Towards a Foundation of an Automated Prover for Expressive Logic

Philippe Suter

February 5, 2009

Introduction

Software verification systems performing static analyses almost invariably rely on external theorem provers or decision procedures to prove the verification conditions they generate. These solvers tend to be specialized for certain theories or logics, forcing verification system designers to choose between two unsatisfactory options: The first one is to restrict oneself to a logic handleable by a solver and to renounce constructs which fall outside of its scope. As a consequence, the usability of the system is lowered, either because the properties it can verify are less expressive or because they need to be encoded into the simpler logic, introducing significant overhead. The second option to circumvent the limitations of individual solvers is to use several of them. This is a difficult task, as verification conditions need to be split into fragments that match the various supported logics and discharged accordingly. The combination of logics is in fact a wide and ongoing research subject.

Among the goals of the VEPAR project is to provide a unified interface to powerful reasoning procedures and automate as much as possible the decomposition of formulas in an expressive logic into formulas in decidable fragments. The implementation of VEPAR started with this semester project. The work done is described in the following sections, as well as some directions for future work.

Overview of the system

Figure 1 shows an overview of the projected system. The parts in grey were implemented as part of this semester project and are described in the following sections.

The general structure is similar to the formDecider component from the Jahob system [8]: formulas in a higher-order logic can be sent to higher-order theorem provers[1] or coerced into first-order logic and subsequently sent to first-order

[1] Note that in formDecider the Isabelle proof assistant [10] can be used either interactively or automatically via the `auto` tactic. Our plans for VEPAR are similar.

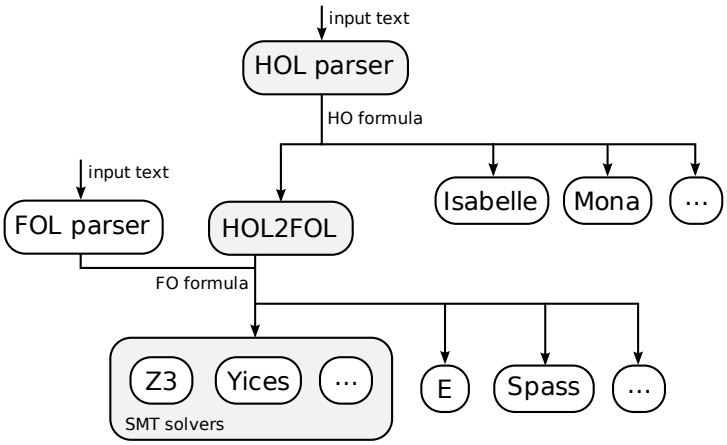

Figure 1: Overview of the VEPAR system. Parts in light grey were implemented during this project.

theorem provers or SMT solvers. All components are written in Scala [12], with the exception of the HOL parser which is built with tools originally designed for Java [7, 5].

Higher-order logic

We use simply-typed lambda calculus to represent formulas in higher-order logic, as in [1]. This approach presents the advantage of simplicity: only four elementary building blocks are used, and all other constructs one expects to find in a logic are encoded using constants to which we attach specific semantics. Figure 2 shows the abstract syntax of our logic, which we simply call "HOL".

As an example, consider the following formula:

$$\forall S_0.\forall S_1.(S_0 \setminus S_1 = \emptyset) \rightarrow (S_0 \subseteq S_1)$$

In HOL, this would be encoded as:

$$(\forall\ (\lambda S_0.\forall\ (\lambda S_1.\ \rightarrow\ ((\setminus\ S_0\ S_1) = \emptyset)\ (\subseteq\ \ S_0\ S_1))))$$

... where $\forall$, $\rightarrow$, $\setminus$, $\emptyset$ and $\subseteq$ are all constants. For instance, $\rightarrow$ has the type $\mathbb{B} \Rightarrow \mathbb{B} \Rightarrow \mathbb{B}$ and can be read as "a function that returns whether its first argument implies its second". The type of $\emptyset$ is harder to define because it depends on the context. If it represents an empty set of integers, its type will be $\mathbb{Z} \Rightarrow \mathbb{B}$, for instance. In other words, $\emptyset$ has the polymorphic type $\alpha \Rightarrow \mathbb{B}$, for some α. Similarly, the constant $\forall$ has the type $(\alpha \Rightarrow \mathbb{B}) \Rightarrow \mathbb{B}$ and we attach to it the semantics of "a function that returns whether a predicate always holds".

