Canonical Abstract Syntax Trees

Antoine Reilles

CNRS & LORIA
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex France

inria-00000967, version 2 - 20 Nov 2006

Abstract

This paper presents GOM, a language for describing abstract syntax trees and generating a Java implementation for those trees. GOM includes features allowing the user to specify and modify the interface of the data structure. These features provide in particular the capability to maintain the internal representation of data in canonical form with respect to a rewrite system. This explicitly guarantees that the client program only manipulates normal forms for this rewrite system, a feature which is only implicitly used in many implementations.

1 Introduction

Rewriting and pattern-matching are of general use for describing computations and deduction. Programming with rewrite rules and strategies has been proven most useful for describing computational logics, transition systems or transformation engines, and the notions of rewriting and pattern matching are central notions in many systems, like expert systems (JRule), programming languages based on rewriting (ELAN, Maude, OBJ) or functional programming (ML, Haskell).

In this context, we are developing the Tom system [10], which consists of a language extension adding syntactic and associative pattern matching and strategic rewriting capabilities to existing languages like Java, C and OCaml. This hybrid approach is particularly well-suited when describing transformations of structured entities like trees/terms and XML documents.

One of the main originalities of this system is to be data structure independent. This means that a *mapping* has to be defined to connect algebraic data structures, on which pattern matching is performed, to low-level data structures, that correspond to the implementation. Thus, given an algebraic data structure definition, it is needed to implement an efficient support for this definition in the language targeted by the Tom system, as Java or C do not provide such data structures. Tools like ApiGen [13] and Vas, which is a human readable language for ApiGen input where used previously for generating

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

such an implementation to use with Tom.

However, experience showed that providing an efficient term data structure implementation is not enough. When implementing computational logics or transition systems with rewriting and equational matching, it is convenient to consider terms modulo a particular theory, as identity, associativity, commutativity, idempotency, or more problem specific equations [9].

Then, it becomes crucial to provide the user of the data structure a way to conveniently describe such rules, and to have the insurance that only chosen equivalence class representatives will be manipulated by the program. This need shows up in many situations. For instance when dealing with abstract syntax trees in a compiler, and requiring constant folding or unboxing operators protecting particular data structures.

GOM is a language for describing multi-sorted term algebras designed to solve this problem. Like ApiGen, Vas or ASDL [15], its goal is to allow the user of an imperative or object oriented language to describe concisely the algebra of terms he wants to use in an application, and to provide an (efficient) implementation of this algebra.

Moreover, it provides a mechanism to describe normalization functions for the operators, and it ensures that all terms manipulated by the user of the data structure are normal with respect to those rules. GOM includes the same basic functionality as ApiGen and Vas, and ensures that the data structure implementation it provides are maximally shared. Also, the generated data structure implementation supports the visitor combinator [14] pattern, as the strategy language of Tom relies on this pattern.

Even though GOM can be used in any Java environment, its features have been designed to work in synergy with Tom. Thus, it is able to generate correct Tom mappings for the data structure (i.e. being *formal anchors* [6]). GOM provides a way to define computationally complex constructors for a data structure. It also ensures those constructors are used, and that no *raw* term can be constructed. Private types [8] in the OCaml language do provide a similar functionality by hiding the type constructors in a private module, and exporting construction functions. However, using private types or normal types is made explicit to the user, while it is fully transparent in GOM. MOCA, developed by Frédéric Blanqui and Pierre Weis is a tool that implements normalization functions for theories like associativity or distributivity for OCaml types. It internally uses private types to implement those normalization functions and ensure they are used, but could also provide such an implementation for GOM.

The rest of the paper is organized as follows: in Section 2, to motivate the introduction of GOM, we describe the Tom programming environment and its facilities. Section 3 presents the GOM language, its semantics and some simple use cases. After presenting how GOM can cooperate with Tom in Section 4, we expose in Section 5 the example of a prover for the calculus of structures [3] showing how the combination of GOM and Tom can help

producing a reliable and extendable implementation for a complex system. We conclude with summary and discussions in Section 6.

2 The Tom language

Tom is a language extension which adds pattern matching primitives to existing imperative languages. Pattern-matching is directly related to the structure of objects and therefore is a very natural programming language feature, commonly found in functional languages. This is particularly well-suited when describing various transformations of structured entities like, for example, trees/terms, hierarchized objects, and XML documents.

The main originality of the Tom system is its language and data-structure independence. From an implementation point of view, it is a compiler which accepts different *native languages* like C or Java and whose compilation process consists in translating the matching constructs into the underlying native language.

