Chapter 7

ATL Applied to the Tableau Method

Author: Joost Jacob

7.1 Introduction

We have created a transformation language called ATL[1] that can be used to define facts and rules in a convenient way that is also suitable for non–programmers. We show how it can be used in a semantic tableau method to generate proofs, where the facts define axioms and the rules define theories. To do this, we create executable functions from ATL rules and these functions are then used in the implementation of a tool for the semantic tableau method. The tableau method we use is extended with equivalence classes in order to provide automatic unification. The resulting proof system is powerful but still transparant and easy to use since the axioms and theories are defined in the user's own notation. Although full automation *is* possible, the prime goal of the proof system is not sophisticated automation of proof, but rather to make it convenient for the user to define axioms and theorems and to help with routine unification of equalities, in order to arrive at a kind of proofs where the emphasis is on elegance and where a high level of abstraction

[1] ATL stands for ASCII Transformation Language, intended for manipulations of symbols that can be formed with an ASCII keyboard. The ASCII aspect is now outdated since also Unicode is supported, but the name stuck.

is encouraged. Such proofs can generally not be created automatically, the human contribution in the design of the proof is very significant. There is a growing demand for such proofs, because they are often easier to understand and thus more convincing[2] than fully automatically generated versions, and because they often help understand the subject matter better.

To demonstrate the above, and to provide a good motivation for the work in this paper, we follow an example that we encountered in the OMEGA[OME] project where we proof a property of a software model. Software is often *in*correct and using a well–known prover like PVS [ORR+96], as we did in OMEGA, only works well for a correct model. What to do if it is not correct? The tableau method, looking for a contradiction, seemed especially suitable for, abstract, high–level, software model verification.

We did investigate literature and the internet to see if we could find a tableau tool that we could use but we were unsuccessful. Our requirements for such a tool were that it would not have a steep learning curve, without for example having to learn a functional programming language like with ACL2 [KMM00], and we wished to be free in syntax notation and the tool should be preferably independent of an operating system, versioning problems, or software libraries. That is the reason we started this work. We have created a web–application, wherein the user can define rules in ATL, that can be used to derive the proof mentioned above. The development of ATL did help considerably to make it possible to create the web–application on schedule and in time and to make it easy to use. The tool also uses the Python [vR95] programming language that generates HTML for presenting the user with forms to fill in and that performs various bookkeeping tasks and is also used for the implementation of unification. We have plans to develop a scripting language that can be combined with ATL, or is an extension, to replace some of the Python software. An ATL interpreter is already available that could in *theory* already -do this, but we need to develop a very high–level language so the user can more easily define for instance proof search strategies. As it is now, the tool takes one derivation step at a time and the user is the scheduler: the user has to click on a button with a rule–name to choose a step. The web–application and the ATL tools are publicly accessible from the URL given in Sect. 7.3. In the web–application the user can derive a proof in the sandbox, or a new project can be started with new notation, new axioms and new theories.

[2] especially for the executive kind of persons

Overview of the next Sections

In the next Section we introduce ATL and we give a language theoretic basis, and an operational semantics for a new reduction rule that we need in Sect. 7.2.4. Following that is a Section about our tableau method. We demonstrate how the α– and β–rules from the semantic tableau method are defined in ATL and how the user can add new rules in possibly new notation. We also discuss the addition of equivalence classes to the tableau to add considerable unification power to the tableau method. Section 7.4 shows how we prove a property of a software model with the tableau. Section 7.5 is a conclusion with some related and future work.

7.2 ASCII Transformation Language (ATL)

The design goal for ATL was to be able to turn derivation rules for our tableau into executable functions, in a general and convenient way. An important aspect of the convenience is syntax independence, the ability to handle user–defined notation. Suppose for example that a user defines the $\alpha 4$ semantic tableau rule in the following syntax

```
~(X -> Y)
    |
  X,~Y
```

where it is clear for a human reader what the user means, especially for a reader that is familiar with semantic tableaux, but for a computer program it is not clear. The "`~`" stands for *not*, the "`->`" for implication, and the comma "`,`" for *and*, or in our case rather the separator between two sentences in a branch in the tableau.

