JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Formal Specification and Verification of Computer Algebra Software

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Muhammad Taimoor Khan

Angefertigt am:
Doktoratskolleg Computational Mathematics
Research Institute for Symbolic Computation

Beurteilung:
A.Univ.-Prof. Dipl.-Ing Dr. Wolfgang Schreiner (Betreuung)
Professor Dr. Renaud Rioboo

Linz, April, 2014

This research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10.

Abstract

In this thesis, we present a novel framework for the formal specification and verification of computer algebra programs and its application to a non-trivial computer algebra package. The programs are written in the language *MiniMaple* which is a substantial subset of the language of the commercial computer algebra system Maple. The main goal of the thesis is the application of light-weight formal methods to *MiniMaple* programs (annotated with types and behavioral specifications) for finding internal inconsistencies and violations of methods preconditions by employing static program analysis. This task is more complex for a computer algebra language like Maple that for conventional programming languages, as Maple supports non-standard types of objects and also requires abstract data types to model algebraic concepts and notions.

As a starting point, we have defined and formalized a syntax, semantics, type system and specification language for *MiniMaple*. For verification, we automatically translate the (types and specification) annotated *MiniMaple* program into a behaviorally equivalent program in the intermediate language Why3ML of the verification tool Why3; from the translated program, Why3 generates verification conditions whose correctness can be proved by various automated and interactive theorem provers (e.g. Z3 and Coq). Furthermore, we have defined a denotational semantics of *MiniMaple* and its specification language and proved the soundness of the translation with respect to the operational semantics of Why3ML. Finally, we discuss the application of our verification framework to the Maple package *DifferenceDifferential* developed at our institute to compute bivariate difference-differential dimension polynomials using relative Gröbner bases.

Keywords: formal methods, program verification, computer algebra software, Maple, formal specification, formal semantics

Zusammenfassung

In dieser Arbeit präsentieren wir ein neuartiges Framework für die formale Spezifikation und Verifikation von Computeralgebra-Programmen und seine Anwendung auf ein nicht-triviales Computeralgebra-Paket. Die Programme werden in der Sprache *MiniMaple* geschrieben, die eine wesentliche Teilmenge der Sprache des kommerziellen Computeralgebra-Systems Maple ist. Das Hauptziel dieser Arbeit ist die Anwendung leichtgewichtiger formaler Methoden auf (mit Typen und Verhaltens-Spezifikationen) annotierte *MiniMaple*-Programme, um interne Inkonsistenzen und Verletzungen von Methodenvorbedingungen durch Einsatz statischer Programmanalyse zu finden. Diese Aufgabe ist für eine Computeralgebra-Sprache wie Maple komplexer als für konventionelle Programmiersprachen, da Maple ungewöhnliche Typen von Objekten unterstützt und auch abstrakte Datentypen zur Modellierung von algebraischen Konzepten und Begriffen benötigt.

Als Ausgangspunkt haben wir eine Syntax, Semantik, Typ-System und Spezifikationssprache für *MiniMaple* definiert und formalisiert. Für die Verifikation übersetzen wir automatisch das (durch Typen und Spezifikationen) annotierte *MiniMaple*-Programm in ein verhaltensgleiches Programm in der Zwischensprache Why3ML des Verifikations-Werkzeugs Why3; aus dem übersetzten Programm generiert Why3 Verifikationsbedingungen, deren Korrektheit mit verschiedenen automatischen und interaktiven Beweisern (z.B. Z3 und Coq) bewiesen werden kann. Weiters haben wir eine denotationale Semantik von *MiniMaple* und ihrer Spezifikationssprache entwickelt und die Korrektheit der Übersetzung in Bezug auf die operationale Semantik von Why3 bewiesen. Schließlich zeigen wir die Anwendung unseres Verifikations-Frameworks auf das Maple-Paket *DifferenceDifferential*, das an unserem Institut für die Berechnung bivariater Differenzen-Differenzial-Dimensions-Polynome unter Verwendung relativer Gröbner Basen entwickelt wurde.

Schlüsselwörter: Formale Methoden, Programmverifikation, Computeralgebra-Software, Maple, formale Spezifikation, formale Semantik.

Acknowledgements

Foremost, I would like to express my sincere gratitude and thanks to my dissertation advisor A.Univ.-Prof. Dipl.-Ing Dr. Wolfgang Schreiner, you have been really a tremendous and wonderful mentor for me. I could not have imagined having a better supervisor for my doctoral studies. I would like to thank you for encouraging my research and providing me this opportunity to work on a very exciting project that has allowed me to grow as a young (formal methods) research scientist. Your invaluable advices on both research as well as on management of life have been priceless. This thesis is the result of your continuous guidance through very explicit and concrete explanations equipped with creative ideas. Thank you very much for your true coordination, care, long patience, feedback and all kind of support!!

I would also especially like to pay thanks to my thesis examiner and research host, Professor Renaud Rioboo (member of Centre d'Études et de Recherches en Informatique, France). His scientific remarks played a very helping role to improve the quality of my work on verification. He was a wonderful colleague who was always willing to help and give his best suggestions during my few months stay at ENSIIE. I also would like to thank for his kind hospitality and support during the stay. Furthermore, I am thankful to him for introducing me to Why3 and other teams at INRIA and CNAM.

I also want to thank the Why3 team at LRI (Laboratoire de Recherche en Informatique), Université Paris-Sud in general and Jean-Christophe Filliâtre and Claude Marché in particular for allowing me to attend their seminars, discussions and above all their technical support.

Afterwards, I am also very thankful to the secretaries, system administrators and all colleagues from RISC and DK for their scientific, technical and organizational support. I especially want to thank Doktratskolleg (DK) and Austrian Science Fund (FWF) for providing funds for such an exciting project.

A special thanks to my beloved parents. I am really short of words to express how grateful I am for their sincere support, prays and encouragement. And last but not the least, I would like to thank my wife (Najma Taimoor) and children (Noor-ul-Eman, M. Mohid Khan and M. Sarmad-Muzaffar Khan) who stood beside me throughout these years and cheered me up in the good and bad times. I am also thankful for their unconditional love and all kinds of support. Above all that, I express my deepest gratitude to Almighty Allah for being infinitely kind throughout the journey of my life.

Contents

List of Figures

1. Introduction

Since the last couple of decades, software has been seamlessly integrated in almost every technology that we use to facilitate our daily life activities. This phenomenon has given rise to the critical question of whether the software is trustworthy and reliable in order to avoid any undesired and unpleasant incidents/situations. To address this question, software reliability has evolved as a major focus area of research in computer science. In fact, behavioral errors of software are main threats to software reliability and cost sixty billion dollars per annum to the US economy as claimed in a study by NIST [146].

One approach to establish the reliability of software is by formal methods. This approach allows to specify the behavior (requirements) of a system using mathematical and logical notations which are then amenable to verification, i.e. to a formal proof that the system's implementation is correct with respect to its specification. Formal methods have been successfully applied in various domains of computer science such as the development of mission and safety critical systems software [9, 10, 60, 86, 87]. In this thesis, we consider the special application domain of computer algebra.

Computer algebra is a branch of symbolic computation that manipulates mathematical expressions and other mathematical objects , e.g. systems of polynomial equations in multiple variables. In contrast to numerical computation, which solves such systems by iterative applications, the goal of computer algebra is to derive exact solutions of such systems (possibly expressed in symbolic form including formal parameters) by symbol manipulation. A computer algebra system is a software that implements computer algebra algorithms, typically within an interactive environment, for the computation with mathematical expressions. There are various such systems, e.g. AXIOM [121], Maple [111] and Mathematica [159] that are widely used for scientific computation in various fields of computer mathematics. The languages supported by these systems for implementing computer algebra algorithms have evolved from simple scripting languages to full-fledged programming languages. However, they have typically not been developed with the consideration of formal methods, such that the correctness of computer algebra algorithms implemented in these languages poses a serious problem.

In the past few decades, there has been a lot of research on applying formal techniques to classical programming languages, e.g. C [17], Java [71] and C# [14]; variously also the application of formal methods to the languages of computer algebra systems has been investigated, e.g. for AXIOM [56], Maple [37] and FoCaLiZe [145]. However,

there has not been significant attention paid to find practical applications of formal methods to computer algebra software implemented in commercial systems such Maple and Mathematica. While computer algebra algorithms and their implementations are "essentially" correct, they often rely on some implicit assumptions, usually dependencies and side conditions which need to be considered, because otherwise the results might be erroneous or misinterpreted.

Therefore, the main goal of this thesis was to design and develop a tool to find by static program analysis behavioral errors in computer algebra programs that are written in a symbolic computation language and are annotated with types and formal specifications. Our focus was on commercial languages such as Maple and Mathematica, because the overwhelming majority of computer algebra software is written in these languages. These languages are more complicated than computer algebra languages developed in the academic context, because they have historically evolved from scripting languages whose fundamental design was not subject to the application of formal methods. Already the task of type checking programs written in these languages is complex as these languages support non-standard objects such as unevaluated expressions and polynomials and also allow dynamic type tests which direct the control flow of the program at runtime.

More concretely, we have developed a verification framework for a well-defined subset of the language of Maple, which we call *MiniMaple* [91]. To be able to demonstrate our framework in a real application scenario, we first studied various computer algebra packages developed at our institute. We then chose as a typical representative the Maple package *DifferenceDifferential* [42] that was developed at our institute by Christian Dönch without formal methods in mind. This package provides algorithms for computing difference-differential polynomials according to the method developed by M. Zhou and F. Winkler [163]. All steps of the development of our verification framework were validated by the application to this package.

Figure 1.1 gives a general overview of our verification framework. First the *MiniMaple* program is parsed to generate an abstract syntax tree (AST). Then the AST is annotated by type information and translated into a semantically equivalent program in the language Why3ML of the verification framework Why3 [21] developed at LRI, France. From this program, Why3 generates verification conditions that may be proved correct by various supported back-end provers. Throughout the whole process, all components may generate error and information messages. Further details of the project and software are available at https://www.dk-compmath.jku.at/people/mtkhan.

In more detail, to approach the goal of this thesis, we have first formally defined the syntax of *MiniMaple*. As type safety is a prerequisite of program correctness, we have formalized a type system for statically type checking *MiniMaple* programs based on the type annotations which Maple has introduced for runtime type checking. Then we have defined a specification language to formally specify the behavior of *MiniMaple* programs [100,101]. The specification language allows to formally describe

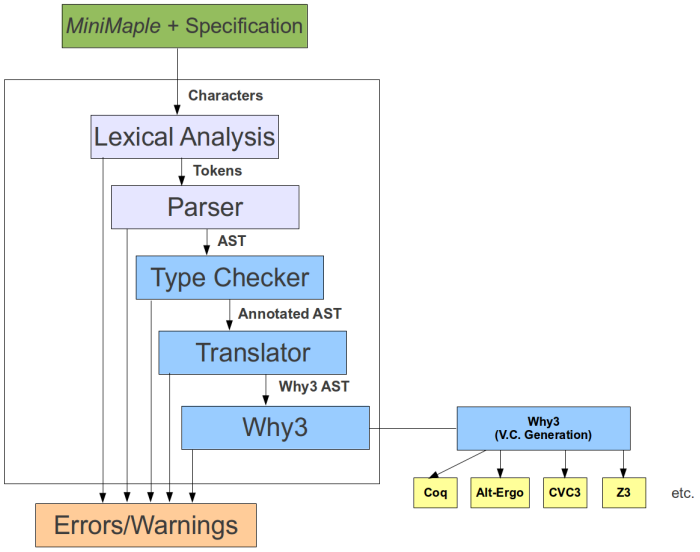

Figure 1.1.: A High-level Overview of the Verification Framework

mathematical theories (types, functions, axioms) and the behavior of procedures (pre- and post-conditions and other constraints), loops (invariants and termination terms) and commands (assertions). The language slightly extends the syntax of Maple, e.g. logical quantifiers use typed variables and numerical quantifiers (binders) use logical conditions that filter values from the specified range of a variable. Moreover, the language also supports abstract data types to specify abstract mathematical notions. We have then formalized the denotational semantics of *MiniMaple* and its specification language [93, 94].

To verify a *MiniMaple* program annotated with types and specifications, we translate this program into a semantically equivalent program in the language Why3ML of the verification tool Why3. Based on the denotational semantics of *MiniMaple* and the operational semantics of Why3ML [63], we have proved the soundness of the translation. The Why3 built-in verification conditions generator is used to produce a set of verification conditions: the pre-conditions of called procedures, the post-conditions of defined procedures, the initial establishing of loop invariants, the preservation of loop invariants after every iteration and the decreasing of termination terms.

Finally, we have applied our verification framework to achieve our original goal: the verification of the package *DifferenceDifferential*. We have type annotated the package, formally specified, translated in Why3ML program and then generated the corresponding verification conditions. Using automatic and interactive provers sup-

ported by the Why3 back-end, we have verified all low-level procedures of the package. However, in order to verify the high-level procedures of the package, we first had to develop a strategy to formally specify and verify such procedures. The problem is that these procedures are implemented using concrete data types but are specified with the help of abstract data types. Based on our strategy, we were able to successfully prove selected high-level procedures of the package.

The results of this thesis are original in that they represent to our knowledge the very first attempt to verify real-life code that was developed in a commercial computer algebra language without formal methods in mind. We have for the first-time formalized a type system, specification language and formal semantics for a subset of the language of the commercial computer algebra system Maple. Our framework is also innovative in that it supports the entire process to statically type check, specify and verify such computer algebra programs. We have also formulated a novel strategy for verifying high-level procedures in such programs and have successfully applied it to a non-trivial example from a real-life application. The contents of this thesis are in parts based on several conference and workshop publications [95, 99–102] and technical reports [91–94, 96–98].

In the following, we discuss the structure of the rest of the thesis: in Chapter 2 we sketch the state of the art of computer algebra systems, type systems, formal semantics, formal methods and their relationship.

In Chapter 3 we introduce the syntax of *MiniMaple* and discuss the language by a running example that will be subsequently used in the other chapters.

In Chapter 4 we first sketch the design of a formal type system for *MiniMaple* consisting of various kinds of judgments and rules to derive these judgments. Then we explain the formalization of the type system in more detail for several commands and expressions of *MiniMaple*. Finally, we discuss the application of type checker to our running example.

Chapter 5 introduces a specification language of *MiniMaple* whose core is a logical formula language embedded into the syntax of *MiniMaple*. Then we discuss various elements of the specification language, i.e. mathematical theories, procedure specifications, loop specifications and assertions. Finally, we demonstrate the use of the specification language by specifying our example program.

The formal semantics of *MiniMaple* and its specification language are discussed in Chapter 6. Here, first we present the formalization of some interesting semantic domains and then give the semantics of selected commands and expressions of *MiniMaple*. Finally, we describe the semantics of the specification language, i.e. its core formula language and other elements of the language.

Chapter 7 explains the various components of our verification framework. In particular, we discuss the translation functions of various constructs of *MiniMaple* and its specification language to corresponding constructs in Why3ML and demonstrate this

translation of our example program. Finally, we discuss the proof of the soundness of our translation with respect to the formal semantics of *MiniMaple* and Why3ML.

In Chapter 8, we discuss the results of the application of our verification framework to the Maple package *DifferenceDifferential*. First, we give an overview of the package and then discuss the results on type checking, specifying and verifying the package.

In the final Chapter 9, we review our work and discuss possible future extensions.

In Appendices A and B we give the complete formal definition of the syntax of *MiniMaple* and its specification language, respectively. Appendix C gives the complete type system for *MiniMaple* programs. The formal semantices of *MiniMaple* and its specification language are defined in the Appendices D and E. In Appendix F, we give the definition of the translation functions from *MiniMaple* to Why3ML. Finally, the proof of the soundness of the translation of *MiniMaple* to Why3ML is discussed in Appendix G. The complete contents of the appendices are not shown in the printout but are part of the electronic version (attached CD) of this thesis.

This research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10 in the frame of Doktoratskolleg "Computational Mathematics" at the Johannes Kepler University, Linz.

2. State of the Art

In this chapter we discuss the state of the art of computer algebra systems, formal semantics, formal methods and the relationship among these topics. The rest of the chapter is organized as follows: Section 2.1 introduces various computer algebra systems and their respective type systems. In Section 2.2 we first give an overview of various approaches for defining the formal semantics of programming languages and then describe the semantics of classical programming and scripting languages, computer algebra languages and their corresponding specification languages; finally we discuss the semantics of various intermediate verification languages. Finally, in Section 2.3 we first sketch the role of formal methods in classical programming languages and then discuss assertions checking in computer algebra languages, the application of formal methods to such languages and the related integration of theorem provers and computer algebra systems.

2.1. Computer Algebra and Type Systems

A variety of computer algebra systems has been developed, e.g. AXIOM [121], Magma [26], Sage [156], FoCaLiZe [145], Mathematica [159], Maple [111], REDUCE [134], Maxima [113] and GAP [147]. Among these the commercial systems Mathematica and Maple are the most widely used ones. In the following, we discuss some of the aforementioned computer algebra systems and their type systems.

Statically Typed Computer Algebra Languages

AXIOM [121] is a general purpose computer algebra system developed by NAG Ltd. Based on the language Aldor [7], AXIOM (forked into FriCAS [68] and OpenAXIOM [120] since 2007) is a strongly typed system [59]. In an interactive (interpreter) mode of AXIOM, a function can be declared with the corresponding signatures and defined by an assignment `==` as follows:

```
f0 : () -> List Integer; f1 : (Integer) -> List Integer
                         Type: Void

f0() == []; f1(x) == [x]
                         Type: Void
```

The types of the functions are known at compile time as shown by the corresponding function applications below:

```
f0()
            Compiling function f0 with type () -> List Integer
                      [ ]
                        Type: List Integer

f1(6)
            Compiling function f1 with type Integer -> List Integer
                      [6]
                        Type: List Integer

f1("12")
           Conversion failed in the compiled user function f3 .

           Cannot convert from type String to Integer for value
           "12"
```

The last function application `f1("12")` indicates that type of the function application is tested against the compile-time type of the function.

The data types in AXIOM are called *domains*; a class of domains is represented by a *category*. In detail, a category defines the exports of domains, i.e. which operations are provided, while the domains implement the corresponding operations. The system supports a hierarchy of parameterized domains and categories, e.g. ordered sets, rings and finite fields. Based on Aldor, AXIOM allows to write programs by combining the properties of functional, aspect-oriented and object-oriented styles.

FriCAS uses the programming language SPAD [7, 142] which is a variant of Aldor. In the following example [142], we define a category in SPAD as follows:

```
)abbrev category MYCAT MyCategory
MyCategory: Category == with
    1: %
    nth: Integer -> %
    _+: (%, %) -> %
```

The header specifies that constructor `MyCategory` is a category. The category declares signatures of three functions. The function `nth` computes the sum of integers up-to a value of the given parameter. A domain `MyDomain` belongs to `MyCategory` and thus implements the corresponding three exports of the category.

```
)abbrev domain MYDOM MyDomain
MyDomain: MyCategory with
    coerce:  Integer -> %
    coerce: % -> Integer
  == add
    Rep ==> Integer
    rep r ==> (r@%) pretend Rep
    per p ==> (p@Rep) pretend %
```

```
    coerce(p: Integer): % == per p
    coerce(r: %): Integer == rep r

    1: % == per(1)
    ((m: %) + (n: %)): % == m + n
    nth(j: Integer): % ==
        r := 1
        for i in 2..j repeat r := r + 1
        r
```

In the domain definition, first two functions are declared in addition to the exported functions of `MyCategory` and then the `add` part provides an implementation of the declared and exported functions. The symbol % refers to *this* domain which is `MyDomain`. In the implementation part (`add`), first three macros (`==>`) define the underlying representation of the elements of the domain; then the two declared operations are defined followed by the definitions of three exports functions of `MyCategory`.

Dynamically Typed Computer Algebra Languages

Magma [25, 26] is a dynamically typed computer algebra system developed at CAG, University of Sydney, Australia. The design of Magma is based on algebraic structures and morphism such that every object has a type *magma* [25]. For an algebraic structure Σ-algebra, Magma supports two-level classification of *magmas*:

1. a class of magmas that satisfies a set of relations Q is called a *variety* and is written as $\mathrm{Var}(\Sigma;\ Q)$ and
2. a class of magmas that belongs to the variety E and shares a common "representation" R is called a *category* and is written as $\mathrm{Cat}(E;R)$.

Here, a *variety* is used to specify generic functions which are independent of the representation of a magma, while a *category* realizes a *magma* in its concrete representation.

Sage [156] is a Python-based dynamically typed computer System for Algebra and Geometry Experimentation with a customized interpreter. In [75] a prototype implementation for Sage has been adapted to a hybrid type checking scheme [90], i.e. static and dynamic type checking. Though Sage has its own libraries for computations in algebra, combinatorics and calculus, Sage mainly provides an interface to several other well-known mathematical and computer algebra tools and libraries, e.g. the combinatorics libraries of GAP, PARI [123] and NTL [139], the commutative algebra tool SINGULAR [52] and the libraries of Maxima [113] for symbolic computation and calculus.

As an example a dynamically typed Sage function, g is defined as follows:

```
sage: def g(x):
         if x <2:
```

```
      return 0
    else:
      return x+"2"
```

The variable g has a type "function". The applications of function `g(‘‘test’’)` and `g(1)` compute valid results. However, the application `g(5)` gives a runtime error, as an integer cannot be concatenated with a string value:

```
sage: type (g)
<type 'function'>

sage: g("test")
'test2'

sage: g(1)
0

sage: g(5)
Error in lines 1-1
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) \
               for '+': 'Integer Ring' and '<type 'str'>'
```

FoCaLiZe [145] is a computer algebra environment based on the dynamically typed language *FoCal* (formerly known as *FoC*). The goal here was to develop a language for the co-design of a program and its corresponding proof of correctness. The language supports an object oriented modularity; code, specifications, and proofs are developed together in the same files. The absence of inconsistencies and correctness of dependencies are analyzed at compile time before the code is translated into Objective Caml, and the corresponding proofs are translated into Coq. Further details on the semantics and the verification framework of FoCaLiZe are discussed in the Sections 2.2 and 2.3, respectively.

Commercial Computer Algebra Languages

Mathematica [159] supports a wide range of packages for symbolic and numeric computations. Mathematica is a rule-based programming language for the manipulation of supported expressions. Besides the availability of optional type annotations, the language of Mathematica is not statically typed (however, in [69] the authors developed a static type system for a small subset of Mathematica). Moreover, the type annotations in Mathematica can be used to select an appropriate rule at runtime.

Maple [111] is a commercial computer algebra software developed by Symbolic Computation Group, University of Waterloo. Maple supports optional type annotations.