It has been designed taking into account experience about efficient compilation of rule-based systems [7], and allows the definition of rewriting systems, rewriting rules and strategies. For an interested reader, design and implementation issues related to Tom are presented in [10].

Tom is based on the notion of formal anchor presented in [6], which defines a mapping between the algebraic terms used to express pattern matching and the actual objects the underlying language manipulates. Thus, it is data structure independent, and customizable for any term implementation.

For example, when using Java as the host language, the sum of two integers can be described in Tom as follows:

```
Term plus(Term t1, Term t2) {
  %match(Nat t1, Nat t2) {
    x,zero   -> { return x; }
    x,suc(y) -> { return suc(plus(x,y)); }
  }
}
```

Here the definition of `plus` is specified functionally, but the function `plus` can be used as a Java function to perform addition. `Nat` is the algebraic sort Tom manipulates, which is mapped to Java objects of type `Term`. The mapping between the actual object `Term` and the algebraic view `Nat` has to be provided by the user.

The language provides support for matching modulo sophisticated theories. For example, we can specify a matching modulo associativity and neutral element (also known as list-matching) that is particularly useful to model the exploration of a search space and to perform list or XML based transformations. To illustrate the expressivity of list-matching we can define the search of a `zero` in a list as follows:

```
boolean hasZero(TermList l) {
  %match(NatList l) {
    conc(X1*,zero,X2*) -> { return true; }
  }
  return false;
}
```

In this example, *list variables*, annotated by a * should be instantiated by a (possibly empty) list. Given a list, if a solution to the matching problem exists, a `zero` can be found in the list and the function returns `true`, `false` otherwise, since no `zero` can be found.

Although this mechanism is simple and powerful, it requires a lot of work to implement an efficient data structure for a given algebraic signature, as well as to provide a *formal anchor* for the abstract data structure. Thus we need a tool to generate such an efficient implementation from a given signature. This is what tools like ApiGen [13] do.

However, ApiGen itself only provides a tree implementation, but does not allow to add behavior and properties to the tree data structure, like defining ordered lists, neutral element or constant propagation in the context of a compiler manipulating abstract syntax tree. Hence the idea to define a new language that would overcome those problems.

3 The GOM language

We describe here the GOM language and its syntax, and present an example data-structure description in GOM. We first show the basic functionality of GOM, which is to provide an efficient implementation in Java for a given algebraic signature. We then detail what makes GOM suitable for efficiently implement normalized rewriting [9], and how GOM allows us to write any normalization function.

3.1 Signature definitions

An algebraic signature describes how a tree-like data structure should be constructed. Such a description contains *Sorts* and *Operators*. *Operators* define the different node shapes for a certain *sort* by their name and the names and sorts of their children. Formalisms to describe such data structure definitions include ApiGen [13], XML Schema, ML types, and ASDL [15].

To this basic signature definition, we add the notion of *module* as a set of sorts. This allows to define new signatures by composing existing signatures, and is particularly useful when dealing with huge signatures, as can be the abstract syntax tree definition of a compiler. Figure 1 shows a simplified syntax for GOM signature definition language. In this syntax, we see that a module can import existing modules to reuse its sorts and operators definitions. Also, each module declares the sorts it defines with the **sorts** keyword, and declares

$$
\begin{array}{lcl}
\langle Gom \rangle & ::= & \langle Module \rangle \\
\langle Module \rangle & ::= & \textbf{module } \langle ModuleName \rangle \\
 & & [\langle Imports \rangle] \ \langle Grammar \rangle \\
\langle Imports \rangle & ::= & \textbf{imports } (\langle ModuleName \rangle)^* \\
\langle Grammar \rangle & ::= & \textbf{sorts } (\langle SortName \rangle)^* \\
 & & \textbf{abstract syntax } (\langle Production \rangle)^* \\
\langle Production \rangle & ::= & \langle Symbol \rangle [\ (\langle Field \rangle (, \langle Field \rangle)^*\ )] \texttt{ -> } \langle SortName \rangle \\
 & | & \langle Symbol \rangle \ \ (\ \langle SortName \rangle \ * \ ) \texttt{ -> } \langle SortName \rangle \\
\langle Field \rangle & ::= & \langle SlotName \rangle \texttt{ : } \langle SortName \rangle \\
\langle ModuleName \rangle & ::= & \langle Identifier \rangle \\
\langle SortName \rangle & ::= & \langle Identifier \rangle \\
\langle Symbol \rangle & ::= & \langle Identifier \rangle \\
\langle SlotName \rangle & ::= & \langle Identifier \rangle
\end{array}
$$

Fig. 1. Simplified GOM syntax

operators for those sorts with productions. This syntax is strongly influenced by the syntax of SDF [12], but simpler, since it intends to deal with abstract syntax trees, instead of parse trees. One of its peculiarities lies in the productions using the $*$ symbol, defining variadic operators. The notation $*$ is the same as in [1, Section 2.1.6] for a similar construction, and can be seen as a family of operators with arities in $[0, \infty[$.