In terms of implementation, all formulas are stored as algebraic data types (or *case classes*, in Scala terminology) representing the basic building blocks.

$$
\begin{array}{rcll}
f & ::= & \lambda x.f & \text{lambda abstraction} \\
 & | & f_1\ f_2 & \text{function application} \\
 & | & f_1 = f_2 & \text{equality} \\
 & | & x & \text{variable or constant} \\
 & | & f :: t & \text{typed formula} \\
 & & & \\
t & ::= & \mathbb{B} & \text{booleans} \\
 & | & \mathbb{Z} & \text{integers} \\
 & | & \mathbb{O} & \text{uninterpreted objects} \\
 & | & t_1 \Rightarrow t_2 & \text{total functions} \\
 & | & t\ \mathsf{list} & \text{lists}
\end{array}
$$

Figure 2: Abstract syntax of formulas and types in HOL. Note that sets of elements of type t are described by their characteristic function of type $t \Rightarrow \mathbb{B}$, and multisets by their multiplicity function $t \Rightarrow \mathbb{Z}$.

Pattern-matching and reasoning in general on these formulas is made easier by the use of extractors [4] which provide different views on the same data. Thus, when pretty-printing formulas, for example, we can use extractors to match on, say, quantifiers without having to deal with their encoding, while computing the set of free variables will be easier to do by matching on the terms in their lambda-calculus form.

Concrete syntax

Needless to say, reading and writing formulas in the prefix or abstract form is unpleasant at best, and we use the conventional infix operators as often as possible. We use a (very small) subset of the Isabelle formula language as our default syntax for HOL. The concrete grammar is depicted in Figure 3.

Our parser uses JFlex [7] for the lexical analysis and the LALR Parser Generator CUP [5] for the syntactical analysis. These tools generate Java classes from which we call Scala factory functions to build the trees.

Formula editor

The Isabelle-like concrete syntax can be considered user unfriendly. `\<subseteq>` is a poor substitute for $\subseteq$, for instance, making large formulas look little like what one would write in math or logic. To palliate this inconvenience, we wrote a small formula editor which automatically substitutes Unicode equivalents to the ASCII sequences used to represent math symbols. Figure 4 shows an example of a formula being edited. When the file is saved, the editor reverts the process and produces a pure ASCII file.

f	::=	`ALL` var `.` f	∀-construct
	|	`EX` var `.` f	∃-construct
	|	`%` var `.` f	lambda abstraction
	|	f f	function application
	|	f `=` f	equality
	|	`(` f `::` t `)`	typed formula
	|	f (`-->` | `|` | `&`) f	boolean connectives
	|	$\sim f$	negation
	|	f (`+` | `-` | `*` | `/`) f	arithmetic operators
	|	f (`<` | `<=` | `>` | `>=`) f	arithmetic relational symbols
	|	$intLit$	integer literals
	|	f (`\<subseteq>` | `\<setminus>`) f	set operators
	|	`\<emptyset>`	empty set
	|	`card` f	set cardinality
	|	`final` f	final set
var	::=	$ident$	variable
	|	($ident$`::`$type$)	typed variable
t	::=	`bool`	boolean type
	|	`int`	integer type
	|	`obj`	uninterpreted object type
	|	t `=>` t	function type
	|	t `list`	list type
	|	`'`$ident$	type variable
$ident$	::=	`[a-zA-Z][a-zA-Z0-9_]`*	identifiers
$intLit$	::=	`0` | `[1-9][0-9]`*	

Figure 3: Isabelle-like concrete syntax of HOL formulas, as recognized by our parser implementation.

