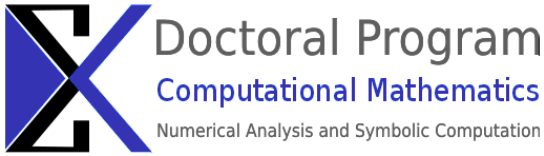


Towards the Formal Specification and Verification of Maple Programs

Muhammad Taimoor Khan and Wolfgang Schreiner

Doktoratskolleg and
Research Institute for Symbolic
Computation
Austria
July 13, 2012

JOHANNES KEPLER
UNIVERSITY LINZ | JKU

Introduction

- Formal specification respectively verification of programs written in (the most widely used) untyped computer algebra languages
  - Mathematica and **Maple**
- Develop a tool to find errors by static analysis
  - for example type inconsistencies
  - and violations of methods preconditions
- Also
  - to bridge the gap between the example computer algebra algorithm and its implementation
  - to formalize properties of computer algebra
- Demonstration example
  - Maple package *DifferenceDifferential* developed by Christian Dönch
- *MiniMaple*
  - A simple but substantial subset (with slight modifications) of Maple
  - Covers all syntactic domains of Maple but fewer expressions

Challenges of Maple type system

- Maple has not a static type system
  - It was developed as scripting language initially
  - Type annotations
  - Gauss: parameterized types (now Maple Domains)
- Type annotations are optional
  - Rises ambiguities in the type information
  - Global variables are always untyped
- Supports some non-standard types of objects
  - e.g. symbols, unions, unevaluated expressions etc.
- No clear difference between declaration and assignment
- Runtime type tests, i.e. $\mathbf{type}(E, T)$
  - Directs the control of flow in the program
- Maple values are organized in a kind of polymorphic type system

Types of objects supported in *MiniMaple*

T ::= **integer**, **float**, **rational**
| **boolean**
| **string**
| { T } # set
| **list(** T **)** # list
| **[** Tseq **]** # tuple
| **procedure[** T **](** Tseq **)**
| **void**
| I(Tseq) # named tuple
| I # user-defined type
| **uneval** # unevaluated
| **Or(** Tseq **)** # union
| **symbol**
| **anything**

Special features of the *MiniMaple* type system

- Uses only *Maple* type annotations
  - *Maple* uses them for *dynamic type checking*
  - *MiniMaple* uses them for *static type checking*
- Context (global vs local)
  - *global*
    - may introduce new identifiers by assignments
    - types of identifiers may change arbitrarily by assignments
  - *local*
    - identifiers only introduced by declarations
    - types of identifiers can only be *specialized*
- Type tests in Maple, i.e. $\mathbf{type}(E, T)$
  - Branches may have different type information for the same variable
    - track type information to allow satisfiable tests only
  - Number of loop iterations might influence the type information
    - least fixed point as an upper bound on the types of the variable
    - as a special case the declared type is the least fixed point

The derived results can also be applied to Mathematica, as Mathematica shares many concepts with Maple, e.g. same basic kinds of runtime objects

A *MiniMaple* program type checked

```
1.  status:=0;
2.  prod := proc(l::list(Or(integer,float)))::[integer,float];
3.          global status;
4.          local i::integer, x::Or(integer,float), si::integer:=1, sf::float:=1.0;
5.          # π={..., status:anything, i:integer, x:Or(integer,float), si:integer, sf:float}
6.          for i from 1 by 1 to nops(l) do
7.              x:=l[i]; status:=i;
8.              # π={..., i:integer, x:Or(integer,float),..., status:integer}
9.              if type(x,integer) then
10.             # π={..., i:integer, x:integer, si:integer, ..., status:integer}
11.                 if (x = 0) then return [si,sf]; end if; si:=si*x;
12.             elif type(x,float) then
13.             # π={..., i:integer, x:float, ..., sf:float, status:integer}
14.                 if (x < 0.5) then return [si,sf]; end if; sf:=sf*x;
15.             end if;
16.             # π={..., i:integer, x:Or(integer,float), si:integer, sf:float, status:integer}
17.         end do;
18.         # π={..., status:anything, i:integer, x:Or(integer,float), si:integer, sf:float}
19.         status:=-1; return [si,sf];
20.         end proc;
21          ...
```