We will now consider a simple example of GOM signature for booleans:

```
module Boolean
  sorts Bool
  abstract syntax
    True                    -> Bool
    False                   -> Bool
    not(b:Bool)             -> Bool
    and(lhs:Bool,rhs:Bool) -> Bool
    or(lhs:Bool,rhs:Bool)  -> Bool
```

From this description, GOM generates a Java class hierarchy where to each sort corresponds an abstract class, and to each operator a class extending this *sort* class. The generator also creates a factory class for each module (in this example, called `BooleanFactory`), providing the user a single entry point for creating objects corresponding to the algebraic terms.

Like ApiGen and Vas, GOM relies on the ATerm [11] library, which provides an efficient implementation of unsorted terms for the C and Java languages, as a basis for the generated classes. The generated data structure can then be characterized by strong typing (as provided by the *Composite* pattern used for generation) and maximal subterm sharing. Also, the generated class hierarchy does provide support for the visitor combinator pattern [14], allow-

ing the user to easily define arbitrary tree traversals over GOM data structures using high level constructs (providing congruence operators).

3.2 Canonical representatives

When using abstract data types in a program, it is useful to also define a notion of canonical representative, or ensure some invariant of the structure. This is particularly the case when considering an equational theory associated to the terms of the signature, such as associativity, commutativity or neutral element for an operator, or distributivity of one operator over another one.

Considering our previous example with boolean, we can consider the De Morgan rules as an equational theory for booleans. De Morgan's laws state $\overline{A \vee B} = \overline{A} \wedge \overline{B}$ and $\overline{A \wedge B} = \overline{A} \vee \overline{B}$. We can orient those equations to get a confluent and terminating rewrite system, suitable to implement a normalization system, where only boolean atoms are negated. We can also add a rule for removing duplicate negation. We obtain the system:

$$\begin{aligned} \overline{A \vee B} &\to \overline{A} \wedge \overline{B} \\ \overline{A \wedge B} &\to \overline{A} \vee \overline{B} \\ \overline{\overline{A}} &\to A \end{aligned}$$

GOM's objective is to provide a low level system for implementing such normalizing rewrite systems in an efficient way, while giving the user control on how the rules are applied. To achieve this goal, GOM provides a *hook* mechanism, allowing to define arbitrary code to execute before, or replacing the original construction function of an operator. This code can be any Java or Tom code, allowing to use pattern matching to specify the normalization rules. To allow *hooks* definitions, we add to the GOM syntax the definitions for *hooks*, and add $\langle OpHook \rangle$ and $\langle FactoryHook \rangle$ to the productions:

$$\begin{aligned} \langle FactoryHook \rangle &::= \textbf{factory \{ } \langle TomCode \rangle \textbf{ \}} \\ \langle OpHook \rangle &::= \langle Symbol \rangle \textbf{ : } \langle Operation \rangle \textbf{ \{ } \langle TomCode \rangle \textbf{ \}} \\ \langle Operation \rangle &::= \langle OperationType \rangle \ \textbf{ (} (\langle Identifier \rangle)^* \textbf{)} \\ \langle OperationType \rangle &::= \textbf{make} \mid \textbf{make_before} \mid \textbf{make_after} \\ &\quad \mid \textbf{make_insert} \mid \textbf{make_after_insert} \mid \textbf{make_before_insert} \\ \langle TomCode \rangle &::= \langle \ldots \rangle \end{aligned}$$

A *factory hook* $\langle FactoryHook \rangle$ is attached to the module, and allows to define additional Java functions. We will see in Section 5.3 an example of use for such a *hook*. An *operator hook* $\langle OpHook \rangle$ is attached to an operator definition, and allows to extend or redefine the construction function for this operator. Depending on the $\langle OperationType \rangle$, the hook redefines the construction function (**make**), or insert code before (**make_before**) or after (**make_after**) the construction function. Those *hooks* take as many arguments as the operator they modify has children. We also define operation types with an appended **insert**, used for variadic operators. Those hooks only take two arguments, when the

operator they apply to is variadic, and allow to modify the operation of adding one element to the list of arguments of a variadic operator.

