paired with a list of zero or more identifiers. In other words, if o is an identifier and $p_1, \ldots, p_n$, for $n \geq 0$, are identifiers, then $o\{p_1, \ldots, p_n\}$ is an operator: o is the *operator identifier* and each p_i is a *parameter*. Parameters provide the ability to inject objects from a model, such as a model for integers, into the abstract syntax. We use identifiers to represent a parameter.

We use operators to construct terms. A term is an object of the form $x_1, \ldots, x_m.a(t_1, \ldots, t_n)$ where

- a is an operator,
- $m \geq 0$ and each x_i is an operator with at least one parameter, and
- $n \geq 0$ and each t_i is a term.

A term is an abstraction if $m > 0$.

We require that each operator used as a binding have at least one parameter, so that each binding will have a name. Therefore, we may represent $\lambda x.(sx)$ in JAS as

$$\mathsf{var\{x\}.lambda(app(var\{}s\mathsf{\}(), var\{}x\mathsf{\}()))}. \tag{1}$$

In (1), the term $\mathsf{var\{}x\mathsf{\}()}$ represents the lambda calculus variable x. JAS does not designate any term as a variable because some object languages may represent variables differently. For example, for a typed language, variables may be tagged with a type. Therefore, we may represent variables in a typed

language as $\mathsf{var}\{x, T\}()$. In an untyped language, however, we may represent variables as $\mathsf{var}\{x\}()$.

We use a *term signature* to identify a particular class of terms as variables. A term signature of a term $x_1, \ldots, x_r.o\{p_1, \ldots, p_n\}(t_1, \ldots, t_m)$ is the triple $\langle o, n, r\rangle$. Variables of an object language must have a term signature where the number of parameters is greater than zero.

A variable is higher order if it has subterms. For example,

$$\mathsf{var\{f\}(var\{x\}(), var\{y\}())}$$

and

$$\mathsf{var\{g\}(var\{x\}.var\{x\}())}$$

are higher order occurrences. A higher order variable occurrence corresponds to application of a variable to one or more terms in the λ-calculus. Therefore, the higher order occurrences above represent the applications $f(x, y)$ and $g(\lambda x.x)$ in the λ-calculus.

In JAS, we substitute for free first and higher order variables. A variable term with signature s is free if it does not occur in the scope of a binding equal to its operator. Two operators are equal if they have the same operator identifier and their parameters are pairwise equal. We use $\mathrm{FV}_s(t)$ to denote the set of operators of free variables in t with signature s. A variable term x of signature s is bound in t if x is a subterm of t and x's operator is not a member of $\mathrm{FV}_s(t)$. We define substitution for JAS below.

Definition 2.1 Let t be a term and s a signature of variables. Let $t_1, \ldots, t_n$ be terms in which no binding or variable of signature s in t occurs in any t_i. Let $x_1, \ldots, x_n$ be variables with signature s. Let $\sigma = [t_1/x_1, \ldots, t_n/x_n]$. Then σt is the term defined inductively as follows.

1. Suppose $t = o(t'_1, \ldots, t'_m)$. If $m = 0$ then $\sigma t = t_i$ if $t = x_i$; otherwise $\sigma t = t$. If $m > 0$ then if $o = x_i$ and $t_i = y_1, \ldots, y_m.t''$, then $\sigma t = [\sigma t'_1/y_1, \ldots, \sigma t'_m/y_m]t''$; otherwise $\sigma t = o(\sigma t'_1, \ldots, \sigma t'_m)$.
2. If $t = y_1, \ldots, y_n.t'$ then $\sigma t = \sigma' t'$ where σ' is σ with t_i/x_i removed if $y_i = x_i$.

3 Specifying Evaluation

To demonstrate how to specify evaluation rules for an object logic, we encode the simply typed lambda calculus with natural numbers and addition in JAS. There are three ways we can encode the natural numbers in JAS. One way is to use a unary constructor, succ, and a single constant, 0. This approach leads to a simple recursive definition of evaluation for addition. From a pragmatic point of view, however, this encoding is inefficient because it represents integers as lists. A more efficient encoding represents each natural number distinctively as a constant. However, to define evaluation for addition requires a separate rule

Terms:

n	$\mathsf{nat}\{\widehat{n}\}()$
v^T	$\mathsf{var}\{v, \widehat{T}\}()$
$a + b$	$\mathsf{add}(a, b)$
$f(x)$	$\mathsf{app}(f, x)$
$\lambda x^T.t$	$\mathsf{var}\{x, \widehat{T}\}.\mathsf{lambda}(t)$

Types:

$\mathcal{N}$	$\mathsf{nat}()$
$T \to T'$	$\mathsf{arrow}(T, T')$

Judgments:

t has type T $\quad \mathsf{typeof}(t) \Downarrow T$

Figure 1: Encoding of λ-calculus in JAS

for each pair of constants. For example, we would have to define separate rules for $\mathrm{add}(3, 4)$ and $\mathrm{add}(30, 40)$.

