Logical Framework Based Program Development

David Basin
Institut für Informatik, Universität Freiburg
D-79110 Freiburg, Germany
basin@informatik.uni-freiburg.de

We propose a methodology for formalizing and using logical frameworks for program development. Calculi for program transformation and synthesis are implemented in a logic, such as first-order logic, by formally deriving proof rules. The derived rules are then applied using higher-order resolution to simultaneously develop programs with their correctness proofs and this development is partially automated by calculus specific tactics. We have applied this approach, using the ISABELLE system, to develop hardware descriptions, and functional and logic programs.

1. FORMAL DEVELOPMENT FROM FOUNDATIONS

We propose a methodology for formalizing and using calculi for program development. It is based on using a logical framework to formally derive the proof rules of the calculi. Programs are constructed semi-automatically by tactics, which apply the proof rules using higher-order resolution. Resolution 'solves' for programs during proof construction and enables users to develop programs simultaneously with their correctness proofs.

Our starting point is the ISABELLE system, a logical framework in which users encode the syntax and deductive systems of logical theories [Paulson 1994]. Given a particular 'target' development calculus, we build in ISABELLE a development environment for it in three steps.

(1) We formalize a base theory, typically first-order or higher-order logic enriched with programming language syntax and theories for relevant data-types, such as numbers or lists.
(2) We formalize proof rules and formally derive them in the base theory. The use of a logical framework as opposed to, say, a theorem prover for first-order logic, is important here, as this step requires formalizing and reasoning about *rules* as opposed to *formulae*. The rules formalized are designed to be adequate to simulate derivations of the target calculus.
(3) We write *tactics* (programs that construct proofs) that help automate the application of proof rules. For example, by adopting ideas from [Bundy et al. 1991; Basin and Walsh 1996], in [Kraan et al. 1996] we completely automated the synthesis of many (albeit relatively simple) logic programs.

This methodology is based on, and arose from, case studies including: unfold/fold style development for functional and logic programs; Deductive Tableaux style development of sorting programs [Ayari and Basin 1996]; synthesis of logic programs [Kraan et al. 1996]; transformation of logic programs using transformation template libraries [Anderson and Basin 1995]; and deductive synthesis of CMOS circuits [Basin and Friedrich 1996]. Our experience indicates that there are a number of advantages to this approach. First, and perhaps surprisingly, formal development from foundations can be achieved at relatively little cost. For example, our environments for Deductive Tableaux synthesis and circuit synthesis were each built in a matter of weeks as we could take advantage of standard Isabelle theories. Second, programs are developed using rules that have been formally verified. This gives us a formal guarantee of the correctness of the calculi and helped us find and correct errors in calculi proposed by other researchers [Basin 1994]. Third, although it sometimes requires considerable work to reinterpret development calculi as collections of derived rules (see [Ayari 1995] for details on the reinterpretation of Deductive Tableaux), doing so sheds new light on the calculi and can lead to extensions and improvements. Moreover, we have found that there is considerable synergy between theories we have reinterpreted: rules and tactics developed in one setting often can be reused in others.

2. AN EXAMPLE

We present an abstracted and simplified example: transformational development of a functional program over lists.

Step 1: Our basis theory formalizes a first-order theory of lists, built from the empty list '[ ]' and the cons operator ':'.

Step 2: We derive rules for constructing recursive programs. Here we give a rule motivated by analysis of unfold/fold proofs of [Burstall and Darlington 1977], which can often be reinterpreted as proofs by induction. The key idea is that the induction used determines how equations are instantiated and use of the induction hypothesis corresponds to 'folding' with the defined program.

The following rule develops programs that structurally recurse over lists.

$$\frac{\begin{array}{l} P(x,y,z) = \ \mathit{if}\ \ x = [\,]\ \mathit{then}\ \ B(y,z)\ \ \mathit{else}\ \ S(hd(x), tl(x), y, z) \\ A([\,],y,z) = B(y,z) \\ A(xs,y,z) = P(xs,y,z) \Rightarrow A(x{:}xs,y,z) = S(x,xs,y,z) \end{array}}{A(x,y,z) = P(x,y,z)}$$

We derive this rule in ISABELLE by structural induction over the list x. The rule states that when the premises (above the line) hold, then so does the conclusion, and it formalizes sufficient conditions for the equality of A and P. When applied, A will constitute a specification and subsequent proof steps will instantiate the program schema P, given by the first premise, to a concrete program.

