Noname manuscript No.
(will be inserted by the editor)

Code as Data – Creating Programs Which Create Programs With Maude

Emanuel Kitzelmann

Before I started to implement my inductive programming algorithm IGOR 2 [2], which I had developed as part of my doctoral research, I had used LISP to implement inductive programming (IP) systems. LISP was a reasonable choice since IP is all about dealing with code as data. In particular, IP is concerned with synthesizing programs from incomplete specifications such as input/output examples or computation traces. So why did I choose the term-rewriting based language MAUDE [1] to implement IGOR 2?

IGOR 2 synthesizes recursive *constructor (term rewriting) systems (CSs)* from non-recursive CSs that specify the desired functions on parts of their domains. In the simplest case, the specifying CS consists of a number of ground rules and denotes a set of I/O examples. CSs can be seen as first-order functional programs: They consist of equations over algebraic types which are interpreted as reduction rules and may contain constructors in their heads (pattern matching). Fig. 1 shows an example of a specification and the synthesized function.

I had two (plus one) reasons for choosing MAUDE. First, MAUDE's so-called functional modules, a certain subset of the language, are an extended form of CSs. Hence IGOR 2's objects – specifications and generated programs – are valid MAUDE programs and I didn't need to care about implementing my own object language. Second, MAUDE has powerful reflection capabilities that facilitate parsing, manipulation and evaluation of MAUDE code from within MAUDE programs, just like quoted expressions in LISP can represent code, can

```
sort NList .  ***sort/type for lists of natural numbers
op nil : → NList [ctor] .                ***empty list
op _:_ : Nat NList → NList [ctor] .      ***cons
vars x, y, z : Nat .  var xs : NList .   *** variables

op last : NList → Nat .   ***signature for last

*** specification                ***synthesized solution
eq last(x:nil)     = x .         eq last(x:nil)   = x .
eq last(x:y:nil)   = y .         eq last(x:y:xs) = last(y:xs) .
eq last(x:y:z:nil) = z .
```

Fig. 1 Example last: Type and specification (input to IGOR 2) and induced solution (IGOR 2 output) in MAUDE syntax.

be manipulated by list functions and evaluated. (And third: I felt like trying something new.)

Reflection means that for all constructs of MAUDE programs (signatures, terms, equations, complete modules) data structures to represent and manipulate them are implemented in MAUDE's standard library. Meta-represented terms, equations, modules etc. are *terms* of types Term, Equation, Module etc. and can be rewritten by a MAUDE program just like any other term. For example, consider a MAUDE module, let's say a module M that contains the two equations of the synthesized solution from Fig. 1. Applying upEqs('M, false) would then yield

```
eq 'last['_:_['x:Nat,'nil.NatList]] = 'x:Nat[none] .
eq 'last['_:_['x:Nat,'_:_['y:Nat,'xs:NatList]]] =
     'last['_:_['y:Nat,'xs:NatList]][none] .
```

which is a term of type EquationSet. Also rewriting and related concepts like matching and substitutions are implemented at the meta-level. For example,

```
metaMatch(upModule('M,false), '_:_['x:Nat,'xs:NatList],
          '_:_['1.Nat, '_:_['2.Nat,'nil.NatList]], nil, 0)
```

The author is funded within the DAAD FIT-programme.

Emanuel Kitzelmann
International Computer Science Institute, Berkeley, USA
E-mail: emanuel@icsi.berkeley.edu

returns the term (of type Substitution):

```
'x:Nat← '1.Nat , 'xs:NatList ← '_:_['2.Nat,'nil.NatList] .
```

Two further features that distinguish MAUDE from other (functional) programming languages are subtypes and operator properties like associativity, commutativity etc. which (together with pattern matching) permit succinct definitions of data structures. Figure 2 shows examples for lists and sets.

```
sort NatC .                ***a sort for Nat collections
subsort Nat < NatC . ***a Nat is a Nat collection , size 1
op none : → NatC [ctor] .  ***the empty collection
***we define a constructor _,_ to build collections from
*** existing ones and make it associative and having none
***as id element; the collection now corresponds to a list
op _,_ : NatC NatC → NatC [ctor assoc id: none] .
var x : Nat .   var xs : NatC .  *** variables
***now getting the last element from a list is just
***pattern matching (like getting its first element)
op last : NatC → Nat .      eq last((xs,x)) = x .
reduce last ((1,2,3)) .  Result: 3  ***a quick test
***let's make the collection a set by adding commutativity
***and eliminating multiple instances of the same element
op _,_ : NatC NatC → NatC [ctor assoc comm id: none] .
eq ((x , x)) = x .
reduce (1,2,3,2) .  Result: (1,2,3)
***every element can be the first one due to commutativity
op member : Nat NatC → Bool .
eq member (x, (x , xs)) = true .
eq member (x, xs)       = false [owise] .
reduce member(3, (1,2,3,4)). Result: true.
```

Fig. 2 Succinctly defining data structures in MAUDE.

I covered some features that let me chose MAUDE for implementing IGOR 2, but MAUDE has much more to offer. Types can be parameterized and besides *functional modules* there are so-called *system modules*, that let you specify and implement concurrent and non-deterministic systems, and even *object-oriented modules*. MAUDE is a logical framework in which more specific languages can be modeled. It is a strictly declarative language and includes a model checker such that properties of a MAUDE program/theory can automatically be checked. The weak points of MAUDE are a rather small library with only few predefined data structures and the lack of suitable input/output handling.

References

1. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. Springer-Verlag (2007)
2. Kitzelmann, E.: A combined analytical and search-based approach for the inductive synthesis of functional programs. Künstliche Intelligenz **25**(2), 179–182 (2011)