Let us forget for a moment the meaning of this rule, and view this rule as a string–rewrite. We note that the `X` and `Y` are like *wildcard characters*. A wildcard character can be used to substitute for any other character or characters in a string[3]. The rule accepts string of the form `~(+ -> +)` where the `+` is a wildcard for one or more characters. If we could *name* the wildcards, remembering the characters they substitute, we can re–use them in the output of the rule. What we need then, is a way to distinguish wildcards

[3]This definition is from FS 1037C, a Telecommunications standard, at `http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm`

from the constant strings, and a way to name them. This idea is explained in more detail and more formal in the rest of this Section, for an early intuitive understanding we give the $\alpha 4$ rule here in ATL:

```
~(var:i -> var:j)
=def
var:i,~(var:j)
```

where `var:` is used to denote a wildcard, and the name that follows immediately is the name of the wildcard. The `=def` is used to separate rule input from rule output. This ATL encoding of the $\alpha 4$ rule can be input into an ATL function that accepts as another argument an input string, and that produces the desired transformation if the input string matches the `~(+ -> +)` pattern. The definition of this $\alpha 4$ rule is convenient for the user since the user can use an own notation of choice. The meaning of the various symbols in the notation remains the responsibility of the user. This approach is different from the approach taken by tools that enforce a notation on the user or tools that enforce a type system. We believe that our approach is more flexible, and can still be extended to the other approaches.

We have created a transformation language, called ATL, that extends the λ–calculus [Chu41] with pattern matching. The λ–calculus contains two reduction rules and our extension consists of one extra reduction rule. With the extra reduction rule it becomes practical to define transformations such as are desired for the rule–based string transformations that we need to define axioms and theorems in our tableau. The λ–calculus is Turing complete, and ATL can mimic the λ–calculus by using only rules that correspond to λ–expressions, with patterns that match only one variable, thereby turning the extra reduction rule in ATL into a dummy transformation. This means that ATL also is a universal model of computation, Turing complete, and leads to a new computational model, but that is out of scope for this paper, however what is interesting here is that as a consequence we are assured that every theorem is expressable in ATL.

7.2.1 Preliminary: λ–calculus

In λ–calculus we have functions that accept input and produce output. The λ–calculus uses the well–known notation with the λ character to distinguish a function expression from an ordinary expression that is the result of applying the function.

Suppose we have a painting function that produces the following outputs for the corresponding inputs:

```
red -> painted red
blue -> painted blue
green -> painted green
```

We can use λ-calculus to describe such a function: `Lx.painted x`. This is called a lambda–abstraction, a kind of lambda–expression. The other kind of lambda–expressions are identifiers and function application. The `L` is supposed to be a lowercase Greek λ character.[4]

We can identify a function with a name: `doPaint === Lx.painted x`. *Application* of a function to an argument is written as:

```
(doPaint red) -> painted red
```

where application of function `f` to argument `x` is written as `(f x)`, like in the LISP programming language. Identifying a function with a name is a useful abstraction, but note that this abstraction is not an official part of the λ–calculus where every function is anonymous.

The evaluation of a λ–expression is from the application of two reduction rules.

The α–reduction rule

The *α–reduction rule* says that we can consistently rename bindings of variables:

```
Lx.E -> Lz.E[z/x]
```

for any z which is neither free nor bound in E, where `E[z/x]` means the substitution of `z` for `x` for any free occurance of `x` in `E`.

[4] We do not use a real λ character here because it is difficult to get it into the *verbatim* character set that we want to use to show "code" examples. As such, the λ character itself can be considered a disadvantage of the λ–calculus since it makes it harder to use in typical source editors. It exemplifies the fact that λ–calculus was not meant for programming, it has mainly theoretical purposes.

