Structural Recursion over Contextual Objects

Sherry Shanshan Ruan and Brigitte Pientka
School of Computer Science
McGill University
Montreal, Canada
shanshan.ruan@mail.mcgill.ca, bpientka@cs.mcgill.ca

Andreas Abel
Department of Computer Science and Engineering
Chalmers and Gothenburg University
Gothenburg, Sweden
andreas.abel@gu.se

Category: Undergraduate
ACM member number: 6888328
Address: McConnell Eng. Bldg., Lab 225, McGill University, 3480 University Street, Montreal, Quebec, Canada H3A 0E9

Extended abstract — The logical framework LF [Harper et al., 1993] is an elegant and robust system which supports concise specifications of formal systems and proofs based on higher-order abstract syntax (HOAS). By modelling binders in the object language using binders in LF, we are able to attain a uniform representation of both variables and assumptions in the object language and a smooth treatment of variables together with discharging and replacing them.

While the elegance of higher-order abstract syntax encodings is widely acknowledged, it has been challenging to reason inductively about LF specifications and to formulate well-founded recursion principles. We cannot specify inductions in the standard sense since HOAS specifications violate the restriction on positivity. In particular, we extend our context of assumptions as we recursively traverse higher-order abstract syntax trees, and LF objects do not remain closed any more. To address this problem, Pientka and collaborators [Pientka, 2008, Pientka and Dunfield, 2008, Cave and Pientka, 2012] propose to pair LF objects together with the context in which they are meaningful. This notion is then internalized as a contextual type $[\Psi \vdash A]$ which is inhabited by term M of type A in the context Ψ [Nanevski et al., 2008]. Contextual objects are then embedded into a computation language which supports pattern matching on contexts, contextual objects and general recursion. Beluga, a programming environment based on these ideas [Pientka and Dunfield, 2010], has been used for a wide range of applications such as encoding normalization proofs [Cave and Pientka, 2013] and type-preserving compiler including closure conversion and hoisting [Belanger et al., 2013]. However, Beluga lacks the justification of inductive principles, thereby cannot guarantee that a given program is total.

In this work, we propose a core programming language corresponding to first-order logic together with an induction principle. This principle, based on a simultaneous pattern matching construct over a specific domain, provides a modular way of writing proofs. This technique is novel since we can model rich domains in LF and uniformly justify their inductive principles over LF signatures. Thus, the system also provides a type-theoretic foundation for Beluga, which verifies programs are total.

We present an example to illustrate our core language which naturally supports higher-order functions and provides a proof language for first-order logic with contextual LF as a domain. We represent only well-typed terms in the logical framework LF by indexing terms with their corresponding types.

$$
\begin{array}{lll}
\texttt{tp} & : & \texttt{type}. \\
\texttt{bool} & : & \texttt{tp}. \\
\texttt{arr} & : & \texttt{tp} \to \texttt{tp} \to \texttt{tp}. \\
 & & \\
\texttt{term} & : & \texttt{tp} \to \texttt{type}. \\
\texttt{app} & : & \texttt{term}\ (A \to B) \to \texttt{term}\ A \to \texttt{term}\ B. \\
\texttt{lam} & : & (\texttt{term}\ A \to \texttt{term}\ B) \to \texttt{term}\ (A \to B).
\end{array}
$$

The recursive program given in Figure 1 computes the number of constructors, where `ctx` describes the schema for contexts containing declarations $x{:}\texttt{term}\ A$ for some A. For better readability, we adopt Beluga's syntax and write .. for identity substitutions.

$$
\begin{array}{l}
\textbf{rec}\ \texttt{count} : (\psi : \texttt{ctx})\ [\psi \vdash \texttt{term A}] \to \texttt{nat} = \\
\textbf{fn}\ \texttt{y} \Rightarrow \textbf{case}\ \texttt{y}\ \textbf{of} \\
\mid\ [\psi \vdash \#\texttt{p..}] \Rightarrow \texttt{0} \\
\mid\ [\psi \vdash \texttt{lam x.M..x}] \Rightarrow \texttt{1} + \texttt{count}\ [\psi, \texttt{x:term B} \vdash \texttt{M..x}] \\
\mid\ [\psi \vdash \texttt{app (M..) (N..)}] \Rightarrow \texttt{1} + \texttt{count}\ [\psi \vdash \texttt{M..}] + \texttt{count}\ [\psi \vdash \texttt{N..}]
\end{array}
$$