Step 3: We use higher-order resolution and tactics to synthesize programs from specifications or transform programs (when the specification is already executable).

Suppose that append, '@', is defined as follows.

$$\begin{aligned} [\,] \mathbin{@} y &= y \\ (x{:}xs) \mathbin{@} y &= x{:}(xs \mathbin{@} y) \end{aligned}$$

Now consider the problem of computing $(l \mathbin{@} m) \mathbin{@} n$. We can directly use the above equations, but this is inefficient since we first must construct $l \mathbin{@} m$ and then retraverse this list to append n. Instead we build a more efficient program P by postulating that it exists and developing it during proof, i.e., we prove the goal

$$\forall l\, m\, n.\, (l \mathbin{@} m) \mathbin{@} n = ?P(l,m,n).$$

We have prefixed P with '?' to emphasize that it is a meta-variable that will be instantiated ('solved for') during proof.

After some preprocessing, this goal can be resolved with the above rule. This entails (higher-order) unifying the goal with the conclusion of the rule and yields three subgoals, one for each rule premise.

$$P(x, y, z) = \ \mathit{if} \ \ x = [\,] \ \ \mathit{then} \ \ ?B(y, z) \ \ \mathit{else} \ \ ?S(hd(x), tl(x), y, z)$$
$$([\,] \ @ \ y) \ @ \ z = ?B(y, z)$$
$$(xs \ @ \ y) \ @ \ z = P(xs, y, z) \Rightarrow (x{:}xs \ @ \ y) \ @ \ z = ?S(x, xs, y, z)$$

The first subgoal constitutes the skeleton of the transformed program. The second can be simplified with the definition of @ yielding $y \ @ \ z = ?B(y, z)$, and then discharged (as equality is reflexive) by unifying $?B(y, z)$ with $y \ @ \ z$. In the third, the right-hand side of the implication is simplified to $x{:}((xs \ @ \ y) \ @ \ z) = ?S(x, xs, y, z)$ and $?S(x, xs, y, z)$ is partially instantiated to $x{:}?S'(x, xs, y, z)$, where $?S'$ is a new metavariable. After simplification the subgoal is

$$(xs \ @ \ y) \ @ \ z = P(xs, y, z) \Rightarrow (xs \ @ \ y) \ @ \ z = ?S'(x, xs, y, z)$$

and we can now unify the left-hand side of the implication (essentially, an induction hypothesis) with the right; this unifies $?S'(x, xs, y, z)$ with $P(xs, y, z)$ and the resulting subgoal is a tautology. This use of the induction hypothesis corresponds to a folding step in an unfold/fold derivation and builds a recursive program.

The second and third proof steps have now be solved and the first is fully instantiated to our transformed program

$$P(x, y, z) = \ \mathit{if} \ \ x = [\,] \ \ \mathit{then} \ \ y \ @ \ z \ \ \mathit{else} \ \ hd(x){:}P(tl(x), y, z)\,.$$

The actual derivation of such programs is fully automated by tactics in ISABELLE.

3. DISCUSSION

The idea of 'development from foundations' goes back to [Gordon et al. 1979] and more recent examples include the work of [Paulson 1993] on verification using set-theory. Note though that the vast majority of development calculi are not formalized and their foundations are part of an informal and usually unstated meta-theory.

Synthesis based on resolution goes back at least to the work of [Green 1969] and the development of theorem provers based on variants of higher-order resolution (see [Pfenning 1991; Paulson 1994]) has lead to new applications, e.g., [Coen 1992; Kolyang et al. 1996] and see [Basin and Krieg-Brückner 1997] for a survey.

The example we gave here is motivated by development in the unfold/fold framework of [Burstall and Darlington 1977] and has also been studied by the partial evaluation community. For instance, [Glück and Sørensen 1996; Sørensen et al. 1997] have shown that this program can be derived by supercompilation and that there are close relationships to partial evaluation and deforestation.