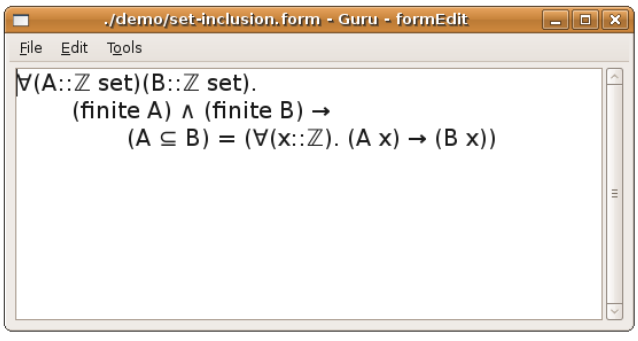

Figure 4: A screenshot of the formula editor displaying Unicode characters

f	::=	$(\forall \mid \exists)\ x.f$	quantified formulas
	$\mid$	$f_1\ (\ \rightarrow\ \mid\ \wedge\ \mid\ \vee\ )\ f_2$	boolean connectives
	$\mid$	$\neg f$	negation
	$\mid$	$f_1\ =\ f_2$	equality
	$\mid$	$(\ sym\ f_1\ \ldots\ f_n\ )$	function or predicate application
	$\mid$	x	variable or constant
sym	::=	$+\ \mid\ -\ \mid\ *\ \mid$	arithmetic function symbols
	$\mid$	$<\ \mid\ \leq\ \mid\ >\ \mid\ \geq$	arithmetic relational symbols
	$\mid$	x	function or predicate symbol
t	::=	$\mathbb{B}$	booleans
	$\mid$	$\mathbb{Z}$	integers
	$\mid$	$\mathbb{O}$	uninterpreted objects
	$\mid$	$(t_1, \ldots, t_n) \Rightarrow t_r$	total functions and predicates

Figure 5: Abstract syntax of FOL

First-order logic

Since most of the solvers with which VEPAR is destined to work are designed to handle formulas in first-order logic, it is important that we have some representation for them built in the system. The abstract syntax for this logic, which we call FOL, is shown in Figure 5.

We currently do not have a parser or a concrete syntax for this logic. Trees can be built using library functions, or by converting a HOL formula as described in the next section.

Translation from high-order logic

At this stage of development, the translation from HOL to FOL is implemented in a straight-forward way: if a HOL formula appears to be encodable entirely in FOL, it is done so recursively. No attempt is made to convert set operations to quantified statements about their characteristic functions, for instance. Neither do we attempt to over- or under-approximate the HOL formula should an exact translation be impossible. More elaborate techniques are described in [3], for instance, and their implementation as part of VEPAR is left for future work.

Connection to SMT solvers

FOL formulas can be dispatched to SMT solvers such as Z3 [11] or CVC3 [2]. These solvers check for the satisfiability of formulas and we therefore use the common fact that a formula ϕ is valid if and only if $\neg\phi$ is unsatisfiable.

The connection to the solvers is achieved by pretty-printing the FOL formula in SMT-LIB format [15] into a file, passing this file to the solver and retrieving the output of the solver. Although CVC3, for example, provides a C interface, using files is the only solution which works consistently over all SMT solvers. Currently, all formulas are generated for the AUFLIA logic and we do not attempt to detect when a simpler logic can be used (such as QF_UFLIA or QF_UF).

Development of a non-clausal SMT solver

During this semester project, some work has also been done on fSTP, an SMT solver, [16] in parallel to the development of the VEPAR project.

Traditionally, SMT solvers work on formulas in conjunctive normal form (CNF). While this has the advantage that it makes their algorithms conceptually simpler, it also means that formulas which are not in CNF must be converted to this form first. This can be done in polynomial time and space by introducing new boolean variables in the formula, but at the cost of destroying the original structure therefore running the risk of making the search for a solution (a satisfiable assignment) harder. fSTP was designed to work on formulas in negation normal form (NNF). Any formula can be converted to NNF without changing its size and, arguably, its structure. The algorithms for reasoning about formulas in NNF were introduced in [6].