However, these type annotations can only be dynamically checked at runtime. Although there does not exist any complete static type system for Maple, various approaches investigated the applications of the type information in Maple for different purposes. The Maple package Gauss [116] introduced parameterized types in Maple and allowed to implement various generic algorithms in an AXIOM-like style. The system also supported parameterized types and parameterized abstract types, however these were only checked at runtime. At start, the package was introduced in Maple V Release 2 and later evolved into the *domains* package. In [37], partial evaluation is applied to Maple. The focus of the work was to apply the available type information in Maple for generating specialized programs from generic Maple programs. The language of the partial evaluator had similar syntactic constructs (but fewer expressions) as our language *MiniMaple* (see Chapter 3) and also supported a limited range of data types e.g. booleans, floats, rationals, and strings.

In comparison to the aforementioned approaches, *MiniMaple* uses the type annotations provided by Maple for static analysis. *MiniMaple* supports a substantial subset of Maple types in addition to user-defined named types as discussed in Chapter 4.

Scripting Languages and Computer Algebra Languages

The problem of statically type-checking dynamically typed computer algebra programs is related to the problem of statically type-checking scripting languages such as JavaScript [8, 150] and Ruby [70].

Object-oriented scripting languages like JavaScript are also popular because of their dynamic features such as the runtime modification of objects (e.g. addition/update of fields or methods). Since static type checking of such languages is a complex task, therefore dynamic typing is used. However, with dynamic typing some errors cannot be detected until runtime, e.g. access to non-existent members (in JavaScript, such errors are reported in a web browser).

On one hand, there are a number of studies on the design and development of type inference algorithms for statically type-checking scripting languages [8, 79, 161]. In [161], an algorithm for type inference for a subset of JavaScript is presented. The focus of the algorithm was on the inference of function types by keeping track of object/function extensions with the help of function calls and assignments. Moreover, the algorithm allowed updates to objects through flexible and unrestricted (but permitted) extensions to objects. The algorithm offered explicit and implicit extension of objects, i.e. with the help of "add" operation and of method calls, respectively. However, the goal here was to allow only legal access of objects, their defined members and operations.

On the other hand, variously annotation-based type systems have been developed for statically type checking scripting languages. For instance, in [107], Anders Hejlsberg at Microsoft developed the language TypeScript which is a typed superset of

JavaScript that provides optional type annotations for JavaScript programs. Based on these annotations, a static type system is developed which makes extensive use of type inference to allow only legal operations and behavior of JavaScript objects. The compiler translates a well-typed TypeScript program into a JavaScript program. Furthermore, the language also supports some object-oriented features; e.g. classes, modules and interfaces can be defined to understand the behavior of even already existing JavaScript components. However, the goal here was to design a language which supports development and maintenance of large scale JavaScript applications.

In comparison to the above approaches, *MiniMaple* had similar typing challenges. For example, Maple also has a polymorphic type system and does support some dynamic features, e.g. runtime type-tests which direct the control flow of a program at runtime and thus makes static type-checking more difficult. Therefore, a type system for *MiniMaple* addressed aforementioned typing issues as discussed in Chapter 4. However, still there are some fundamental differences due to the two different language paradigms.

2.2. Formal Semantics

While the syntax of a programming language is formally defined by some grammar (e.g. BNF), still not every syntactically correct program is well-typed: consequently only well-typed programs are of value. Therefore, the semantics of a programming language specifies a relation between a well-typed derivation of the program and its meaning [50].

The formal semantics of a computer programming language can be defined in an operational, axiomatic or denotational style [118]. Each style has been defined to achieve a different purpose. For example, some styles make reasoning about programs very easy; others make the meanings of programs accessible to a large audience and some can be used to make the implementation of programming languages easier. In the following, we discuss these approaches.

Operational Semantics

In the operational style introduced by Gordon Plotkin [127], the meaning of a program is described by specifying an execution of the program on an abstract machine. Furthermore, this method specifies the execution of a program with the help of rules which are directed by the syntax of the language. There are numerous variants of operational semantics, i.e. small-step, big-step and modular operational semantics. Gordon Plotkin originally introduced the small-step variant of operational semantics, which is also known as structural operational semantics. The focus here was to define the execution of a program in terms of the execution of its parts. The big-step variant defines the semantics of a program construct as a whole by hiding the intermediate

executions of parts of the construct. The modular operational semantics is a variant of structural operational semantics: in this style the rules for a programming construct are defined incrementally such that the rules do not need reformulation when new constructs are added in the language. The goal here was to provide high degree of modularity in the language which was the shortcoming of original structural operational semantics [117]. The examples of some variants of the operational semantics are discussed later in this section.

Axiomatic Semantics

In the axiomatic style formulated by Floyd and Hoare [67, 80], the meaning of a program is defined with the help of rules which specify the properties of the program. The initial goal of the style was to explain the meaning of programs with the help of "axioms" (more generally inference rules). The rules specify how to prove properties for a given program construct. After this reason, this style is known as axiomatic semantics [50]. The rule for the axiomatic semantics of a typical conditional-statement is:

$$\frac{\{P \wedge B\}\ C_1\ \{Q\}, \quad \{P \wedge \neg B\}\ C_2\ \{Q\}}{\{P\}\ \textbf{if}\ B\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{endif}\{Q\}}$$

In detail, the rule says that execution of the conditional in a pre-condition P yields a postcondition Q, iff

- either the execution of C_1 in a pre-condition P and B yields a post-condition Q
- or the execution of C_2 in a pre-condition P and $\neg B$ yields a post-condition Q.

Here a boolean expression B has no side-effects and is thus identified with a logical formula.

Denotational Semantics

In the denotational style devised by Scott-Strachey [143], the meaning of a program is defined as a mathematical function that maps the well-typed derivation of the program to its semantic value (denotation). The semantics of a syntactic phrase is formulated in terms of the denotations of its sub-phrases. Thus, the corresponding proof of the program's correctness is typically based on the proof technique of structural induction. The denotational semantics of *MiniMaple* is discussed in Chapter 6.

In the following, first we present various attempts to define the formal semantics of classical programming languages: then we describe approaches to define the semantics of scripting languages, computer algebra and specification languages and finally explain the semantics of intermediate verification languages.

Semantics of Classical Languages

In the literature, the semantices of most popular programming languages have been defined only informally. However, there have been also attempts to formally define the semantices of subsets of a few widely used classical programming languages, e.g. Ada [31], Scheme [135], C [122], Java [16, 84], C++ [40], Pascal [48, 133], Standard ML [132], Cobol [112], Prolog [58] and Algol 60 [47].

The semantics and operational design of Ada were developed based on denotational semantics of Scott-Strachey [31]. In the work, the static (compile-time) semantics was formulated only for the sequential constructs of Ada. The semantics of the functional programming language Scheme is also based on the variety of the denotational approach [135]. In [122], mathematical monads were employed to define the denotational semantics of C. In order to achieve semantic respectively operational modularity and readability, the work transformed monads into denotations.

In [24], based on transition rules of abstract state machines [23], the semantics for a substantial subset of Java was formalized which however excluded visibility of Java names and packages and class loading. Here the focus was to define the rigorous semantices of Java programs which can help to identify the design inconsistencies of the language and also can serve as the basis for the standardization of Java language. For example, the transition rule for a Java conditional was formulated as follows:

$$\begin{array}{l}\textbf{if } \mathit{task} \textbf{ is } \texttt{if } (E)\ C_1\ \texttt{else}\ C_2 \textbf{ then}\\ \quad \textbf{if } \mathit{val}(E) \textbf{ then } \mathit{task} := \mathit{fst}(C_1)\\ \qquad\qquad \textbf{else } \ \mathit{task} := \mathit{fst}(C_2)\end{array}$$

In detail, the rule states that in order to execute a conditional command, if evaluation of expression E yields *true* then the task is to execute the first command of a command sequence C_1 otherwise, the task is to execute the first command of command sequence C_2.

Later in [84], the formal semantices of object-oriented and concurrency features of Java were defined in the styles of big-step and small-step operational semantics respectively. The definition of semantics was based on informal Java specification of Sun [72]. The semantics of Java were formalized in the Centaur system which is a programming environment where from the semantics of a language one can automatically generate semantic tools, e.g. type checkers and interpreters. The semantics definition results in a list of objects and threads, which denotes the behavior of a Java program. For example, the rule for the Java conditional is defined as follows:

Assignment_end:
–CS : Ident1 = Value1;
–C : ”assign“: makes assignment.
assign(ObjL1, ClVarL1, Env1, Ident1, Value1

$$\frac{\begin{array}{l}\text{–>}\\ \text{ObjL2, ClVarL2, Env2)}\end{array}}{\begin{array}{l}\text{ObjL1, ClVarL1, Env1, OThId1, ObjId1, Mode}\\ \quad\text{|– } \textbf{assign\_comp}\text{(void(), Ident1, Value1) –>}\\ \qquad\text{ObjL2, ClVarL2, Env2, Value1, inst\_l[], nil;}\end{array}}$$

In detail, the rule says that the behavior of an assignment statement "Ident1 = Value1" is formalized as:

- if in a given list of current objects (ObjL1), a list of class variables (ClVarL1), an environment (Env1), an identifier (Ident1) and value (Value1) the assignment-phrase introduces a list of objects (ObjL2), a list of class variables (ClVarL2) and an environment (Env2), then
- in a given ObjL1, ClVarL1, Env1, thread identifier (OThid1) and thread mode (Mode) the execution of the assignment yields ObjL2, ClVarL2, Env2, value (Value1), an empty list of running threads (inst_l[]).

Here "nil" indicates that the assignment statement is a part of a sequential (non-threaded) Java program.

Semantics of Scripting Languages

Also variously attempts have been made to formalize the semantics of scripting languages, e.g. the early work [8, 150] encoded the formal semantics of a small subset of JavaScript in a corresponding type system. However, later a small-step operational semantics for the complete language ECMAScript of JavaScript was defined in [106]. In fact, the semantics is a relational variant of structural operational semantics: here each semantic function represents a corresponding semantic transition relation which transforms a heap, a pointer to the scope, and the term into a new heap-scope-term triple. Moreover, the evaluation of expression terms yields either a value or an exception, while the statements evaluates with a notion of *completion*. The *completion* is a flag-value-identifier triple where

$$\text{flag} \in \{\text{Normal, Break, Continue, Return, Throw}\}$$

In the triple, the value represents the return or exception value while the identifier represents the corresponding break or continue. The focus here was to analyze various security properties of JavaScript based web applications. In the following, we show the semantic rules for an exception statement and a '@PutValue' expression of JavaScript. An exception statement has the following semantic rule

$$\text{H,I,}\texttt{throw}\text{ va;} \xrightarrow{s} \text{H,I,<Throw, va, \&empty>}$$

which returns a completion with a flag 'Throw' and value 'va'.

The semantic rule for the specification of a '@Put' expression (which is used to set properties of objects) is defined as follows:

$$\frac{\begin{array}{c}\text{H,I1.@CanPut(m)} \\ \text{m <> H(I1)} \quad \text{H(I1.m=v1\{\}) = H1}\end{array}}{\text{H,I,I1.@Put(m,v)} \xrightarrow{e} \text{H1,I,v1}}$$

The rule says that if the predicate 'H,l1.@CanPut(m)' holds (which shows that an object 'm' is not 'ReadOnly'), then the fresh properties are added with an empty set of attributes.

Semantics of Computer Algebra Languages

There have been various attempts to formalize the formal semantics of computer algebra languages: for instance, the formal semantics of the (former) language FoCaL of the computer algebra system FoCaLiZe has been studied respectively formalized in [62, 131, 144]. In [154], the semantics was formalized for the translation of proofs into Coq. The denotational semantics is hard to maintain, particularly when new features are added in the language, so the later work [55] attempted to formalize the semantics in the $\lambda\Pi$-calculus.

In [119], the formal semantics of the language Lisp of the computer algebra system Maxima was defined. In fact, the operational semantics was defined for the language M-Lisp (Meta-Lisp) which was a subset of Lisp. For example, the operational semantics of a M-Lisp conditional was defined by the following two rules:

$$\frac{M_1 \underset{v}{\longrightarrow} \text{TRUE} \quad M_2 \underset{v}{\longrightarrow} M}{\mathbf{IF}\, M_1 M_2 M_3 \underset{v}{\longrightarrow} M}$$

$$\frac{M_1 \underset{v}{\longrightarrow} \text{FALSE} \quad M_3 \underset{v}{\longrightarrow} M}{\mathbf{IF}\, M_1 M_2 M_3 \underset{v}{\longrightarrow} M}$$

The former rule says that if the M-expression M_1 yields value "TRUE" then M_2 yields value M which is the result of execution of the conditional. The latter rule says that if the M-expression M_1 yields value "FALSE" then M_3 yields the resulting value M.

In [35] an abstract interpretation is used to analyze whether a certain relationship holds between the two semantic interpretations of a Maple program for a particular property. One of the interpretations is used as a template while the other as its abstract version with a certain property. The focus here was to exploit the operational semantics of Maple against certain properties.

The denotational semantics of *MiniMaple* (see Chapter 6) is different from the aforementioned variants, as *MiniMaple* has some non-standard semantic domains, e.g. symbol, union and polynomial etc.; moreover, it also supports a polymorphic type system with corresponding runtime type tests. In contrary to functional programming languages, *MiniMaple* has expressions with side effects. Thus we have developed the denotational semantics as a mathematical function which defines a relationship between pre and post-states to describe a program behavior. The *MiniMaple* semantics is defined to formalize the runtime behavior of *MiniMaple* programs. So far, there is no formally defined semantics of Maple and hence we consider the current implementation of Maple as a basis of our semantics.

Semantics of Specification Languages

Some approaches have also been investigated to formalize the semantics of specification languages. However, the task of defining the formal semantics of specification languages is more complex because the underlying semantic domains are very different from the conventional Scott-Strachey denotational domains. For example, in [115] the semantic space of the specification language Z is modeled as the world of "theories" and their corresponding meaning as the collection of all of its models. The semantics of a schema (i.e. an abstract object with certain properties) and its related operations is defined with the help of the notion of "variety" based on the typed set theory and relational algebra. For example, the semantic function *sexpr* is defined for the conjunction of the Z schema-expression *SEXPR*. The function maps schema-expressions to varieties with a given environment:

$$
\begin{array}{|l}
sexpr : ENV \rightarrow SEXPR \nrightarrow VARIETY \\
\hline
\ldots \\
sexpr(\rho, [\![S_1 \wedge S_2]\!]) == combine(sexpr(\rho, [\![S_1]\!]), sexpr(\rho, [\![S_2]\!])) \\
\ldots
\end{array}
$$

The varieties corresponding to the schema-expressions S_1 and S_2 are put together by the auxiliary operation *combine*. In principle, the *combine* function joins the signatures sig_1 and sig_2 of the given varieties by another auxiliary operation *join*. In fact, varieties correspond to the meaning/semantics of the schema here. Then the function defines a class of models, which recovers a structure of those models which satisfy the properties of the schema. The function *combine* is defined as follows:

$$
\begin{array}{|l}
\hline
combine : VARIETY \times VARIETY \nrightarrow VARIETY \\
\hline
combine(\theta\, VARIETY_1, \theta\, VARIETY_2) == \\
\quad \mu\, VARIETY' \mid \\
\qquad sig' == join(sig_1, sig_2) \\
\qquad models' = \\
\qquad\quad (M : Struct(sig') \mid \\
\qquad\qquad restrict(sig_1, M \in models_1)\& \\
\qquad\qquad restrict(sig_2, M \in models_2)) \\
\hline
\end{array}
$$

The denotational semantics of a specification language Meta-IV of VDM was defined in [49]. The work was later applied to derive the compiler that translated VDM denotations into Ada. The system allowed to generate an implementation through design specification where the specifications were added incrementally. Each increment to the specification generated corresponding proof obligations, which could be proved for the correctness of the design. Later the semantics for the specification language BSI/VDM SL was formalized based on a variant of denotational semantics [126]. For example, the semantic function for the evaluation of expressions (Expr) has signature:

$$EvalExpr\colon \text{Expr} \rightarrow \text{MODEL} \rightarrow \mathbb{P}(\text{VAL})$$

where VAL is the semantic value and MODEL is a mapping of identifiers to denotations. The semantic function for the conditional statement (without side-effects) is defined as:

$$
\begin{array}{l}
EvalExpr(\underline{mk}\text{-}\texttt{If}(t, c, a)(m)) \triangleq \\
\quad \textbf{let } t_s = EvalExpr(t)(m) \textbf{ in} \\
\qquad \textbf{if } t_s = \{True\} \textbf{ then} \\
\qquad\quad EvalExpr(c)(m) \\
\qquad \textbf{else if } t_s = \{False\} \textbf{ then} \\
\qquad\qquad EvalExpr(a)(m) \\
\qquad\quad \textbf{else if } t_s = \{True, False\} \textbf{ then} \\
\qquad\qquad\quad EvalExpr(c)(m) \cup EvalExpr(a)(m) \\
\qquad\qquad \textbf{else } \{\bot\}
\end{array}
$$

In detail, the rule says that, if the evaluation of the expression t yields a singleton set, then the corresponding branch of the conditional statement is evaluated. However, if the evaluation of t yields a set with both values, i.e. $True$ and $False$, then both branches of the conditional are evaluated and the result is the union of the evaluations. If the evaluation yields some other value, then the result is a set with an element $\bot$. The semantic definition is the result of looseness of the expressions. The loose expressions are demonstrated with the help of the following let-be-such expression:

$$\underline{\text{let}}\ x \in \{3,5\}\ \underline{\text{in}}\ \underline{\text{if}}\ x = 3\ \underline{\text{then}}\ 2\ \underline{\text{else}}\ 4\ .$$

The evaluation of the let-be-such expression results in a set $\{2,4\}$ because during execution x can be chosen for the other value, i.e. 5: hence both branches of the conditional are evaluated.

The formal semantics of an interface specification language of Larch was formalized in [41]. The focus here was to formalize the semantics of object-oriented features of Larch, e.g. specification inheritance. The semantics was defined as a modular variant of operational semantics as a pre-requisite for modular reasoning about object-oriented programs of Larch. Also the formal semantics of a specification language for Java (JML) is defined in [27] as a variant of denotational semantics.

In comparison to the above semantics approaches, the semantic domains for the specification language of *MiniMaple* has more complex structures, e.g. mathematical theories, loop and procedure specifications. Moreover, in addition to the basic arithmetic and logical expressions, the specification language also supports guarded numerical and sequence quantifiers. We have defined the semantics of the specification language as a relational variant of denotational semantics in order to overcome the complexity of its semantic domains and other non-standard constructs.

Semantics of Intermediate Verification Languages

There also have been various attempts to formalize the semantices of intermediate verification languages, e.g. for Why3ML [63] and BoogiePL [155]. The operational semantics of Why3ML was defined in [63]. Each semantic function transforms a current state and an expression to the new state and the value yielded by the evaluation of the expression (the evaluation yields a special value *void* for the command expressions). For example, an assignment statement has the following rule:

$$\frac{s, e \longrightarrow s', c}{s, x := e \longrightarrow s' \oplus \{x \mapsto c\}, \text{void}}$$

which says that evaluation of an assignment statement $(x := e)$ in state s yields a value 'void' and the post-state results by an update of value c (of expression e) to an identifier x in state s'. For further details on the formal semantics of Why3ML, please see Chapter 7.

The operational semantics for selected constructs of the intermediate verification language BoogiePL of the verification environment Boogie was formalized in [155]. The single step semantics was defined by a function on states, i.e. $\sigma \rightsquigarrow \sigma'$. For example, the semantic rule for an assignment statement in BoogiePL takes a command sequence $x := e;\ c$ and store μ and results in command c and an updated store μ:

$$\overline{(x := e;\ c \mid \mu) \rightsquigarrow (c \mid \mu, x \mapsto e[\mu])}$$

The focus was to prove the soundness of the verification conditions generator, i.e. the corresponding weakest precondition calculus.

Our main goal was to formalize the runtime behavior of *MiniMaple* annotated programs. Also the semantics is a pre-requisite of our translation to an intermediate verification language where we show that the translation preserves the semantics (see Chapter 7).

2.3. Formal Specification and Verification

In this section, we discuss the role of formal methods in classical programming languages, assertion checking in computer algebra languages, an application of formal methods to computer algebra languages, and the integration of theorem proving and computer algebra systems, respectively.

Formal Methods and Classical Languages

The interest of applying formal methods in computer science for modeling and reasoning has surged during last couple of decades. Thus on one hand, some programming-language independent specification languages have been developed, e.g. Z [160], Alloy [85], VDM [88], Larch [105], B [2] and Object Constraint Language (OCL) which is part of the UML standard [103]; on the other hand, some specification languages have also been developed to formally specify the behavior of programs written in classical languages, e.g. the Java Modeling Language (JML) [71] for Java, ACSL [17] for ANSI C, Larch/C++ for C++ [105], Spec# [14] for C# and Spark for Ada [14,83]. Based on the recent developments in SMT (satisfiability modulo theories) solving [15,140] various tools are making use of automated reasoning techniques [32,38] by employing the aforementioned specification languages. Moreover, various development environments have been integrated with proving assistants for the specification and verification of systems [19,21,83].

Assertion Checking in Computer Algebra Languages

In general, programming languages of most of the computer algebra systems, e.g. Mathematica, Maple, Sage, and Maxima (to name a few) support assertion checking to increase the reliability of programs. For example, Mathematica supports runtime assertion checking only if assertions are enabled. The following example function has an assertion `Assert[c > 1]` which is not violated at the first application.

```
In[6]:=  testFunc[a_] :=
              Block[{c},
                c = a*a;
```

```
              Assert[c > 1];
            ]

In[7]:= testFunc[1]
Out[7]= 1
```

Then, the assertion checking is enabled by command `On[Assert]` and the subsequent function call results in the violation of the assertion.

```
In[8]:= On[Assert];
        testFunc[1]

        Assert::asrtf: Assertion c > 1 failed.
```

Sage also supports runtime assertion checking: however, here the assertions are enabled by default. An assertion `assert x > 0` is introduced in the example function but function application shows the violation of the assertion.

```
sage: def g(x):
         if x <2:
           assert x > 0
           return 0
         else:
           return x+"2"

sage: g(0)
 Error in lines 1-1
 ...
AssertionError
```

The last line reports the error to be `AssertionError`.

Formal Methods and Computer Algebra Languages

Later also several pragmatic applications of formal methods to computer algebra systems have been investigated. Here, the focus was to develop a logical framework such that the language of computer algebra system was equipped with a corresponding formal specification language. For instance, [56] presents an integration of the behavioral specification language Larch [76] to the programming language Aldor of the computer algebra system AXIOM. The methodology for the verification of Aldor programs was devised by defining abstract specifications for AXIOM (respective Aldor) primitives and generating verification conditions which can be proved correct with the help of the prover Larch [57,89]. Also, to define and prove the correctness of computer algebra programs, the language of prover Coq was used such that executable OCaml code can be extracted from the corresponding Coq definitions [74,108–110,149].