Such *hooks* can be used to define the boolean normalization system:

```
module Boolean
  sorts Bool
  abstract syntax
    True                    -> Bool
    False                   -> Bool
    not(b:Bool)             -> Bool
    and(lhs:Bool,rhs:Bool) -> Bool
    or(lhs:Bool,rhs:Bool)  -> Bool
    not:make(arg) {
      %match(Bool arg) {
        not(x)   -> { return `x; }
        and(l,r) -> { return `or(not(l),not(r)); }
        or(l,r)  -> { return `and(not(l),not(r)); }
      }
      return `make_not(arg);
    }
```

We see in this example that it is possible to use Tom in the *hook* definition, and to use the algebraic signature being defined in GOM in the *hook* code. This lets the user define *hooks* as rewriting rules, to obtain the normalization system. The signature in the case of GOM is extended to provide access to the default construction function of an operator. This is done here with the `make_not(arg)` call.

When using the *hook* mechanism of GOM, the user has to ensure that the normalization system the hooks define is terminating and confluent, as it will not be enforced by the system. Also, combining hooks for different equational theories in the same signature definition can lead to non confluent systems, as combining rewrite systems is not a straightforward task.

However, a higher level strata providing completion to compute normalization functions from their equational definition, and allowing to combine theories and rules could take advantage of GOM's design to focus on high level tasks, while getting maximal subterm sharing, strong typing of the generated code and *hooks* for implementing the normalization functions from the GOM strata. GOM can then be seen as a reusable component, intended to be used as a tool for implementing another language (as ApiGen was used as basis for ASF+SDF [2]) or as component in a more complex architecture.

4 The interactions between GOM and Tom

The GOM tool is best used in conjunction with the Tom compiler. GOM is used to provide an implementation for the abstract data type to be used in a

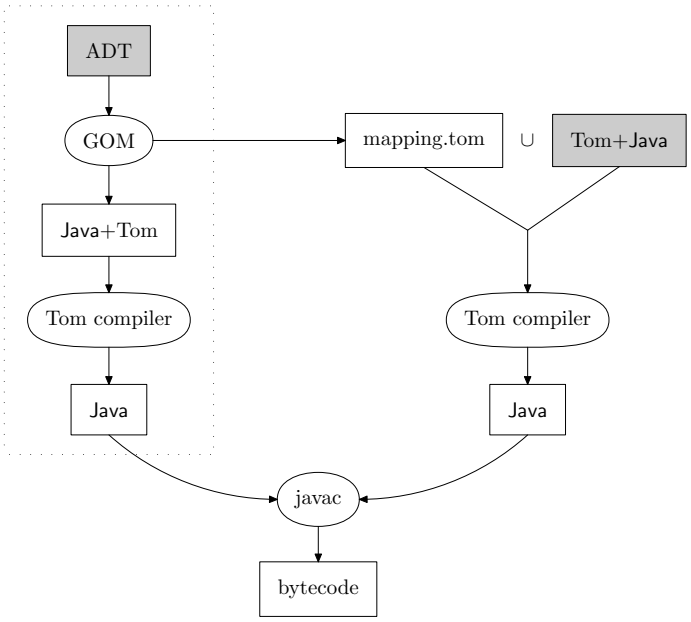


Fig. 2. The interaction between Tom and GOM

Tom program. The GOM data structure definition will also contain the description of the invariants the data structure has to preserve, by the mean of *hooks*, such that it is ensured the Tom program will only manipulate terms verifying those invariants. Starting from an input datatype signature definition, GOM generates an implementation in Java of this data structure (possibly using Tom internally) and also generates an anchor for this data structure implementation for Tom (See Figure 2). The users can then write code using the match construct on the generated mapping and Tom compiles this to plain Java. The dashed box represents the part handled by the GOM tool, while the grey boxes highlight the source files the user writes. The generated code is characterized by strong typing combined with a generic interface and by maximal sub-term sharing for memory efficiency and fast equality checking, as well as the insurance the hooks defined for the data structure are always applied, leading to canonical terms. Although it is possible to manually implement a data structure satisfying those constraints, it is difficult, as all those features are strongly interdependent. Nonetheless, it is then very difficult to let the data structure evolve when the program matures while keeping those properties, and keep the task of maintaining the resulting program manageable.