The third encoding uses only one evaluation rule for addition without representing terms as lists. This encoding uses partial functions to map identifiers to and from elements of a model of the natural numbers. We define a model as a semantic algebra M and the partial functions $\Phi_D : D \to \mathcal{I}$ and $\Psi_D : \mathcal{I} \to D$ for each domain D in M. A model also contains a function Ω_D for each domain D that maps elements of D to JAS terms. Each function Φ_D, Ψ_D, and Ω_D is one-to-one. Furthermore, for each $d \in D$, $\Psi_D(\Phi_D(d)) = d$. For the remainder of this abstract, we write $\Phi_D(d)$ as $\widehat{d}$ where $d \in D$.

$$
\begin{array}{rl}
\mathsf{nat}\{n\}() : & \longrightarrow \mathsf{nat}\{n\}() \\
\mathsf{add}(t, t') : & t \Downarrow \mathsf{nat}\{n\}() \;\&\; t' \Downarrow \mathsf{nat}\{n'\}() \longrightarrow \mathsf{nat}\{\Phi_{\mathcal{N}}\big(\Psi_{\mathcal{N}}(n) + \Psi_{\mathcal{N}}(n')\big)\}() \\
\mathsf{var}\{v, T\}() : & \longrightarrow \mathsf{var}\{v, T\}() \\
\mathsf{app}(f, x) : & f \Downarrow \mathsf{var}\{y, T\}.\mathsf{lambda}(t) \;\&\; x \Downarrow x' \longrightarrow t[x'] \\
\mathsf{var}\{x, T\}.\mathsf{lambda}(t) : & \longrightarrow \mathsf{var}\{x, T\}.\mathsf{lambda}(t)
\end{array}
$$

Figure 2: Reduction Rules of λ calculus in JAS

We give an encoding of the typed λ-calculus with natural numbers based on a model of natural numbers and a model of type tags in Figure 1. The model of natural numbers contains the domain $\mathcal{N}$ and operator $+ : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$. The model of type tags contains the domain $\mathcal{P}$ of strings created with the alphabet $\{\mathtt{n}, \mathtt{->}\}$. The column on the right in Figure 1 contains the JAS version of the lambda calculus object on the left. We encode the judgment t has type T as an evaluation rule. In particular, we say that t has type T, if $\mathsf{typeof}(t)$ evaluates to

$$
\begin{aligned}
\mathsf{typeof}(\mathsf{nat}\{n\}()) : &\quad \longrightarrow \mathsf{nat}() \\
\mathsf{typeof}(\mathsf{add}\{t,t'\}()) : &\quad \mathsf{typeof}(t) \Downarrow \mathsf{nat}() \ \&\ \mathsf{typeof}(t') \Downarrow \mathsf{nat}() \ \longrightarrow \mathsf{nat}() \\
\mathsf{typeof}(\mathsf{var}\{v,T\}()) : &\quad \longrightarrow (\Omega_{\mathcal{P}} \circ \Psi_{\mathcal{P}})(T) \\
\mathsf{typeof}(\mathsf{app}(f,x)) : &\quad \mathsf{typeof}(f) \Downarrow \mathsf{arrow}(A,B) \ \&\ \mathsf{typeof}(x) \Downarrow A \ \longrightarrow B \\
\mathsf{typeof}(\mathsf{var}\{x,T\}.\mathsf{lambda}(t)) : & \\
& \mathsf{typeof}(\mathsf{var}\{x,T\}()) \Downarrow A \ \&\ \mathsf{typeof}(t) \Downarrow B \ \longrightarrow \mathsf{arrow}(A,B)
\end{aligned}
$$

Figure 3: Typing Rules of λ calculus in JAS

T.

We define the evaluation rules of our encoding of the λ-calculus in Figure 2 using a specification language. We omit the details of the language here, but describe components of the language as needed. The general form of an evaluation rule is t : clauses $\longrightarrow$ t'. Intuitively, an evaluation rule states that t evaluates to t' if all the clauses are true. The clause $r \Downarrow r'$ is true if a term that matches r evaluates to a term that matches r'. The expression $t[x']$ represents substituting x' for the first binding of t. Notice that the evaluation rule for addition uses addition from the model of natural numbers. More specifically, the evaluation rule converts the identifers n and n' into natural numbers, adds them, and converts the result into an identifer.