Our work bridges two areas, computation and deduction, and two communities, program transformation/development and theorem proving. There is much to be gained from bringing these two communities closer together. Logical frameworks like Isabelle offer many advantages in implementing, exploring, and applying calculi. Moreover, there are also conceptual advantages. The relationship between the proof rules of the target calculus and those which are appropriate for synthesis based on resolution is not always straightforward (e.g., the relationship between our example and an unfold/fold derivation). However, by recasting calculi we gain insights and a uniform formal background for comparing them, understanding their correctness, and developing improvements.

REFERENCES

ANDERSON, P. AND BASIN, D. 1995. Deriving and applying logic program transformers. In *Algorithms, Concurrency and Knowledge (1995 Asian Computing Science Conference)*, Volume 1023 of *LNCS* (Pathumthani, Thailand, December 1995), pp. 301–318. Springer-Verlag.

AYARI, A. 1995. A reinterpretation of the deductive tableaux system in higher-order logic. Master's thesis, University of Saarbrücken.

AYARI, A. AND BASIN, D. 1996. Generic system support for deductive program development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS-96)*, Volume 1055 of *LNCS* (Passau, Germany, March 1996), pp. 313–328. Springer-Verlag.

BASIN, D. 1994. Isawhelk: Whelk interpreted in Isabelle. In *11th International Conference on Logic Programming (ICLP94)* (1994).

BASIN, D. AND FRIEDRICH, S. 1996. Modeling a hardware synthesis methodology in Isabelle. In *Theorem Proving in Higher Order Logics (TPHOLs'96)*, Volume 1125 of *LNCS* (Turku, Finland, August 1996), pp. 33–50. Springer-Verlag.

BASIN, D. AND KRIEG-BRÜCKNER, B. 1997. Formalization of the development process. In ASTESIANO, KREOWSKI, AND KRIEG-BRÜCKNER Eds., *Algebraic Foundations of System Specification*. To appear.

BASIN, D. AND WALSH, T. 1996. Annotated rewriting in inductive theorem proving. *Journal of Automated Reasoning 16*, 1–2, 147–180.

BUNDY, A., VAN HARMELEN, F., HESKETH, J., AND SMAILL, A. 1991. Experiments with proof plans for induction. *Journal of Automated Reasoning 7*, 303–324.

BURSTALL, R. M. AND DARLINGTON, J. 1977. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery 24*, 1, 44–67.

COEN, M. D. 1992. Interactive program derivation. Technical Report 272 (November), Cambridge University Computer Laboratory, Cambridge.

GLÜCK, R. AND SØRENSEN, M. H. 1996. A roadmap to metacomputation by supercompilation. In O. DANVY, R. GLÜCK, AND P. THIEMANN Eds., *Partial Evaluation*, Volume 1110 of *Lecture Notes in Computer Science* (1996), pp. 137–160. Springer-Verlag.

GORDON, M. J., MILNER, R., AND WADSWORTH, C. P. 1979. *Edinburgh LCF: A Mechanized Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

GREEN, C. 1969. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69* (1969), pp. 219–239.

KOLYANG, SANTEN, T., AND WOLFF, B. 1996. Correct and user-friendly implementation of transformation systems. In M.-C. GAUDEL AND J. WOODCOCK Eds., *FME '96 — Industrial Benefits and Advances in Formal Methods*, LNCS 1051 (1996), pp. 629–648. Springer Verlag.

KRAAN, I., BASIN, D., AND BUNDY, A. 1996. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning 16*, 1–2, 113–145.

PAULSON, L. C. 1993. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning 11*, 353–389.

PAULSON, L. C. 1994. *Isabelle : a generic theorem prover; with contributions by Tobias Nipkow*, Volume 828 of *LNCS*. Springer, Berlin.

PFENNING, F. 1991. Logic programming in the LF logical framework. In *Logical Frameworks*, pp. 149 – 181. Cambridge University Press.

SØRENSEN, M. H., GLÜCK, R., AND JONES, N. D. 1997. A positive supercompiler. *Journal of Functional Programming*. To appear.