Formally Specified Computer Algebra Software - DK10

Muhammad Taimoor Khan
Supervisor: Prof. Wolfgang Schreiner

Doktoratskolleg Computational Mathematics
Johannes Kepler University
Linz, Austria

October 6, 2010

Outline

Introduction

- Project goals
  - Formal specification of programs written in untyped computer algebra languages
  - Especially to find errors/inconsistencies
    - for example violation of method preconditions
- Computer algebra software at RISC as examples
  - DK11: rational parametric algebraic curves (Maple)
  - DK6: computer algebra tools for special functions in numerical analysis (Mathematica)
  - DK1: automated theorem proving (Mathematica)

Past Activities (Oct. 2009 to Sep. 2010)

- Course work (Oct. 2009 - Jun. 2010)
  - Computer Algebra
  - Automated Theorem Proving
  - Formal Methods in Software Development
  - Formal Specification of Software
  - Formal Specification of Abstract Data Types
- Literature study (Oct. 2009 - Jun. 2010)
  - Type systems
    - Polymorphism
    - Abstract data types
  - Denotational semantics
  - Functional programming languages
    - Pattern matching
    - Type checking and inference

Past Activities (Oct. 2009 to Sep. 2010)

- Summer school and seminars (Oct. 2009 - Aug. 2010)
  - Marktoberdorf summer school (Aug. 3 - 15, 2010)
    - Software and Systems Safety: Specification and Verification
  - Formal Methods Seminar
- Software Study (Nov. 2009 - Feb. 2010)
  - Bivariate difference-differential dimension polynomials and their computation - Maple package *DifferenceDifferential*
  - Advanced applications of holonomic systems approach - Mathematica package - *HolonomicFunctions*
  - Theorema set theory prover (STP) - Mathematica package *SetTheory‘Prover‘*
- Type Checker for *MiniMaple* (Mar. 2010 - Sep. 2010)
  - Syntactic definition of *MiniMaple*
  - Defined judgements/rules for type checking the syntactic definitions

Software Study - Computer Algebra

- Bivariate difference-differential dimension polynomials and their computation
- Relative Gröbner bases computation (using M. Zhou and F. Winkler's algorithm)
- Software
  - Maple package *DifferenceDifferential* by Christian Dönch
- Potential considerations
  - Limited types used i.e. integer and list
  - Not much use of Maple libraries - mostly standalone
  - No destructive update of data structures
  - Imperative style of development

Procedural/functional Maple package

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*
- Why type system in this project?
  - 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

Need to develop such a type system first

A Type System

- What is a type system?
  - A *type* is an upper bound on the range of values of a variable that can be deduced from the text
  - A *type system* is a set of formal typing rules to extract the contents (type information) from the text (syntax)
    - A simple (decidable) logic
    - $\pi \vdash E{:}(\tau)exp$
  - A type system is *sound*, if every well-typed program doesn't cause *forbidden errors*
    - if $\pi \vdash E{:}(\tau)exp$ and $e \in Env_\pi$ then
      $[[\ \pi \vdash E{:}(\tau)exp\ ]]e \in [[\tau\ ]]$

A Type System for *MiniMaple*

- Challenges of Maple type system
  - Maple has not a static type system
    - It was developed as scripting language initially
  - Type assignments are optional/volatile
    - Rises amibiguities in the type information
    - Global variables are untyped
  - No complete static type system for Maple
    - Type annotations
    - Gauss: parameterized types (now Maple Domains)
- *MiniMaple* type system
  - Syntactic definition (language grammar) of *MiniMaple*
  - Typing rules/judgements
    - Auxiliary functions
    - Predicates
  - Stronger and weaker types
    - boolean, string, integer, ... (*stronger types*)
    - anything, Or(integer, string, ...), ... (*weaker types*)