We specify the typing rules also as evaluation rules. We give the typing rules in Figure 3. The typing rule for variables uses the function $\Omega_{\mathcal{P}} \circ \Psi_{\mathcal{P}}$ to convert the type tag on a variable into an actual term. For example, $\Omega_{\mathcal{P}} \circ \Psi_{\mathcal{P}}$ converts the identifer n–>n into the term arrow(nat(), nat()). The typing rule for abstraction makes the term var{x, n}.lambda(var{x, n–>n}()) typeable. This term should be typeable because var{x, n}() and var{x, n–>n}() are not equal.

Our encoding of the λ-calculus is not adequate because there are JAS terms, such as var{x, n}.lambda(zoo()), that are not λ-terms. Although we expected the encoding to be inadequate, our reduction rules do not guarantee that they will not produce values for invalid λ-terms. For example, the reduction rule for abstraction generates a value for var{x, n}.lambda(zoo()). Likewise the typing rule for nat{n}() makes invalid λ terms typeable. For example, suppose $\Psi_{\mathcal{N}}$ is undefined on the identifier -3. Then nat{-3}() is not a λ term, but it is typeable. Fortunately, we can decide whether a JAS term is a λ-term. More specifically, we can determine whether a JAS term t is a λ-term if the term islamb(t) has a value, namely tt(). We use the following evaluation rules to produce a value for islamb(t).

$$
\begin{aligned}
\mathsf{islamb}(\mathsf{nat}\{n\}()) : &\quad (\Omega_{\mathcal{N}} \circ \Psi_{\mathcal{N}})(n) \Downarrow x \ \longrightarrow \mathsf{tt}() \\
\mathsf{islamb}(\mathsf{add}(t,t')) : &\quad \mathsf{islamb}(t) \Downarrow \mathsf{tt}() \ \&\ \mathsf{islamb}(t') \Downarrow \mathsf{tt}() \ \longrightarrow \mathsf{tt}() \\
\mathsf{islamb}(\mathsf{var}\{v,T\}()) : &\quad (\Omega_{\mathcal{P}} \circ \Psi_{\mathcal{P}})(T) \Downarrow x \ \longrightarrow \mathsf{tt}() \\
\mathsf{islamb}(\mathsf{app}(f,x)) : &\quad \mathsf{islamb}(f) \Downarrow \mathsf{tt}() \ \&\ \mathsf{islamb}(x) \Downarrow \mathsf{tt}() \ \longrightarrow \mathsf{tt}() \\
\mathsf{islamb}(\mathsf{var}\{x,T\}.\mathsf{lambda}(t)) : &\quad \mathsf{islamb}(\mathsf{var}\{v,T\}()) \Downarrow \mathsf{tt}() \ \&\ \mathsf{islamb}(t) \Downarrow \mathsf{tt}() \ \longrightarrow \mathsf{tt}()
\end{aligned}
$$

The use of the model functions in the evaluation rules for islamb(nat{n}())

and $\mathsf{islamb}(\mathsf{var}\{v, T\}())$ determine whether the parameters n and T are valid representations of elements of $\mathcal{N}$ and $\mathcal{P}$, respectively. Recall that $\Omega_{\mathcal{N}} \circ \Psi_{\mathcal{N}}$ and $\Omega_{\mathcal{P}} \circ \Psi_{\mathcal{P}}$ are partial functions. Therefore, they will only return a value if the parameter represents an element in $\mathcal{N}$ or $\mathcal{P}$.

4 Specifying Inference Rules

In addition to a language for specifying the operational semantics of an object language, JaLoF also contains a language for specifying inference rules of a proof system. Currently, this language is still under development. However, we intend the language to be able to adequately define proof rules such as the following proof rule taken verbatim from [5].

> The sequent $P \longrightarrow \forall_\tau x.B$ is provable if and only if the sequent $P \longrightarrow B[y/x]$ is provable, where y is some (eigen)variable that does not occur free in P or in $\forall_\tau x.B$.

Although, JaLoF's specification language for defining inference rules is under construction, it will meet the following requirements.

1. Support specification of rules for top-down and bottom-up proofs.
2. Support matching against JAS terms and parameters within JAS terms.
3. Provide a construct for term evaluation.
4. Provide a construct to invoke model functions.
5. Provide a construct for referring to free variables of a term.
6. Provide a construct for stating a proviso.
7. Provide a construct for stating that a term belongs to a class of terms.
8. Provide a construct for substitution.
9. Provide a construct to express that an inference rule can contain an arbitrary number of antecedents.

5 Implementation

We have begun implementing a prototype of JaLoF in Java. In addition to being widely availiable and multi-threaded, we choose Java because of its ability to dynamically load classes. This feature makes it easier to extend our system with new terms, evaluation rules, inference rules, and models.