As already mentioned, the FoCaLiZe (former FoCal and FoC) project [61, 130, 145] has been developed to provide an integrated programming and specification based environment for computer algebra to develop certified programs to achieve higher reliability. The environment is based on its functional programming language FoCaLiZe (influenced by Ocaml) that additionally supports some object-oriented features and thus allows to write both specifications (based on an axiomatic type language) and proofs of programs together. For verification, the FoCaLiZe compiler extracts the computational code to executable Ocaml programs and also generates verification conditions in the language of an interactive prover Coq. The verification conditions can be proved interactively in Coq; as a result, proofs are produced as Coq scripts that can be verified by Coq.

In FoCaLiZe, a *specie* (domain) can be formally specified by its name and declarations of operations, values and properties. The environment supports a *refinement* process from formal modeling of requirements to design and implementation: and thus allows the *incremental* addition of more definitions of the domain. At any stage of development, the corresponding proofs can be produced such that an implementation meets the specification. Consequently, hierarchal development is possible in the environment, where the higher level corresponds to the specification while the lower one correspond to implementation and every node in the hierarchy refers to a refinement towards the goal.

Later, several new and powerful features were added to the language of FoCaLiZe, e.g. patter matching, inheritance, parameterization and lazy binding. Also an automatic prover Zenon was integrated in the compiler which allowed to develop recursive functions and checks for corresponding termination proofs.

Variously the applications of formal methods to Maple have been investigated. For instance, Maple was integrated with provers to empower the reasoning capabilities of Maple, e.g. [5, 73] provided a Maple-PVS interface where the validity of the Maple procedure calls (i.e. preconditions and postconditions) can be checked with the help of PVS. Also, [3, 4] focused on the verification of necessary side conditions of arguments of the Maple procedures by invoking PVS which matches the entries to a symbolic integral table.

Later, a mathematical description of the interfaces between existing Maple routines was studied in [36]. The goal here was to study the actual contracts that are in use by Maple routines. The contracts were statements with certain (static and dynamic) logical constraints. In fact, the work was just focused on the collection of requirements for the pure type inference engine for existing Maple routines. The work was extended to develop the partial evaluator for Maple [37] as discussed in Section 2.1.

Theorem Proving and Computer Algebra Systems

Various attempts have been made to enhance the role of formal methods in the computer algebra systems beyond runtime assertion checking: one of the earlier works [141] provided a generic interface between the proof planing system Omega and computer algebra systems. The focus here was to verify the computation results based on additional information computed by computer algebra systems. Also the integration of reasoning and computation was discussed in, e.g. Theorema [29, 30, 148] which is built on top of Mathematica and employs proving, computing, and solving method: the method is an iterative proof heuristics. Similarly, the reasoning systems were embedded into computer algebra systems as discussed in Analytica [18, 44] and REDLOG [54].

Later, on one hand some investigations focused on developing computer algebra programs based on the principle of "correct by construction". For example, the Atypical project [128, 129, 152] modified the type system of Aldor to describe propositions and specifications of Aldor's categories. Here, the goal was to modify the dependent types of Aldor such that the category specifications become equivalent to axiomatic data-type specifications [151]. Also a type theory based formalization of polynomial rings and the Gröbner bases algorithm is discussed in [125]. On the other hand, several attempts focused on building computer algebra software on the top of proving systems, e.g. [39] built a computer algebra system on top of HOL Light such that rewriting proofs can be generated based on the computations.

Also, several projects attempted to enhance the computing capabilities of the theorem provers by interfacing provers with back-end computer algebra systems. For instance, in [77, 78] the proving assistant HOL used Maple as a "canon" to find answers for some computational tasks which are later verified by HOL (checking an answer is often much easier than finding it). Similarly an integration of Isabelle and Maple was discussed in [12] where simply the answers are trusted to be correct.

In comparison to the approaches discussed above, the goal of our verification framework was not only to reason about the full correctness of a program but also to apply light-weight formal methods to computer algebra programs for finding internal inconsistencies in the program such as violations of methods preconditions by employing static program analysis. Furthermore, *MiniMaple* supports some non-standard types of objects and runtime type tests, while the specification language of *MiniMaple* supports abstract data types to formalize abstract mathematical concepts; many existing specification languages are weaker in this aspect. In contrast to the computer algebra specification languages above, our specification language is defined for the commercially supported language Maple, which is widely used but was not designed to support static analysis (type checking respectively verification). The challenge here was to overcome those particularities of the language that hinder static analysis.

3. MiniMaple

Based on our study and syntactic analysis of the Maple package *DifferenceDifferential*, we have defined a substantial subset of the language of the computer algebra system Maple, which we call *MiniMaple*. In this chapter, we discuss the syntax and various other interesting features of *MiniMaple*. The rest of the chapter is organized as follows: in Section 3.1, we discuss the results on our study of various computer algebra packages available at our institute. In Section 3.2, we highlight the challenges of static program analysis in Maple and demonstrate them with examples. An overview of the *MiniMaple* syntax is given in Section 3.3; this section also provides some examples of selected syntactic constructs. Section 3.4 elaborates *MiniMaple* in more detail with the help of an example.

3.1. Background Study

At the very beginning of our project, we studied various packages developed at RISC by experts in the area of numerical and symbolic computation in general and algorithmic combinatorics, automated theorem proving and computer algebra in particular. The goal here was to choose one of these packages as a test-case for our envisioned verification framework. In the following, we briefly discuss the investigated packages respectively.

Algorithmic Combinatorics

In the research area of algorithmic combinatorics, the Mathematica package *HolonomicFunctions* [104] was developed by Christoph Koutschan. The goal here was to develop advanced applications of the holonomic systems approach, i.e. computations in Ore algebras, non-commutative Gröbner bases and solving linear systems of differential equations. Characteristically, this package

- was based on pattern matching,
- used more of an imperative style of programming,
- used abstract data types,
- on the one hand made use of customized Mathematica functionality and
- on the other hand did not use many Mathematica libraries.

In essence, this package can mainly be considered as a procedural/functional Mathematica program with abstract data types.

Automated Theorem Proving

The Mathematica package *STProver* [157] was developed by Wolfgang Windsteiger in the area of automated theorem proving. This package provides a prover based on the Prove-Compute-Solve (PCS) strategy, i.e. *proving* by applying standard inference techniques from the predicate logic, *computing* the facts by rewriting the formulas using assumptions in the knowledge base, and finally *solving* by applying computer algebra methods to solve quantified formulas in general and existentially formulas in particular.

We identified the following characteristics of the package *STProver* as it mainly made use of:

- pattern matching rules,
- implicit type definitions and a
- declarative style of programming paradigm.

This package is a Mathematica program based on pattern matching which is now integrated with the Theorema [28] infrastructure.

Computer Algebra

In the area of computer algebra, the Maple package *DifferenceDifferential* [42] was developed by Christian Dönch to compute bivariate difference differential polynomials using relative Gröbner bases using an algorithm of M. Zhou and Franz Winkler [162].

As the main characteristics, this package:

- made use of limited types i.e. integers and lists only,
- was mainly standalone, i.e. did not made much use of Maple libraries,
- did not use destructive update of data structures and
- made use of imperative style of development.

In principle, this package is a Maple functional program. Further details of this package *DifferenceDifferential* are discussed in Chapter 8.

Summary

Based on our study of the aforementioned packages developed in the most prominent dynamically typed computer algebra languages, i.e. Mathematica and Maple, we have chosen Maple for our subsequent study for the following reasons:

- Maple has an imperative style of programming which has a simpler semantics than the rule-based programming style of Mathematica.

- Maple has type annotations for runtime checking which can be directly applied for static analysis. There are also parameter annotations in Mathematica but they are used for selecting the appropriate rule at runtime.

Still many of the results we derive with the static analysis (e.g. type checking) of Maple can be applied to Mathematica, as Mathematica has almost the same kinds of runtime objects as Maple. In the following section, we demonstrate various challenges for this static analysis of Maple programs by examples.

3.2. Challenges

During our study, we found the following special features respectively challenges for the static analysis of Maple programs (which are typical for most other computer algebra languages):

- The language has no static type system. It allows runtime type checking by type annotations but these annotations are optional.
- The language does support some non-standard objects, e.g. symbols and unevaluated expressions.
- There is no clear difference between declaration and assignment. A global variable is introduced by an assignment; a subsequent assignment may modify the dynamic (runtime) type of the variable.
- The language uses type information to direct the flow of control in the program, i.e. it allows some runtime type-tests to select the further execution path.
- The (dynamic) type system of Maple is kind of polymorphic [34]; since Maple has a hierarchy of types in a sub-typing relationship, values of different types can satisfy the same type test.

In the following, we demonstrate the aforementioned challenges for various Maple language constructs by some example Maple programs.

Runtime Type Checking

As already explained, Maple has optional type annotations, which Maple uses for runtime type checking. A Maple kernel routine *kernelopts* allows to change the mode of type checker at different levels to check type assertions at runtime. To do so, one needs to set the value of variable *assertlevel* of routine *kernelopts* to either 0, 1 or 2 where

- 0 (the default value) means, no assertion checking at all,
- 1 means, only calls of the ASSERT function are checked, and
- 2 implies checking of calls of the ASSERT function and of assignments such that these calls respect the type annotations of variable declarations.

We demonstrate the runtime type checking in Maple by a simple Maple procedure which takes an integer as an argument and returns an integer value. If the value of the parameter is less than 10, then the procedure adds 10 to it and stores the result in a local variable x, otherwise it assigns a string value "test" to a local variable x and returns x.

```
> p := proc(s::integer)::integer;
local x::integer;
    if s < 10 then
      x := s + 10
    else
      x := "string"
    end if;
return x;
end proc;
```

Now, we test the behavior of the procedure by a corresponding call with an integer argument 12 as follows:

```
> p(12);
                        "string"
```

The call to the procedure p returns "string" because the parameter value is greater than 10, which is clearly not a valid result as indicated in the procedure header, i.e. the procedure must return an integer value. No checking of any assertion at all allows this program to be executed without warnings, as the kernel routine *kernelopts* is operating in the default mode, i.e. with a value of *assertlevel*.

Now, we change the mode of operation of the kernel routine *kernelopts* such that it checks all the assertions:

```
> kernelopts(assertlevel=2);
                                    0
```

To test the type assertions, we call the procedure p again with an integer value 12.

```
> p(12);
Error, (in p) assertion failed in assignment, expected integer,\
got string
>
```

In the body of the procedure, the else branch of the conditional is executed, where an assignment is made such that a string value is assigned to an integer type variable. This is a typing error and caught by the Maple type checker at runtime.

Runtime Type Tests

Maple also supports some non-standard types of objects, e.g. union types, symbols and unevaluated expressions. For example, the *Or(integer, string)* indicates that a value may be an integer or a string. The type predicate $type(e, t)$ allows one to test whether a Maple expression e is of a given type t; it returns *true* if the expression e is of type t, and returns the value *false* otherwise. Such type tests can be used to direct the flow of control in the program, which complicates reasoning about the correctness of the behavior of such programs.

For instance, type test is applied to the parameter x. The procedure calls show that the corresponding conditional branch is executed depending on the type of the value of the procedure parameter.

```
 p := proc(x::Or(integer, string))::integer;
local y::integer;
    if type(x, integer) then
      y := x;
    elif type(x, string) then
      y := 10;
    end if;
return y;
end proc;

> p(12);
                                    12

> p("test");
                                    10
```

As shown in Section 3.2, the type inference of such expressions (with union types) is complex because in order to identify the correct use of such variables, one needs to keep the track of their types.

Special Types

A symbol is a Maple name and stands for itself until some value is assigned to it. In the following script, a is an unassigned variable whose runtime type is correspondingly *symbol*. Subsequent assignments of values to a change its type to "string" and "integer" respectively.

```
 > a;
                                    a
```

```
> type(a,symbol);
                                    true

> a:="test";
                                    a := "test"

> type(a,string);
                                    true

> a:=12;
                                    a := 12

> type(a, integer);
                                    true
```

Maple supports a subtype relationship among types such that Maple can satisfy multiple type tests:

```
 > a:=12;
                                    a := 12

> type(a, integer);
                                    true

> type(a, rational);
                                    true

> type(a, anything);
                                    true
```

Every expression in Maple is of type *anything* as *anything* is a the root of the type hierarchy in Maple as *Object* is the root type in Java. This sub-typing and polymorphic typing phenomena in Maple makes type inferences more complex.

Enclosing a Maple expression with unevaluated (right) quotes delays its evaluation. Such expressions are called "unevaluated expressions", which are correspondingly annotated with type *uneval*:

```
 > type(''2+3'', uneval);
                                    true
```

An unevaluated expression can be evaluated as show below:

```
> eval(''2+3'', 1);
                                    '2+3'

> eval(''2+3'', 2);
                                    5
```

By default each evaluation strips off one quote from the unevaluated expression. An operation *eval* is also supported to evaluate an unevaluated expression until a desired level. The delayed evaluation of a certain expression increases the complexity to the semantics and behavior of program using such expressions.

3.3. Overview of Syntax

Based on our previous investigations, we have defined a simple but substantial subset of Maple, which we call *MiniMaple*. *MiniMaple* covers all the syntactic domains of Maple but has fewer alternatives in each domain than Maple; in particular, Maple has many expressions which are not supported in our language. The complete syntactic definition of *MiniMaple* is given in Appendix A.

In the following, we briefly explain the major syntactic domains of *MiniMaple* and their informal semantics, while the corresponding complete semantic details are discussed in Chapter 6. The grammar of *MiniMaple* has been formally specified in Backus-Naur-Form (BNF) from which a parser for the language has been automatically generated with the help of the parser generator ANTLR [11]. The top level syntax for *MiniMaple* is shown in Figure 3.1.

Commands

A *MiniMaple* program (*Prog*) is a sequence of commands (*Cseq*); commands are represented by the syntactic domain *C*. There is no separation between declaration and an assignment. In addition to the classical while-loop, the return statement and one and two-sided conditionals statements, *MiniMaple* also supports four variations of for-loops.

For example, in the variation "**for** I **from** E **by** E **to** E **while** E **do** Cseq **end do**", the for-loop iterates starting **from** an initial bound **to** a terminating bound with the steps as specified in the **by** clause. Additionally, the **while** expression condition is also tested, before it executes the body *Cseq* of the loop. In this variation, **to** expression is evaluated only once at the start of the loop and tested at each iteration for the termination of the loop. The **while** expression is evaluated and tested before every iteration. Using this variant of the for-loop, the code fragment

Prog ::= Cseq
Cseq ::= EMPTY | C;Cseq
C ::= **if** E **then** Cseq Elif **end if** | **if** E **then** Cseq Elif **else** Cseq **end if**
| **while** E **do** Cseq **end do**
| **for** I **in** E **do** Cseq **end do**
| **for** I **in** E **while** E **do** Cseq **end do**
| **for** I **from** E **by** E **to** E **do** Cseq **end do**
| **for** I **from** E **by** E **to** E **while** E **do** Cseq **end do**
| **return** E; | **return**; | **error** | **error** I,Eseq
| **try** Cseq Catch **end** | **try** Cseq Catch **finally** Cseq **end**
| I,Iseq := E,Eseq | E(Eseq) | **‘type/I‘** := T
...
Eseq ::= EMPTY | E,Eseq
E ::= I | N | **module()** S;R **end module**;
| **proc**(Pseq) S;R **end proc**;| **proc**(Pseq)::T; S;R **end proc**;
| E1 Bop E2 | Uop E | Esop | E1 **and** E2 | E1 **or** E2 | E(Eseq)
| I1:-I2 | E,E,Eseq | **type(** I,T **)** | E1 = E2 | E1 <> E2
S ::= EMPTY | **local** It,Itseq;S | **global** I,Iseq;S | **uses** I,Iseq;S
| **export** It,Itseq;S
R ::= Cseq | Cseq;E
...

Figure 3.1.: High-level Syntactic Domains

```
> s := "The quick brown fox jumped over the lazy dog.":
> for c from "a" to "z" while searchtext(c,s) > 0 do end do; c;

                                    "s"
```

iterates over the letters of the alphabet for a letter missing in s.

The assignment statement “I,Iseq := E,Eseq“ represents a simultaneous assignment where first the expressions on the right hand side are evaluated and then the resulting values are assigned to the respective variables on the left hand side:

```
> x,y := 1, 2;
                                x, y := 1, 2

> x,y := x+y, x-y;
                                x, y := 3, -1
```

An exception handling mechanism ”**try** Cseq Catch **end**” allows the execution of

MiniMaple commands in a controlled environment, while the execution of command sequence can be interrupted by a corresponding **error** statement:

```
p := proc(x::integer)::integer;
local y::integer;
    try
      if x < 10 then
         error "invalid"
      else y := x - 10
      end if
    catch "invalid":
      y := -1; print("Exception caught")
    end;
  return y
end proc
```

In the example above, a procedure has an exceptional behavior such that when a parameter has a value less than 10 it throws an exception "invalid"; this exception is caught by an exception handler `catch "invalid"`, which assigns -1 to a local variable y and prints a message. Otherwise the procedure subtracts 10 from the value of its argument, assigns this to a local variable y. In both cases, the procedure returns the value of y.

This behavior is demonstrated by the corresponding procedure calls.

```
> p(12);
                                    2

> p(1);
                                "Exception caught"

                                 -1
```

Sometimes, when the names of types get too long, it is helpful to use an abbreviated name. In the following example, by an assignment to variable 'type/myList', we define a new name "myList" for the type *list*(*integer*):

```
> `type/myList`:=list(integer);
                          type/myList := list(integer)

> l:=[54,23,98];
                          l := [54, 23, 98]
```

```
> type(l, myList);
                              true
```

Expressions

MiniMaple supports almost all classical basic expressions, e.g. arithmetic (addition, subtraction, multiplication, division and remainder computation) and logical operations (equal, less, greater, less equal and greater equal). Moreover, *MiniMaple* also supports *procedure* and *module* expressions.

Syntactically, a procedure expression "**proc**(Pseq)::T; S;R **end proc**;" consists of a header and a body. The sequence of parameters (typed identifiers) *Pseq* and the return type of the procedure T are part of the procedure header, while the body of the procedure contains various (local and global) declarations S and a sequence of statements R. The return type T is a type assertion, which in Maple is checked at runtime when a procedure is called with the operational mode 2 of *assertlevel* in the kernel routine *kernelopts*.

In the example program

```
> m;
                          m

> f := proc(k::Or(integer, string))::integer;
local n::integer;
global m;
    if type(k, integer) then
      n := k + 1; m := n
    elif type(k, string) then
      m := "test"
    end if;
return m
end proc
```

the procedure f takes an argument k of union type of integer or string and returns an value of type integer. After the header, there are local and global declarations. Here, one can notice that the global variable m has no type information attached to it. This is because of the fact that the global declarations respectively variables cannot be type annotated in Maple and therefore values of arbitrary types can be assigned to them.

In the body of the procedure, the local declaration respectively variable is type annotated. In the body of the loop, we assign an integer or a string value to the global variable m based on the evaluation of the respective type tests for integer

```
1. status:=0;
2. sum := proc(l::list(Or(integer,float)))::[integer,float];
3.         global status;
4.         local i, x::Or(integer,float), si::integer:=0, sf::float:=0.0;
5.         for i from 1 by 1 to nops(l) do
6.             x:=l[i];
7.             status:=i;
8.             if type(x,integer) then
9.                     if (x = 0) then
10.                        return [si,sf];
11.                    end if;
12.                    si:=si+x;
13.            elif type(x,float) then
14.                     if (x < 0.5) then
15.                        return [si,sf];
16.                    end if;
17.                    sf:=sf+x;
18.            end if;
19.        end do;
20.        status:=-1;
21.        return [si,sf];
22.        end proc;
```

Figure 3.2.: An Example *MiniMaple* Program

($type(k, integer)$) or string ($type(k, string)$). In any case, the procedure returns the value of the global variable, i.e. m.

To test the return type assertion of the procedure f, we call this procedure with a string value "s", which results in an error because f expects its return value to be of type integer, but returned 's'.

```
> f("s");
Error, (in f) assertion failed: f expects its return value to\
be of type integer, but computed test
```

In addition to the aforementioned expressions, *MiniMaple* also supports other special expressions, e.g. constructors for list, tuple and set and also their corresponding various operands, i.e. select, length, substitution etc.

In addition to basic types, e.g. integers, booleans, *MiniMaple* also supports composite and extended types, e.g. anything, union and unevaluated. Further details on the type system of *MiniMaple* are discussed in the Chapter 4.

3.4. Running Example

For showing more details of the *MiniMaple* syntax, we introduce in Figure 3.2 an example procedure, which we will use in the following chapters to demonstrate type checking, specification and verification.

The program consists of a command followed by a procedure definition and an application of the procedure. The procedure takes a list of integers and floats and computes the sum of these integers and floats separately; it returns a tuple of integer and float as the sum of respective integers and floats in the list. The procedure may also terminate prematurely for certain inputs, i.e. either for an integer value 0 or for a float value less than 0.5 in the list; in this case the procedure computes the respective sums just before the index at which the aforementioned terminating input occurs.

As one can see from the example, we make use of the type annotations that Maple introduced for runtime type checking. In particular, we demand that function parameters, function results and local variables are correspondingly type annotated.

4. Formal Type System

Based on the *MiniMaple* type annotations introduced in the previous chapter, we have defined a language of types and a corresponding type system for the static type checking of *MiniMaple* programs. In this chapter, we discuss a corresponding formal type system for *MiniMaple*. The rest of the chapter is organized as follows: in Section 4.1, we discuss the motivation for the design and development of the type system. In Section 4.2, we present various elements of the type system, while in Section 4.3, we demonstrate the implementation of a corresponding type checker by its application to our example *MiniMaple* program.

4.1. Background

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 determine the types of variables from the text of a program. A type system prevents *forbidden errors* during the execution of the program. It completely prevents the *untrapped errors* and also a large class of *trapped errors*. *Untrapped errors* may go unnoticed for a while and later cause an arbitrary behavior during the execution of a program, while *trapped errors* immediately stop execution [34].

A type system is a simple decidable logic with various kinds of *judgments*; for example the typing judgment

$$\pi \vdash E{:}(\tau)\mathbf{exp}$$

can be read as "in the given type environment π, E is a well-typed expression of type τ".