The β–reduction rule

The *β–reduction rule* says that application of a λ–expression to an argument is the consistent replacement of the argument for the λ–expression's bound variable in its body:

```
(Lx.E)Q -> E[Q/x]
```

where `E[Q/x]` means the substitution of `Q` for `x` for any free occurrance of `x` in `E`. The Church–Rosser Theorem states that the final result of a chain of substitutions does not depend on the order in which the substitutions are performed.

7.2.2 ATL

- Like in λ–calculus, in ATL we also have functions that accept input and produce output, we call them rules. The input for a rule is a string, and the output of a rule is also a string.

The λ-expression `Lx.2x` is written in ATL as the rule `var:x =def 2var:x`. Everything before the `=def` we call the *antecedent* of the rule, everything after it we call the *consequent* of the rule. Roughly comparing ATL with λ-calculus, the rule antecedent maps to "λ", and the `=def` maps to the "." .[5]

The input for a function in λ–calculus is exactly one argument, whereas the input for an ATL rule is a string, and from this string we can extract more than one argument. The string is matched with a pattern (the rule antecedent) that extracts the arguments from the input string and binds them to variable names, like names of function formal parameters. ATL applies the language–theoretic idea of regular expressions in its design; the matching pattern is a template with variables. The consequent of an ATL rule is also a pattern like that in the antecedent, this pattern is not used for matching but for construction of the output: the variables are replaced with their bindings from the match.

For a function expression in ATL that corresponds with "`Lx.painted x`" in the paint example we write `var:x =def painted var:x` where we see

[5] In the example the string 2 is used and as such it does not mean the number 2, but it *does* after we have identified the string 2 with the Church integer, see [Chu41], defined as `Lf.(Lx.(f (f x)))`, a higher–order function that takes a function f as argument and returns the 2–fold composition $f \circ f$.

the `var:` notation to denote variables that together with the `=def` notation is all that we need to define rules. Every variable name start with `var:` and ends with a name-string, where we define the name—string like an XML–name[6].

To the α– and β–reduction rules in λ–calculus we add a γ reduction rule that reduces a *rule application* to a *consequent application*. Where the λ–calculus applies the β rule, in ATL we have to apply the γ rule first, so we have a two–step reduction instead of one step:

```
((A =def C) inputstring) -> (C frame) -> output
```

where `A =def C` is the rule and `C` is the consequent of the rule. The matching of the antecedent creates a frame[7], a set of name–value pairs, that provides a binding for the variables. The `frame` dictionary is created from the `inputstring` via a matching algorithm described below. We write a frame between curly braces like {`var:x=foo, var:y=bar`} where the name `var:x` is bound to `foo` and the name `var:y` is bound to `bar`.

We will now first show how the λ–calculus paint example is expressed in ATL. The γ rule applied to the example:

```
((var:x =def painted var:x) red) ->
                                    ((painted var:x) {var:x=red)
```

removes the antecedent and the `=def` and builds a frame from the input wherein variable `var:x` is bound to `red`.

For application of the ATL consequent we use the same parenthesis–syntax as in λ–calculus, except that the input is now a frame. The application of the consequent goes via simple substitution like in the β—reduction from λ-calculus:

```
((painted var:x) {var:x=red}) -> painted red
```

In the given example the creation of a frame in the γ step looks pointless since there is only one binding in the frame and the whole input string is the value. To see why the two–step reduction is useful, suppose that our

[6]See the XML Specification at `http://www.w3.org/TR/REC-xml`. The choice for the XML name definition is because of the Unicode support in XML, which is used in other work were we combine ATL with XML transformations. For the purpose of this paper consider name–strings just to be strings that start with a letter