Typing rule - **And** boolean expression

- **Typing rule**

$$\frac{\begin{array}{c}\pi \vdash \mathsf{E}_1{:}(\pi')\text{boolexp} \\ \mathit{canSpecialize}(\pi,\pi') \\ \pi \vdash \mathsf{E}_2{:}(\pi'')\text{boolexp} \\ \mathit{andCombinable}(\pi',\pi'')\end{array}}{\pi \vdash \mathsf{E}_1 \textbf{ and } \mathsf{E}_2{:}(\mathit{andCombine}(\pi',\pi''))\text{boolexp}}$$

- **Definitions**
  - $\mathit{canSpecialize}(\pi_1, \pi_2) \Leftrightarrow$
    $\forall(I : \tau_2) \in \pi_2 : \exists(I : \tau_1) \in \pi_1 \wedge \mathit{superType}(\tau_1, \tau_2)$
  - $\mathit{andCombinable}(\pi_1, \pi_2) \Leftrightarrow$
    $\forall(I : \tau_2) \in \pi_2 : \exists(I : \tau_1) \in \pi_1 \wedge \mathit{andCombinable}(\tau_1, \tau_2)$
  - $\mathit{andCombinable}(\tau_1, \tau_2) =$ *false*, if $[[\tau_1]] \cap [[\tau_2]] = \emptyset$
    // actually simpler
    true, otherwise

Implementation of a Type Checker

- Application of the type checker to **DifferenceDifferential**
  - A Maple package to compute bivariate difference-differential polynomials using relative Gröbner basis
  - Manual translation to *MiniMaple* program
  - No crucial errors found but some bad code parts
    - variables that are declared but not used
    - variables that have duplicate declarations

```
#comm#

*************COMMAND-SEQUENCE-ANNOTATION START*************
PI -> [
status:integer
prod:procedure[[integer,float]](list(Or(integer,float)))
result:[integer,float]
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************COMMAND-SEQUENCE-ANNOTATION END***************

Annotated AST generated.
The program type-checked correctly.
```

http://www.risc.jku.at/people/mtkhan/dk10/

A Specification Language for *MiniMaple*

- Logical formula language
  - Syntax influenced by Java Modeling Language (JML)
  - Uses Maple notations, but also has its own notations
- Supports
  - Basic formulas and expressions
  - Logical quantifiers over typed variables
    - exists
    - forall
  - Numerical quantifiers with logical condition
    - add, mul, min and max
  - Sequence quantifier
    - seq

Elements of the Specification Language

- **Mathematical theories**
  - Types
    - user defined data-types
    - abstract data types
  - Functions and predicates (declared/defined)
  - Axioms
- **Procedure specifications**
  - Pre-post conditions
  - Exceptions
  - Global variables
- **Loop specifications**
  - Invariants
  - Termination terms
- **Assertions**
  - To constrain the state of execution

Challenges of specification language for *MiniMaple*

- Support of some non-standard types of objects
  - e.g. symbols, unevaluated expressions etc.
- Additional functions and predicates
  - e.g. type test **type**(E, T)
- Specification of abstract mathematical concepts by abstract data types
  - Weaker support in current classical specification languages
  - e.g., ring, variables and ordering of a polynomial
  - ADDO as an abstract data type represented by list of tuples
    - Abstract Difference Differential Operator

