Compiling Logics

Mihai Codescu¹, Fulya Horozal², Till Mossakowski¹, and Florian Rabe²

DFKI GmbH Bremen, Germany
Computer Science, Jacobs University Bremen, Germany

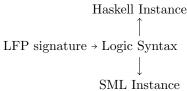
In [1], we presented an extension of the Heterogeneous Tool Set HETS [4] with a framework for representing logics independently of their foundational assumptions. The key idea [5] is that a graph of theories in a type theoretical logical framework like LF [2] can fully represent a model theoretic logic. Our integration used this construction to make the process of extending HETS with a new logic more declarative, on one side, and fully formal, on the other side.

However, the new logic in HETS inherits the syntax of the underlying logical framework. This is undesirable for multiple reasons. Firstly, a logical framework unifies many concepts that are distinguished in individual logics. Examples are binding and application (unified by higher-order abstract syntax), declarations and axioms (unified by the Curry-Howard correspondence), and different kinds of declarations (unified by LFP – LF with declaration patterns – as introduced in [3]). Therefore, users of a particular logic may find it unintuitive to use the concrete syntax (and the associated error messages) of the logical framework.

Secondly, only a small fragment of the syntax of the logical framework is used in a particular logic. For example, first-order logic only requires two base types term for terms and form for formulas and not the whole dependent type theory of LF. Therefore, it is unnecessarily complicated if implementers of additional services for a particular logic have to work with the whole abstract syntax of the logical framework. Such services include in particular logic translations from logics defined in LF to logics implemented by theorem provers.

Therefore, we introduce an architecture that permits compiling logics defined in LF into custom definitions in arbitrary programming languages. This is similar to parser generators, which provide implementations of parsers based on a language definition in a context-free grammar. Our work provides implementations of a parser and a type-checker based on a context-sensitive language definition in the recent extension of LF with *declaration patterns* (LFP) [3]. Here declaration patterns give a formal specification of the syntactic shape of the declarations in the theories of a logic.

An overview of the architecture is given on the right. The centerpiece is the compiler, which takes an LFP signature and produces an abstract representation of the syntax of the defined logic, including parsing and type-checking algorithms in a generic functional



programming language. In a second step, the logic syntax can be easily serialized in, e.g., Haskell or SML.

A logic syntax corresponds to the straightforward implementation one would choose in a functional programming language. Given an LFP signature Σ , the compiler generates one abstract data type for every type family declaration in Σ . Here all type dependencies are erased, and only those LFP symbols with higher-order arguments are supported that can be readily interpreted as binders. Moreover, for every declaration pattern in Σ , one data type of declarations is generated, and lists of such declarations form the type of signatures. Finally, one type family must be distinguished as the type of formulas, which permits the definition of a type of theories.

For example, for propositional logic, the LF type form: type generates one data type form whose constructors are the connectives. The declaration pattern $pattern PropVar = \{F : form\}$ for propositional variables gives rise to a data type PropVar with one constructor taking a string argument, i.e., the name of the declared symbol.

In addition, the compiler produces two families of recursive functions. The first one parses a generic representation of concrete syntax into the above data types. The second one translates them into a generic abstract syntax for static analysis.

We are currently integrating this framework into HETS, using a re-implementation of first-order logic as an easy test case. This will provide insight about how much functionality is still missing in order to make the implementation of new logics in HETS fully declarative. Important such functionality includes signature colimits, amalgamability checks, and theorem prover interfaces. The plan for the future is to extend the framework with these features, and successively replace HETS' Haskell-coded logics with compiled LF-defined logics.

References

- M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In T. Mossakowski and H. Kreowski, editors, Recent Trends in Algebraic Development Techniques 2010, volume 7137 of Lecture Notes in Computer Science, pages 139–159. Springer, 2012.
- R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. Journal of the Association for Computing Machinery, 40(1):143–184, 1993.
- 3. F. Horozal and F. Rabe. Representing categories of theories in a proof-theoretical logical framework. https://svn.kwarc.info/repos/fhorozal/pubs/theory-categories_abst.pdf. Submitted to WADT 2012.
- T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editor, TACAS 2007, volume 4424 of Lecture Notes in Computer Science, pages 519–522, 2007.
- 5. F. Rabe. A Logical Framework Combining Model and Proof Theory. see http://kwarc.info/frabe/Research/rabe_combining_10.pdf, 2010.