A type system is *sound*, if the deduced types indeed capture the program values exhibited at runtime. For example, if we can derive the simplified typing judgment

$$\pi \vdash E{:}(\tau)\mathbf{exp}$$

and e is an environment which is consistent with π, then

$$[\![E]\!]e \in [\![\tau]\!]$$

i.e. at runtime the expression E in environment e indeed denotes a value of type τ ($[\![E]\!]$ describes the runtime value of E and $[\![\tau]\!]$ describes the set of values specified by type τ) as will be formally explained in the Chapter 6.

4.2. Type System for MiniMaple

We have defined a typing judgment for each syntactic domain of *MiniMaple*. Logical rules are defined to derive the typing judgments by using auxiliary functions and predicates. In this section, we first sketch the design of our type system and then we presents its corresponding implementation and application by an example. A proof of the soundness of the type system is still a future task. The complete formalization of the type system is presented in Appendix C.

4.2.1. Types and Sub-typing

MiniMaple uses Maple's type annotations for static type checking, which gives rise to the following language of types:

$$
\begin{aligned}
T ::= \ & \textbf{integer} \mid \textbf{boolean} \mid \textbf{string} \mid \textbf{float} \mid \textbf{rational} \mid \textbf{anything} \\
& \mid \textbf{\{}\ T\ \textbf{\}} \mid \textbf{list(}\ T\ \textbf{)} \mid \textbf{[}\ \mathit{Tseq}\ \textbf{]} \mid \textbf{procedure[}\ T\ \textbf{](}\ \mathit{Tseq}\ \textbf{)} \\
& \mid I\textbf{(}\ \mathit{Tseq}\ \textbf{)} \mid \textbf{Or(}\ \mathit{Tseq}\ \textbf{)} \mid \textbf{symbol} \mid \textbf{void} \mid \textbf{uneval} \mid I
\end{aligned}
$$

The language supports various atomic data types (e.g. integer, boolean, float, rational), sets of values of type T ($\{\ T\ \}$), lists of values of type T (**list**(T)) and tuples whose members have the values of types denoted by a type sequence *Tseq* ([*Tseq*]). Type **anything** is the super-type of all types. Type **Or**(*Tseq*) denotes the union type of various types, type **uneval** denotes the values of unevaluated expressions, e.g. polynomials, and type **symbol** is a name that stands for itself, because no value has been assigned to it yet. User-defined data types are referred by I while I(*Tseq*) denotes tuples (of values of types *Tseq*) tagged by a name I.

As discussed in the previous chapter, Maple supports a sub-typing relation ($<$) among types, e.g. **integer** $<$ **rational** $<$ ... $<$ **anything**, i.e. **integer** is a sub-type of **rational** and **anything** is the super-type of all types. A Maple function $subtype(s, t)$ determines such relation between any two Maple types. The call $subtype(s, t)$ returns true, if type s is a subtype of type t and both types are Maple types and returns false otherwise. In general, not every Maple type qualifies for this subtype test and hence the aforementioned routine returns false. In the following example, a variable c has an integer value assigned to it, which consequently returns true for various type tests, i.e. for integer, rational and anything.

```
> c:=12;
                                    x := 12

> type(c, integer);
                                     true
```

```
> type(c, rational);
                                        true

> type(c, anything);
                                        true

> subtype(integer, rational);
                                        true

> subtype(rational, anything);
                                        true

> subtype(integer, anything);
                                        true
```

Also, a sub type test $subtype(integer, rational)$ returns true, which reflects our notion of sub-typing as above. The other two sub typing tests, show that type *anything* is the super type of both *integer* and *rational*.

In *MiniMaple*, we have defined the sub-typing relation by a predicate

$$matchType(\tau_1, \tau_2)$$

which returns *true* if the former type τ_1 is a super-type of type τ_2. In the following we define this predicate for selected types, for its complete definition, see Appendix C.

$$matchType(integer, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = integer \\ false & \text{otherwise} \end{cases}$$

$$matchType(boolean, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = boolean \\ false & \text{otherwise} \end{cases}$$

$$matchType(string, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = string \\ false & \text{otherwise} \end{cases}$$

$$matchType(anything, \tau) \Leftrightarrow true$$

$$matchType([\tau seq], \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists\, \tau seq_1 : \tau = [\tau seq_1] \\ & \wedge matchTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$matchType(\{\tau\}, \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists\, \tau_2 : \tau_1 = \{\tau_2\} \\ & \wedge matchType(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$matchType(list(\tau), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists\, \tau_2 : \tau_1 = list(\tau_2) \\ & \wedge matchType(\tau, \tau_2) \\ false & \text{otherwise} \end{cases}$$

$$matchType(procedure[\tau](\tau seq), \tau_1) \Leftrightarrow \begin{cases} true & \text{if } \exists\, \tau_2, \tau seq_1 : \tau_1 = procedure[\tau_2](\tau seq_1) \\ & \wedge matchType(\tau, \tau_2) \wedge matchTypeSeq(\tau seq_1, \tau seq) \\ false & \text{otherwise} \end{cases}$$

$$matchType(I(\tau seq), \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists\, \tau seq_1 : \tau = I(\tau seq_1) \\ & \wedge matchTypeSeq(\tau seq, \tau seq_1) \\ false & \text{otherwise} \end{cases}$$

$$matchType(Or(\tau seq), \tau) \Leftrightarrow \begin{cases} true & \text{if } hasTypeAnything(\tau seq) \\ true & \text{if } \exists\, \tau_1 \in \tau seq : matchType(\tau_1, \tau) \\ true & \text{if } \exists\, \tau seq_1 : \tau = Or(\tau seq_1) \\ & \wedge \forall\, \tau_1 \in \tau seq_1 : \exists\, \tau_2 \in \tau seq : matchType(\tau_2, \tau_1) \\ false & \text{otherwise} \end{cases}$$

$$matchType(symbol, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = symbol \\ false & \text{otherwise} \end{cases}$$

$$matchType(void, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = void \\ false & \text{otherwise} \end{cases}$$

$$matchType(uneval, \tau) \Leftrightarrow \begin{cases} true & \text{if } \tau = uneval \\ false & \text{otherwise} \end{cases}$$

$$matchType(I, \tau) \Leftrightarrow \begin{cases} true & \text{if } \exists\, \tau : \tau = I \\ false & \text{otherwise} \end{cases}$$

As shown above, in addition to the surface types the predicate *matchType* also defines the sub-typing relationship for the structured types. The predicates *hasTypeAnything* and *matchTypeSeq* are defined in Section 4.2.5.

4.2.2. Type Environment

In order to track the types of *MiniMaple* identifiers, we define a type environment π as a partial function

$$\pi : \text{Identifier} \xrightarrow{partial} \text{Type}$$

from identifiers to types. In the following, we discuss the problems arising from type checking *MiniMaple* programs using the example presented in the previous chapter.

Type Tests

As already explained in the previous chapter, a predicate **type**(E, T) (which is true if the value of expression E has type T) may direct the control flow of a program. If this predicate is used in a conditional, then different branches of the conditional may have different type information for the same variable. We keep track of the type information introduced by the different type tests from different branches to adequately reason about the possible types of a variable.

In our example program, the variable x has a union type Or(integer,float) and this variable x is used in a conditional statement where the "then" branch is guarded by a test type(x,integer), and similarly the other branch is guarded by a corresponding test type(x, float).

The correspondingly inferred type environments are:

...
(a). # π={..., x:***Or(integer, float)***, ...}
...
if type(x,integer) **then**
(b). # π={..., i:***integer***, x:***integer***, si:***integer***, ..., status:***integer***}
...
elif type(x,float) **then**
(c). # π={..., i:***integer***, x:***float***, ..., sf:***float***, status:***integer***}
...
end if;
(d). # π={..., i:***integer***, x:***Or(integer,float)***,..., status:***integer***}
...

The use of type tests in the conditional expressions introduce more type information for the identifier x to direct the program control flow as depicted by the type environments at lines (b) and (c).

By analyzing the conditional command as a whole, the type of variable x is combined to **Or(integer, float)** as depicted at line (d). For this purpose, we use the auxiliary function

combine: *Type_Environment* $\times$ *Type_Environment* $\rightarrow$ *Type_Environment*

$$
\begin{aligned}
combine(\pi_1, \pi_2) = &\{(I : \tau_1) \in \pi_1 : \neg\, \exists (I : \tau_2) \in \pi_2)\} \\
&\cup \{(I : \tau_2) \in \pi_2 : \neg\, \exists (I : \tau_1) \in \pi_1)\} \\
&\cup \{(I : \tau_3) : \exists\, \tau_3 : (I : \tau_1) \in \pi_1 \wedge \exists (I : \tau_2) \in \pi_2 \\
&\qquad \wedge \tau_3 = orCombine(\tau_1, \tau_2)\}
\end{aligned}
$$

that combines the identifiers of the former type environment with the identifiers in latter type environment. The resulting type environment has the disjoint identifiers with their corresponding types and the common identifiers with an or-type τ_3 of the two corresponding types.

The function *orCombine* is defined in Section 4.2.5.

Global Variables

As shown in our example program, global variables (declarations) can not be type annotated:

```
...
global status;
...
```

therefore values of arbitrary types can be assigned to global variables as shown below for a global variable *status*:

```
> status:=12;
              status := 12

> status:="test";
              status := "test"

> status:=[12,31,43];
              status := [12, 31, 43]
```

We have introduced *global* and *local* contexts to handle the semantics of the variables inside and outside of the body of a procedure respective loop.

- In a *global* context new variables may be introduced by assignments and the types of variables may change arbitrarily by assignments.
- In a *local* context variables can only be introduced by declarations. The types of variables can only be *specialized* i.e. the new value of a variable should be a sub-type of the declared variable type, which is defined by an auxiliary function

$$\begin{aligned} &\mathit{specialize}\colon \mathit{Type\_Environment} \times \mathit{Type\_Environment} \to \mathit{Type\_Environment} \\ &\mathit{specialize}(\pi_1, \pi_2) = \{(I : \tau_1) \in \pi_1 : \neg\, \exists (I : \tau_2) \in \pi_2)\} \\ &\qquad\qquad \cup \{(I : \tau_2) \in \pi_2 : \neg\, \exists (I : \tau_1) \in \pi_1)\} \\ &\qquad\qquad \cup \{(I : \tau_3) : \exists \tau_3 : (I : \tau_1) \in \pi_1 \wedge (I : \tau_2) \in \pi_2 \\ &\qquad\qquad\qquad\qquad \wedge \tau_3 = \mathit{superType}(\tau_1, \tau_2)\} \end{aligned}$$

that specializes the identifiers of former type environment to the identifiers in the latter type environment w.r.t. their types.

Moreover, the sub-typing relation (i.e. *matchType*) is observed while specializing the types of variables, which is correspondingly defined by an auxiliary function

$$\begin{aligned} &\mathit{superType}\colon \mathit{Type} \times \mathit{Type} \to \mathit{Type} \\ &\mathit{superType}(\tau_1, \tau_2) = \begin{cases} \tau_1 & \text{if } \mathit{matchType}(\tau_2, \tau_1) \\ \tau_2 & \text{if } \mathit{matchType}(\tau_1, \tau_2) \end{cases} \end{aligned}$$

that returns the super-type between the two given types.

Depending on the current context, the different typing rules will be used to constrain variable assignments as shown in Section 4.2.3.

4.2.3. Typing Judgments

In this subsection we explain the typing judgments and typing rules for the boolean expressions and commands of *MiniMaple*. These judgments use the following kinds of objects ("Identifier" and "Type" are the syntactic domains of identifiers/variables and types of *MiniMaple* respectively):

- πset: A set of type environments introduced by type checking the corresponding syntactic phrase.
- $c \in \{\text{global, local}\}$: A context flag to check if the corresponding syntactic phrase is type checked inside/outside of the procedure/loop.
- $asgnset \subseteq$ Identifier: A set of assignable identifiers introduced by type checking the declarations.
- $expidset \subseteq$ Identifier: A set of exported identifiers introduced by type checking the export declarations in procedure/module.
- $\epsilon set \subseteq$ Identifier: A set of thrown exceptions introduced by type checking the corresponding syntactic phrase.
- $\tau set \subseteq$ Type: A set of return types introduced by type checking the corresponding syntactic phrase.
- $rflag \in \{\text{aret, not\_aret}\}$: A return flag to check if the last statement of every execution of the corresponding syntactic phrase is a *return* command.

Boolean Expressions

MiniMaple supports various types of expressions but boolean expressions are treated specially because of the test $\mathbf{type}(I, T)$ that gives additional type information about the expression. The typing judgment for boolean expressions

$$\pi \vdash E{:}(\pi_1)\mathbf{boolexp}$$

can be read as "with the given type environment π, E is a well-typed boolean expression which generates a new type environment π_1". The new type environment is produced as a fact of type test that might introduce new type information for an identifier.

Commands

The typing judgment for commands

$$\pi,\ c,\ asgnset \vdash C{:}(\pi_1,\ \tau set,\ \epsilon set,\ rflag)\mathbf{comm}$$

can be read as "in the given type environment π, context c and an assignable set of identifiers *asgnset*, C is a well-typed command and produces (π_1, τset, ϵset, *rflag*) as type information".

4.2.4. Typing Rules

In this subsection, we discuss the typing rules for selected boolean expressions and commands. These rules use various auxiliary functions and predicates which are defined in Subsection 4.2.5.

Boolean Expression

The typing rule for $\mathbf{type}(I,T)$ is as follows:

$$\frac{\pi \vdash I{:}(\tau_1)\mathbf{id} \quad \pi \vdash T{:}(\tau_2)\mathbf{type} \quad superType(\tau_1,\tau_2)}{\pi \vdash \mathbf{type}(I,T){:}(\{\mathrm{I}{:}\tau_2\})\mathbf{boolexp}}$$

The phrase "$\mathbf{type}(I,T)$" is a well-typed boolean expression if the declared type of identifier (τ_1) is the super-type of T (τ). The boolean expression may introduce new type information for the identifier.

In our example program, the local variable x has a union type of integer and float (as depicted at line a below), which is used in two corresponding type tests (at lines b and c) in the conditional.

```
...
a: local i, x::Or(integer,float), ...;
...
a0: # π0={..., i:integer, x:Or(integer,float),..., status:anything}
b: if type(x,integer) then
...
c: elif type(x,float) then
...
```

d: **end if**;
...

By the above typing rule, the type test **type**(x, integer) is evaluated in a step-wise way as follows:

- $\pi_0 \vdash x$:(Or(integer,float))**id**,
- $\pi_0 \vdash$ *integer*:(integer))**type** and
- *superType*(Or(integer, float), integer) evaluates to *true*.

All of the above evaluations results in the conclusion/judgment

$$\pi_0 \vdash \textbf{type}(x,integer):(\{\text{x:integer}\})\textbf{boolexp}$$

where a new type environment is produced, in which a variable x has a type **integer**. Similarly the typing judgment

$$\pi_0 \vdash \textbf{type}(x,float):(\{\text{x:float}\})\textbf{boolexp}$$

can be derived for the second type test **type**(x, float).

Conditional Command

The typing rule for the conditional command, i.e. **if** E **then** *Cseq Elif* **end if** is given below:

$$\frac{\begin{array}{c}\pi \vdash E\text{: }(\pi')\textbf{boolexp} \quad canSpecialize(\pi,\pi') \\ specialize(\pi,\pi'),\ c,\ asgnset \vdash Cseq\text{:}(\pi_1,\tau set_1,\epsilon set_1,rflag_1)\textbf{cseq} \\ \pi,\ c,\ asgnset \vdash Elif\text{:}(\pi_2,\pi set,\ \tau set_2,\epsilon set_2,rflag_2)\textbf{elif}\end{array}}{\begin{array}{c}\pi,\ c,\ asgnset \vdash \textbf{if}\ E\ \textbf{then}\ Cseq\ \text{Elif}\ \textbf{end} \\ \textbf{if}\text{:}(combine(\pi_1,\pi_2),\tau set_1 \cup \tau set_2,\epsilon set_1 \cup \epsilon set_2,\text{ret}(rflag_1,\ rflag_2))\textbf{comm}\end{array}}$$

The phrase "**if** E **then** *Cseq Elif* **end if**" is a well typed conditional command if the type of expression E does not conflict global type information. The conditional command combines the type environment of its two conditional branches (*if* and *elif*), because we are not sure which of the branches will be executed at runtime.

For demonstration of this typing rule, let's consider the following conditional code snippet from our example program, which is manually type annotated (with corresponding type environment) for elaboration.

...
a: **local** i, x::**Or**(**integer**,**float**), ...;
...
a_0: # π_0={..., *i*:***integer***, *x*:***Or(integer,float)***,..., *status*:***anything***}
b: **if type**(x,integer) **then**

b_0: # π_1={..., *i*:***integer***, *x*:***integer***, *si*:***integer***, ..., *status*:***integer***}
b_1: **if** (x = 0) **then**
return [si,sf];
end if;
b_2. si:=si+x;
b_3: # π_{11}={..., *i*:***integer***, *x*:***integer***, *si*:***integer***, ..., *status*:***integer***}
c: **elif type**(x,float) **then**
c_0: # π_2={..., *i*:***integer***, *x*:***float***, ..., *sf*:***float***, *status*:***integer***}
c_1: **if** (x < 0.5) **then**
return [si,sf];
end if;
c_2. sf:=sf+x;
c_3: # π_{22}={..., *i*:***integer***, *x*:***float***, ..., *sf*:***float***, *status*:***integer***}
d: **end if**;
d_0: # π_3={..., *i*:***integer***, *x*:***Or(integer,float)***,..., *status*:***integer***}
...

The above typing rule is evaluated with respect to its premises as follows:

- $\pi_0 \vdash$ *type(x, integer)*: ({x:integer})**boolexp** (as given π_0 above)
- *canSpecialize*(π_0,{x:integer}) evaluates to *true*
- *specialize*(π_0,{x:integer}) results in a type environment π_1
- π_1, *c*, *asgnset* $\vdash$ *Cseq*:(π_{11},{[integer,float]},$\emptyset$,not_aret)**cseq**, where *Cseq* represents the command sequence with labels b_1 and b_2
- π, *c*, *asgnset* $\vdash$ *Elif*:(π_{22},$\emptyset$, {[integer,float]},$\emptyset$,not_aret)**elif**, where *Elif* represents the command with label *c*.

The above premise evaluations results in the judgment

$$\pi_0,\ c,\ asgnset \vdash \textbf{if}\ type(x,integer)\ \textbf{then}\ Cseq\ \text{Elif}\ \textbf{end}\ \textbf{if}{:}(\pi_3,\{[\text{integer,float}]\},\emptyset,\text{not\_aret})\textbf{comm}$$

where π_3 is an application of the auxiliary function *combine* and all other components of **comm** are computed as derived by the typing rule. Moreover, this conditional command not always returns and also has no exceptions as depicted by the corresponding values not_aret and $\emptyset$.

Assignment Command

The typing rule for the assignment command *I,Iseq* := *E,Eseq* in a *local* context is defined below:

$$\frac{\begin{array}{c}\pi \vdash I{:}(\tau_1)\mathbf{id} \quad \pi \vdash Iseq{:}(\tau seq_1)\mathbf{idseq} \quad isNotRepeated(I,Iseq) \\ \pi \vdash E{:}(\tau_2)\mathbf{exp} \quad \pi \vdash Eseq{:}(\tau seq_2)\mathbf{expseq} \\ matchTypeSeq((\tau_1,\tau seq_1),(\tau_2,\tau seq_2)) \quad isAssignable((I,Iseq),\ asgnset)\end{array}}{\begin{array}{c}\pi,\ local,\ asgnset \vdash I,Iseq := \\ E,Eseq{:}(update(\pi,(I,Iseq),(\tau_2,\tau seq_2)),\{\},\{\},not\_aret)\mathbf{comm}\end{array}}$$

In a *local* context, the phrase "*I,Iseq* := *E,Eseq*" is a well typed assignment command which updates the types of the identifiers only, if the types of the expressions (E and *Eseq*) are the subtypes of the declared types of identifiers (I and *Iseq*).

$$\frac{isNotRepeated(I,Iseq) \quad \pi \vdash E{:}(\tau)\mathbf{exp} \quad \pi \vdash Eseq{:}(\tau seq)\mathbf{expseq}}{\begin{array}{c}\pi,\ global,\ asgnset \vdash I,Iseq := \\ E,Eseq{:}(update(\pi,(I,Iseq),(\tau,\tau seq)),\{\},\{\},not\_aret)\mathbf{comm}\end{array}}$$

In a *global* context, the phrase "*I,Iseq* := *E,Eseq*" is a well typed assignment command that allows to change the types of identifiers arbitrarily.

The rule for the global context can easily be practiced, so we demonstrate the typing rule in the *local* context by considering the our example program as follows:

```
a: status:=0;
b: sum := proc(l::list(Or(integer,float)))::[integer,float];
...
c:      # π0={..., i:integer, x:Or(integer,float),..., status:anything}
d:      for i from 1 by 1 to nops(l) do
d0.         x:=l[i];
d1.         status:=i;
d2.         # π1={..., i:integer, ..., status:integer}
...
```

Based on the assignment command labeled d_1, the premises of the corresponding typing rule evaluates in the following order:

- $\pi_0 \vdash$ *status*:(anything)**id** as global variables cannot be type annotated and are assigned the super type by default
- $\pi_0 \vdash$ *EMPTY*:(EMPTY)**idseq**, as it is not a simultaneous assignment
- $\pi_0 \vdash i$:(integer)**exp**
- $\pi_0 \vdash$ *EMPTY*:(EMPTY)**expseq**
- *isNotRepeated(I,EMPTY)* returns *true*
- *matchTypeSeq*((anything, EMPTY),(integer, EMPTY)) returns *true* and
- *isAssignable((status,EMPTY), asgnset)* returns *true* because the identifier *status* became available for assignment after its (global) declaration.

The above premises results in

$$\pi_0,\ local,\ asgnset \vdash status := i{:}(\pi_1,\emptyset,\emptyset,not\_aret)\mathbf{comm}$$

where π_1 is the result of the *update* function. Furthermore, an assignment command is neither an exception statement nor a return statement and also not always returns as represented by the corresponding values $\emptyset$, $\emptyset$ and not_aret.

Further details on the typing judgments and rules for various syntactic domains of *MiniMaple* are discussed in [91].

4.2.5. Auxiliary Functions and Predicates

In this subsection, we give definitions of the selected auxiliary functions and predicates, which are used in the Subsection 4.2.4.