[7]In this Section we use the concepts of a **frame** from [AwJS96]

input is not just the string "`red`" but the string "`bg=blue fg=red`". This is a common situation, the longer string could for example be attributes of an XML element that define foreground and background colors. This situation can be handled with the rule

```
var:y fg=var:x
=def
painted var:x
```

The γ and β rule applications

```
((var:y fg=var:x =def painted var:x) bg=blue fg=red)
->
((painted var:x) {var:y=bg=blue,var:x=red})
->
painted red
```

now also result in "`painted red`". The γ–rule matches "`var:y`" with "`bg=blue`" and "`var:x`" with "`red`", creating the frame {`var:y=bg=blue var:x=red`}. The "`fg=`" substring from the input is discarded. By adapting the antecedent we extract the desired "`red`" value from the input and bind it to `var:x`. We leave the definition of the rule consequent unchanged.

We give another example of the usefulness of the two–step reduction. The function $f(x, y) = x + y$ is written in λ–calculus as `Lx.Ly.x + y`, a higher—order function of one argument that returns a function of one argument. The γ–reduction step in ATL can eliminate some need for higher–order, this function is written more readable as "var:x var:y =def var:x + var:y", expecting an inputstring with x and y separated by whitespace. A direct translation of the λ–expression is also possible in ATL, but not recommended: "var:x =def var:y =def var:x + var:y". This one first accepts an inputstring with the value for x, and then generates a new ATL rule with that value. The new rule accepts the value for y and returns `x + y`. For the definition of axioms and theorems in our tableau method we avoid higher–order functions. An example rule for AND elimination can be defined as simple as "var:x AND var:x =def var:x", without the need for higher–order functions and with the additional benefit of constraining valid input to two equal strings separated by an `AND` string. It was this kind of rules that ATL was designed for in the first place.

7.2.3 Implementation

A library is available[8] that contains a function that takes an inpu-t-string and an ATL rule as arguments, and returns the output–string.

7.2.4 Definition of the γ–reduction

The γ—reduction maps a *rule* application with an *inputstring* parameter to a *consequent* application with a $frame$ parameter:

`(rule inputstring)` $\mapsto$ `(consequent frame)`

The rule is a string that contains the substring `=def`. The *antecedent* of the rule is everything before the `=def` and the *consequent* is everything after it, so the production of *consequent* is a simple string tail extraction. The γ–reduction results in a (possibly empty) frame if the *inputstring* is an element of the set of strings defined by the *pattern* formed by the *antecedent*. A *pattern* is an ordered list of interleaved constant strings $cs_{1..n}$ and named wildcards $v_{1..k}$ and it defines the (infinite) set of strings that can be formed by substituting every wildcard by any string. If a wildcard occurs at more than one place in the pattern, i.e., with the same name, then it has to be substituted by the same string. A wildcard v_j, with $1 \leq j \leq k$, that is substituted by a substring s of *inputstring* adds the pair $(name, s)$ to the set $frame$, where *name* is the name of the wildcard. If the substring s contains parens then it must be well–formed, i.e., all opening parens must be closed. This is an important constraint on valid bindings, it makes it possible for the user to disambiguate rules when necessary[9]. This well–formedness constraint holds also for braces and square brackets in the current ATL implementation. The tool can be configured to add a well–formedness constraint for angle brackets, or to remove for example the constraint for braces. If there is more than one way to match *inputstring* with *pattern* in this way, then we take for every v_i, with $1 \leq i \leq k$, the shortest possible match before matching

[8]Library `atl` at `http://homepages.cwi.nl/~jacob/atl/`. The mentioned function is called `transform`. Python was chosen because Python is very interoperable with other languages, Python code can for example be translated to Java byte code. The `atl.transform` function is used in our semantic tableau web–application.

[9]Consider the difference between matching "a and b and c" and "(a and b) and c" with the pattern "var:x and var:y". In the first case x will be bound to `a`, in the second case x will be bound to `(a and b)`, because `(a`, being the first match found from left–to–right, is not well–formed and therefore rejected.

v_{i+1}. If *inputstring* is not an element of the set defined by *antecedent*, then the γ–reduction returns the unchanged *rule* and inputstring.

