Reflecting Data: Formally Correct Results for Efficient (and Dirty) Algorithms

Lucas Dixon

University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, UK.
l.dixon@ed.ac.uk
http://dream.inf.ed.ac.uk/

Abstract. We describe an approach to writing efficient algorithms in fully formal proof systems by reflecting the data the algorithm uses, but not the algorithm itself. We illustrate the approach with an efficient algorithm for normalisation of arithmetic terms. Our approach ensures the correctness of the algorithm's result without extending the trusted code of the proof system and without having to prove any properties about the algorithm itself, such as its termination. This approach allows correct results to be ensured even for 'dirty' algorithms - such as those which lack a proof of termination.

1   Introduction

This paper presents an approach to implementing efficient algorithms that manipulate mathematical objects within a fully formal system. A common view is that fully formal tactic-based systems are not sufficiently extensible to implement efficient computer-algebra algorithms. The aim of this paper is to show that, contrary to this common belief, it is possible to provide efficient implementations with results that have been proved to be correct. We believe this provides a strong argument that fully-formal proof systems can enjoy efficient and correct computer-algebra algorithms.

We consider fully-formal proof systems to be those with a fixed and trusted logical kernel of inference rules. This is the so called LCF-style, also referred to as tactic-based theorem proving. Examples include systems such as Isabelle [9] and HOL [4]. Extensions to the system are made by writing tactics which combine the primitive inference rules with previously derived theorems to derive new theorems. This ensures that extensions do not endanger the system's correctness. Following this methodology, our approach avoids introducing any additional code that needs to be trusted. The key trick we employ is to observe that, to verify the result, only the representations used by an algorithm need to characterised formally. In contrast to approaches based on computational reflection or program extraction, such as those used by MetaPrl [8], Coq [2], Isabelle [1], we avoid the need to represent the algorithm in the proof system. Thus we avoid needing to prove properties about the algorithm, such as its termination. Furthermore,

we avoid needing extend the trusted code by to include the code-extraction machinery. Instead, the algorithm can be written outside the logical kernel as a tactic that justifies its steps using a representation within the proof system's object logic.

We note that while you do less, you also get less: within Coq and Isabelle, you also prove that your algorithm terminates and that it covers all cases. Our approach lets the algorithm behave badly, but ensures that the results are still right. In the rest of this paper, we introduce an example problem, that of term normalisation, outline other approaches, describe an efficient algorithm, and then describe our approach, illustrated by how it applies to term normalisation. Finally we conclude and remark on further work.

2 An Example Problem: Term Normalisation

This work was initially motivated by an example problem posed by Bruno Buchberger during a visit to Edinburgh in June 2005. He suggested that, without extending the logical kernel of an LCF-style proof system, an efficient normalisation algorithm for Peano arithmetic terms involving zero, addition and successor cannot be defined. In particular, given a term such as "$a + (Suc\ (0 + b) + (c + (Suc\ d)))$", an efficient normalisation algorithm can be written by traversing the term tree once. The idea is that the traversal builds a list of the non-zero elements and counts the number of successors. The result is a normalised representation of a term with the successors on the outside, i.e. "$(2, [a, b, c, d]) \equiv Suc(Suc(a + b + c + d))$". The efficiency of such an algorithm is linear with respect to the size of the term.

It is clear that the same level of efficiency is not achieved by a naive algorithm based on rewriting the term. The naive rewriting based approach would be to use the rewrite rules:

$$a + (Suc\ b) \Rightarrow Suc(a + b)$$
$$(Suc\ a) + b \Rightarrow Suc(a + b)$$
$$0 + b \Rightarrow b$$
$$b + 0 \Rightarrow b$$

and normalisation is achieved by exhaustive rewriting. However, such an algorithm will move successors out one step at a time. The overall runtime will be asymptotically equal to the number of successors multiplied by the size of the term, multiplied by the time taken to find the rule's left hand side in the term. This is polynomial in the term size, significantly worse than the linear time for a single term traversal.

The question is how can we implement the efficient algorithm in a fully formal proof system? Before we present our approach, we briefly discuss other approaches.

2.1 Related Work

Verification of Oracles

For problems where the result can be verified quickly, but the computation of that result is expensive, it is possible to perform the computation outside the prover and simply verify the result within the prover [6]. Example's that use this technique include factorisation and integration. However, this approach is only useful for problems where the verification of the result is significantly quicker than the computation. For many problems, including our problem of normalising arithmetic terms, it is not clear how the correct result can be used as an oracle without re-performing the full computation. The approach we suggest applies more generally than the verification of oracles.