- *orCombine*: $Type \times Type \to Type$ returns the general type (if there) between the two (former and later type), otherwise returns the union of the two types.

$$orCombine(integer,\tau) = \begin{cases} integer & \text{if } \tau = integer \\ anything & \text{if } \tau = anything \\ Or(integer,\tau) & \text{if } \tau \notin \{integer, anything\} \end{cases}$$

$$orCombine(boolean,\tau) = \begin{cases} boolean & \text{if } \tau = boolean \\ anything & \text{if } \tau = anything \\ Or(boolean,\tau) & \text{if } \tau \notin \{boolean, anything\} \end{cases}$$

$$orCombine(string,\tau) = \begin{cases} string & \text{if } \tau = string \\ anything & \text{if } \tau = anything \\ Or(string,\tau) & \text{if } \tau \notin \{string, anything\} \end{cases}$$

$$orCombine(anything,\tau) = anything$$

$$orCombine(\{\tau\},\tau_1) = \begin{cases} \{\tau\} & \text{if } \exists\,\tau_2 : \tau_1 = \{\tau_2\} \\ anything & \text{if } \tau = anything \\ Or(\{\tau\},\tau_1) & \text{if } \tau_1 \neq anything \wedge \neg\exists\,\tau_2 : \tau_1 = \{\tau_2\} \end{cases}$$

$$orCombine(list(\tau),\tau_1) = \begin{cases} list(\tau) & \text{if } \exists\,\tau_2 : \tau_1 = list(\tau_2) \\ anything & \text{if } \tau = anything \\ Or(list(\tau),\tau_1) & \text{if } \tau_1 \neq anything \wedge \neg\exists\,\tau_2 : \tau_1 = list(\tau_2) \end{cases}$$

$$orCombine([\tau seq],\tau_1) = \begin{cases} [orCombineSeq(\tau seq,\tau seq_1)] & \text{if } \exists\,\tau seq_1 : \tau_1 = [\tau seq_1] \\ anything & \text{if } \tau = anything \\ Or([\tau seq],\tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge\neg\exists\,\tau seq_1 : \tau_1 = [\tau seq_1] \end{cases}$$

$$orCombine(procedure[\tau](\tau seq), \tau_1) =$$
$$\begin{cases} procedure[\tau][orCombineSeq(\tau seq, \tau seq_1)] & \text{if } \exists\, \tau_2, \tau seq_1 \\ & : \tau_1 = procedure[\tau_2](\tau seq_1) \\ anything & \text{if } \tau = anything \\ Or(procedure[\tau](\tau seq), \tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge \neg\, \exists\, \tau_2, \tau seq_1 \\ & : \tau_1 = procedure[\tau_2](\tau seq_1) \end{cases}$$

$$orCombine(I(\tau seq), \tau_1) =$$
$$\begin{cases} I(orCombineSeq(\tau seq, \tau seq_1)) & \text{if } \exists\, I_1, \tau seq_1 : \tau_1 = I_1(\tau seq_1) \\ & \wedge I = I_1 \\ anything & \text{if } \tau = anything \\ Or(I(\tau seq), \tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge \neg\, \exists\, I_1, \tau seq_1 : \tau_1 = I_1(\tau seq_1) \\ & \wedge I = I_1 \end{cases}$$

$$orCombine(Or(\tau seq), \tau_1) =$$
$$\begin{cases} Or(orCombineSeq(\tau seq, \tau seq_1)) & \text{if } \exists\, \tau seq_1 : \tau_1 = Or(\tau seq_1) \\ & \wedge \neg hasTypeAnything(\tau seq) \\ & \wedge hasTypeAnything(\tau seq_1) \\ anything & \text{if } \tau = anything \\ Or(\tau seq, \tau_1) & \text{if } \tau_1 \neq anything \\ & \wedge \neg hasTypeAnything(\tau seq) \end{cases}$$

$$orCombine(symbol, \tau) = \begin{cases} symbol & \text{if } \tau = symbol \\ anything & \text{if } \tau = anything \\ Or(symbol, \tau) & \text{if } \tau \notin \{symbol, anything\} \end{cases}$$

$$orCombine(void, \tau) = \begin{cases} void & \text{if } \tau = void \\ anything & \text{if } \tau = anything \\ Or(void, \tau) & \text{if } \tau \notin \{void, anything\} \end{cases}$$

$$orCombine(uneval, \tau) = \begin{cases} uneval & \text{if } \tau = uneval \\ anything & \text{if } \tau = anything \\ Or(uneval, \tau) & \text{if } \tau \notin \{uneval, anything\} \end{cases}$$

- $matchTypeSeq \subset Type\_Sequence \times Type\_Sequence$: returns true (in most cases) if every type from the former sequence of types is general to the corresponding type in the latter sequence of types, i.e. the former type is a super type of the latter type.

$$matchTypeSeq((\tau_1, \tau seq_1), (\tau_2, \tau seq_2)) \Leftrightarrow$$
$$\begin{cases} true & \text{if } \exists\, \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'') \\ & \wedge matchTypes(\tau', \tau'') \\ & \wedge matchTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ true & \text{if } \exists\, \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'') \\ & \wedge \neg(matchTypes(\tau', \tau'') \\ & \wedge matchTypeSeq(\tau seq_1, (\tau_2, \tau seq_2))) \\ true & \text{if } \exists\, \tau' : \tau_1 = seq(\tau') \\ & \wedge \neg\, \exists\, \tau'' : \tau_2 = seq(\tau'') \\ & \wedge matchTypes(\tau', \tau'') \\ & \wedge matchTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ true & \text{if } \exists\, \tau' : \tau_1 = seq(\tau') \\ & \wedge \neg\, \exists\, \tau'' : \tau_2 = seq(\tau'') \\ & \wedge \neg(matchTypes(\tau', \tau'') \\ & \wedge matchTypeSeq(\tau seq_1, (\tau_2, \tau seq_2))) \\ true & \text{if } \neg(\exists\, \tau', \tau'' : \tau_1 = seq(\tau') \\ & \wedge \tau_2 = seq(\tau'')) \\ & \wedge matchTypes(\tau', \tau'') \\ & \wedge matchTypeSeq((\tau_1, \tau seq_1), \tau seq_2) \\ false & \text{otherwise} \end{cases}$$

$$matchTypeSeq((\tau, \tau seq), empty) \Leftrightarrow \begin{cases} true & \text{if } \tau seq = seq(\tau) \\ & \wedge matchTypeSeq(\tau seq, empty) \\ false & \text{otherwise} \end{cases}$$

$$matchTypeSeq(empty, (\tau, \tau seq)) \Leftrightarrow false$$

$$matchTypeSeq(empty, empty) \Leftrightarrow true$$

- $canSpecialize \subset Type\_Environment \times Type\_Environment$: returns true if all the common identifiers (in both type environments) have a super-type between their corresponding types, i.e.
  $$canSpecialize(\pi_1, \pi_2) \Leftrightarrow (\forall\, I, \tau_1, \tau_2 : (I : \tau_1) \in \pi_1 \wedge (I : \tau_2) \in \pi_2 \Rightarrow \exists\, \tau_3 : \tau_3 = superType(\tau_1, \tau_2))$$
- $hastTypeAnything \subset Type\_Sequence$ returns true if any of the type is anything in the given sequence of types.

  $$hasTypeAnything(empty) \Leftrightarrow true$$

```
/home/taimoor/antlr3/Test6.m parsed with no errors.
Generating Annotated AST...
...
**********COMMAND-SEQUENCE-ANNOTATION START**********
PI -> [
sum:procedure[[integer,float]](list(Or(integer,float)))
status:integer
result:[integer,float]
]
RetTypeSet -> {}
ThrownExceptionSet -> {}
RetFlag -> not_aret
**********COMMAND-SEQUENCE-ANNOTATION END**********
Annotated AST generated.
The program type-checked correctly.
```

Figure 4.1.: Parsing and Type Checking Output

$$hasTypeAnything(\tau, \tau seq) \Leftrightarrow \tau \neq anything \wedge hasTypeAnything(\tau seq) \ldots$$

The other auxiliary predicate *orCombineSeq* used above is defined in Appendix C.

4.3. A Type Checker for MiniMaple

In this section, we discuss the implementation of the type checker [91, 92, 99] and its application by an example. As discussed in Chapter 1, the type checker annotates the abstract syntax tree (AST) generated by the parser with type information respectively it may generates warning and error messages. The type checker is operational and its implementation consists of approx. 100 Java classes and 10K lines of code.

To type check the *MiniMaple* program shown in Chapter 3, we execute the command;

```
java fmrisc/typechecker/MiniMapleTypeChecker -typecheck Test6.m
```

where the file `Test6.m` contains the example program.

The output of the type checker applied to a file containing the source code of the example program is shown in Figure 4.1.

Figure 4.1 shows that the file has been successfully parsed and presents the type annotations for the assignment command. In the second part, it shows the resulting type environment with the associated program identifiers and their respective types introduced while type checking. The last message indicates that the program has type checked correctly.

The current implementation has the following limitations:

- All the code must be contained in a single *MiniMaple* file.
- Procedure and module definitions must precede their application. In the future, we may consider the following alternatives:

  - We can use forward declarations i.e. procedure/module prototypes embedded in comments in the *MiniMaple* program.
  - We can use two-pass type checking. In the first pass we can collect the procedure and module information and in second pass we can type check rest of the program with the given procedure/module definitions.
- Procedure parameter(s) and return types have to be explicitly given in the *MiniMaple* program. In the future, we may use type inference (to determine the parameter(s) and return types) by the applications of the parameter(s) and procedure(s).
- Type checking terminates at very first error message, so one cannot see all the type information flow if the type checking fails.

An application of the type checker to the Maple package *DifferenceDifferential* is discussed in Chapter 8.

5. Formal Specification Language

Based on the formal type system discussed in the previous chapter, we have developed a formal specification language for *MiniMaple*. In this chapter, we discuss the characteristic features of this language. The rest of the chapter is organized as follows: in Section 5.1, we discuss the underlying formula language of the specification language; in Section 5.2, we explain the main elements of the specification language and in Section 5.3, we give a specification example.

5.1. Formula Language

The formal specification language for *MiniMaple* is essentially a first order logic, which is mainly based on Maple notations but has been extended also by new notions as shown in Figure 5.1.

Apart from atomic formulas, the formula language supports logical connectives (**and**, **or**, **implies**, **equivalent**), various forms of quantifiers, i.e. logical quantifiers (**exists** and **forall**), numerical quantifiers/binders (**add**, **mul**, **min** and **max**) and a sequence quantifier (**seq**) representing truth values, numeric values and sequences of values respectively.

In Maple, such quantifiers are computable expressions as:

```
> l:=[2,4,6,8];
                        l := [2, 4, 6, 8]

> add(k, k in l);
                        20

> mul(k, k in l);
                        384

> seq(k+k, k in l);
                        4, 8, 12, 16
```

where **add** and **mul** quantifier are used to compute the sum and the product of all elements k **in** list l respectively. The **seq** expression generates the sequence of values

spec-expr ::= spec-expr **and** spec-expr | spec-expr **or** spec-expr
| spec-expr **equivalent** spec-expr | spec-expr **implies** spec-expr
| **forall**(Itseq, spec-expr) | **exists**(Itseq, spec-expr)
| it-op(spec-expr, binding, (EMPTY | spec-expr))
| spec-expr Bop spec-expr | Uop spec-expr | **type**(spec-expr,T)
...
| **'if'**(spec-expr1, spec-expr2, spec-expr3)
| **LET** Iseq=eseq **IN** spec-expr
| I | N | **OLD** I | **RESULT**
...
binding ::= I = spec-expr1...spec-expr2 | I **in** spec-expr
it-op ::= **add** | **mul** | **max** | **min** | **seq**
...

Figure 5.1.: Syntactic Domains of Formula Language and the Related Domains

$k + k$ iterating over all the elements k of the list l.

The conditional operator **'if'** in Maple accepts three arguments, a conditional expression and two other expressions of the same type; based on the evaluation of the conditional expression, it evaluates one of the other expressions and returns its value.

```
> x:=12;
                        x := 12

> 'if'(x*x<100, true, false);
                        false
```

In the specification language for *MiniMaple*, we have extended the Maple syntax, e.g., logical quantifiers use typed variables and numerical quantifiers are equipped with logical conditions that filter values from the specified variable range by a corresponding property. Thus the following are legal specification expressions:

```
mul(k, k in l, k*k < 50)
seq(k+k, k in l, k < 8)
add(k, k in l, k < 5)
```

For example, the application of the last quantifier sums those elements k in the list l, which are less than 5.

The specification language also supports the local definition of variables (by **LET-IN** construct). The definition

```
LET
    l = [1,-2,3,-4]
IN
    mul(k, k in l, k > 0)
```

locally introduces a list l which is used in the numerical quantifier.

For the complete syntactic definition of the formula language, please see Appendix B. In the following section, we explain the use of the formula language for the formal specification of various *MiniMaple* constructs.

5.2. Specification Elements

The formal specification language allows to formally describe the behavior of *MiniMaple* procedures by pre- and post-conditions and other constraints; it also supports loop specifications and command annotations. The specification language also allows the definitions of abstract data types to formalize mathematical concepts in general and computer algebra concepts in particular; existing behavioral specification languages (such as the Java Modeling Language [71]) are weaker in this respect. For specifying mathematical notions, the use of abstract data types is more simpler than specifying with their underlying representation, i.e. by concrete data types. Also other related facts of abstract concept can be formalized for better and easier reasoning.

The *MiniMaple* type checker also checks the correct typing of formal specifications. We have used the specification language to formally specify the Maple package *DifferenceDifferential*, which demonstrates the adequacy of the language for the intended purpose. In the following, we discuss the main elements of the specification language by examples.

5.2.1. Mathematical Theories

At the top of a *MiniMaple* program one can define mathematical theories by declaring respectively defining mathematical functions, named types, abstract data types and axioms. The syntax of specification declarations

decl ::= EMPTY
| (**define**(*I(Itseq)::T,rules*); | **‘type/**I‘:=T; | **‘type/**I‘;
| **assume**(*spec-expr*);) *decl*

is mainly borrowed from Maple. For example, the mathematical function *fac* can be defined using the Maple **define** construct as follows:

```
> define( fac,
              fac(0) = 1
              fac(n::integer) = n*fac(n - 1) );
```

```
> fac(5);
                          120
```

However, as we believe type checking to be the pre-requisite of program correctness, we demand type annotations in mathematical function definitions. In *MiniMaple*, thus the factorial function is defined as follows:

define(fac(i::integer)::integer, fac(0) = 1, fac(n::integer) = n * fac(n -1));

Names data types can be defined with the phrase "**‘type/**I‘:=T“, e.g. in the following declaration an identifier ListInt stands for the type **list(integer)**:

‘type/ListInt‘:=**list**(**integer**);

The phrase "**‘type/**I‘" can be used to declare an abstract data type with name I, e.g. the following example shows the declaration of an abstract data type “difference differential operator (DDO)”:

‘type/addo‘;

Axioms can be introduced by the phrase “**assume**(*spec-expr*)“; the following example shows an axiom that an operator 't' is a difference-differential operator, if each of it's term is a difference-differential term:

assume(forall(t::list(ddoterm), d::ddo_data, isddo(t, d) equivalent
forall(i::integer, 1<=i and i<=nops(t) implies isddo_term(d, t[i]))));

The abstract data type "addo“ is used in the specification and verification of the package *DifferenceDifferential* described in Chapter 8.

5.2.2. Procedure Specifications

A specification of a *MiniMaple* procedure consists of a pre-condition, the set of global variables that can be modified (by an execution of the body of the procedure) and the post-condition, describing the relationship between pre- and post-state. By an optional clause we can also specify the exceptional behavior of a procedure. The procedure specification syntax is influenced by the Java Modeling Language [71]:

proc-spec ::= **requires** *spec-expr*;
global *Iseq*;
ensures *spec-expr*; *excep-clause*

Figure 5.2 shows an example for the procedure specification. The specification is a big logical disjunction to formulate two possible behaviors of the procedure:

1. when the procedure terminates normally and
2. when the procedure terminates prematurely.

The figure gives a formal specification of the example procedure introduced in Chapter 3. The procedure has no pre-condition as shown in the **requires** clause; the **global** clause says that a global variable *status* can be modified by the body of the procedure.

```
(*@
  requires true;
  global status;
  ensures
    (status = -1 and RESULT[1] = add(e, e in l, type(e,integer))
    and RESULT[2] = add(e, e in l, type(e,float))
    and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],integer) implies l[i]<>0)
    and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],float) implies l[i]>=0.5))
    or
    (1<=status and status<=nops(l)
    and RESULT[1] = add(l[i], i=1..status-1, type(l[i],integer))
    and RESULT[2] = add(l[i], i=1..status-1, type(l[i],float))
    and ((type(l[status],integer) and l[status]=0)
        or (type(l[status],float) and l[status]<0.5))
    and forall(i::integer, 1<=i and i<status and type(l[i],integer) implies l[i]<>0)
    and forall(i::integer, 1<=i and i<status and type(l[i],float) implies l[i]>=0.5));
  @*)
proc(l::list(Or(integer,float)))::[integer,float]; ... end proc;
```

Figure 5.2.: A *MiniMaple* Procedure Formally Specified

The normal behavior of the procedure is specified in the **ensures** clause.

The post-condition specifies that, if the complete list is processed then we get the result as the sum of all integers and floats in the list; if the procedure terminates pre-maturely, then we only get the sum of integers and floats till the index denoted by the variable *status*. The variable **RESULT** is a keyword of the specification language, which represents the return value of the procedure.

In the example, the numerical quantifier **add** sums those elements e of the input list l that satisfy the given property, i.e. a type test **type**(e,**integer**) or **type**(e,**float**), respectively.

5.2.3. Loop Specifications

The specification language allows to formally specify the total correctness of a *MiniMaple* loop by an invariant and by a termination term that denotes a non-negative integer.

loop-spec := **invariant** *spec-expr*; **decreases** *spec-expr*;

An invariant is used to generate the conditions to verify partial correctness of the loop; it must hold before and after each iteration of the loop body. A termination term is added to specify the total correctness of the loops; it denotes a value that is decremented by every iteration of the loop such that the loop eventually terminates.

Figure 5.3 shows the formally specified loop from the example program. In analogy to the post-condition of the procedure specification, the loop invariant is a big logical disjunction which specifies the relationship among the variables that are modified in the loop body

1. when the body of the loop executes normally and
2. when the loop terminates abnormally, i.e. with the execution of the **return** statement.

In detail, the loop specification says that at each iteration of the loop,

- either *si* and *sf* equal the sum of the corresponding integer and float values of the elements of list *l* until index *status*. Furthermore no integer of value 0 or float of value less than 0.5 has been found before index *status*;
- or *si* and *sf* equal the sum of the corresponding integer and float values of the elements of the list *l* until index *status*-1; at index *status*, either there is an integer of value 0 or a float of value 0.5. Moreover, there is no integer of value 0 or a float value of less than 0.5 until index *status*-1.

The termination term (as specified by the **decreases** clause) decrements after each iteration because the subtrahend of the termination term, i.e. the loop counter *i* gets incremented after each iteration. After the last iteration, the subtrahend equals one more than the length of the list *l* and hence the termination term still becomes non-negative.

5.2.4. Assertions

Any *MiniMaple* statement can be specified by assertions. Semantically, an assertion constrains the state of the execution at the point where it occurs. Furthermore, an assertion splits the verification proof into two parts,

1. a proof obligation and
2. an assumption for the rest of the proof.

In the formal specification language of *MiniMaple*, an assertion has the syntax borrowed from Maple:

ASSERT(*spec-expr*, (EMPTY | "*I*"));

An assertion can be named by an optional identifier *I* as in

ASSERT(**type**(y,**integer**), "y is not an integer");

where, if the assertion **type**(y, integer) fails, the message "y is not an integer" is printed.

In Maple, an assertion is checked when the variable *assertlevel* of the kernel routine *kernelopts* is set to 1. For instance, in the example

```
> kernelopts(assertlevel=1);
                0
```

```
for i from 1 by 1 to nops(l) do
(*@
invariant (status <= i and
           (si = add(l[j], j=1..status, type(l[j],integer)) and
           sf = add(l[j], j=1..status, type(l[j],float)) and
           forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],integer)
                  implies l[i0]<>0) and
           forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],float)
                  implies l[i0]>=0.5)
           ))
           or
           ( si = add(l[j], j=1..status-1, type(l[j],integer)) and
           sf = add(l[j], j=1..status-1, type(l[j],float)) and
           ((type(l[status],integer) and l[status]=0)
                  or (type(l[status],float) and l[status]<0.5)) and
           forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],integer)
                  implies l[i0]<>0) and
           forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],float)
                  implies l[i0]>=0.5)
           );
decreases (nops(l) + 1 - i);
@*)
x:=l[i];
status:=i;
if type(x,integer) then
        if (x = 0) then
           return [si,sf];
        end if;
        si:=si+x;
elif type(x,float) then
          if (x < 0.5) then
             return [si,sf];
          end if;
          sf:=sf+x;
end if;
end do;
```

Figure 5.3.: A *MiniMaple* Loop Formally Specified

```
> x:=12;
                x := 12

> ASSERT(x>10);
> ASSERT(x<10);
Error, assertion failed
```

when an assertion is violated, Maple reports an error, otherwise it continues the execution.

5.3. Example

The complete specification of the example program presented in the previous sections is given:

```
status:=0;
sum := proc(l::list(Or(integer,float)))::[integer,float];
(*@
requires true;
global status;
ensures
(status = -1 and RESULT[1] = add(e, e in l, type(e,integer))
and RESULT[2] = add(e, e in l, type(e,float))
and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],integer) implies l[i]<>0)
and forall(i::integer, 1<=i and i<=nops(l) and type(l[i],float) implies l[i]>=0.5))
or
(1<=status and status<=nops(l)
and RESULT[1] = add(l[i], i=1..status-1, type(l[i],integer))
and RESULT[2] = add(l[i], i=1..status-1, type(l[i],float))
and ((type(l[status],integer) and l[status]=0)
or (type(l[status],float) and l[status]<0.5))
and forall(i::integer, 1<=i and i<status and type(l[i],integer) implies l[i]<>0)
and forall(i::integer, 1<=i and i<status and type(l[i],float) implies l[i]>=0.5));
@*)
global status;
local i,x::Or(integer,float), si::integer:=0, sf::float:=0.0;
for i from 1 by 1 to nops(l) do
(*@
invariant (status <= i and
        (si = add(l[j], j=1..status, type(l[j],integer)) and
        sf = add(l[j], j=1..status, type(l[j],float)) and
          forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],integer)
                                              implies l[i0]<>0) and
        forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],float)
                                              implies l[i0]>=0.5)
        )
        or
```

```
        ( si = add(l[j], j=1..status-1, type(l[j],integer)) and
        sf = add(l[j], j=1..status-1, type(l[j],float)) and
        ((type(l[status],integer) and l[status]=0)
                or (type(l[status],float) and l[status]<0.5)) and
          forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],integer)
                                                implies l[i0]<>0) and
        forall(i0::integer, 0 <= i0 and i0 <= status and type(l[i0],float)
                                                implies l[i0]>=0.5)
        );
decreases (nops(l) + 1 - i);
@*)
        x:=l[i];
        status:=i;
        if type(x,integer) then
                if (x = 0) then
                        return [si,sf];
                end if;
                si:=si+x;
        elif type(x,float) then
                if (x < 0.5) then
                        return [si,sf];
                end if;
                sf:=sf+x;
        end if;
end do;
status:=-1;
return [si,sf];
end proc;
```

We type check this program by executing the following command:

```
java fmrisc/typechecker/MiniMapleTypeChecker -typecheck Test66.m
```

which accepts the specified program as correctly typed. The application of the type checker to the formally specified Maple package *DifferenceDifferential* is discussed in Chapter 8.