7.3 A webapplication

In this Section we introduce the tableau method as we use it and we introduce the web–application that can be used for the method. The web–application is publicly available at `http://homepages.cwi.nl/~jacob/st/index.html`. Because it is a web–application, a user does not have to download software but always has the latest version via a browser, and browser–features like the "Back"-button are available to redo steps in a derivation, making it suitable to iteratively develop a proof. We will explain by example how to define derivation rules with ATL and how our application handles them. First we show how the standard α– and β—rule are implemented and then how the user can add his or her own rules in a notation of choice.

For the notation of the α– and β–rule we had to decide on a notation for propositional logic, it is:

```
~           NOT
&           AND
v           OR
->          IMPLIES
<->         IF AND ONLY IF
```

We did choose a text notation because the web–application has a HTML Textarea that contains the derivation tree and we want the user to be able to edit the derivation tree, so proof derivations can be varied and retried in an easier way than would be the case if we choose real mathematic symbols in a more comples user interface.

The webapplicaton consists of password–protected projects where each project has the standard α– and β–rules predefined and where new rules and notation can be added. An interested user could also experiment with a redefinition of the α– and β–rules in his own notation if so desired, but sticking to the predefined notation of choice is of course less work. There is also one project without password that is called the Sandbox.

An example of the α–4 rule as implemented in ATL was already shown in Section 7.2. If you skipped Section 7.2 but are familiar with wildcard matching, a common technique in for example shell programming, then look

at α–4 example there, the rule can be understood by `var:` being prefix–notation for a wildcard name and `=def` being a separator between input and output.

We now show how to implement a β–rule in ATL, where branching in the derivation tree occurs. The derivation tree in the textarea consists of lines with sentences, where every line is a branch in the derivation tree. This representation has the advantage that every line can now be worked on independently, a disadvantage is that in the case of a β–rule applied to a sentence, the other sentences in the branch have to be copied to a new line. In our work we found that the advantage outweighs the disadvantage, but of course this depends on the kind of proofs that you want to derive.

Beta–rule 2 rewrites implication, creating a new branch:

```
X -> Y
  / \
~X   Y
```

In ATL this rule is implemented with

```
var:i -> var:j
=def
~var:i
var:j
```

changing a one–line sentence to two lines. The web–application copies all the other sentences in a branch if a rule creates a new line like above.

Once more, it is not necessary to implement α– or β–rules yourself, they are predefined for every project since the tool is make for a tableau method.

As an example let's input modus ponens in the application, see if it handles correctly

```
p -> q, p, ~(q)
```

as is done in example project called "modus ponens" on the website. If you go to this project then you see this formula in the textarea as shown in Figure 7.1.

Pressing the Hint button will make the application try all the defined rules for that project, and suggest one that changes the contents of the textarea. This suggestion is shown beside the Hint button after clicking on it. The application finds the β2–rule as expected, and pressing the b2 button that implements it results in