In the following example, we see how the use of GOM for the data structure definition and Tom for expressing both the invariants in GOM and the rewriting rules and strategy in the program leads to a robust and reliable implementation for a prover in the structure calculus.

$$\frac{}{\circ}\ \circ\downarrow \qquad \frac{S\{\circ\}}{S[a,\overline{a}]}\ ai\downarrow \qquad \frac{S([R,T],U)}{S[(R,U),T]}\ s \qquad \frac{S\langle[R,U];[T,V]\rangle}{S[\langle R;T\rangle,\langle U;V\rangle]}\ q\downarrow$$

Fig. 3. System BV

5 A full example: the structure calculus

We describe here a real world example of a program written using GOM and Tom together. We implement a prover for the calculus of structure [3] where some rules are promoted to the level of data structure invariants, allowing a simpler and more efficient implementation of the calculus rules. Those invariants and rules have been shown correct with respect to the original calculus, leading to an efficient prover that can be proven correct. Details about the correctness proofs and about the proof search strategy can be found in [5]. We concentrate here on the implementation using GOM.

5.1 The approach

When building a prover for a particular logic, and in particular for the system BV in the structure calculus, one needs to refine the strategy of applying the calculus rules. This is particularly true with the calculus of structure, because of deep inference, non confluence of the calculus and associative-commutative structures.

We describe here briefly the system BV, to show how GOM and Tom can help to provide a robust and efficient implementation of such a system.

Atoms in BV are denoted by $a, b, c, \ldots$ Structures are denoted by $R, S, T, \ldots$ and generated by

$$S ::= \circ \mid a \mid \langle \underbrace{S; \ldots; S}_{>0} \rangle \mid [\underbrace{S, \ldots, S}_{>0}] \mid (\underbrace{S, \ldots, S}_{>0}) \mid \overline{S}$$

where $\circ$, the *unit*, is not an atom. $\langle S; \ldots; S\rangle$ is called a *seq structure*, $[S, \ldots, S]$ is called a *par structure*, and $(S, \ldots, S)$ is called a *copar structure*, $\overline{S}$ is the *negation* of the structure S. A structure R is called a *proper par structure* if $R = [R_1, R_2]$ where $R_1 \neq \circ$ and $R_2 \neq \circ$. A *structure context*, denoted as in $S\{\ \}$, is a structure with a hole. We use this notation to express the deduction rules for system BV, and will omit context braces when there is no ambiguity.

The rules for system BV are simple, provided some equivalence relations on BV terms. The seq, par and copar structures are associative, par and copar being commutative too. Also, $\circ$ is a neutral element for seq, par and copar structures, and a seq, par or copar structure with only one substructure is equivalent to its content. Then the deduction rules for system BV can be expressed as in Figure 3.

Because of the contexts in the rules, the corresponding rewriting rules can

be applied not only at the top of a structure, but also on each subterm of a structure, for implementing deep inference. Deep inference then, combined with associativity, commutativity and $\circ$ as a neutral element for seq, par and copar structures leads to a huge amount of non-determinism in the calculus. A structure calculus prover implementation following strictly this description will have to deal with this non-determinism, and handle a huge search space, leading to inefficiency [4].

The approach when using GOM and Tom will be to identify canonical representatives, or preferred representatives for equivalence classes, and implement the normalization for structures leading to the selection of the canonical representative by using GOM's *hooks*. This process requires to define the data structure first, and then define the normalization. This normalization will make sure all units $\circ$ in seq, par and copar structures are removed, as $\circ$ is a neutral for those structures. We will also make sure the manipulated structures are *flattened*, which corresponds to selecting a canonical representative for the associativity of seq, par and copar, and also that subterms of par and copar structures are ordered, taking a total order on structures, to take commutativity into account.

When implementing the deduction rule, it will be necessary to take into account the fact that the prover only manipulates canonical representatives. This leads to simpler rules, and allow some new optimizations on the rules to be performed.