Fig. 1. Counting constructors in a term

The example illustrates recursion over elements of type $[\psi \vdash \texttt{term A}]$. The central difficulty lies in traversing a lambda term. This is tackled by extending the context ψ associated with the contextual object $[\psi \vdash \texttt{lam}\ \lambda\texttt{x.M..x}]$ to $\psi, \texttt{x:term B}$. The totality is verified by our coverage checking and termination analysis built on [Schürmann and Pfenning, 2003, Dunfield and Pientka, 2009, Pientka, 2005]. Instead of simply splitting a given object into different cases as in a general matching construct, our system introduces well-founded recursive calls based on a specified invariant. In the given example, the system splits an object of type $[\psi \vdash \texttt{term}\ A]$ into three cases: parameter variable $[\psi \vdash \#\texttt{p..}]$ with $\#p$ standing for a variable in context ψ, lambda term $[\psi \vdash \texttt{lam x.M..x}]$, and application term $[\psi \vdash \texttt{app (M..) (N..)}]$. While the parameter variable case corresponds to the base case, the other two cases

represent step cases. The system thereby generates all valid induction hypotheses for them, i.e., `count` $[\psi, \texttt{x:term B} \vdash \texttt{M..x}]$ and `count` $[\psi \vdash \texttt{M..}]$, `count` $[\psi \vdash \texttt{N..}]$. This dynamic on-the-fly generation of well-founded recursive calls contrasts with a generic induction principle which is statically derived from the inductive definition such as in Coq. Since the impossible cases are not generated, our language reduces the amount of cases needed. We describe a measure for contextual objects and prove the completeness and the validity of the set of call patterns and the recursive calls generated. Therefore, our type system not only warrants that we are manipulating well-typed objects but also ensures that a given collection of cases is covered and recursive calls are well-founded.

To establish consistency, we define a call-by-value small-step semantics for our core language and prove the termination of every well-typed program using logical relations. This justifies the interpretation of well-founded recursive programs in our core language as first-order inductive proofs. Our proof of normalization follows Taits method of logical relations. Such a proof is absent in previous work [Schürmann, 2000].

To summarize, we propose a domain specific language with well-founded recursion for implementing total functions about LF specification.We also develop and prove a sound splitting and induction hypotheses generating algorithm. Moreover, the proof is modular in the sense that we can model different domains in LF. Our language serves as a core programming language, and as justified by the Curry-Howard isomorphism, can be interpreted as a language for the development of interactive proofs. At present we concentrate on generating recursive calls which are structurally smaller. In the future, we plan to implement and design a proof engine as such and to generalize this notion to support lexicographic orderings and generally well-founded recursions.

REFERENCES

Olivier Savary Belanger, Stefan Monnier, and Brigitte Pientka. Programming type-safe transformations using higher-order abstract syntax. In Georges Gonthier and Michael Norrish, editors, *Third International Conference on Certified Programs and Proofs (CPP'13)*, Lecture Notes in Computer Science (LCNS 8307), pages 243–258. Springer, 2013.

Andrew Cave and Brigitte Pientka. Programming with binders and indexed data-types. In *39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 413–424. ACM Press, 2012.

Andrew Cave and Brigitte Pientka. First-class substitutions in contextual type theory. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, pages 15–24. ACM Press, 2013.

Joshua Dunfield and Brigitte Pientka. Case analysis of higher-order data. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'08)*, volume 228 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 69–84. Elsevier, June 2009.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9(3):1–49, 2008.

Brigitte Pientka. Verifying termination and reduction properties about higher-order logic programs. *J. Autom. Reasoning*, 34(2):179–207, 2005.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

Brigitte Pientka and Joshua Dunfield. Programming with proofs and explicit contexts. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 163–173. ACM Press, July 2008.

Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer-Verlag, 2010.

Carsten Schürmann. *Automating the Meta Theory of Deductive Systems.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 2000. CMU-CS-00-146.

Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, pages 120–135. Springer, 2003.