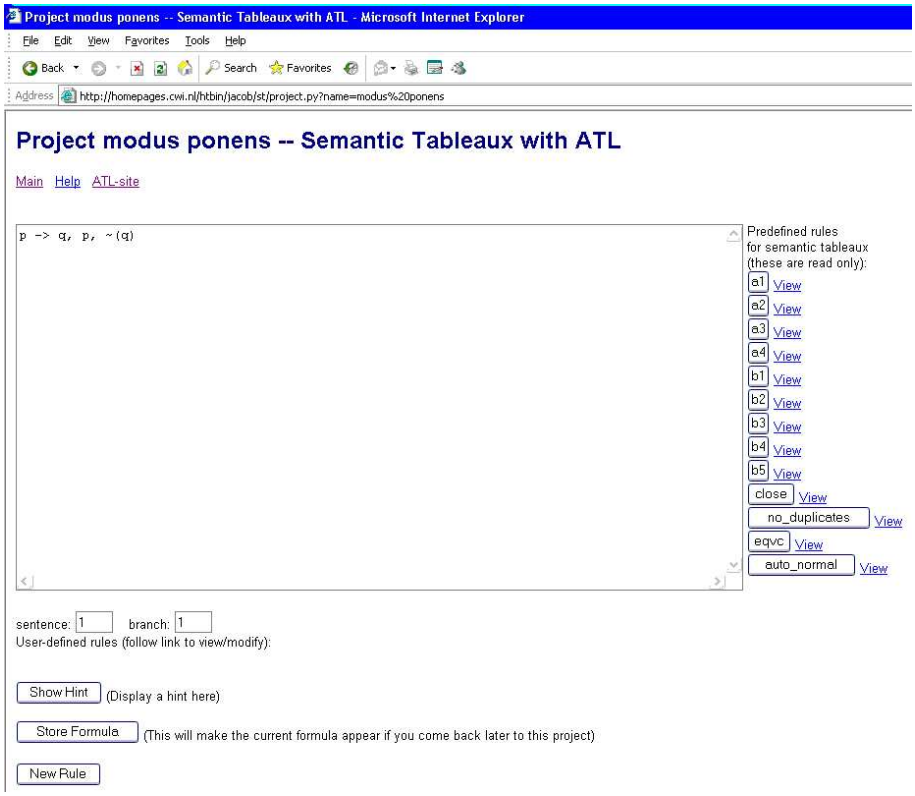

Figure 7.1: A project window

```
~(p),p,~(q)
q,p,~(q)
```

in the textarea. We see how the $\beta 2$–rule was applied to the first sentence and how the second and third sentences were copied to the new branches. Pressing Hint again will suggest "Close" for the first branch (line), since it contains the contradiction *notp*, written as `~(p)`, and p. Two clicks on the Close button will result in the line "`The proof succeeds.`".

The tool supplies input boxes to choose the branchnumber and the sentencenumber that you wish to apply a rule to. Sometimes only a branchenumber is enough, suppose we want to close the second branch in the example above, we must supply the number 2 in the input box labeled with `branch:`.

Besides the rule buttons, the "Hint" button and a button to add "New Rules" there is a button called "no duplicates" that removes duplicate sentences from a branch, e.g., `p,q,p` will become `p,q`. An explanation of a

button is available via the "View" link beside it, in the case of an ATL rule the link leads to the ATL definition. The buttons "eqvc" and "auto normal" are explained in the rest of this Section.

7.3.1 Equivalence classes and conflict relations

We have extended the basic tableau with propositional logic as presented so far, with identity and equivalence classes, in order to add the power of unification. If a branch contains a sentence that is an identity, notation `p = q`, then the user can use the "eqvc" button on that branch to turn this identity into an equivalence class. The notation for an equivalence class is `[p;q]`, with `p` being the canonical representative of the class. Equivalence classes are added upon creation to the end of the line that represents a branch and they can contain more than two instances. If there is already a canonical representative for one of the terms in the identity, then only the new term will be added to the equivalence class. The negation of an equivalence class, notation `~[p;q]` can only contain two terms, it is a conflict relation, and it is also generated via the "eqvc" button, replacing a `~(p = q)` sentence.

With the "auto normal" button every occurence of `q` is rewritten as `p` if `p` is a canonical representative of `q`. While not always strictly necessary for a proof, this can help considerable in makeing a proof better readable, since long formulas can be represented with a single short symbol.

Equivalence classes and conflict relations are useful for unification: a contradictory situation results, after appropriate use of the "eqvc" and "auto normal" buttons, in a conflict–relation of the form `~[P;P]` and that means the branch can be closed.