An example utility procedure of *DifferenceDifferential*

```
(*@
    'type/ADDO';
    define(terms, terms(ad::ADDO)=...);
    define(getTerm, getTerm(ad::ADDO,i::nat, j::nat)=...);
    isADDO(d);
    isADDOTerm(c,n,z,e);
    ...
    assume(isADDO(d) equivalent forall(i::integer, 1<=i and i<=terms(d) implies
            isADDOTerm(getTerm(d,i,1), getTerm(d,i,2), getTerm(d,i,3), getTerm(d,i,4)));
    assume(isADDOTerm(c,n,z,e) equivalent inField(c) and isGenerator(e));
   ...
    define(power, power(a::integer,0)=1, power(a::integer,b::integer)= mul(a,1...b));
    define(maps, maps(d::DDO)=...);
   @*)
global noauto, generators, ...;
   ...
(*@
    requires 1 <= z and z <= power(2,length(noauto)) and
             forall(i::integer, 1<=i and i<=terms(maps(a)) implies isGenerator(getTerm(maps(a),i,4))) and
             forall(i::integer, 1<=i and i<=terms(maps(b)) implies isGenerator(getTerm(maps(b),i,4)));
    global EMPTY;
    ensures
       ( forall(j::integer, 1<=j and j<=nops(RESULT) implies isGenerator(RESULT[j][1],maps(a),maps(b)) and
                    RESULT[j][2] = isLT(maps(a),z) and RESULT[j][3] = isLT(maps(b),z)) )
       or
       (nops(RESULT) = 0 and ...);
    @*)

VGB := proc (z::integer, a::DDO, b::DDO)::list([symbol,list(symbol),list(symbol)]) ... return v; end proc;
```

Computes generators w.r.t. the leading terms of given difference-differential operators

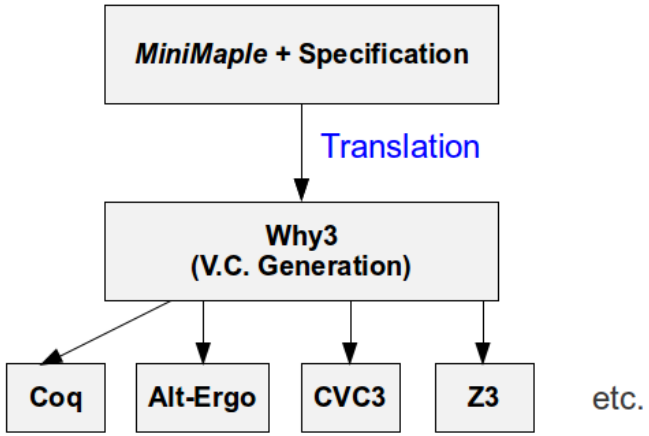
Formal semantics of *MiniMaple* and its specification language

- Defined as a state relationship between pre and post-states
- *MiniMaple* has expressions with side effects
  - Not supported in functional programming languages, e.g. Haskell
- Semantic domain of values have some non-standard types of objects
  - Symbol, Uneval and Union etc.
- *MiniMaple* supports additional functions and predicates
  - e.g. type test i.e. **type**(E, T)
- *MiniMaple* procedure is defined by an assignment command
  - Static scoping is used to evaluate a *MiniMaple* procedure
- Specification language supports abstract data types to formalize mathematical concepts

Verification conditions generated by a verification calculus for *MiniMaple* must be sound w.r.t. formal semantics