Our prototype of JaLoF is a command line interpreter that contains four separate environments that map names to commands, *term families*, terms,

and models.[1] Each command is represented as an instance of a subclass of the class `Command`. When a user types in a command, the runtime system obtains the object corresponding to the command from the command environment. Afterwards, the runtime system executes the command invoking the command object's `execute` method on the command line arguments. A user may define a new command by creating a subclass of `Command` and loading it into JaLoF. JaLoF has a built–in command that loads a command subclass into JaLoF, creates an instance of the subclass, and stores the instance in the command environment.

JaLoF also contains a built–in command for loading term family classes. A term family represents the set of terms that have the same term signature. Recall from Section 2 that a term signature is the operator identifier of a term, the number of parameters that occur in its operator, and the number of bindings. A class representing a term family contains methods to create and evaluate terms belonging to the term family. A term family is defined by a user by creating a subclass of `TermFamily` that overrides the `instanceOf` and `eval` methods of `TermFamily` . The `instanceOf` method creates a term in the term family and the `eval` evaluates a term in the term family. We intend to develop a GUI that will assist a user to create a term family class instead of explicitly programming it in Java. Using the GUI, a user will create a term family by completing a form. The GUI will automatically generate Java code for the `instanceOf` method based upon the user's input. Also, the GUI will implement the language for specifying evaluation rules described in Section 3. The GUI will automatically convert specifications of the evaluation rules into Java code for the `eval` method.

Users create terms using the `def-term` built–in command. The arguments of this command is a name and the concrete syntax of a term. From the concrete syntax, the `def-term` command determines the signature of the term and uses it to invoke the `instanceOf` method of the term family with the same signature. The term produced by the `instanceOf` method is stored in the term environment.

Models are represented as instances of subclasses of the class `Model`. In our implementation, each model class contains only one domain and three methods representing the marshalling functions Φ, Ψ, and Ω. To create a model, a user develops a subclass of `Model` and loads it into JaLoF using the built–in command for loading model classes. When a model class is loaded into JaLoF an instance of the class is created and stored in the model environment. Once a model object is stored in the model environment, the `eval` method of a term family class can access any of its public methods.

6 Conclusion

We are currenlty developing JaLoF, a program development environment implemented in Java for rapid development of deduction systems. Unlike most logical frameworks, deduction systems created with JaLoF will allow users to reason

[1] We do not have an environment for proof objects because we have not determined how we will represent them in JaLoF.

directly within the object logic. Our motivation for creating JaLoF comes from discovering that the Nuprl Proof Development System is implemented as a program development environment for building reasoning system for the Nuprl type theory. We believe that Nuprl is inadequate as a general purpose development environment for automated deduction systems for the following reasons. Nuprl defines two separate functions for performing substitution, designates a specific class of variables as terms, and does not provide users with the capability to extend the semantics of the Nuprl term language. JaLoF overcomes these problems by creating a modification of the Nuprl term language called JAS. The binding structure of JAS permits the definition of a single function for substitution of first order and higher order variables. Futhermore, JAS defines substitution to operate over any class of terms that a user designates as variables of an object langauge. JaLoF also allows users to extend the operational semantics of JAS with evaluation rules for constructs of an object langauge. JaLoF utitlizes Java's ability to load classes dynamically to make it extendable. An object logic is defined as a collection of classes which are loaded dynamically into JaLoF. These classes are used to create Java objects that JaLoF uses for reasoning within the object logic.

7 Acknowledgement

I thank Jesus Christ for giving me the ability to pursue this work.

References

[1] Robert Constable. The Structure of Nuprl's Type Theory . http://www.cs.cornell.edu/Info/Projects/NuPrl/documents/Constable/1stfile.ps, 1997.

[2] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, NJ, 1986.

[3] Joëlle Despeyroux, Frank Pfenning, and Carsten Schrmann. Primitive recursion for higher-order abstract syntax. In *Proceedings of the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, number 1210 in Lecture Notes in Computer Science, pages 147–163. Springer-Verlag, April 1997.

[4] Jason J. Hickey. Nuprl-Light: An implementation framework for higher-order logics. In *Logic and Computer Science*. Springer Verlag, 1997.

[5] Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*, pages 434 – 445, 1997.

[6] Roderick Moten. *Concurrent Refinement in Nuprl*. PhD thesis, Cornell University, 1997.

[7] Roderick Moten. Exploiting parallelism in interactive theorem provers. In *Proceedings of the Eleventh International Conference, TPHOLs 98*, number 1479 in Lecture Notes In Computer Science, pages 315–330, 1998.

[8] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1994.

[9] Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, number 1632 in Lecture Notes in Artificial Intelligence, pages 202–206. Springer-Verlag, July 1999.

[10] Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, number 1421 in Lecture Notes in Artificial Intelligence, pages 286–300. Springer-Verlag, July 1998.