Original submission WADT'04

Towards Eliminating Conditional Rewrite Rules

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin, Urbana, Illinois 61802, USA

Conditional equations are a continuous source of difficulties in the automation of equational reasoning by term rewriting, because in order to evaluate a condition one may need to use again conditional rewriting rules, and so on, for any arbitrary level of nesting. Thus, like implementations of functional programming languages which need to maintain directly or indirectly stacks of calling environments in order to continue the execution of a function after returning from a function call, equational provers or rewriting engines need to maintain *control context* information encoding the status of the evaluation of conditions of a potentially very large sequence of applications of conditional equations.

Let us consider the following simple equational specification where boolean tests odd/even are defined mutually recursively on top of Peano natural numbers and booleans using conditional equations:

Signature
  $\mathit{operation}\ o \ :\ \mathit{Nat} \to \mathit{Bool}$
  $\mathit{operation}\ e \ :\ \mathit{Nat} \to \mathit{Bool}$
Equations
  $o(s(N)) = \mathit{true}$ `if` $e(N) = \mathit{false}$
  $o(s(N)) = \mathit{false}$ `if` $e(N) = \mathit{true}$
  $o(0) = \mathit{false}$
  $e(s(N)) = \mathit{true}$ `if` $o(N) = \mathit{false}$
  $e(s(N)) = \mathit{false}$ `if` $o(N) = \mathit{true}$
  $e(0) = \mathit{true}$

With this admittedly inefficient conditional equational definition, in order to see whether $o(n)$ is *true* or *false* for some natural number n, a proof tree of size $O(2^n)$ may need to be investigated. Moreover, an automatic prover or rewriting engine would build such a tree "on-the-fly", so it would have to store enough control context information not only to continue an incomplete path, but especially to backtrack and try another path in case of failure.

Therefore, conditional rewriting is currently a complex procedure to implement efficiently, with significant effects on the overall efficiency of equational reasoning. Additionally, most known optimizations, heuristics, confluence and termination criteria, etc., that work for non-conditional equations or rewriting rules do not trivially extend to the conditional case. From now on, all equations are regarded as rewriting rules. For the specification above, the symbol $=$ should be regarded as $\Rightarrow$ unless it occurs in a condition, where it should be regarded as $\Rightarrow^*;\Leftarrow^*$. So we follow the mainstream rewriting approach to equational specification, where equations are applied only from left to right, while their conditions are evaluated by rewriting both terms and then checking whether their normal forms are identical.

Stimulated by the power of *continuations* in functional programming and many other places, the core idea of our conditional equation removal technique is to *transform the control context into data context.* More precisely, we enrich terms with information about what conditional rewriting rules can be applied and where; once a conditional rule is tried and failed, the term is modified accordingly, so that that rule is not tried anymore. In order to do this, we replace each operation $\sigma : s_1 \times \cdots \times s_n \to s$ by an operation $\sigma : s_1 \times \cdots \times s_n \times b \times \cdots \times b \to s$, where b is a fresh sort occurring as many times as conditional rules are in the spec having the operation σ at the top of their left-hand-side terms. The sort b contains two constants, t and f. Intuitively, a term $\sigma(u_1, ..., u_n, t, f, t)$ stores the information that there are three conditional rules that might be applied on its top, but that the second had been tried and failed.

Our technique builds on a rewriting safe axiomatization of equality. The procedure starts by adding operations $eq : s \times s \to b$ and $? : b \times s \times s \to s$ for each sort s, together with equations $(\forall x : s)\ eq(x, x) = t$ and $(\forall x : s, y : s)\ ?(t, x, y) = x$ and $(\forall x : s, y : s)\ ?(f, x, y) = y$ for each sort s, and $(\forall X, Y)\ eq(\sigma(X), \tau(Y)) = f$ for any distinct operations σ and τ of same result sort, where X and Y are appropriate lists of sorted variables. We call these equations "rewriting safe" because they have the property that for any terms u, v of same sort, $u(\Rightarrow^*; \Leftarrow^*)v$, i.e., u and v rewrite to the same term, if and only if $eq(u, v) \Rightarrow^* t$. Note, however, that they may *not* be equationally safe: one can use an equality $(\forall X, Y)\ \sigma(X) = \tau(Y)$ provable in the spec to prove that $t = f$ using the equations of eq, and then use the equations of ? to show that any two terms are equal.

We next show how the conditional specification above can be converted into a non-conditional one, claiming that this example is sufficient in order for the reader to foresee the general translation procedure:

$$\begin{aligned}
o(s(N), t, B) &= ?(eq(e(N, t, t), \mathit{false}), \mathit{true}, o(s(N), f, B)) \\
o(s(N), B, t) &= ?(eq(e(N, t, t), \mathit{true}), \mathit{false}, o(s(N), B, f)) \\
o(0, B, B') &= \mathit{false} \\
e(s(N), t, B) &= ?(eq(o(N, t, t), \mathit{false}), \mathit{true}, e(s(N), f, B)) \\
e(s(N), B, t) &= ?(eq(o(N, t, t), \mathit{true}), \mathit{false}, e(s(N), B, f)) \\
e(0, B, B') &= \mathit{true}
\end{aligned}$$

One can show that $u \Rightarrow^* v$ in the original conditional specification if and only if $u_t \Rightarrow^* v'$ in the non-conditional one, where u_t modifies u by translating each operation accordingly and adding t for the new arguments, and v' is a term over the new syntax which becomes v by forgetting the extra-arguments of the new operations. Thus, one can automatically "wrap" a conditional rewriting engine into a non-conditional one.

We have manually applied this automatic translation on several examples using world's fastest rewriting engines, and have always seen significant improvements in efficiency. For the example above, the non-conditional version is about 10 times faster in ELAN and about 3 times faster in Maude than the conditional one. Moreover, there is an almost (modulo only syntactic considerations) one-to-one mapping of interesting properties between the original and the translated specification, such as confluence and termination. In fact, the non-conditional rewriting system conceptually performs exactly the same computations as the original one, except that it maintains the control context explicitly in the term to rewrite rather than implicitly in the rewriting engine. We believe that this translation opens the door to novel developments of rewriting engines, which can now focus on optimizing non-conditional rewriting, which is by far simpler to implement and optimize than conditional rewriting.