6. Formal Semantics

In this chapter, we define a formal semantics of *MiniMaple* programs as a pre-requisite of our translation, which will be discussed in Chapter 7: this translation of a *MiniMaple* annotated program into a corresponding Why3ML program must be sound with respect to the semantics. The rest of the chapter is organized as follows: Section 6.1 highlights selected features of *MiniMaple* program's semantics. Section 6.2 introduces the background for the definition of the formal semantics. In Section 6.3, we discuss the formal semantics of *MiniMaple* programs, while Section 6.4 and 6.5 sketch the formal semantics of the formula language and of the specification annotations, respectively. The complete definitions of the semantics of *MiniMaple* and its specification language are presented in Appendices D and E, respectively.

6.1. Introduction

There is no formally defined semantics for Maple such that only the implementation of Maple can be considered as a basis of our semantics definition which attempts to depict the internal behavior of Maple. Based on this semantics, we can formalize the question about the correct behavior of any *MiniMaple* program.

Our formal semantics of *MiniMaple* correspondingly shows the following features:

- *MiniMaple* has expressions with side-effects, which is not supported in functional programming languages, e.g. Haskell [82] and Miranda [153]. As a result the evaluation of an expression may change the program execution state.
- The semantics is correspondingly defined in a denotational style as a state relationship between pre- and post-states.
- Static scoping [114] is used to evaluate a *MiniMaple* procedure.
- *MiniMaple* and its specification language share various semantic domains of values that have some non-standard types of objects, for example symbol, uneval and union etc. These languages also support additional functions and predicates, for example type tests i.e. $\mathbf{type}(E, T)$.

In the following Section, we introduce some background notions which will be used to define the formal semantics in the subsequent sections.

6.2. Background

In this section, we discuss the structure of our definition of denotational semantics. A denotational semantics is defined with the help of various *semantic domains* [1], which represents sets of elements that share some common properties. A semantic domain is accompanied by a set of operations as functions over the domain. A domain and its operations together form a *semantic algebra* [136]. A *valuation function* defines a mapping from an abstract syntax structure of a language to its corresponding meaning which is an element of a semantic domain. A valuation function VF for a syntax domain VF is usually formalized by a set of equations, one per alternative in the corresponding BNF rule for the syntactic domain. The abstract syntax domains for *MiniMaple* and its specification language are defined in Appendices A and B respectively, while the corresponding semantic algebras and the valuation functions are defined in Appendices D and E.

The most important semantic domains are introduced in the following.

6.2.1. Semantic Values

A *Value* is a disjunctive union domain composed of all kinds of primitive semantic values supported in *MiniMaple*:

$$Value := Module + Procedure + Function + List + Set + ... + Uneval + Value^*$$

Some of these domains, e.g. *Module*, *Procedure* and *Function* are explained in the following subsections. Note that the domain *Value* is a recursive domain, e.g. *List* is defined by a sequence of values *Value** as discussed in the Section 6.2.5.

6.2.2. Module Values

A *Module* can be thought of as a collection of name bindings:

$$Module := Identifier\_Sequence \rightarrow Value^*$$

These bindings are accessible outside the module, once the module has been constructed. They are defined by the **exports** of the module.

6.2.3. Procedure Values

The semantic domain *Procedure* represents *MiniMaple* procedures. It is defined as a relation on a sequence of (parameter) values, a pre- and a post-state and a return value.

$$Procedure := \mathbb{P}(Value^* \times State \times StateU \times ValueU)$$

A *Procedure* is one of the values that can be stored in the *Environment* values as discussed in Subsection 6.2.7. The domain *State* and other lifted domains *StateU* and *ValueU* are defined in Subsections 6.2.8 and 6.2.9 respectively.

6.2.4. Function Values

The semantic domain *Function* defines and formalizes the mathematical functions supported in the specification language as follows:

$$Function := \bigcup_{n \in \mathbb{N}} Function^n$$

where

$$Function^n := Value^n \rightarrow Value$$

i.e. a value of type *Function* maps n parameter values to a return value. A predicate is a special case of a mathematical function which returns a boolean value.

6.2.5. List Values

The structure of domain *List* is defined as a finite sequence of elements *Value*:

$$List := Value^*$$

The semantic domain *List* is used as a building block for some other domains, e.g. *Tuple* and *Set*. Furthermore, the domains *List* and *Set* are defined as a sequence of values belonging to a single domain.

6.2.6. Sequence Values

The domain for a finite sequence of values $Value^*$ is defined by two constructors

$emptyValue : () \rightarrow Value^*$
$cons : Value \times Value^* \rightarrow Value^*$

which create an empty and the finite sequences respectively.

The formalism of our semantics does require some auxiliary semantic domains; the important of which are discussed in the following sections.

6.2.7. Environment Values

The domain *Environment* holds for the environment of a *MiniMaple* program. *Environment* is formalized as a Cartesian product of domains *Context* and *Space*.

$$Environment := Context \times Space$$

where the domain *Context* is a mapping of identifiers to the environment values (*Variable*, *Procedure*, *Function* and *Type-Tag*):

$Context := Identifier \to EnvValue$
$EnvValue := Variable + Procedure + Function$

The domain *Space* models the memory space

$$Space := \mathbb{P}(Variable)$$

as a pool of variables that are not assigned to any identifiers and can be used for allocation of program variables.

6.2.8. State Values

This section defines the domain for the *State* of the program, which is composed of a *Store* and a *Data* object:

$$State := Store \times Data$$

A *Store* holds for every *Variable* a *Value*, while *Data* stores the control information of a particular state.

$Store := Variable \to Value$
$Data := Flag \times Exception \times Return$
$Flag := \{execute, exception, return, leave\}$

An *Exception* and *Return* domains give the corresponding exception and return values based on the value of *Flag*.

6.2.9. Lifted Values

The evaluation of some semantic domains might result in an illegal state or an undefined value. To address these unsafe evaluations we lifted the domains of *State* and *Value* to domains *StateU* and *ValueU*, which are disjoint sums of the basic domains and the domains *Error* and *Undefined*, respectively.

$ValueU := Value + Undefined$
$StateU := State + Error$

Undefined ($:= \{()\}$) and *Error* ($:= \{()\}$) are unit domains.

6.3. Semantics of Programs

Based on the semantic domains introduced in the previous section, we define the valuation functions for selected syntactic domains of *MiniMaple* in this section.

6.3.1. Commands

As the formal semantics of *MiniMaple* commands is defined as a state relationship, we define the result of the corresponding valuation functions as a predicate. A valuation function for commands takes the abstract syntax of a command as a value of type C and results in a *ComRelation*:

$$[\![C]\!]\colon \mathit{ComRelation}$$

where

$\mathit{ComRelation} := \mathit{Environment} \to \mathit{StateRelation}$
$\mathit{StateRelation} := \mathbb{P}(\mathit{State} \times \mathit{StateU})$

If we unfold the definition of the above valuation function signature can be rewritten as follows:

$$[\![]\!]\colon \mathit{Command} \to \mathit{Environment} \to \mathit{StateRelation}$$

A valuation function for a command thus takes a command and an environment and results in a power set of pairs of pre- and post-states of the execution of the command.

In the following, we give some examples for the definition of the valuation function of a command.

Assignments

MiniMaple supports a simultaneous multi-assignment statement, whose semantics is defined as a relationship between pre-state s and post-state s' as shown below:

$[\![I,Iseq := E,Eseq]\!](e)(s,s') \Leftrightarrow$
$\exists\, v \in \mathit{ValueU}, s'' \in \mathit{StateU} : [\![E]\!](e)(s,s'',v) \wedge$
cases v of
$\quad \mathit{isUndefined}() \to s' = \mathit{inError}()$
$\quad [] \mathit{isValue}(v') \to$
$\quad\quad$ cases s'' of
$\quad\quad\quad \mathit{isError}() \to s' = \mathit{inError}()$
$\quad\quad\quad [] \mathit{isState}(p) \to \exists\, v'' \in \mathit{ValueU}^*, s''' \in \mathit{State} : [\![\mathit{Eseq}]\!](e)(p,s''',v'') \wedge$
$\quad\quad\quad\quad$ cases s''' of
$\quad\quad\quad\quad\quad \mathit{isError}() \to s' = \mathit{inError}()$
$\quad\quad\quad\quad\quad [] \mathit{isState}(p_1) \to$
$\quad\quad\quad\quad\quad\quad$ IF $\mathit{undefinedSeq}(v'')$ THEN
$\quad\quad\quad\quad\quad\quad\quad \exists\, var \in \mathit{Variable}, l_1 \in \mathit{List}, vars \in \mathit{Variable}^*, l_n \in \mathit{List}^* :$
$\quad\quad\quad\quad\quad\quad\quad [\![I]\!](e)(var,l_1) \wedge [\![\mathit{Iseq}]\!](e)(vars,l_n) \wedge$
$\quad\quad\quad\quad\quad\quad\quad s' = \mathit{inValueU}(\mathit{update}(p_1, <var, vars>, <l_1, l_n>, <v', \mathit{valSeq}(v'')>))$
$\quad\quad\quad\quad\quad\quad$ ELSE $s' = \mathit{inError}()$

```
            END //if
        END //cases-s'''
    END //cases-s'''
END //cases-v
```

Semantically, with the given environment e and a pre-state s, first the expression sequence (E and $Eseq$ respectively) of the assignment command is evaluated:

- if none of them yields an unsafe evaluation (i.e. an error state or an undefined value), then
- the identifier sequence (i.e. left-hand-side) of the assignment statement is evaluated, and
- consequently, a post-state s' is computed by simultaneously updating the value of identifier sequence to the corresponding values in the state p_1 (computed by the evaluation of the corresponding expression sequence).

If any of the evaluation is unsafe, then the post-state of the assignment command is an error state.

Command Sequences

Also the semantics of *MiniMaple* command sequence states the relationship between a pre-state s and a post-state s' as follows:

$$
\begin{array}{l}
[\![C;\ Cseq]\!](e)(s, s') \Leftrightarrow \\
\ \exists\ s'' \in StateU : [\![C]\!](e)(s, s'') \wedge \\
\quad \text{cases } s'' \text{ of} \\
\qquad isError() \rightarrow s' = inError() \\
\qquad [] isState(p) \rightarrow \\
\qquad\ \text{IF } executes(data(p)) \text{ THEN} \\
\qquad\qquad \text{LET } e' = Env(e, C) \text{ IN} \\
\qquad\qquad\qquad [\![Cseq]\!](e')(p, s') \\
\qquad\ \text{ELSE } s' = inStateU(p) \\
\qquad\ \text{END //if} \\
\quad \text{END //cases-s''}
\end{array}
$$

If the execution of a command C yields a post-state s'', then the execution of a command sequence $Cseq$ in a pre-state s'' results in a post-state s'.

While-loops

MiniMaple supports the typical while-loop, whose semantics is given below:

$$[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end do}]\!](e)(s, s') \Leftrightarrow$$

$\exists\, k \in Nat, t, u \in StateU^* :$
$t(0) = inStataU(s) \wedge u(0) = inStateU(s) \wedge$
$(\forall\, i \in Nat_k : iterate(i, t, u, e, [\![E]\!], [\![Cseq]\!])) \wedge$
$((u(k) = inError() \wedge s' = u(k)) \vee$
$(returns(data(inState(u(k)))) \wedge s' = t(k)) \vee$
$(\exists\, v \in ValueU : [\![\mathrm{E}]\!](e)(inState(t(k)), u(k), v)$
$\wedge v <> inValue(inBoolean(True)) \wedge$
IF $v = inValue(inBoolean(False))$ THEN
$s' = t(k)$
ELSE $s' = inError()$
END
)
)

The semantics of the while-loop is determined by the two sequences of *pre* and *post* states [137]. Both sequences are constructed from the pre-state of the loop. Any *ith* iteration (execution of the body) of the loop transforms state *pre(i)* into state *post(i+1)* from which the state *pre(i+1)* is constructed. No iteration is allowed from the *Error* as *pre* state. The loop terminates when the guard expression E evaluates to *false* or when the body of the loop evaluates to an error post-state. The corresponding *iterate* predicate formalizes the aforementioned while-loop semantics, which is defined as a relation on

- number of iterations i,
- a sequence of pre-states t,
- a sequence of post-states u,
- an environment e in which the body of the loop (command sequence) has to be evaluated,
- a valuation function for the loop condition expression E and
- a valuation function for the body o the loop (command sequence) C.

Here the pre and post-states refer to the corresponding pre and post-states of the execution of the body of the loop.

$iterate \subseteq Nat \times StateU^* \times StateU^* \times Environment \times$
$StateValueRelation \times StateRelation$
$iterate(i, t, u, e, E, C) \Leftrightarrow$
cases $t(i)$ of
$isError() \rightarrow false$
$[]\, isState(m) \rightarrow executes(data(m)) \wedge$
$\exists\, v \in ValueU, s' \in StateU : E(e)(m, s', v) \wedge$
cases s' of
$isError() \rightarrow u(i+1) = inError() \wedge t(i+1) = u(i+1)$

$$
\begin{array}{l}
[]\mathit{isState}(p) \rightarrow \\
\quad \text{cases } v \text{ of} \\
\quad\quad \mathit{isUndefined}() \rightarrow u(i+1) = \mathit{inError}() \wedge t(i+1) = u(i+1) \\
\quad\quad []\mathit{isValue}(v') \rightarrow \\
\quad\quad\quad \text{cases } v' \text{ of} \\
\quad\quad\quad\quad \mathit{isBoolean}(b) \rightarrow b \wedge C(e)(p, u(i+1)) \wedge t(i+1) = u(i+1) \\
\quad\quad\quad\quad [] \ldots \rightarrow u(i+1) = \mathit{inError}() \wedge t(i+1) = u(i+1) \\
\quad\quad\quad \text{END } //\text{cases-v'} \\
\quad\quad \text{END } //\text{cases-v} \\
\quad \text{END } //\text{cases-s'} \\
\text{END } //\text{cases-t(i)}
\end{array}
$$

The predicate *iterate* is defined such that, at an arbitrary iteration i of the loop,

- if the pre-state $t(i)$ is a non-error state m, then
- if the state m is an executing state (i.e. no-exception and no return), then, with the given environment e and a pre-state m, the loop condition expression E evaluates to value v and results in the post-state s', then
- if the resulting post-state s' is a non-error state and the value of expression v is not undefined, then
- if the value of expression v is a boolean *true* value, then, with the given environment e and a pre-state p, the execution of the body of the loop C produces $u(i+1)$ as a post-state, which is then transformed to the pre-state $t(i+1)$ for the next iteration of the loop.

In the alternative of any of the aforementioned conditions, the post-state $u(i+1)$ is set to an error state, which consequently results in an error pre-state $t(i+1)$. However, if the pre-condition $t(i)$ for any iteration i is an error state, then the predicate returns $false$.

6.3.2. Expressions

The valuation function for the abstract syntax domain of expression values E is defined as:

$$[\![E]\!]\colon \mathit{ExpRelation}$$

where

$$\mathit{ExpRelation} := \mathit{Environment} \rightarrow \mathit{StateValueRelation}$$
$$\mathit{StateValueRelation} := \mathbb{P}(\mathit{State} \times \mathit{StateU} \times \mathit{ValueU})$$

The valuation function for an expression takes an expression and an environment and results in a power set of triples of pre-state, post-state and the value of the expression.

In the following, we give some examples for the definition of the valuation functions of a *MiniMaple* expressions.

Binary Expressions

MiniMaple supports various kind of binary operations, e.g. arithmetic and logical expressions, whose abstract syntax is represented by the domain of value of type *Bop*. The semantics of such expressions state that the evaluation of the binary operator *Bop* (operating over expression E_1 and E_2) in a pre-state s yields a post-state s' and a value v as defined below:

$$
\begin{array}{l}
[\![E_1\ Bop\ E_2]\!](e)(s, s', v) \Leftrightarrow \\
\ \exists\, s_1 \in StateU, v_1 \in ValueU : [\![E_1]\!](e)(s, s_1, v_1) \wedge \\
\ \text{cases } s_1 \text{ of} \\
\quad isError() \rightarrow s' = inError() \wedge v = inUndefined() \\
\quad [] isState(s11) \rightarrow \\
\quad\ \text{cases } v_1 \text{ of} \\
\quad\quad isUndefined() \rightarrow s' = inError() \wedge v = inUndefined() \\
\quad\quad [] isValue(v_{11}) \rightarrow \\
\quad\quad\ \exists\, s_2 \in StateU, v_2 \in ValueU : [\![E_2]\!](e)(s_{11}, s_2, v_2) \wedge \\
\quad\quad\ \text{cases } s_2 \text{ of} \\
\quad\quad\quad isError() \rightarrow s' = inError() \wedge v = inUndefined() \\
\quad\quad\quad [] isState(s22) \rightarrow \\
\quad\quad\quad\ \text{cases } v_2 \text{ of} \\
\quad\quad\quad\quad isUndefined() \rightarrow s' = inError() \wedge v = inUndefined() \\
\quad\quad\quad\quad [] isValue(v_{22}) \rightarrow \\
\quad\quad\quad\quad\ \exists\, v' \in Value : [\![Bop]\!](v_{11}, v_{22})(v') \wedge \\
\quad\quad\quad\quad\quad\quad\quad\quad s' = inStateU(s_{22}) \wedge v = inValueU(v') \\
\quad\quad\quad\ \text{END } //\text{cases-}v_2 \\
\quad\quad\ \text{END } //\text{cases-}s_2 \\
\quad\ \text{END } //\text{cases-}v_1 \\
\ \text{END } //\text{cases-}s_1
\end{array}
$$

Semantically, first the expression E_1 is evaluated in a given environment e and pre-state s, if

- the evaluation yields a value v_1, then
- the expression E_2 is evaluated in a pre-state s_{11} (which is a yielded post-state by the evaluation of expression E_1) and if this evaluation yields a value v_2, then
- the application of the binary operator *Bop* to the values v_{11} and v_{22} computes the result value v' which equals v.

Any unsafe evaluation results in an undefined value v.

Procedures

As discussed earlier in Section 6.2.3, a *MiniMaple* procedure expression evaluates to a *Procedure* value, which is defined as a predicate. Moreover, static scoping is used to evaluate a *MiniMaple* procedure.

In the following we define the corresponding definition time valuation function where a procedure expression evaluates to a procedure predicate value p. Here, *Pseq*, S and R represent the parameter sequence (identifiers with corresponding types), declarations and body (command sequence) of the procedure, respectively.

$[\![\textbf{proc}(Pseq)\ S;\ R\ \textbf{end proc}]\!](e)(s, s', v) \Leftrightarrow$
LET $p \in Procedure, p(valseq, s_0, s_1, v') \Leftrightarrow$
 LET $e' = push(e, identifiers(Pseq))$
  $\exists varseq \in Variable*, s'', s_3 \in StateU,$
   $e'', e''' \in Environment$:$[\![\text{Pseq}]\!](e')(e'', valseq) \wedge$
    $[\![\text{S}]\!](e'')(s_0, s'', e''') \wedge$
    cases s'' of
     $isError() \rightarrow inError()$
     $[]isState(s_4) \rightarrow \exists\, s_2 \in State, v'' \in ValueU$:
      $s_2 = update(s_4, varseq, valseq) \wedge [\![\text{R}]\!](e''')(s_2, s_3, v'')$
    END
 IN cases s_3 of
   $isError() \rightarrow inError()$
   $[]isState(s_5) \rightarrow$
    cases v'' of
     $isUndefined() \rightarrow s_1 = inError() \wedge v' = inUndefined()$
     $[]isValue(v_1) \rightarrow s_1 = inStateU(s_5) \wedge v' = inValueU(v_1)$
    END
   END
 END
IN $s' = inStateU(s) \wedge v = inValueU(p)$ END

The valuation function for a procedure expression in a given environment e and a pre-state s evaluates to a procedure value v, i.e. a procedure $p(valseq, s_0, s_1, v')$, where *valseq* is the sequence of parameter values, s_0 and s_1 are the corresponding pre and post-states of the procedure, while v' is the return value of the procedure. Note here that the evaluation of the procedure expression does not change the post-state of the expression, i.e. the post-state s' is equivalent to the pre-state s.

Procedure Calls

In a procedure call, first, the argument expression sequence is evaluated; if any of them yields an unsafe result, then the call-expression evaluates to an *Undefined* value

and an *Error* as a post-state.

$[\![I(Eseq)]\!](e)(s, s', v) \Leftrightarrow$
LET $vseq \in ValueU^*, s_1 \in StateU$: $[\![Eseq]\!](e)(s, s_1, vseq)$
IN
cases s_1 of
  $isError() \rightarrow s' = inError() \wedge v = inUndefined()$
  $[]\, isState(s_2) \rightarrow$
   IF $hasUndefinedValue(vseq)$ THEN
      $s' = inError() \wedge v = inUndefined()$
   ELSE
   cases $[\![I]\!](e)$ of
     $isProcedure(p) \rightarrow \exists\, s_3 \in StateU, v_1 \in ValueU : p(vseq, s_2, s_3, v_1) \wedge$
      cases s_3 of
        $isError() \rightarrow s' = inError() \wedge v = inUndefined()$
        $[]\, isState(s_4) \rightarrow$
         cases v_1 of
           $isUndefined() \rightarrow s' = inError() \wedge v = inUndefined()$
           $[]\, isValue(v') \rightarrow s' = inStateU(s_4) \wedge v = inValueU(v')$
         END //cases-v_1
        END //cases-s_3
      $[] \ldots \rightarrow s' = inError()$
   END //cases-$[\![I]\!]$
   END //IF-$hasUndefinedValue$
  END //cases-s_1
END //LET

Otherwise, the environment e is looked up for the procedure named I with value $p(vseq, s_2, s_3, v_1)$. This procedure p is applied to the argument values which yields a command behavior; the post-state of the command sequence execution is set to the post-state of the procedure call expression and the procedure call expression evaluates to the value of the procedure.