Here are some examples of using the "eqvc" button:

```
p, x = y                          --->    p, [x;y]

p, z = x, [x;y]                   --->    p, [x;y;z]

p, [x;i], [y;j], ~(x = y)         --->    p, [x;i], [y;j], ~[x;y]

p, [x;i], [y;j], ~(i = j)         --->    p, [x;i], [y;j], ~[x;y]
```

As can be seen, a created conflict relation uses the canonical representatives where possible: in the last line `~[i;j]` is not created but `~[x;y]`, since `i` is in an equivalence class with `x` as representative, and likewise for `j` and `y`.

7.3.2 Adding user–defined rules

So far we have only seen formulas in propositional logic and equivalences. In the next section we will see a proof of a property of software and for that proof we wish to use additional notation for the concepts that occur in the proof. New axioms and theories that are needed for the proof must be definable in that new notation, and that is possible with ATL by defining them as rules.

Figure 7.2: Editing a rule

The web–application has a "New Rule" button that leads to a form like in Figure 7.2 where the user can input a new rule and give it a name. After rule submission with the "Send" button, the new rule appears as a new button with the rulename in the project window under the label "User defined rules".

In the next section we show how such new rules can be applied to a derivation. It is the responsibility of the user that a new rule "makes sense", i.e., has a correct semantics in the context of the notation used. This is more similar to modifying formulas in a proof on paper than to many proof tools where it is not immediately possible to add rules, but only to add axioms. Also, there is no such thing as typechecking a rule in ATL, since we only deal with symbolic computations. For the kind of proofs we encountered, the resulting simplicity was desirable, but your mileage may vary.

In a proof we also want to be able to instantiate axioms, they have to be instantiated with canonical representatives of the equivalence classes in the branch where the axiom is to be placed. For this purpose in the web–application we use the string `axiom` in the antecedent of a rule, the web–application then generates a new button on the project screen just like a new rule button, but now with drop down list input boxes where the user can choose how the axiom is instantiated. In the next Section there are two examples of axioms defined with ATL.

7.4 The Sieve example

In the OMEGA project[OME] we did prove several properties of software. During this work we felt the need for an easy to use tool that can apply a tableau method with equivalence classes and that resulted in the web–application as introduced. As an example we will prove the "Main sieve property" of software that models the Sieve of Erastosthenes. The software is modeled with UML classes and statemachines and was chosen as an example because it has several interesting characteristics like the dynamic creation of objects. In our proof we abstract from some implementation details and we use a new notation for properties of the software and theories and axioms about those properties.

The notation used is:

```
p           a sequence of data, a pipeline of sieve objects
            The complete list of sieve objects or any
            consecutive part of it (at least 2 elements).
sieve(p)    The sequence of data resulting from the sieve
            process applied to p.
f(p)        The first element of p
t(p)        The tail of p
o           A sieve object
??(o)       The sequence received by object o
?(o)        sent to o
!(o)        sent by o
<=          subsequence operator (infix)
~(...)      negation of ...
[a;...]     equivalence class with representative a
~[a;b]      conflict relation between representatives a and b
```

The "Main Sieve Property" can now be written as

```
!(f(t(p))) <= sieve(!(f(p)))
```

meaning that what is sent by a second sieve object in a pipeline, is a subsequence of the sieve process applied to that what is sent by the first sieve object in the pipeline. For this proof we created a project with the name `sieve`.

The proof can be derived automatically by following the "Hint" button as suggested by the tool at each step, except for the input in Step 1, the definitions in Step 2 and the axiom instantiations in Steps 5 and 7.

Besides the functionality provided by the "eqvc" and "auto normal" buttons in the tool, there are four new user defined rules necessary for the proof:

The sieve monotonicity rule:

```
??(var:n) <= ?(var:n)
=def
sieve(??(var:n)) <= sieve(?(var:n))
```

The subsequence transitivity rule:

```
var:p <= var:q,var:q <= var:s
=def
var:p <= var:s
```

Two rules are axioms, they will be instantiated with canonical representatives.