Formalised Algorithms and Computational Reflection

Perhaps the most obvious way to get an efficient computer algebra algorithm within a formal proof system is to write it within the logic of the proof system. Then algorithm can be applied by simply evaluating it within the proof system. Doing this naively will have a significant linear slowdown. However, efficient evaluation techniques has been demonstrated in systems such as ACL2 [5].

The main drawback of this approach is that formalising algorithms is a lot of work. The user is required to prove properties about the algorithm, such as its termination, even though they may only be interested in the correctness of the result. Furthermore, certain algorithms cannot even be defined as functions within the proof system. In particular, functions within a proof system cannot recurse over the system's abstract syntax.

The way to overcome the limits of recursion on the system's syntax is to 're-flect' the syntax as object inside the prover. This involves creates an equivalent represention of the systems syntax within the system itself. This can be a lot of work, but once this is done, the algorithm can again be expressed as a function on the reflected representation. However, to apply the algorithm within the proof system requires extracting the algorithm from the formalisation so that it can be applied to the systems original syntax. Such an approach has been developed for the MetaPrl system [8]. Other techniques that extract efficient functional programs from formalised algorithms have been demonstrated in Coq [2, 7] and Isabelle [1]. In terms of the trusted code, extraction and reflection-based approaches, require significantly more code trusted.

The main difference with our approach is that we avoid needing to formalise and prove properties of the algorithm, while preserving the correctness of its results. We also avoid the need to extend the trusted code base.

3 The Efficient Algorithm

Before we describe our suggested approach, we clarify the introduced example by detailing the efficient algorithm for normalisation of arithmetic terms. We will

present the efficient algorithm over an abstract syntax. For clarity and brevity we will hide the types. This leaves us with the following datatype for terms:

```
T = App of T * T
  | Abs of T
  | Bound of int
  | Const of string
```

Using this syntax, the term $a + (Suc\ b)$ is represented by (App (App (Const '+') (Const 'a')) (App (Const 'Suc') (Const 'b'))). However, because the abstract syntax is rather unreadable, we will write terms within $[\![$ and $]\!]$, and let ourselves use a natural syntax with infix operators. With this notation, the main part of our efficient normalisation algorithm can be written as:

```
intern ⟦x + y⟧ (s,l) = intern ⟦y⟧ (intern ⟦x⟧ (s,l))
intern ⟦Suc x⟧ (s,l) = intern ⟦x⟧ (Suc 1, l)
intern ⟦0⟧ (s,l) = (s,l)
intern ⟦n⟧ (s,l) = (s,n::l)
```

where the infix function :: is the cons operator for lists. The last case is a catch-all for any other constants, variables or subterms that are outside the scope of the normalisation procedure. The intern function computes the number of successors in the term and creates a list of all non-normalisable, non-zero leaf nodes. Observe that this function cannot be expressed within the object logic of a proof-system. This is because the function is defined recursively on the object-logic's syntax. Furthermore, it evaluates to different results for the terms $[\![0 \cdot a]\!]$, where $\cdot$ represents multiplication, and $[\![0]\!]$, even though they are semantically the same within the proof system [1].

Once the intermediate representation is calculated, we extract the normalised expression for it. This is done by evaluating the following function:

```
extract (a, l) = foldr (op ⟦+⟧) ⟦a⟧ l
```

Unlike intern, this can be expressed within a fully formal proof system.

The full normalisation function is then simply the combination of the extraction with the computation of the intermediate representation:

```
normalise x = extract (intern x)
```

By using a list to represent the non-normalisable subterms of the expression, we normalise away addition's associative structure. If we wanted to keep the original structure of the term, then our intermediate representation would simply be a tree rather than a list.

---

[1] In general, this issue occurs whenever the representation function is not injective to the original representation; whenever the efficient representation removes symmetries in the original.

4 Reflecting Algorithmic Data

Our approach is to implement the efficient version of the algorithm as a tactic. Unlike functions in the object logic, a tactic can recurse on the structure of terms. To ensure correctness of the result, the tactic maintains a representation of the value being computed within the proof system. Steps that transform the representation require derived theorems to justify their correctness. However, the algorithm as a whole remains outside the proof system and thus avoids needing a proof of termination and totality. We illustrate the idea by implementing the efficient term normalisation.

