Formally Specified Computer Algebra Software - DK10

Muhammad Taimoor Khan
Supervisor: Prof. Wolfgang Schreiner

Doktoratskolleg Computational Mathematics
Johannes Kepler University
Linz, Austria

January 20, 2011

Outline

Project Goals

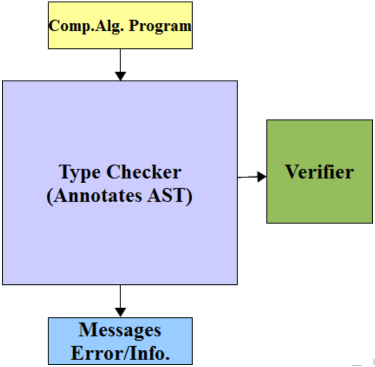
- Find errors in computer algebra programs by static analysis
  - Without executing programs (not by testing)
- Programs written in untyped computer algebra languages
  - Maple and Mathematica
  - DK11: rational parametric algebraic curves (Maple)
  - DK6: computer algebra tools for special functions in numerical analysis (Mathematica)
  - DK1: automated theorem proving (Mathematica)
- Program annotated with formal specification
  - Types and pre/post conditions of a method
- Develop a tool to find errors/inconsistencies in the annotated program
  - Type inconsistencies and violations of method preconditions

Initial Activities (Oct. 2009 to Sep. 2010)

- Course work (Oct. 2009 - Sep. 2010)
  - Computer Algebra, FM Seminar, ATP, Formal Methods in Software Development, ...
- Software Study (Nov. 2009 - Feb. 2010)
  - Maple package - *DifferenceDifferential*
  - Mathematica package - *HolonomicFunctions*
  - Mathematica package - *SetTheory‘Prover‘*
- Literature study (Oct. 2009 - Jun. 2010)
  - Type systems
    - Polymorphism
    - Abstract data types
  - Denotational semantics
  - Functional programming languages
    - Pattern matching
    - Type checking and inference
- Marktoberdorf summer school (Aug. 3 - 15, 2010)
  - Software and Systems Safety: Specification and Verification

Role of Type Checker in DK10

- Type safety as a pre-requisite of correctness
  - Type information allows only the legal use of instructions
- Easier to verify than general correctness
  - Later general verifier may use this information



A Computer Algebra Type System

- Why Maple?
  - Maple is simpler than Mathematica
  - The type system can be re-used for Mathematica
- *MiniMaple*
  - A simple computer algebra language
- Type system for *MiniMaple*
  - Typing judgements
  - Logical rules to derive the judgements
  - Auxiliary functions and predicates used in the rules
- Checker for the type system

MiniMaple

- Formal syntax
  - Prog ::= Cseq
    Cseq ::= EMPTY | C;Cseq
    C ::= ... | **if** E **then** Cseq **else** Cseq **end if**; |
    ... | **while** E **do** Cseq **end do**; | ...
    E ::= ... | E_1 **and** E_2 | ...
    ...

Example - Syntax

```
p := proc(y::integer)
    global x; local c::integer;
    if (y < 2) then
        x:=y;
    else
        x:="testString";
    end if;
    c:=y;
    while c < 10 do
        if type(x,integer) and c <= y then
            c:=c*x;
        else
            x:=c-1;
            c:=c+x;
        end if;
    end do;
end proc;
```

A Type System

- Why a type system?
  - To prevent *forbidden errors* during the execution of a program
    - *untrapped errors* completely
    - a large class of *trapped errors*
- What is a type system?
  - A *type* is (an upper bound on) the range of values of a variable
  - A *type system* is a set of formal typing rules to extract the type information from the text (syntax)
    - A simple (decidable) logic
    - $\pi \vdash$ E:(τ)*exp*
  - A type system is *sound*, if every well-typed program doesn't cause *forbidden errors*
    - if $\pi \vdash$ E:(τ)*exp* and $e \in Env_\pi$ then
      $[[\ \pi \vdash \text{E:}(\tau)exp\ ]]e \in [[\tau\ ]]$

Challenges of Maple Type System

- Maple has no complete static type system
  - It was developed as scripting language initially
  - Type annotations as predicates for runtime checking
  - Gauss: parameterized types (now Maple Domains)
- Type assignments are optional/volatile
  - Global variables are untyped
  - Raise amibiguities in the type information
- No switch-like statement for type differentiation in Maple
  - Alternatively **type**(E,T) can be used
  - Type checking is more complex

