Postdoc subject of the ARC Quotient

Automatic generation of certified construction functions guaranteeing algebraic invariants on concrete data types

Frédéric Blanqui (INRIA Lorraine) and Pierre Weis (INRIA Rocquencourt)

http://quotient.loria.fr/

Concrete data types and pattern-matching are salient features of modern programming languages as powerful ways of defining and manipulating data structures. Among other things (eg. abstract syntax trees, regular expressions, DNA sequences, chemistry, cellular automata), the developments of XML greatly increases the interest in more complex pattern-matching for easily writing programs transforming or querying XML documents. Various functional programming languages are extended with XML specific data types and matching (eg. OcamlDuce). Even some well-known non-functional programming languages (eg. C, Java) are extended with such complex pattern-matching (eg. TOM[1]).

Although concrete date types are very useful in defining complex data structures, they are not always sufficient to adequately specify the data structures manipulated by the algorithms. Often, only a subset of the concrete data type is in fact used since some invariants between the components are mandatory to ensure the correctness of the program. For instance, some list have to be sorted and should never have the same element twice. The usual way to solve this problem is to use abstract data types: no constructor is declared but instead construction functions that are supposed to guarantee the invariants are used to build the values of the data structure. However, by using an abstract data type, the programmer loses the ability to do pattern-matching, although this would not harm. Indeed, to maintain the invariants, the only important thing is to make sure that the user can only build values by using the construction functions. Pierre Weis's "private" data types in OCaml solves this problem [4].

Now, many data structures use common algebraic properties as invariants. For instance, a sorted list is a particular representant of the equivalence class of lists modulo commutativity. A list without the same element twice is a particular representant of the equivalence class of lists modulo idempotence. And, as soon as various such algebraic invariants must be combined, it becomes very difficult to write correct and efficient construction functions. That is why we propose to study the automatic generation of certified construction functions guaranteeing algebraic invariants on concrete data types.

Together with Thérèse Hardin (LIP6, Paris), we already have preliminary

[1] http://tom.loria.fr

results [1] and began to develop a prototype extension of OCaml[2], called Moca, allowing algebraic invariants like commutativity, associativity, neutral elements, idempotence, involution and nilpotence. This raises many theoretical and practical problems.

From a theoretical point of view, having construction functions implementing such algebraic invariants implies that the corresponding combination of equational theories is decidable. Deciding that such a combination is decidable is not decidable in general. There has been many work on this subject. There is however an important semi-decision procedure: the Knuth-Bendix completion. Given a well-founded ordering, it consists in finding a confluent and terminating rewriting system deciding the same equational theory [2]. But even simple equations may lead to infinite systems (eg. associativity with idempotence). To go round this problem, some people developped completion procedures on schemas representing infinite sets of rules. However, to our knownledge, no implementation is available. Another problem is that completion fails as soon as an equation is orientable with no well-founded ordering, like it is the case with associativity and commutativity (AC). Again, to go round this problem, one can consider rewriting with pattern-matching modulo AC. However, rewriting modulo AC is difficult to implement and much less efficient than rewriting with syntactic matching. Therefore, we would like to see to which extent normalization with rewriting modulo AC is equivalent to normalization with rewriting by syntactic matching. A recent work in this direction is [3].

Now, from a pratical point of view, the development of Moca, the prototype extension of OCaml with algebraic invariants, raises many interesting problems. As any other program, various pragmatic aspects have to be taken into account. Moreover, in OCaml, there is not only concrete data types but also polymorphic type constructors (eg. lists, arrays), higher-order types, objects, etc. Finally, to be efficient, it seems reasonable to have maximal sharing.

Another important aspect is related to the proof of programs. In one hand, if someone uses the construction functions generated by Moca and wants to prove some properties about his own program, he needs to know the properties ensured by Moca. These properties could be generated by Moca in various formats like Why, Coq[3], Isabelle... Finally, it is important to be able to certify that the functions generated by Moca indeed satisfy these properties. One way to obtain this certification could be to specify the Moca compiler, then to state and prove its correction property using the framework and tools that are available in Focal[4] for instance. Alternatively, the same proof could be done with any other logical framework, for instance Coq. Another open path is to make the Moca compiler to systematically equip each construction function it generates with a devoted correction proof of the properties ensured by the function.

[2] `http://caml.inria.fr`
[3] `http://coq.inria.fr`
[4] `http://focal.inria.fr`

References

[1] F. Blanqui, T. Hardin, and P. Weis. On the implementation of construction functions for non-free concrete data types. In *Proc. of ESOP'07*, LNCS ?

[2] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. North-Holland, 1990.

[3] J.-P. Jouannaud. Associative commutative rewriting via flattening, 2006. Draft.

[4] P. Weis. Private constructors in OCaml. Caml Weekly News `http://alan.petitepomme.net/cwn/2003.07.01.html#5` and `http://alan.petitepomme.net/cwn/2003.07.08.html#7`, 2003.