Returning to the efficient normalisation algorithm, its intermediate internal representation is defined in the proof system. This is simply a pair of a natural number - the successor count - with a list of un-normalisable terms: the pair type `(nat * (nat list))`. We will call this type E.

Given the intermediate representation within the proof system, we then prove its relationship to the original representation. For our example, we define the relationship between objects of type E and those of type `nat`. This is based on two functions, firstly, the function *fin* which creates an E from a `nat`:

$$fin \ 0 \ (s,l) = (s,l) \tag{1}$$

$$fin \ (Suc \ x) \ (s,l) = fin \ x \ (Suc \ s,l) \tag{2}$$

This function takes an extra accumulator argument. This is to characterise the behaviour of the `intern` function. If `intern` was not recursive then *fin* would not need the accumulator argument.

The second function we need is a fully formal version of the `extract` function, which we will call *fex*, which creates a `nat` from an E:

$$fex \ (a,l) = foldr \ (op \ +) \ a \ l \tag{3}$$

In the proof system we then derive theorems, based on these functions, to justify the cases that make up the main part of the efficient algorithm (the `intern` function):

$$fex \ (fin \ (x+y) \ (s,l)) = fex \ (fin \ y \ (fin \ x \ (s,l))) \tag{4}$$

$$fex \ (fin \ (Suc \ x) \ (s,l)) = fex \ (fin \ x(Suc \ s,l)) \tag{5}$$

$$fex \ (fin \ 0 \ (s,l)) = fex \ (s,l) \tag{6}$$

$$fex \ (fin \ x \ (s,l)) = fex \ (s,x::l) \tag{7}$$

These exactly characterise the cases of `intern` under the extract function. We will use these to justify the steps taken by the efficient normalisation algorithm. Although all these theorems can be proved automatically by IsaPlanner [3], we anticipate that for more complex representations, it will be more difficult.

In addition to these properties, in order to let us introduce normalisation, we also derive:

$$fex\ (fin\ x\ (0, [])) = x \tag{8}$$

Using these we can now write the efficient normalisation procedure as a tactic. The tactic traverses a term of interest and applies (8) from right to left to initialise the normalisation procedure on a subterm. The subterm matching x is then traversed and the rules (4) - (7) are applied by the tactic using pattern matching. This mirrors exactly the `intern` part of the efficient algorithm. Once this is completed, the definition of fex is unfolded to apply the `extract` function, completing the normalisation. Because each step is justified by an equation, the final theorem has been derived fully formally.

It is then interesting to ask how this could have gone wrong. Bugs in the algorithm's implementation cannot yield a false result, however, a buggy algorithm may not terminate. Another way the algorithm can be buggy is that it does not complete the normalisation procedure. In these cases extra constants such as fex and fin will be left within the term.

5   Conclusion

The approach we have presented provides a fully formalised proof of the result of an algorithm without requiring any proofs about the algorithm's correctness. Unlike reflection techniques, properties such as totality and termination do not need to be proved and no extra trusted code is needed. In comparison with verified oracles, our approach is more general. Our approach still requires more work than just trusting novel code. In particular, we still require some proofs about the representation to justify the algorithms steps. However, for this extra work, we ensure the soundness of the results. Thus our approach falls somewhere between verification of oracles and the reflection of algorithms. We have shown that correct results can be ensured independently of the correctness of the algorithm used to derive them.

We suggest that whatever representation is used, it can be modelled in a formal proof system, and thus our approach can be used. For this example, datatypes suffice. Working with pointers is more complex, but also has an analogue within a proof system as a function from natural numbers (the pointer) to the value pointed to. Dealing with the details of more complex representations is left as further work.

To see if this provides a feasible level of efficiency for more complex algorithms is further work. However, given that this incurs only a linear slowdown we expect that it will be sufficient for many applications. Extending our approach appropriately for algorithms which use pointer-based data-structures is another area further work.

References

1. U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.
2. Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.
3. L. Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2006.
4. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
5. D. A. Greve, M. Kaufmann, P. Manolios, J S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient Execution in an Automated Reasoning Environment. *Journal of Functional Programming*, 18(1):15–46, January 2007.
6. J. Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.
7. D. Hendriks. *Metamathematics in Coq*. PhD thesis, Department of Philosophy, Utrecht University, 2003.
8. J. Hickey, A. Nogin, X. Yu, and A. Kopylov. Practical reflection for sequent logics. *Electron. Notes Theor. Comput. Sci.*, 174(5):79–94, 2007.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.