The work done on fSTP during this semester involved bug fixes, and identification of performance bottlenecks. We are currently exploring the role of fSTP and its algorithsm in a prover for expressive formulas, such as VEPAR.

Future work

We have presented the current status of the VEPAR project. The system is currently in an early development stage and we expect significant development in the coming months.

Near-term infrastructure development

One feature we will need to have implemented very soon is type inference. Currently, only variables are identified and their type, if explicitly declared, is shared among their instances. We will add full Hindley-Milner type inference [13, Chapter 22] to the system.

Integration of SAT solver

An important component we will need to integrate is a SAT solver, which will be used for boolean reasoning and, arguably more importantly, for DPLL-like satisfiability checking of HOL formulas. The applicability of these techniques to rich logics remains to be studied.

Congruence closure and unification

A congruence closure algorithm is often at the center of a or a combination of decision procedures. We will study the applicability of such an approach to higher-order formulas.

Reasoning about recursive functions and relations

(Purely) functional programs can in general be translated into higher-order logic with minimal effort, with the exception of recursive definitions. We will study techniques to reason about these definitions, possibly by integrating some form of bounded unrolling into the theorem proving process.

Combination techniques for rich theories

While decision procedures exist for some decidable fragments of HOL, their potential for combination is still a subject of investigation. Some recent work [9] has been done in that direction, but some questions still need to be addressed, in particular regarding the practicality of the (theoretically sound) approach. Given its HOL foundation, VEPAR appears to be an adequate platform to experiment with these new combination ideas.

Integration of related techniques

VEPAR is a collective effort. The following are among the ongoing developments in VEPAR being pursued concurrently with my efforts:

- Abhinav Kumar is working on the connection to MONA, a decision procedure for WS2S;
- Ali Sinan Köksal is doing a semester project on connecting VEPAR with finite model finding tools;
- Steven Obua is designing a tactic language for within VEPAR, and developing the set-theoretic foundations;
- Ruzica Piskac will be developing and integrating a new decision procedure for multisets with cardinality constraints [14].

I will collaborate with the researchers involved in these efforts and ensure that they integrate into the overall VEPAR infrastructure.

References

[1] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Springer (Kluwer), 2nd edition, 2002.

[2] C. Barrett and C. Tinelli. CVC3. *Lecture Notes in Computer Science*, 4590:298, 2007.

[3] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using First-Order Theorem Provers in the Jahob Data Structure Verification System. *Verification, Model Checking and Abstract Interpretation, November*, 2007.

[4] B. Emir, M. Odersky, and J. Williams. Matching Objects with Patterns. *Lecture Notes in Computer Science*, 4609:273, 2007.

[5] S. E. Hudson and M. Petter. CUP: LALR Parser Generator for Java. `http://www2.cs.tum.edu/projects/cup/`.

[6] H. Jain. *Verification using Satisfiability Checking, Predicate Abstraction, and Craig Interpolation*. PhD thesis, Carnegie Mellon University, School of Computer Science, 2008.

[7] G. Klein and R. Décamps. JFlex - The Fast Scanner Generator for Java. `http://jflex.de`.

[8] V. Kuncak. *Modular Data Structure Verification*. PhD thesis, Massachusetts Institute of Technology, 2007.

[9] V. Kuncak and T. Wies. On set-driven combination of logics and verifiers. Technical Report LARA-REPORT-2009-001, EPFL, February 2009.

[10] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/Hol: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[11] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. *Lecture Notes in Computer Science*, 4963:337, 2008.

[12] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2nd Edition). Technical report, 2006.

[13] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] R. Piskac and V. Kuncak. Decision Procedures for Multisets with Cardinality Constraints.

[15] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[16] P. Suter. Non-Clausal Satisfiability Modulo Theories. 2008.