The fifo axiom

```
axiom
=def
??(var:f) <= ?(var:f)
```

The sieve IO axiom

```
axiom
=def
!(var:sio) <= sieve(??(var:sio))
```

The two axioms are to be instantiated with sieve objects in a pipeline as defined.

The proof itself[10]:

[10] right–justified indentation is used in Step 7 to fit the line on the page

```
    Step 1: Take the negation for the semantic tableau method
~(!(f(t(p))) <= sieve(!(f(p))))

    Step 2: Add definitions
~(!(f(t(p))) <= sieve(!(f(p)))), o = f(t(p)), k = f(p), ?(f(t(p))) = !(f(p))

    Step 3: (Hint) eqvc
~(!(f(t(p))) <= sieve(!(f(p)))), [o;f(t(p))], [k;f(p)], [?(f(t(p)));!(f(p))]

    Step 4: (Hint) auto normal
~(!(o) <= sieve(?(o))), [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 5: fifo AXIOM var:f=o
??(o) <= ?(o),~(!(o) <= sieve(?(o))), [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 6: (Hint) RULE sieve monotonicity
sieve(??(o)) <= sieve(?(o)), ~(!(o) <= sieve(?(o))),  [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 7: Sieve IO AXIOM var:sio=o
!(o) <= sieve(??(o)), sieve(??(o)) <= sieve(?(o)), ~(!(o) <= sieve(?(o))),
                                                    [o;f(t(p))], [k;f(p)], [?(o);!(k)]

    Step 8: (Hint) RULE subsequence transitivity
!(o) <= sieve(?(o)), ~(!(o) <= sieve(?(o))), [o;f(t(p))], [k;f(p)], [?(o);!(k)]
    Step 9: (Hint) close: The proof succeeds.
```

7.5 Related work and the future

The Main Sieve Property was also proven with PVS [ORR+96] in the OMEGA project, but that was much more work and resulted in a much longer proof that could not be summarized in a short readable form. The PVS theories incorporated much more detail about the software, and this was necessary to be able to use PVS, so it is unfair to say that the PVS proof was longer or worse. Although about the same topic, the PVS proof was very different and targeted at different goals. Our ATL approach is hard to compare with work in literature, were we find mostly full–blown theorem provers, among many others there are PVS, ACL2 [KMM00] and Isabelle [Pau94]: they are much more powerful and sophisticated. But the sophistication leads to some problems with using those tools in practice: although for instance Isabelle is aiming for human–readable proofs, their success in this respect is questionable, and the, excellently written, ACL2 documentation for example is honest enough to state that it would take months for a highly educated person, familiar with LISP, to use their tool efficiently. The tableau approach with ATL is more like writing a proof on paper, but with the computer helping with the bookkeeping, checking for various types

of errors, and providing repetitious tasks like unification. We could not find literature on applying the tableau method to high–level and human–readable proofs like we like to have for software properties, but the tableau world is not our field of specialization and we are very interested if there are people who can point out relevant work that we overlooked.

With respect to future work we would like to replace much of the functionality that is now implemented in the programming language Python in the web–application with a high–level special purpose scripting language that captures ATL rules by name. Such a language can then also be used to implement proof search stategies.

In the OMEGA project there was also a proof on paper created for the example, and the notation chosen in the ATL proof was based on that. By using ATL however we found that some of the theories and axioms as used on the paper proof were not absolutely necessary, and we also were able to avoid having to use induction. Of course, the rules used in the ATL proof would *formally* be needed to be proven themselves, and there the induction would return. However, the rules are rather obvious, and we consider it a nice example of how the tableau method based on ATL can also be used to make a proof shorter or more elegant.

Acknowledgments This work was made possible by the OMEGA [OME] project, an EU sponsored research initiative, IST–2001–33522 OMEGA. Special thanks go to Frank de Boer at CWI for the enlightening discussions.