Our Approach to *MiniMaple* Type Sytem

- 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

Example - Type Checking/Specified

```
p := proc(y::integer)
    global x; local c::integer;     # π = {x : anything, y : integer, c : integer}
    if (y < 2) then
        x:=y;                       # π = {x : integer, y : integer, c : integer}
    else
        x:="testString";            # π = {x : string, y : integer, c : integer}
    end if;                         # π = {x : Or(integer, string), ...}
    c:=y;
    while c < 10 do
        if type(x,integer) and c <= y then
            c:=c*x;                 # π = {x : integer, y : integer, c : integer}
        else
            x:=c-1;
            c:=c+x;                 # π = {x : integer, y : integer, c : integer}
        end if;                     # π = {x : integer, y : integer, c : integer}
    end do;                         # π = {x : Or(integer, string), ...}
end proc;
```

Types of Objects in *MiniMaple*

T ::= **integer** | **boolean** | **string**
| **float** } *under implementation*
| **rational** }
| **uneval**
| **symbol**
| **{** T **}**
| **list(** T **)**
| **[** Tseq **]**
| I(Tseq)
| I
| **procedure[** T **](** Tseq **)** | **void**
| **Or(** Tseq **)**
| **anything**

Syntax and Top Level Judgements

- Syntax
  - Prog $\in$ Program
    Cseq $\in$ Command Sequence
    C $\in$ Command
    E $\in$ Expression
    ...
- Judgements
  - |– Prog : prog
    π,c, asgnset |– Cseq : (π_1, τset, ϵset, rflag)cseq
    π,c, asgnset |– C : (π_1, τset, ϵset, rflag)comm
    π |– E : (π')boolexp
    ...
  - Declarations
    - π, π_1 : Identifier $\rightarrow$ Type (*partial*)
    - c $\in$ {global, local}
    - asgnset, ϵset $\subseteq$ Identifier
    - τ*set* $\subseteq$ Type
    - rflag $\in$ {aret, not_aret}

Example *Expression*

- Syntactic definition
  - E ::= ... | E_1 **and** E_2 | ...
- Typing rule
  - π |– E_1:(π')boolexp *canSpecialize*(π,π')
    specialize(π,π') |– E_2:(π'')boolexp
    canSpecialize(π',π'')

    π |– E_1 **and** E_2:(*specialize*(π',π''))boolexp
- Definitions
  - $\mathit{canSpecialize}(\pi_1, \pi_2) \Leftrightarrow \forall I, \tau_1, \tau_2 : (I : \tau_1) \in \pi_1 \wedge (I : \tau_2) \in \pi_2 \Rightarrow \exists \tau_3 : \tau_3 = \mathit{superType}(\tau_1, \tau_2)$
  - $\mathit{specialize}\ (\pi_1, \pi_2) = \{(I : \tau_1) \in \pi_1 | \neg\exists (I : \tau_2) \in \pi_2\} \cup \{(I : \tau_2) \in \pi_2 | \neg\exists (I : \tau_1) \in \pi_1\} \cup \{(I : \tau_3) | \exists (I : \tau_1) \in \pi_1 \wedge \exists (I : \tau_2) \in \pi_2 \wedge \tau_3 = \mathit{superType}(\tau_1, \tau_2)\}$

Example *Expression* - Type Checking

- Type Checking
  π ={x:Or(integer,string), y:integer, c:integer}
  |– **type**(x,integer):(π'={x:integer})*boolexp* *canSpecialize*(π,π')=true
  specialize(π,π')={x:integer, y:integer, c:integer}
  |– c <= y:(π"={})*boolexp*
  canSpecialize(π',π")=true
  ____
  π ={x:Or(integer,string), y:integer, c:integer}
  |– **type**(x,integer) **and** c<=y:(*specialize*(π',π")={x:integer})*boolexp*

Example *Command*

- Syntactic definition
  - C ::= ... | **while** E **do** Cseq **end do;** | ...
- Typing rule
  - π |– E:(π')boolexp   *canSpecialize*(π,π')
    specialize(π,π'),local,asgnset |– Cseq:(π_1,τset,ϵset, rflag)cseq
    ——————————————————————————————————-

    π, c,asgnset |– **while** E **do** Cseq **end do**:
    (π ,τset,ϵset,not_aret)comm

Example *Command* - Type Checking (loop)

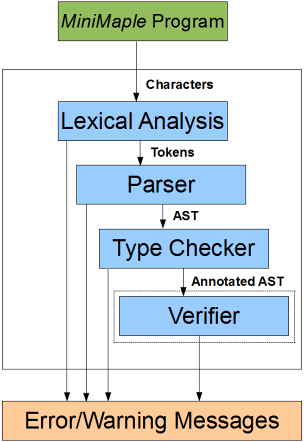
- Type Checking
  π={x:Or(integer,string),y:integer,c:integer}
  |– c < 10:(π'={})*boolexp* *canSpecialize*(π,π')=true
  {x:Or(integer,string),y:integer,c:integer},c=local,asgnset={x,c}
  |– **if type**(x,integer) **and** c <= y **then** c:=c*x; **else** x:=c-1;c:=c+x; **end if**;:
  (π_1={x:integer,y:integer,c:integer},{},{}, not_aret)*cseq*
  ———————————————————————————
  π={x:Or(integer,string),y:integer,c:integer}, c=local,asgnset={x,c}
  |– **while** c<10 **do if type**(x,integer) **and** c <= y **then** c:=c*x; **else** x:=c-1;c:=c+x; **end if**; **end do**;:
  (π={x:Or(integer,string),y:integer,c:integer} ,{},{},not_aret)*comm*