5.2 The data structure

We first have to give a syntactic description of the structure data-type the BV prover will use, to provide an object representation for the *seq*, *par* and *copar* structures ($\langle R;T\rangle$, $[R,T]$ and (R,T)). In our implementation, we considered these constructors as unary operators which take a *list of structures* as argument. Using GOM, the considered data structure can be described by the following signature:

```
module Struct
  imports
  public
    sorts Struc StrucPar StrucCop StrucSeq
  abstract syntax
    o -> Struc
    a -> Struc
    b -> Struc
    c -> Struc
    d -> Struc
    ...other atom constants
    neg(a:Struc) -> Struc
    concPar( Struc* )  -> StrucPar
```

```
concCop( Struc* )  -> StrucCop
concSeq( Struc* )  -> StrucSeq
cop(copl:StrucCop) -> Struc
par(parl:StrucPar) -> Struc
seq(seql:StrucSeq) -> Struc
```

To represent structures, we define first some constant atoms. Among them, the `o` constant will be used to represent the unit $\circ$. The `neg` operator builds the negation of its argument. The grammar rule `par(StrucPar) -> Struc` defines a unary operator `par` of sort `Struc` which takes a `StrucPar` as unique argument. Similarly, the rule `concPar(Struc*) -> StrucPar` defines the `concPar` operator of sort `StrucPar`. The syntax `Struc*` indicates that `concPar` is a *variadic-operator* which takes an indefinite number of `Struc` as arguments. Thus, by combining `par` and `concPar` it becomes possible to represent the structure $[a, [b, c]]$ by `par(concPar(a,b,c))`. Note that this structure is flattened, but with this description, we could also use nested `par` structures, as in `par(concPar(a,par(concPar(b,c))))` to represent this structure. (R, T) and $\langle R; T\rangle$ are represented in a similar way, using `cop, seq`, `concCop`, and `concSeq`.

5.3 The invariants, and how they are enforced

So far, we can manipulate objects, like `par(concPar())`, which do not necessarily correspond to intended structures. It is also possible to have several representations for the same structure. Hence, `par(concPar(a))` and `cop(concCop(a))` both denote the structure `a`, as $\langle R\rangle \approx [R] \approx (R) \approx R$.

Thus, we define the canonical (prefered) representative by ensuring that

- $[]$, $\langle\rangle$ and $()$ are reduced when containing only one sub-structure: $par(concPar(x)) \to x$
- nested structures are flattened, using the rule:

$$par(concPar(a_1, \ldots, a_i, par(concPar(x_1, \ldots, x_n)), b_1, \ldots, b_j)) \\ \to par(concPar(a_1, \ldots, a_i, x_1, \ldots, x_n, b_1, \ldots, b_j))$$

- subterms are sorted (according to a given total lexical order $<$): $concPar(\ldots, x_i, \ldots, x_j, \ldots) \to concPar(\ldots, x_j, \ldots, x_i, \ldots)$ if $x_j < x_i$.

This notion of canonical form allows us to efficiently check if two terms represent the same structure with respect to commutativity of those connectors, neutral elements and reduction rules.

The first invariant we want to maintain is the reduction of singleton for *seq*, *par* and *copar* structures. If we try to build a `cop`, `par` or `seq` with an empty list of structures, then the creation function shall return the unit `o`. Else if the list contains only one element, it has to return this element. Otherwise, it will just build the requested structure. As all manipulated terms are canonical forms, we do not have for this invariant to handle the case of a structure list

containing the unit, as it will be enforced by the list invariants. This behavior can be implemented as a *hook* for the `seq`, `par` and `cop` operators.

```
par(parl:StrucPar) -> Struc
par:make (l) {
  %match(StrucPar l) {
    concPar() -> { return ‘o(); }
    concPar(x)-> { return ‘x; }
  }
  return ‘make_par(l);
}
```

This simple *hook* implements the invariant for singletons for `par`, and use a call to the Tom constructor `make_par(l)` to call the intern constructor (without the normalization process), to avoid an infinite loop. Similar hooks are added to the GOM description for `cop` and `seq` operators. We see here how the pattern matching facilities of Tom embedded in GOM can be used to easily implement normalization strategies.

The *hooks* for normalizing structure lists are more complex. They first require a total order on structures. This can be easily provided as a function, defined in a `factory` hook. The comparison function we provide here uses the builtin translation of GOM generated data structures to text to implement a lexical total order. A more specific (and efficient) comparison function could be written, but for the price of readability.

```
factory {
  public int compareStruc(Object t1, Object t2) {
    String s1 = t1.toString();
    String s2 = t2.toString();
    int res = s1.compareTo(s2);
    return res;
  }
}
```

Once this function is provided, we can define the hooks for the variadic operators `concSeq`, `concPar` and `concCop`. The hook for `concSeq` is the simplest, since the $\langle\rangle$ structures are only associative, with $\circ$ as neutral element. Then the corresponding hook has to remove the units, and flatten nested `seq`.

```
concSeq( Struc* )  -> StrucSeq
concSeq:make_insert(e,l) {
  %match(Struc e) {
    o()              -> { return l; }
    seq(concSeq(L*)) -> { return ‘concSeq(L*,l*); }
  }
  return ‘make_concSeq(e,l);
}
```