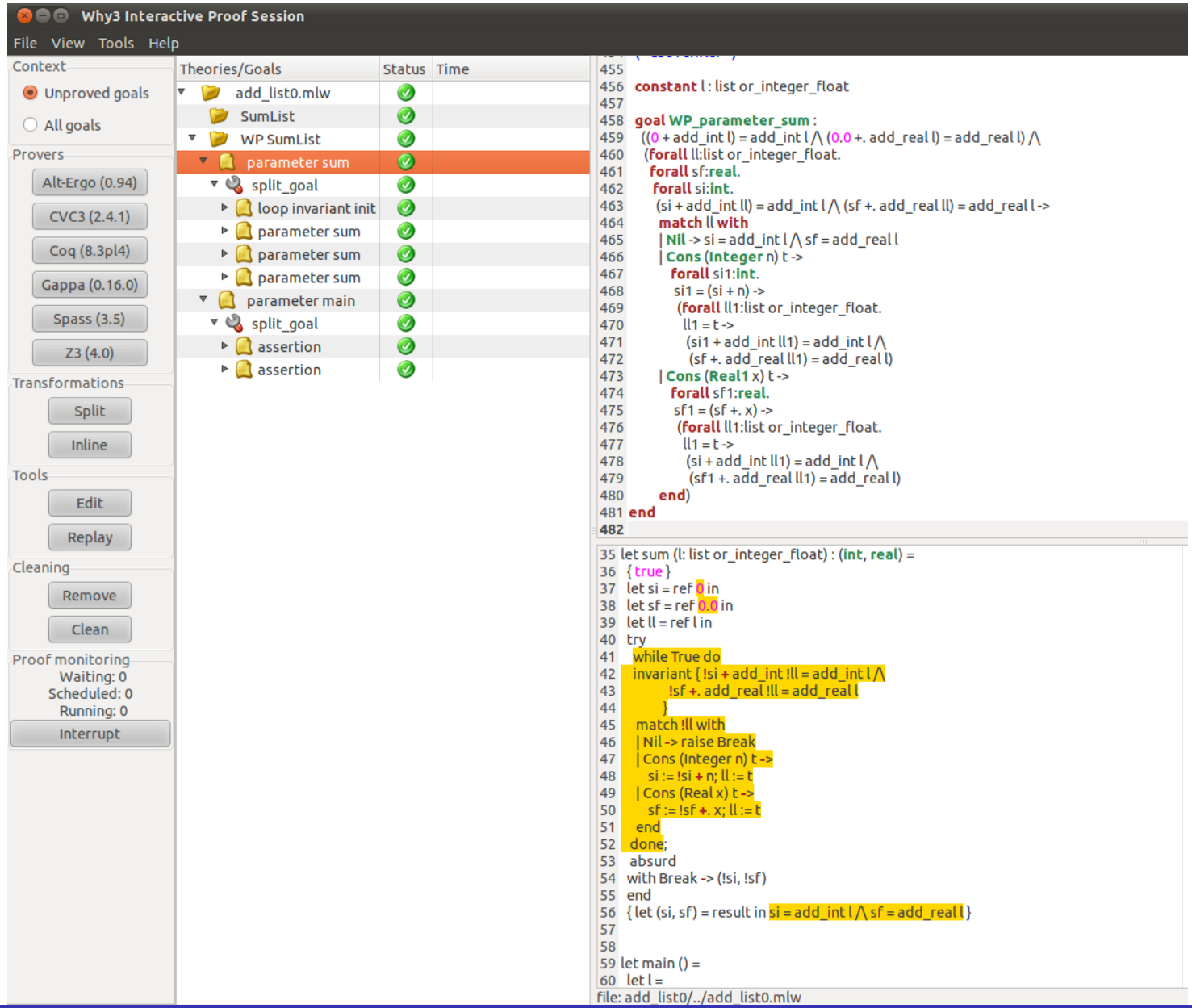
Verification calculus for *MiniMaple*

1. **Verification conditions generation** by using existing framework Why3 by LRI, France (`http://why3.lri.fr/`)
   - Verification conditions generated must be sound w.r.t. formal semantics
2. **Proving correctness of conditions** by Why3 back-end provers
   - In particular method preconditions



- Some features of Why3 (influenced by ML)
  - Supports algebraic and abstract data types
  - Also supports pattern matching
  - Has WP-based semantics
  - Provides collaborative proofs by both automated and interactive provers

An example translation of annotated *MiniMaple* to *Why3*



Current status and activities

Achievements

- Defined *MiniMaple* and its specification language
  - Formal grammars
    - implemented parsers correspondingly
  - Type systems
    - implemented type checkers correspondingly
  - Formal semantics
    - as a state relationship between pre- and post-states
    - also as a pre-requisite of our verification calculus

Current activities

- Developing a **verification calculus** for *MiniMaple*
  - Translation of annotated *MiniMaple* to Why3