Example *Command* - Type Checking (if-else)

- Type Checking
  π={x:Or(integer,string),y:integer,c:integer}
  |– E:(π'={x:integer})*boolexp* *canSpecialize*(π,π')
  {x:integer,y:integer,c:integer},local, {x,c}
  |– c:=c*x:(π_1={x:integer,y:integer,c:integer},{},{},not_aret)cseq
  π,local,{x,c} |– x:=c-1;c:=c+x;:
  ({x:integer,y:integer,c:integer},{},{},not_aret)cseq
  ———————————————————————————
  π,c=local,asgnset={x,c}
  |– **if type**(x,integer) **and** c <= y **then** c:=c*x; **else** x:=c-1;c:=c+x; **end if**;
  ({x:integer,y:integer,c:integer},{},{},not_aret))comm

Implementation of the Type Checker

- Workflow
  - Lexical Analyser/Parser
    Input: a *MiniMaple* program
    Output: an abstract syntax tree (AST) and error/warning messages
  - Type Checker
    Input: an AST
    Output: warning, error and acknowledgement messages
    - generates an annotated AST
- Tools and technologies used
  - Java - for the development of library
  - ANTLR - for lexical analysis and parsing



Demo of the Type Checker - Test4.m

```
p := proc(y::integer)
        global x; local c::integer;
        if (y < 2) then
                x:=y;
        else
                x:="testString";
        end if;
        c:=y;
        while c < 10 do
                if type(x,integer) and c<=y then
                        c:=c*x;
                else
                        x:=c-1;
                        c:=c+1;
                end if;
        end do;
end proc;
```

Demo of the Type Checker - Type Checking(1)

```
*************|ASSIGNMENT| COMMAND-ANNOTATION START*************

PI -> [
c:integer
x:integer
y:integer
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************|ASSIGNMENT| COMMAND-ANNOTATION END**************
```

Demo of the Type Checker - Type Checking(2)

```
*************|ASSIGNMENT| COMMAND-ANNOTATION START*************

PI -> [
c:integer
x:string
y:integer
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************|ASSIGNMENT| COMMAND-ANNOTATION END***************
```

Demo of the Type Checker - Type Checking(3)

```
*************|CONDITIONAL| COMMAND-ANNOTATION START*************

PI -> [
c:integer
x:Or(integer,string)
y:integer
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************|CONDITIONAL| COMMAND-ANNOTATION END**************
```

Demo of the Type Checker - Type Checking(4)

```
*************|WHILE-LOOP| COMMAND-ANNOTATION START*************

PI -> [
x:Or(integer,string)
c:integer
y:integer
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************|WHILE-LOOP| COMMAND-ANNOTATION END***************
```

Demo of the Type Checker - Type Checking(5)

```
*************COMMAND-SEQUENCE-ANNOTATION START*************

PI -> [
p:procedure[void](integer)
c:integer
x:Or(integer,string)
y:integer
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
*************COMMAND-SEQUENCE-ANNOTATION END***************

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

Limitations of the Type Checker

- All the code is in a single Maple file
- Procedure/module definition must preceed its application
  - Alternative 1: forward declarations
  - Alternative 2: two-pass type checking
- Procedure parameter(s) and return types have to be explicitly given
  - Alternative: type inference
- Type checking terminates at very first error message
- Exhaustive testing of the type-checker is still required
  - Maple package *DifferenceDifferential* will be the test run for *MiniMaple*

Current and Future Activities

- Current status
  - Defined syntactic grammar for *MiniMaple*
    - Syntactic domains: 22 (2 pages)
  - Defined typing judgements/logical rules for *MiniMaple*
    - Typing Judgements: 23
    - Logical rules: 105 (32 pages)
  - Parser
    - Defined the EBNF grammar using ANTLR for *MiniMaple*
  - Type checker
    - Developed the user defined library
    - Classes: 159
    - Lines of Code: 15K+
  - M.T.Khan, *A Type Checker for MiniMaple*, Technical Report, RISC, JKU, Linz, January 2011 (in progress).
- Future activities
  - Next - Formal specification language