This *hook* only checks the form of the element to add to the arguments of the variadic operator, but does not use the shape of the previous arguments. The *hooks* for `concCop` and `concPar` are similar, but they do examine also the previous arguments, to perform sorted insertion of the new argument. This leads to a sorted list of arguments for the operator, providing a canonical representative for commutative structures.

```
concPar( Struc* )  -> StrucPar
concPar:make_insert(e,l) {
  %match(Struc e) {
    o() -> { return l; }
    par(concPar(L*)) -> { return `concPar(L*,l*); }
  }
  %match(StrucPar l) {
    concPar(head,tail*) -> {
      if(!(compareStruc(e, head) < 0)) {
        return `make_concPar(head,concPar(e,tail*));
      }
    }
  }
  return `make_concPar(e,l);
}
```

The associative matching facility of Tom is used to examine the arguments of the variadic operator, and decide whether to call the builtin construction function, or perform a recursive call to get a sorted insertion.

As the structure calculus verify the De Morgan rules for the negation, we could write a hook for the `neg` construction function applying the De Morgan rules as in Section 3.2 to ensure only atoms are negated. This will make implementing the deduction rules even simpler, since there is then no need to propagate negations in the rules.

5.4 The rules

Once the data structure is defined, we can implement proof search in system BV in a Tom program using the GOM defined data structure by applying rewriting rules corresponding to the calculus rules to the input structure repeatedly, until reaching the goal of the prover (usually, the unit $\circ$).

Those rules are expressed using Tom's pattern matching over the GOM data structure. They are kept simple because the equivalence relation over structures is integrated in the data structure with invariants. In this example, [] and () structures are associative and commutative, while the canonical representatives we use are sorted and flattened variadic operators.

For instance, the rule s of Figure 3 can be expressed as the two rules $[(R,T),U] \to ([R,U],T)$ and $[(R,T),U] \to ([T,U],R)$, using only associative matching instead of associative commutative matching. Then, those rules are

encoded by the following match construct, which is placed into a strategy implementing rewriting in arbitrary context (congruence) to get deep inference, the c collection being used to gather multiple results:

```
%match(Struc t) {
  par(concPar(X1*,cop(concCop(R*,T*)),X2*,U,X3*)) -> {
    if(‘T*.isEmpty() || ‘R*.isEmpty() ) { }
    else {
      StrucPar context = ‘concPar(X1*,X2*,X3*);
      if(canReact(‘R*,‘U)) {
        StrucPar parR = cop2par(‘R*);
            // transform a StrucCop into a StrucPar
        Struc elt1 = ‘par(concPar(
              cop(concCop(par(concPar(parR*,U)),T*)),context*));
        c.add(elt1);
      }
      if(canReact(‘T*,‘U)) {
        StrucPar parT = cop2par(‘T*);
        Struc elt2 = ‘par(concPar(
              cop(concCop(par(concPar(parT*,U)),R*)),context*));
        c.add(elt2);
}  }  } }
```

We ensure that we do not execute the right-hand side of the rule if either R or T are empty lists. The other tests implement restrictions on the application of the rules reducing the non-determinism. This is done by using an auxiliary predicate function canReact(a,b) which can be expressed using all the expressive power of both Tom and Java in a factory hook. The interested reader is referred to [5] for a detailed description of those restrictions.

Also, the search strategy can be carefully crafted using both Tom and Java constructions, to achieve a very fine grained and evolutive strategy, where usual algebraic languages only allow breadth-first or depth-first strategies, but do not let the programmer easily define a particular hybrid search strategy. While the Tom approach of search strategies may lead to more complex implementations for simple examples (as the search space has to be handled explicitly), it allows us to define fine and efficient strategies for complex cases.

The implementation of a prover for system BV with GOM and Tom leads not only to an efficient implementation, allowing to cleanly separate concerns about strategy, rules and canonical representatives of terms, but also to an implementation that can be proven correct, because most parts are expressed with the high level constructs of GOM and Tom instead of pure Java. As the data structure invariants in GOM and the deduction rules in Tom are defined algebraically, it is possible to prove that the implemented system is correct and complete with respect to the original system [5], while benefiting from the expressive power and flexibility of Java to express non algebraic concerns (like

building a web applet for the resulting program, or sending the results in a network).

