Normalising the Associative Law: An Experiment with Martin-Löf's Type Theory

Michael Hedberg
Programming Methodology Group, Department of Computer Sciences, Chalmers University of Technology, S-412 96 Göteborg, Sweden

Keywords: Functional programming; Type theory; Inductive types; Associative law

Abstract. Martin-Löf's type theory contains a logic, a specification language and a programming language, so it is a tool with different uses. Although it is traditionally used as an integrated programming logic, it may well be used as an external logic, which is necessary if one wants to use the formalism of type theory to verify the correctness of an external program. Different tools, such as well founded recursion, measure functions, or the separation of correctness into termination and partial correctness, may be used to obtain a correct type theory program. Type theory is viewed as an open system with respect to inductively defined types and predicates, which makes it easy to represent an external program as a graph. Formal proofs have been edited using Larry Paulson’s ISABELLE.¹

1. Introduction

1.1. Conditional Expressions

In Boyer and Moore [BoM79], a tautology checker for classical propositional logic is presented. The logic is based on the ternary if-connective and names of

¹The paper is a slightly revised version of a Thesis for the Licentiate Degree in Computer Science at University of Göteborg with the same title. The Thesis lists some ISABELLE proof sessions in its appendix.
the form at(...) for atomic propositions. The expressions denoting propositions are also called conditional expressions. The difficult part of the tautology checker is to eliminate tested ifs from an expression. This subproblem may well be described as normalisation with respect to the equation

$$\text{if}(\text{if}(x, y, z), u, v) = \text{if}(x, \text{if}(y, u, v), \text{if}(z, u, v))$$

This leads to a derivation problem, that of constructing a normaliser. Boyer and Moore also give a general recursive program,

$$\begin{align*}
\text{norm-cond}(\text{at}(a)) &= \text{at}(a) \\
\text{norm-cond}(\text{if}(\text{at}(a), u, v)) &= \text{if}(\text{at}(a), \\
&\quad \text{norm-cond}(u), \text{norm-cond}(v)) \\
\text{norm-cond}(\text{if}(x, y, z), u, v)) &= \text{norm-cond}(\text{if}(x, \\
&\quad \text{if}(y, u, v), \text{if}(z, u, v)))
\end{align*}$$

which leads to the verification problem of proving that this program is indeed a normaliser. The difficult thing here is termination.

1.2. Programming Logics

In order to reason about programs, one has to have a programming logic in which to express proofs of termination or correctness. What logic to use depends on the programming language. Hoare logic, for instance, is suitable for imperative programs, Hennessy–Milner logic talks about the parallel and nondeterministic agents of CCS, and LCF is about functional programs. All these are what Girard [Gir86] calls external logics, because programs and proofs live in different worlds—the correctness proof is thought of as a comment, external to the program. Martin-Löf’s type theory, on the other hand, is an integrated logic. The concept of integrated logic is made possible through the Curry–Howard analogy between proofs and functional expressions, so the dichotomy between external and integrated logic concerns primarily functional programming. The Curry–Howard analogy is the main idea behind type theory as a programming logic and it is a useful conceptual tool that may play a central rôle in the implementation of functional programming logics. When enhanced to a doctrine, however, it becomes problematic, because it leads to the unnatural contamination of values with proof garbage. If you ask your computer to print an even natural number of its liking, and it chooses the number 6, then you want it to print this number, without justifying its answer by explaining why 6 is an even number. Although silly, the example illustrates the essence of the contamination problem, which has inspired to the formulation of type theories with a relaxed Curry–Howard doctrine.

Other respects in which programming logics may differ are the following:

Is the logic constructive or classical?
Are objects total or partial?
Is recursion primitive or general?

---

2 Properly speaking, there are names for the truth values as well, but they will have no bearing on the subproblem addressed.