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;
```

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 supported in *MiniMaple*

T ::= **integer**
| **boolean**
| **string**
| **{** T **}**   # set
| **list(** T **)**   # list
| **[** Tseq **]**   # record
| **procedure[** T **](** Tseq **)**
| **void**
| I(Tseq)
| I
| **unevaluated**
| **Or(** Tseq **)**   # union
| **symbol**
| **anything**

Syntax and top level Judgements

- 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 |
    ...
- Judgements
  - |– Cseq : prog
  - π,c, asgnset |– Cseq : (π_1, τset, ϵset, rflag)cseq
  - π |– E : (τ)exp
  - Definitions
    - 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 | ...
- Judgement
  - π |– E : (τ)exp
  - π |– E : (π_1)boolexp
- Conversion rules
  - $$\frac{\pi \mid\!- E:(\text{boolean})exp}{\pi \mid\!- E:(\{\})boolexp}$$
  - $$\frac{\pi \mid\!- E:(\pi_1)boolexp}{\pi \mid\!- E:(\text{boolean})exp}$$

Example *Expression*

- Typing rule
  - π |– E_1:(π')boolexp
    andCombine(π,π') |– E_2:(π'')boolexp
    andCombinable(π,π')
    andCombinable(π',π'')

    π |– E_1 **and** E_2:(andCombine(π',π''))boolexp
- Definitions
  - $\mathit{andCombinable}(\pi_1, \pi_2) \Leftrightarrow$
    $(\forall(I : \tau_2) \in \pi_2 : \exists \tau_1 : (I : \tau_1) \in \pi_1 \wedge \mathit{andCombinable}(\tau_1, \tau_2))$
  - $\mathit{andCombinable}(\tau_1, \tau_2)$ = *false*, if $[[\tau_1]] \cap [[\tau_2]] = \emptyset$
    // actually simpler
    true, otherwise

Example *Expression* - Type Checking

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

Example *Command*

- Syntactic definition
  - Cseq ::= EMPTY | C;Cseq
  - C ::= ... | **while** E **do** Cseq **end do;** | ...
- Judgement
  - π,c, asgnset |– Cseq : (π_1, τset, ϵset, rflag)comm

where

  - π, π_1 : Identifier $\rightarrow$ Type
  - c $\in$ {global, local}
  - asgnset, ϵ*set* $\subseteq$ Identifier
  - τ*set* $\subseteq$ Type
  - rflag $\in$ {aret, not_aret}

Example *Command*

- Typing rule
  - π |– E:(π')boolexp
    specialize(π,π'),local,asgnset
    |– Cseq:(π_1,τset,ϵset, rflag)cseq
    canSpecialize(π,π')

    π, c,asgnset |– **while** E **do** Cseq **end do**:
    (π ,τset,ϵset,rflag)comm
- Definitions
  - $canSpecialize(\pi_1, \pi_2) \Leftrightarrow$
    $(\forall (I : \tau_2) \in \pi_2 : \exists \tau_1 : (I : \tau_1) \in \pi_1 \wedge matchType(\tau_1, \tau_2))$
  - $$specialize\ (\pi_1, \pi_2) = \{(I : \tau_1) \in \pi_1 | \neg\exists (I : \tau_2) \in \pi_2\} \cup \\ \{(I : \tau_2) \in \pi_2 | \neg\exists (I : \tau_1) \in \pi_1\} \cup \\ \{(I : \tau_2) | \exists (I : \tau_1) \in \pi_1 \wedge \exists (I : \tau_2) \in \pi_2 \wedge matchType(\tau_1, \tau_2)\} \cup \\ \{(I : \tau_1) | \exists (I : \tau_1) \in \pi_1 \wedge \exists (I : \tau_2) \in \pi_2 \wedge matchType(\tau_2, \tau_1)\}$$

Example *Command* - Type Checking (loop)

- Type Checking
  π={x:Or(integer,string),y:integer,c:integer}
  |– c < 10:(π'={c:integer})*boolexp*
  {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*
  canSpecialize(π,π')=true
  ———————————————————————————
  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)

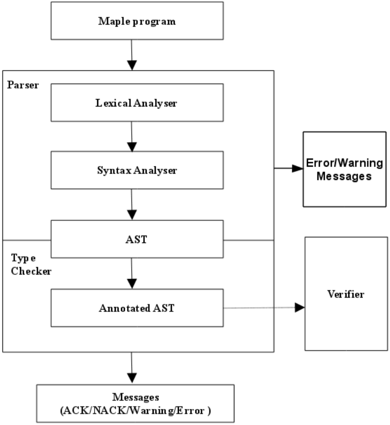
- Type Checking
  π={x:Or(integer,string),y:integer,c:integer}
  |– E:(π'={x:integer,y:integer,c:integer})*boolexp*
  {x:integer,y:integer,c:integer},local, {x,c}
  |– c:=c*x;:(π_1={x:integer,y:integer,c:integer},{},{},not_aret)cseq
  π,local,{x,c} |– x:=c-1;c:=c+x;:
  ({x:integer,y:integer,c:integer},{},{},not_aret)cseq
  canSpecialize(π,π')
  ―――――――――――――――――――――――――――――――
  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

Special features of the *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
- No switch statement in Maple
  - **type**(I,T) can be used to differentiate among types in *MiniMaple*
  - type checking is more complex

Implementation of a Type Checker

- Workflow
  - Parser
    - Input: a Maple 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



Current and Future Activities

- Current status
  - Defined syntactic grammar for *MiniMaple* (22 syntactic domains - 2 pages)
  - Defined typing rules/judgements for *MiniMaple* (105 rules and 23 judgements - 32 pages)
  - Parser
    - defined the EBNF grammar using ANTLR for *MiniMaple*
  - Type checker
    - developed the user defined library - partially complete
- Future activities
  - Functional implementation of type checker by November
  - Experiments with software fragments available at RISC
  - Next - Formal specification language