6 Conclusion

We have presented the GOM language, a language for describing algebraic signatures and normalization systems for the terms in those signatures. This language is kept low level by using Java and Tom to express the normalization rules, and by using *hooks* for describing how to use the normalizers. This allows an efficient implementation of the resulting data structure, preserving properties important to the implementation level, such as maximal subterm sharing and a strongly typed implementation.

We have shown how this new tool interacts with the Tom language. As Tom provides pattern matching, rewrite rules and strategies in imperative languages like C or Java, GOM provides algebraic data structures and canonical representatives to Java. Even though GOM can be used simply within Java, most benefits are gained when using it with Tom, allowing to integrate formal algebraic developments into mainstream languages. This integration can allow to formally prove the implemented algorithms with high level proofs using rewriting techniques, while getting a Java implementation as result.

We have applied this approach to the example of system BV in the structure calculus, and shown how the method can lead to an efficient implementation for a complex problem (the implemented prover can tackle more problems than previous rule based implementation [5]).

As the compilation process of Tom's pattern matching is formally verified and shown correct [6], proving the correctness of the generated data structure and normalizers with respect to the GOM description would allow to expand the trust path from the high level algorithm expressed with rewrite rules and strategies to the Java code generated by the compilation of GOM and Tom. This allows to not only prove the correctness of the implementation, but also to show that the formal parts of the implementation preserve the properties of the high level rewrite system, such as confluence or termination.

Acknowledgments: I would like to thank Claude Kirchner, Pierre Étienne Moreau and all the Tom developers for their help and comments. Special thanks are due to Pierre Weis and Frederic Blanqui for fruitful discussions and their help in understanding the design issues.

References

[1] Comon, H. and J.-P. Jouannaud, *Les termes en logique et en programmation* (2003), master lectures at Univ. Paris Sud.
URL
http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/articles/cours-tlpo.pdf

[2] de Jong, H. and P. A. Olivier, *Generation of abstract programming interfaces from syntax definitions*, Journal of Logic and Algebraic Programming **59** (2004), pp. 35–61.

[3] Guglielmi, A., *A system of interaction and structure*, Technical Report WV-02-10, TU Dresden (2002), To app. in ACM Transactions on Computational Logic.

[4] Kahramanoğulları, O., *Implementing system BV of the calculus of structures in maude*, in: L. A. i Alemany and P. Égré, editors, *Proceedings of the ESSLLI-2004 Student Session*, Université Henri Poincaré, Nancy, France, 2004, pp. 117–127, 16th European Summer School in Logic, Language and Information.

[5] Kahramanoğulları, O., P.-E. Moreau and A. Reilles, *Implementing deep inference in TOM*, in: P. Bruscoli, F. Lamarche and C. Stewart, editors, *Structures and Deduction* (2005), pp. 158–172, iSSN 1430-211X.

[6] Kirchner, C., P.-E. Moreau and A. Reilles, *Formal validation of pattern matching code*, in: P. Barahone and A. Felty, editors, *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming* (2005), pp. 187–197.

[7] Kirchner, H. and P.-E. Moreau, *Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories*, Journal of Functional Programming **11** (2001), pp. 207–251.

[8] Leroy, X., D. Doligez, J. Guarrigue, D. Rémy and J. Vouillon, *The Objective Caml system* (2004), http://caml.inria.fr/pub/docs/manual-ocaml/.

[9] Marché, C., *Normalized rewriting: an alternative to rewriting modulo a set of equations*, Journal of Symbolic Computation **21** (1996), pp. 253–288.

[10] Moreau, P.-E., C. Ringeissen and M. Vittek, *A Pattern Matching Compiler for Multiple Target Languages*, in: G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, LNCS **2622** (2003), pp. 61–76.

[11] van den Brand, M., H. de Jong, P. Klint and P. Olivier, *Efficient annotated terms*, Software, Practice and Experience **30** (2000), pp. 259–291.

[12] van den Brand, M., J. Heering, P. Klint and P. Olivier, *Compiling language definitions: The ASF+SDF compiler*, ACM Transactions on Programming Languages and Systems **24** (2002), pp. 334–368.

[13] van den Brand, M., P.-E. Moreau and J. Vinju, *A generator of efficient strongly typed abstract syntax trees in java*, Technical report SEN-E0306, ISSN 1386-369X, CWI, Amsterdam (Holland) (2003).

[14] Visser, J., *Visitor combination and traversal control*, in: *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (2001), pp. 270–282.

[15] Wang, D. C., A. W. Appel and J. L. Korn, *The zephyr abstract syntax description language*, in: *USENIX Workshop on Domain-Specific Languages*, 1997.