6.4. Semantics of Specification Expressions

In this section, we first discuss the signatures of a valuation function of the specification expression and then define the valuation functions for various interesting expressions of the formula language.

The valuation function for the abstract syntax domain specification expression of values *spec-expr* is defined as:

$$[\![spec\text{-}expr]\!]: Environment \rightarrow StateResultValueRelation$$

where

$$StateResultValueRelation := \mathbb{P}(State \times StateU \times ValueU \times ValueU)$$

is a power set of a pre-state, a post-state, a (procedure) result value and the value of the expression. Here, the post-state can be an *Error* state and also the evaluated value can be *Undefined*.

Variables

OLD I is an expression that refers to the value of identifier I in the previous state.

$$[\![\textbf{OLD } I]\!](e)(s, s', r, v') \Leftrightarrow v' = inValueU(store(s)([\![I]\!](e))) \wedge s' = inStateU(s)$$

The semantics of the old expression is the value v' of the identifier I looked up in the previous state s.

The expression **RESULT** refers to the result (return) value of a *MiniMaple* procedure expression and is defined as:

$$[\![\textbf{RESULT}]\!](e)(s, s', r, v') \Leftrightarrow v' = inValueU(r) \wedge s' = inStateU(s)$$

The value of this expression is provided as the third parameter of the predicate.

Conditionals

The specification language supports a conditional operator whose semantics is defined as follows:

$$\begin{array}{l}
[\![\textbf{'if'}(\textit{spec-expr}_1, \textit{spec-expr}_2, \textit{spec-expr}_3)]\!](e)(s, s', r, v') \Leftrightarrow \\
\exists\, v_1 \in ValueU : [\![\textit{spec-expr}_1]\!](e)(s, s', r, v_1) \wedge \\
\text{IF } v_1 = inValueU(inValue(inBoolean(inTrue()))) \text{ THEN} \\
\quad [\![\textit{spec-expr}_2]\!](s, s', r, v') \\
\text{ELSE} \\
\quad [\![\textit{spec-expr}_3]\!](s, s', r, v') \\
\text{END } //\text{if-}b_1 = inTrue()
\end{array}$$

The semantics of a conditional expression says that the $\textit{spec-expr}_1$ is evaluated first, if it yields *true* then the specification expression $\textit{spec-expr}_2$ is evaluated that gives the semantic value v', otherwise the specification expression $\textit{spec-expr}_3$ is evaluated to a result value v'.

Local Definitions

The specification language supports an evaluation of a specification expression with a local definition (by the **LET-IN** construct).

$$
\begin{array}{l}
[\![\textbf{LET } \mathit{Iseq} = \mathit{eseq} \textbf{ IN } \mathit{spec\text{-}expr}]\!](e)(s, s', r, v') \Leftrightarrow \\
\ \exists\, \mathit{vs} \in \mathit{ValueU}* : [\![\mathit{eseq}]\!](e)(s, s', r, \mathit{vs}) \wedge \\
\ \text{IF } \mathit{hasUndefinedValue}(\mathit{vs}) \text{ THEN} \\
\quad v' = \mathit{inUndefined}() \\
\ \text{ELSE} \\
\quad \exists\ e_1 \in \mathit{Environment} : e_1 = \mathit{push}(e, \mathit{Iseq}, \mathit{vs}) \wedge\ [\![\mathit{spec\text{-}expr}]\!](e_1)(s, s', r, v') \\
\ \text{END } //\text{if}
\end{array}
$$

First the local definitions (**LET** part) is evaluated and the specification expression sequence is evaluated; then, if none of them yields the *Undefined* value, *Environment* is updated with the identifiers (*Iseq*) mapped to the correspondingly evaluated values (expression sequence). Then the specification expression *spec-expr* (**IN** part) is evaluated in the updated *Environment*, the result of the whole **LET-IN** construct is the evaluated value of *spec-expr*.

Numerical Quantifiers/Binders

As discussed in Chapter 5, the specification language also supports numerical quantifiers (of the form $\mathit{IOp}(\mathit{SE}_1, B, \mathit{SE}_2)$) to apply a binary arithmetic operation to a range of values those satisfy a certain property. For example, the numerical quantifier `mul(e, e in l, e > 0)` computes the product of the those elements e of the list l which are greater than 0; here `mul` is the quantifier's name *IOp*, `e` is the base expression SE_1, `e in l` is the range B of the quantifier and `e > 0` is the property SE_2 to be satisfied by the quantifier. The semantics of the numerical quantifier/binder is a relationship among the pre-state (s), post-state (s'), (procedure) result value (r) and the evaluated value (v) of the iterator as defined below:

$$
\begin{array}{l}
[\![\mathit{IOp}(\mathit{SE}_1, B, \mathit{SE}_2)]\!](e)(s, s', r, v) \Leftrightarrow \\
\exists \mathit{vseq} \in \mathit{Value}* : [\![B]\!](e)(s, s', r, \mathit{inValueU}(\mathit{vseq})) \wedge \\
\ \exists k' \in \mathit{Nat}', e_1 \in \mathit{Environment}, \mathit{vs} \in \mathit{Value}* : \\
\quad e_1 = \mathit{push}(e, \mathit{getIdentifiers}(B)) \wedge \\
\quad (\ \forall i \in \mathit{Nat}'_{k'} : \mathit{iterate}(i, I, e_1, \mathit{vseq}, \mathit{vs}, [\![\mathit{SE}_1]\!], [\![\mathit{SE}_2]\!])\ ) \wedge \\
\qquad (\ k' < \mathit{length}(\mathit{vseq}) \wedge \\
\qquad\quad (\ \mathit{access}(k', \mathit{vseq}) = \mathit{isUndefined}() \vee \\
\qquad\quad \forall s \in \mathit{State}, r \in \mathit{Value} : \exists\, v_1 \in \mathit{Value}, n \in \mathit{StateU} : \\
\qquad\quad\ [\![\mathit{SE}_2]\!](e_1)(s, \mathit{inStateU}(s), r, \mathit{inValueU}(v_1)) \wedge \\
\qquad\quad \mathit{inBoolean}(v_1) = \mathit{inFalse}()\ ) \wedge v = \mathit{inUndefined}() \\
\qquad) \vee (\ k' = \mathit{length}(\mathit{vs}) \wedge v = \mathit{doIterate}(\mathit{IOp}, \mathit{vs})\ )
\end{array}
$$

Semantically, first the range B is computed to get the sequence of values; if none of these values evaluates to undefined, then the environment e is iteratively updated with each value in the range computed previously. At each iteration the (property/filter) SE_2 is evaluated; if it holds, then SE_1 is evaluated and its value is collected. If all these evaluations are safe, then we get a range of those values of SE_1 for which SE_2 holds. At the end we apply the operator IOp to these filtered values and compute the result value. The corresponding auxiliary predicate *iterate* formalizes the collection of filtered values. The relation *iterate* is defined on

- the number of iterations i (over the range of the quantifier),
- an identifier I, which is subject to a corresponding iteration in the quantifier's binding,
- an environment e,
- a sequence of all the values *vseq* of the corresponding bound expression (i.e. quantifier/binder),
- a sequence of values *vs* that are filtered from *vseq* for
- an expression SE_1 which satisfies
- the property formulated by the expression SE_2.

The corresponding definition of *iterate* is as follows:

$$
\begin{aligned}
&iterate \subseteq Nat' \times Identifier \times Environment \times Value^* \times Value^* \times \\
&\qquad StateResultValueRelation \times StateResultValueRelation \\
&iterate(i, I, e, vseq, vs, SE_1, SE_2) \Leftrightarrow \\
&\quad \exists\ e_1 \in Environment : e_1 = push(e, I, access(i, vseq)) \wedge \\
&\quad\ \forall\ s \in State, r \in Value : \\
&\quad\quad \exists\ v' \in Value : SE_1(e_1)(s, inStateU(s), r, inValueU(v')) \wedge \\
&\quad\quad\ SE_2(e_1)(s, inStateU(s), r, inValueU(inBoolean(inTrue()))) \wedge \\
&\quad\quad\ v' = access(i, vs)
\end{aligned}
$$

In detail, the relation *iterate* says that at any arbitrary iteration i,

- a given environment is extended such that identifier I is assigned the *ith* value of the value sequence *vseq*, then
- in pre-state s the expression SE_1 evaluates to the next value v' such that
- in a pre-state s the expression SE_2 (i.e. a property or filter) evaluates to *true* and
- this value v' is in the filtered values of sequence *vs*.

In essence, this predicate describes the relationship of the filtered (sequence) range of value *vs* from the given full (sequence) range of values *vseq* for a given expression SE_1 and the corresponding property SE_2.

6.5. Semantics of Specification Annotations

In this section, we define the semantics (correspondingly valuation functions) of the specification annotations for *MiniMaple*. The main specification annotations includes the syntactic domains of specification declarations, procedure specifications, loop specifications and assertions.

6.5.1. Specification Declarations

A specification declaration can be used to specify a mathematical theory and its semantics; it produces a new environment that has the corresponding theory declarations and definitions. The valuation function for a specification declaration *decl* has signature:

$$[\![decl]\!]\colon \mathit{Environment} \to \mathit{Environment}$$

The specification declaration introduces a new environment that contains the mathematical function declarations/definitions as defined below:

$[\![decl]\!](e)(e') \Leftrightarrow$
LET
$(id_1, \ldots, id_n, T_1, \ldots, T_n) = \mathit{getFunctionIdentifersAndTypes}(decl)$
$(iseq_1, \ldots, iseq_n, Tseq_1, \ldots, Tseq_n) = \mathit{getFunctionParametersAndTypes}(decl)$
$(i_1, \ldots, i_n, Td_1, \ldots, Td_n) = \mathit{getTypeIdentifiersAndTypes}(decl)$
$(ax_1, \ldots, ax_n) = \mathit{getAxioms}(decl)$
$(r_1, \ldots, r_n) = \mathit{getRules}(decl)$
IN
$\exists\, f_1, \ldots, f_n = \mathit{Function}^{n_1}, \ldots, \mathit{Function}^{n_n}, n_1, \ldots, n_n \in \mathit{Nat}',$
$tag_1, \ldots, tag_n \in \mathit{Type\text{-}Tag}, e_1, \ldots, e_n \in \mathit{Environment}:$
$n_1 = \mathit{length}(iseq_1) \wedge \cdots \wedge n_n = \mathit{length}(iseq_n) \wedge$
$[\![Td_1]\!](e)(\mathit{inType\text{-}TagU}(tag_1)) \wedge e_1 = \mathit{push}(e, i_1, tag_1) \wedge \ldots \wedge$
$[\![Td_n]\!](e_{n-1})(\mathit{inType\text{-}TagU}(tag_n)) \wedge e_n = \mathit{push}(e, i_n, tag_n) \wedge$
$e' = \mathit{push}(e_n, id_1, \ldots, id_n, f_1, \ldots, f_n) \wedge [\![r_1]\!](e') \wedge \ldots \wedge [\![r_n]\!](e')$
$\wedge$
$(\forall\ b_1, \ldots, b_n \in \mathit{Boolean}, s \in \mathit{State}, r \in \mathit{Value}:$
$[\![ax_1]\!](e')(s, \mathit{inStateU}(s), r, \mathit{inValueU}(\mathit{inValue}(b1))) \wedge \ldots$
$[\![ax_n]\!](e')(s, \mathit{inStateU}(s), r, \mathit{inValueU}(\mathit{inValue}(bn)))$
$\Rightarrow b_1 = \mathit{inTrue}() \wedge \ldots \wedge b_n = \mathit{inTrue}()$
) //END //let-in

In detail, first from a given declaration (*decl*) all the function definitions (function identifiers and corresponding rules), axioms (specification expressions) and type declarations (type identifiers and corresponding types) are collected and then

- the type of each type identifier is evaluated which introduces a new environment where the type identifier is mapped to its corresponding type. The evaluation of all the type identifiers produces the environment e_n;
- the environment e_n is updated to the result environment e' with the function identifiers mapped to corresponding *Function* values where each function is of some arity which equals the number of its parameters;
- in the updated environment e' all the rules must hold;
- also in e' all the axioms evaluate to *true*.

For the above used auxiliary functions, predicates and the definition of the other alternatives of the declaration domain *decl*, please see Appendix E.

6.5.2. Procedure Specifications

The valuation function for a procedure specification *proc-spec* has signature:

$$[\![\textit{proc-spec}]\!]\colon \mathbb{P}(\textit{Environment})$$

The procedure specification holds in the given environment.

The semantics of a procedure specification is defined below:

$[\![$**requires** *spec-expr*$_1$;
global *Iseq*;
ensures *spec-expr*$_2$;
excep-clause;
proc$(Pseq) :: T;\ S;\ R$ **end**$]\!](e) \Leftrightarrow$
LET $(iseq, Tseq) = getIdentifiersAndTypes(Pseq)$
IN
$\forall\ valseq \in [\![Tseq]\!], e_1 \in Environment, s_1, s_2 \in State, v, r \in Value, b, b_1 \in Boolean :$
$e_1 = push(e, iseq, valseq) \wedge$
$[\![spec\text{-}expr_1]\!](e_1)(s_1, inStateU(s_1), r, inValueU(inValue(b))) \wedge b = inTrue() \wedge$
$\exists\, p \in Procedure, tag \in Type\text{-}Tag, tagseq \in Type\text{-}Tag^* :$
$[\![\text{proc}(Pseq) :: T;\ S;\ R;\ \text{end}\ ]\!](e_1)(s_1, inStateU(s_1), inValueU(inValue(p))) \wedge$
$p(valseq, s_1, inStateU(s_2), inValueU(v), tag, tagseq) \wedge isType(v, tag)$
$\Rightarrow equalsExcept(s_1, s_2, Iseq) \wedge$
IF $exceptions(data(s_2))$ THEN
$[\![excep\text{-}clause]\!](e_1)(s_2, inStateU(s_2), v, inValueU(inValue(b_1))) \wedge b_1 = inTrue()$
ELSE
$[\![spec\text{-}expr_2]\!](e_1)(s_2, inStateU(s_2), v, inValueU(inValue(b_1))) \wedge b_1 = inTrue()$
END //if-$exceptions(data(inState(s_2)))$
END //let-$(iseq, Tseq)$

In detail, if for any pre-state s_1 and post-state s_2

- we update the given environment e by mapping all the identifiers (from the given parameter sequence *Pseq*) to their possible values (w.r.t. their types) and
- the precondition expression (*spec-expr*$_1$) holds in the pre-state s_1 and
- the evaluation of a procedure expression (**proc**(Pseq)::T; S;R; **end proc;**) in a pre-state s_1 evaluates to a procedure relation p and
- the procedure relation p holds for all the possible values of parameter identifiers

then

- the two states s_1 and s_2 are equal except for the values of identifiers *Iseq* and
- if the post-state s_2 is an exception-state then the exceptional behavior of the procedure *excep-clause* holds in the post-state s_2, otherwise normal behavior *spec-expr*$_2$ holds in the post-state s_2.

For the definition of various auxiliary functions and predicates (which are not defined in this chapter), please see Appendix E.

6.5.3. Loop Specifications

The valuation function for a loop specification *loop-spec* has signature:

$$[\![\textit{loop-spec}]\!]\colon \textit{Environment} \to \mathbb{P}(\textit{State} \times \textit{StateU})$$

The loop specification must hold in the given environment and in the pre- and post-states.

The semantics of a loop specification is defined as a relationship between the pre-state (s) and post-state (s') of the loop. *MiniMaple* supports different variations of a loop; for simplicity, we only discuss here the semantics of a while-loop specification.

$[\![$**invariant** SE_1; **decreases** SE_2;
while E **do** *Cseq* **end do;**$]\!](e)(s, s') \Leftrightarrow$
$(\forall b \in \textit{Boolean}, r \in \textit{Value} :$
$[\![SE_1]\!](e)(s, \textit{inStateU}(s), r, \textit{inValueU}(b) \Rightarrow b = \textit{inTrue}())$
$\wedge$
$(\forall i \in \textit{Integer}, r \in \textit{Value} : [\![SE_2]\!](e)(s, \textit{inStateU}(s), r, \textit{inValueU}(i)) \Rightarrow i > 0)$
$\wedge$
$(\forall s_1, s_2 \in \textit{State}, r \in \textit{Value} :$
$(\forall b_1 \in \textit{Boolean} :$
$[\![SE_1]\!](e)(s_1, \textit{inStateU}(s_1), r, \textit{inValueU}(b_1)) \Rightarrow b_1 = \textit{inTrue}()) \wedge$
$(\forall j \in \textit{Integer} :$
$[\![SE_2]\!](e)(s_1, \textit{inStateU}(s_1), r, \textit{inValueU}(j)) \Rightarrow j > 0) \wedge$
$(\forall b_2 \in \textit{Boolean} :$
$[\![E]\!](e)(s_1, \textit{inStateU}(s_1), r, \textit{inValueU}(b_2))$
$\Rightarrow b_2 = \textit{inTrue}()) \wedge [\![\textit{Cseq}]\!](e)(s_1, \textit{inStateU}(s_2))$

$$\Rightarrow (\ \forall\, b_3 \in \mathit{Boolean} : [\![SE_1]\!](e)(s, \mathit{inStateU}(s_2), r, \mathit{inValueU}(b_3)) \Rightarrow b_3 = \mathit{inTrue}()\ ) \wedge$$
$$(\ \forall\, k \in \mathit{Integer} : [\![SE_2]\!](e)(s_2, \mathit{inStateU}(s_2), r, \mathit{inValueU}(k)) \Rightarrow k >= 0 \wedge k < j\ )$$
$$)$$

The semantics of an annotated while-loop says that:

- in a pre-state (s) an invariant (boolean specification expression) *spec-expr*$_1$ evaluates to *true* and
- the termination term (a numeral specification expression) *spec-expr*$_2$ evaluates to a non-negative integer value and
- also for any arbitrary pre-state s_1 and post-state s_2, if we make an iteration step for the body of the loop (*Cseq*) where in the pre-state s_1
  - the loop expression E holds and
  - the invariant *spec-expr*$_1$ evaluates to *true* and
  - the termination term *spec-expr*$_2$ evaluates to an integer value that is greater than or equal to zero

  then (after iteration step) in the post-state s_2
  - the invariant *spec-expr*$_1$ evaluates to *true* and
  - the termination term *spec-expr*$_2$ evaluates to an non-negative integer value and
  - the value of the termination term in the post-state s_2 must be less than its value in the pre-state s_1

Based on the same idea, the corresponding semantics of the for-loop specification can easily be derived.

6.5.4. Assertions

The valuation function for an assertion *asrt* has signature:

$$[\![\mathit{asrt}]\!]\colon \mathit{Environment} \rightarrow \mathbb{P}(\mathit{State})$$

The assertion holds in the given environment and a state.

The semantics of an assertion is similar to the semantics of a boolean specification expression as defined below:

$$[\![\mathbf{ASSERT}(\textit{spec-expr})]\!](e)(s) \Leftrightarrow$$
$$\forall\ r \in \mathit{Value}, b \in \mathit{Boolean} : [\![\textit{spec-expr}]\!](e)(s, \mathit{inStateU}(s), r, \mathit{inValueU}(b))$$
$$\Rightarrow b = \mathit{inTrue}()$$

The result of the evaluation of the boolean command-specification (assertion) expression *spec-expr* evaluates to *true* in the given e and state s.

7. Formal Verification

In this chapter, we discuss the formal verification of *MiniMaple* programs. For verification, we first translate an annotated *MiniMaple* program into the language Why3ML of the intermediate verification tool Why3 [21] developed at LRI, France; then we generate verification conditions by the corresponding component of Why3; finally, we prove the correctness of these conditions by various automatic and interactive theorem provers supported by Why3 as back-ends. The rest of the chapter is organized as follows: Section 7.1 introduces the intermediate verification tool Why3. In Section 7.2 we give an overview of the translation of *MiniMaple* and its specification language to Why3ML. In Section 7.3 we discuss the *MiniMaple* to Why3ML translation and the verification of our example program. Section 7.4 sketches the structure and strategy of the proof of the soundness of the translation in general and the proof of the soundness of the translation of command sequences and while-loops in particular.

7.1. Why3

For the verification of an annotated *MiniMaple* program, we can generate verification conditions either on our own (as in the RISC ProgramExplorer [138]) or by some existing verification framework, e.g. Why3 [21] developed at LRI, France or Boogie [13] developed by Microsoft. Based on preliminary investigations, we decided to use Why3, which we discuss in this section.

Why3 is a verification tool for the programming language Why3ML whose core is a verification condition generator as depicted in Figure 7.1. The generated verification conditions are translated into a logical specification language called Why for which translation to various back-end theorem provers is provided [65].

In general, Why3 provides an environment for deductive program verification [64]. The system was originally developed as a generic intermediate verification platform supporting various front-end tools, e.g. Krakatoa [33] (for Java programs) and Frama-C [46] (for C programs); currently the focus of Why3 is the verification of Why3ML programs. Why3ML is a first order functional language influenced by ML that supports pattern matching, inductive predicates, algebraic data types and also supports typical imperative constructs (loops, sequences, exceptions, etc.).

Why3 supports various automated provers (e.g. Z3 and CVC3) and proof assistants (e.g. Coq). This wide range of proof support was one of the reasons why we chose

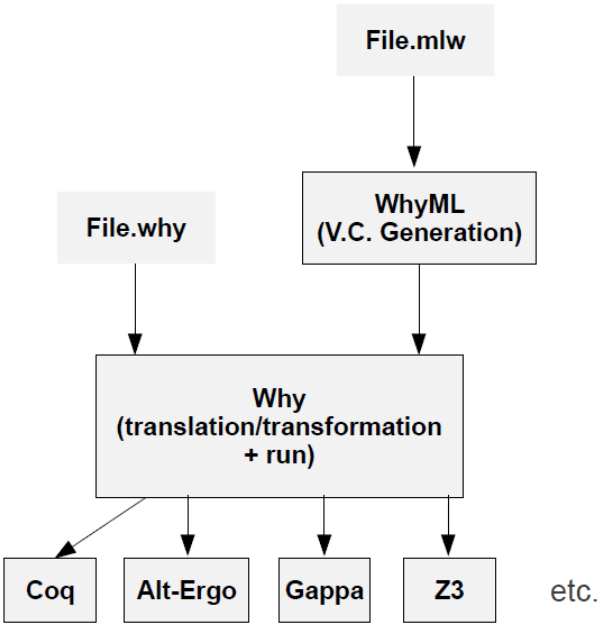

Figure 7.1.: Overview of Why3

Why3, as we are, e.g. dealing with non-linear arithmetic which requires in general an interactive prover. The existence of a formal semantics of Why3ML (first based on weakest precondition [64], later operational [63, 66]) is the other reason for choosing this system, because one can precisely argue whether the generated verification conditions are sound with respect to the *MiniMaple* semantics.

7.2. MiniMaple to Why3 Translation

In this section we discuss the translation of annotated *MiniMaple* to Why3ML. The goal is to automatically translate a *MiniMaple* program into a semantically equivalent Why3ML program. Some of the main features respectively challenges of the translation are as follows:

- *MiniMaple* supports a **return** statement which is not supported in Why3ML. The **return** statement is translated with the help of the Why3ML exception-handling mechanism: where-ever **return** statement occurs, we assign values to the corresponding exception-object and then raise an exception which is caught by a corresponding handler in the program.
- In contrast to Why3ML, *MiniMaple* supports a multi-assignment command. We translate this statement by a local binding in Why3ML.

- Why3ML supports very limited data types, e.g. integers, reals, strings, tuples and lists. We axiomatize all other *MiniMaple* types and their corresponding operations. For example, type **set**(T) is axiomatized with the underlying Why3ML list representations where the elements of the set are some permutation of the list elements.
- The union type **Or**(*Tseq*) is defined as an algebraic data type with one constructor for each type in *Tseq*. The type tests for such types are translated using pattern matching over the corresponding type constructor.

The formal definition of our translation has 40 valuation functions, approx. 50 auxiliary functions and predicates and contains 45 pages [97]. For further details on the translation, please see Appendix F. In the following, we discuss the definition of translation functions for the selected syntactic constructs of *MiniMaple*.

7.2.1. Commands

The translation function T for a command C has the following signature:

$$\text{T } [\![C]\!]\text{: } Env_m \times Env_w \times Decl_w \times Thry_w \rightarrow Exp_w \times Env_w \times Decl_w \times Thry_w$$

The function takes as arguments a *MiniMaple* type environment (Env_m), a Why3 environment (Env_w), Why3 global declarations ($Decl_w$) and a Why3 theory ($Thry_w$); the function returns the corresponding translated Why3ML expression (Exp_w) and generates the respective extended Why3 environment (Env_w), global and theory declarations ($Decl_w$ and $Thry_w$).

Assignment

The translation of a *MiniMaple* assignment command depends on the value of the *Context* flag (*local* or *global*) as determined by the type system.

In a *global* context, the translation function looks as follows:

$$\begin{aligned}
&\text{T}[\![\text{I, Iseq} := \text{E, Eseq}]\!](te, we, mdecl, wt) = \\
&\quad <inWhy3_ExpU(I := w_expr_1;\ w_expr_3), we_2, \\
&\qquad combine(w_mdecl_2, \textbf{val } I\text{: } \textbf{ref } w_type_1, decls), wt_2> \\
&\text{where} \\
&\quad <w_expr_1, we_1, w_mdecl_1, wt_1> = \text{T } [\![\text{E}]\!](te, we, mdecl, wt) \\
&\quad <inWhy3_ExpU(w_expr_2), we_2, w_mdecl_2, wt_2> = \text{T } [\![\text{Eseq}]\!](te, we_1, mdecl_1, wt_1) \\
&\quad w_type_1 = getType(I, we_1) \\
&\quad <inWhy3_ExprU(w_expr_3), decls> = getAssignments(Iseq, we_2, w_expr_2)
\end{aligned}$$

The translation function first translates the right hand side expressions E and $Eseq$ to corresponding Why3 expressions w_expr_1 and w_expr_2, respectively. Then the ordered

sequence of assignments is constructed. For example, if w_expr_1 is the translation of E then an assignment $I := w\_expr_1$ corresponds to the translation of $I := E$. However, this assignment introduces a declaration **val** I: **ref** w_type_1 where w_type_1 is the corresponding type of expression w_expr_1. Similarly, the assignment of *Eseq* to *Iseq* is translated to expression w_expr_3 with corresponding declarations *decls*. Additionally, an extended Why3 environment we_2, declarations w_mdecl_2 and a theory wt_2 are produced.

The translation function for a *local* context is similar:

$$
\begin{aligned}
&\mathrm{T}[\![\text{I, Iseq} := \text{E, Eseq}]\!](te, we, mdecl, wt) = \\
&\quad <inWhy3\_ExpU(I := w\_expr_1;\ w\_expr_3), we_2, w\_mdecl_2, wt_2> \\
&\text{where} \\
&\quad <w\_expr_1, we_1, w\_mdecl_1, wt_1> = \mathrm{T}\ [\![\mathrm{E}]\!](te, we, mdecl, wt) \\
&\quad <inWhy3\_ExpU(w\_expr_2), we_2, w\_mdecl_2, wt_2> = \mathrm{T}\ [\![\mathrm{Eseq}]\!](te, we_1, mdecl_1, wt_1) \\
&\quad <inWhy3\_ExprU(w\_expr_3), decls> = getAssignments(Iseq, we_2, w\_expr_2)
\end{aligned}
$$

However, here the variables in *local* context are already declared so we can just introduce a typical assignment.

The translations of assignments statements in the *global* and *local* contexts from our example program are shown in Section 7.3.

For-loop

The translation of a *MiniMaple* **for-while** loop command is as follows:

$$
\begin{aligned}
&\mathrm{T}[\![\textbf{for } I \textbf{ in } E_1 \textbf{ while } E_2 \textbf{ do } Cseq \textbf{ end do}]\!](e_m, e_w, d_w, t_w) = \\
&\quad (inWhy3\_Exp(\textbf{let } I_0 = \textbf{ref } 0 \textbf{ in} \\
&\qquad\qquad \textbf{while } I_0 < \mathrm{op\_length}(exp1_w)\ \&\ exp2_w \textbf{ do} \\
&\qquad\qquad\quad \textbf{let } I = \mathrm{op\_nth}(I_0, exp1_w) \textbf{ in} \\
&\qquad\qquad\qquad exp3_w;\ I_0 := !I_0 + 1 \\
&\qquad\qquad \textbf{done}), e3_w, d3_w, t3_w) \\
&\text{where} \\
&\quad <exp1_w, e1_w, d1_w, t1_w> = \mathrm{T}\ [\![E_1]\!](e_m, e_w, d_w, t_w), \\
&\quad <exp2_w, e2_w, d2_w, t2_w> = \mathrm{T}\ [\![E_2]\!](e_m, e1_w, d1_w, t1_w), \\
&\quad <exp3_w, e3_w, d3_w, t3_w> = \mathrm{T}\ [\![Cseq]\!](e_m, e2_w, d2_w, t2_w), \\
&\quad exp\_type1 = getExpType(exp1_w, e1_w), \\
&\quad op\_length = access(\text{“length”}, exp\_type1, e1_w), \\
&\quad op\_nth = access(\text{“select”}, exp\_type1, e1_w)
\end{aligned}
$$

The *MiniMaple* **for-while** loop checks at the start of every iteration both loop conditions, i.e. I **in** E_1 and E_2; if any of them is $false$, the body of the loop is not executed. Moreover, the identifier I is used in the body of the loop representing the

ith (iteration) element of expression E_1. For a semantically equivalent translation we proceeded as follows:

1. We declare a (locally bound) auxiliary variable I_0 to track the iteration number and initialize it with value 0.
2. We translate the member-based loop condition, i.e. I **in** E_1 into a corresponding iteration-bounded condition $I_0 < op\_length(exp1_w)$ and combine it with the while-loop condition ($exp2_w$) which we get from the translation of the corresponding *MiniMaple* expression E_2.
3. We declare I as a local variable and at the *ith* iteration (represented by I_0) assign it the *ith* value of the translated expression $exp1_w$.
4. We increment I_0 at the end of the iteration.

The generic operation $access(op, exp\_type, e_w)$ returns the name of the concrete operation op as generated by the translator for the type expression exp_type in the environment e_w. In above example, the *access* function returns the names of the concrete operations "length" and "select" of the expression type exp_type1.

Our example program has a typical for-loop, which is simpler than the *MiniMaple* for-while loop; thus the translation for the typical for-loop is pretty simple as shown in Section 7.3.

Return

The *MiniMaple* **return** command is translated with the help of the Why3 exception-handling mechanism.

$$
\begin{aligned}
&\mathrm{T}[\![\textbf{return } \mathrm{E}]\!](te, we, mdecl, wt) = \\
&\quad <inWhy3\_ExprU(\textbf{raise } I, w\_expr_1), we_1, \\
&\qquad inWhy3\_MDeclU(combine(w\_mdecl_1, \textbf{exception } I)), wt_1> \\
&\text{where} \\
&\quad <w\_expr_1, we_1, w\_mdecl_1, wt_1> = \mathrm{T}[\![\mathrm{E}]\!](te, we, mdecl, wt)
\end{aligned}
$$

The translation generates a Why3 raise statement **raise** I, w_expr_1, where I is an auxiliary identifier denoting an exception and w_expr_1 is the corresponding handling expression. Moreover, this translation extends the module declarations by an exception declaration **exception** I. The return statement is replaced by a corresponding translated raise statement; by executing this raise statement, the corresponding catch-block returns the expression w_expr_1 and thus conforms to the semantics of *MiniMaple* return statement.

The translation of the return statement of our example program is shown in Section 7.3.

7.2.2. Expressions

The translation function T for an expression E has the following signature:

T $[\![E]\!]$: $Env_m \times Env_w \times Decl_w \times Thry_w \rightarrow Exp_w \times Env_w \times Decl_w \times Thry_w$

The translation function for E is similar to the corresponding function for command C as discussed in Subsection 7.2.1 because both syntactic domains command C and expression E are mutually recursive.

Procedure

A *MiniMaple* procedure is translated into Why3 let-in construct. The translation function T for procedure first translates parameter sequence 'Pseq' and return type 'T' into w_expr_1 and w_type respectively and then, translates the procedure declarations 'S' and the body of the procedure 'R' into corresponding Why3 expressions w_expr_2 and w_expr_3 respectively.

T$[\![$**proc**(Pseq)::T S;R **end**$]\!](te, we, mdecl, wt) = (proc_expr, we_4, w_mdecl_4, wt_4)$
where
$<inWhy3\_ExprU(w\_expr_1), we_1, w\_mdecl_1, wt_1>$ = T$[\![$Pseq$]\!](te, we, mdecl, wt)$
$te_1 = typeEnv(te, Pseq)$
$<w\_type, we'_1, w\_mdecl'_1, wt'_1>$ = T$[\![$T$]\!](te_1, we_1, mdecl_1, wt_1)$
$<inWhy3\_ExprU(w\_expr_2), we_2, w\_mdecl_2, wt_2>$ = T$[\![$S$]\!](te_1, we'_1, mdecl'_1, wt'_1)$
$te_2 = typeEnv(te_1, S)$
$<inWhy3\_ExprU(w\_expr_3), we_3, w\_mdecl_3, wt_3>$ = T$[\![$R$]\!](te_2, we_2, mdecl_2, wt_2)$
$proc\_expr = defineProcedure(inWhy3\_ExprU(w\_expr_1), inWhy3\_ExprU(w\_expr_2),$
$inWhy3\_ExprU(w\_expr_3), w\_type)$

Finally, a resulting Why3 procedure expression $proc\_expr$ is constructed with the help of an auxiliary function $defineProcedure$.

Our example program contains a procedure definition; the corresponding application of the procedure translation function is shown in Section 7.3.

Type Test

The translation function T translates a type test with the help of Why3 pattern-matching construct.

T$[\![$**type**(I,T)$]\!](te, we, mdecl, wt) =$
<**match** I **with**
constructors

end, $we_2, w\_mdecl_2, wt_2>$
where
$<we_1, w\_mdecl_1, wt> = \mathrm{T}\ [\![\mathrm{I}]\!](te, we, mdecl, wt)$
$<w\_type, we_2, w\_mdecl_2, wt_2 = \mathrm{T}[\![\mathrm{T}]\!](te, we_1, mdecl_1, wt_1)$
$constructors = getTestConstructors(I, w\_type, we_2, wt_2)$

The function, first translates the testing type T into a Why3 type *w_type* then, the *constructors* of the corresponding union type of the identifier I are extracted from the given Why3 type environment we_2 and the theory declarations wt_2. Finally, a match construct is defined with the given constructors.

Please remember here that a *MiniMaple* union type is translated with the help of an algebraic type with respective constructors. The example translation of our program in Section 7.3 shows the translation function of the union type $Or(integer, float)$ and its corresponding type tests.

7.2.3. Specification Expressions

The translation function T for a specification expression *SE* has signature:

$\mathrm{T}\ [\![SE]\!]$: $Env_m \times Env_w \times Decl_w \times Thry_w \rightarrow Exp_w \times Env_w \times Decl_w \times Thry_w$

The function takes as arguments a *MiniMaple* type environment (Env_m), a Why3 environment (Env_w), Why3 global declarations ($Decl_w$) and a Why3 theory ($Thry_w$); the function returns the corresponding translated Why3ML expression (Exp_w) and generates the respective extended Why3 environment (Env_w), global and theory declarations ($Decl_w$ and $Thry_w$).

Numerical Quantifiers/Binders

The translation function T for a numerical quantifier specification expression into a corresponding Why3 specification/theory function.

$\mathrm{T}[\![IOp(SE_1, B, SE_2)]\!](te, we, mdecl, wt) =$
$<$**function** $func\_name\ (w\_sexpr_1) : w\_type = func\_def,$
$we_4, w\_mdecl_3, wt_4)>$
where
$<(w\_sexpr_1, we_1, w\_mdecl_1, wt_1)> = \mathrm{T}\ [\![SE_1]\!](te, we, mdecl, wt)$
$te_1 = typeEnv(te, SE_1)$
$w\_type = getQuantifierType(SE_1, te, w\_sexpr_1, we_1, wt_1)$
$<(w\_sexpr_2, we_2, w\_mdecl_2, wt_2)> = \mathrm{T}\ [\![B]\!](te_1, we_1, w\_mdecl_1, wt_1)$
$te_2 = typeEnv(te, B)$
$<(w\_sexpr_3, we_3, w\_mdecl_3, wt_3)> = \mathrm{T}\ [\![SE_2]\!](te_2, we_2, w\_mdecl_2, wt_2)$

$$
\begin{aligned}
&< func\_name = getQuantifierName(IOp) \\
&< func\_def = getQuantifierDefinition(w\_sexpr_1, w\_sexpr_2, w\_sexpr_3) \\
&wt_4 = combine(wt_3, \textbf{function}\ func\_name\ (w\_sexpr_1) : w\_type = func\_def) \\
&we_4 = wtypeEnv(we_3, func\_name, w\_sexpr_1, w\_type)
\end{aligned}
$$

First, this function translates the corresponding elements SE_1, B and SE_2 of the numerical quantifier IOp into corresponding Why3 specification expressions w_sexpr_1, w_sexpr_2 and w_sexpr_3 respectively. Then the resulting Why3 specification function is constructed with the help of auxiliary functions. Furthermore, the translation function returns an updated Why3 environment we_4 and the theory declarations wt_4.

The translation of the numerical quantifier *add* used in our example program is shown in Section 7.3.

7.3. Example

In this section, we discuss the implementation of the translator, the translation of our example program and finally the verification of the translated program. The corresponding translator is implemented in Java and contains approximately 80+ classes and 5K+ lines of code.

To translate the *MiniMaple* program shown in Section 5.3, we execute the command;

```
java fmrisc/typechecker/MiniMapleTypeChecker -translate Test6.m
```

where the file `Test6.m` contains the example program.

7.3.1. Translation

In the following, we show the example translation (manually modified for readability) of our example *MiniMaple* (discussed in Section 7.2) into a Why3ML program. The translated program consists of a theory (specification) and a module (program). For further illustration, the various code parts of the translation are annotated with Why3ML comments (* ... *).

```
theory SumList

  use export int.Int
  use export real.RealInfix
  use export list.List
  use export list.Length
  use export list.Nth

  type or_integer_float = Integer int | Real real
```

```
  (* sum integers among the first j elements of e *)
  function add_int (e: list or_integer_float) (j: int) : int =
    if j <= 0 then 0 else
    match e with
    | Nil -> 0
    | Cons (Integer n) t -> n + add_int t (j-1)
    | Cons _ t -> add_int t (j-1)
    end

  (* sum reals among the first j elements of e *)
  function add_real (e: list or_integer_float) (j: int) : real =
    if j <= 0 then 0.0 else
    match e with
    | Nil -> 0.0
    | Cons (Real x) t -> x +. add_real t (j-1)
    | Cons _ t -> add_real t (j-1)
    end

end

module SumListImpl

  use import SumList
  use import module ref.Ref

  val status: ref int

  exception Break

  val get (n: int) (l: list 'a) :
    { 0 <= n < length l } 'a { nth n l = Some result }

  let sum (l: list or_integer_float) : (int, real) =
    { true }
    status := 0;
    let si = ref 0 in
    let sf = ref 0.0 in
    try
      for i = 0 to length l - 1 do
        invariant { ( i = 0 /\ !status = 0 /\ !si = 0 /\ !sf = 0.0   )
                    \/
                  ( i > 0 /\ !status = i-1 /\
                         forall j: int. 0 <= j <= !status ->
                         match nth j l with
                         | None -> false
                         | Some y -> match y with
                                 | Integer n -> n <> 0
                                 | Real r -> r >=. 0.5
                             end
                         end /\
                    !si = add_int l (!status + 1) /\
```

```
                    !sf = add_real l (!status + 1))
        }
        status := i;
        match get i l with
        | Integer n -> if n = 0 then raise Break; si := !si + n
        | Real r -> if r <. 0.5 then raise Break; sf := !sf +. r
        end
      done;
      status := -1;
      (!si, !sf)
    with Break ->
      (!si, !sf)
    end
    { let (si, sf) = result in
        ( !status = -1 /\
            forall j: int. 0 <= j < (length l) ->
            match nth j l with
            | None -> false
            | Some y -> match y with
                    | Integer n -> n <> 0
                    | Real r -> r >=. 0.5
                    end
            end /\
            si = add_int l (length l) /\ sf = add_real l (length l) )
        \/
        ( 0 <= !status < length l /\
                    match nth !status l with
                    | None -> false
                    | Some y -> match y with
                            | Integer n -> n = 0
                            | Real r -> r <. 0.5
                        end
                    end /\
            forall j: int. 0 <= j < !status ->
            match nth j l with
            | None -> false
            | Some y -> match y with
                    | Integer n -> n <> 0
                    | Real r -> r >=. 0.5
                    end
            end /\
          si = add_int l !status /\ sf = add_real l !status )
    }

end
```

In detail, the corresponding Why3 theory defines the types and functions arisen from the translation of the Why3 module and *MiniMaple* program; e.g. the *MiniMaple* union type **Or(integer, float)** is translated to an algebraic data type with two corresponding constructors for integers and floats respectively. The module con-

tains the declarations arising from the translation of the *MiniMaple* procedure, a global variable *status*, the auxiliary exception *Break*, and the translation of the procedure *sum* itself. This procedure also contains a translation of *MiniMaple* **for** loop to a corresponding Why3ML loop. The type tests of *MiniMaple* are translated using the pattern matching feature of Why3ML: the **match** construct matches the *ith* element of the list with the corresponding constructor of the type of the list elements. The *MiniMaple* **return** statement is translated into an equivalent exception-handling mechanism by an auxiliary exception object *Break*, i.e. we throw the exception *Break* whereever the **return** statement occurred and then catch this exception in the corresponding handler as shown in the **with** construct. Finally, in the handler we return the value of the corresponding resulting tuple.

The application of our translator to the test package *DifferenceDifferential* package is discussed in Chapter 8.

7.3.2. Verification

In this section we discuss the verification of the example program which was generated by the translator in the previous subsection. For this purpose, we use the GUI-based interface of Why3 to generate verification conditions and to prove them as shown in Figure 7.2.

The Why3 GUI displays three columns:

1. The left column lists the configured theorem provers.
2. The middle column shows the verification conditions generated (respectively required to be proved correct).
3. The right column shows the contents of the goals (verification conditions). Actually, the right column has two parts, the upper part shows the corresponding Why contents of the selected goals, while the lower part highlights the corresponding Why3ML code from which the selected goal is generated.

In our example, the proof of the correctness of the procedure results in the following four goals as shown in the middle column of Figure 7.2:

1. a normal postcondition,
2. the for-loop (invariant) initialization,
3. the for-loop (invariant) preservation and
4. a normal postcondition.

The first and the last goal are about proving the correctness of postconditions. The first goal is to prove the postcondition when the loop body is not executed, while the last goal is to prove the postcondition when the loop body is executed, which requires invariant-based reasoning.

The second goal is to prove that the loop invariant holds at the start of the execution of the loop. The third goal is to prove that the loop invariant is preserved by the

Figure 7.2.: Verification of Example Program

execution of the loop and must also hold when the loop terminates.

To run through the prove of these generated verification conditions, it was required to add some lemmas manually (at Why3 level) because the function definitions generated by the translator from the corresponding numeric quantifiers (**add**) appearing in the *MiniMaple* procedure specification were not adequate for this proof:

```
(* the following two lemmas are about "add" function over integers *)
lemma add_int_right:
  forall e: list or_integer_float, j: int.
  0 <= j < length e -> forall n: int.
  nth j e = Some (Integer n) -> add_int e (j+1) = add_int e j + n

lemma add_int_right2:
  forall e: list or_integer_float, j: int.
  0 <= j < length e -> forall x: real.
  nth j e = Some (Real x) -> add_int e (j+1) = add_int e j

(* the following two lemmas are about "add" function over floats *)
lemma add_real_right:
  forall e: list or_integer_float, j: int.
  0 <= j < length e -> forall x: real.
  nth j e = Some (Real x) -> add_real e (j+1) = add_real e j +. x

lemma add_real_right2:
  forall e: list or_integer_float, j: int.
  0 <= j < length e -> forall n: int.
  nth j e = Some (Integer n) -> add_real e (j+1) = add_real e j
```

Thus, the above lemmas introduce the facts that the translated addition functions (*add_int* and *add_real* over lists) correctly handle the integers and reals in the list. By the introduction of these lemmas, all of the verification conditions could be proved with the automatic decision procedures Alt-Ergo and Z3. On the other hand, we had to prove these lemmas manually by induction using the interactive theorem prover Coq. In the future, we will generate these lemmas automatically as axioms: then the proof of the aforementioned generated verification conditions is automatic.

The verification of the test package *DifferenceDifferential* is discussed in Chapter 8.

7.4. Soundness of Translation

In order to show that the verification of the translated Why3ML program implies the correctness of the original *MiniMaple* program , we have to prove that the translation preserves the semantics of the program. In detail, we have to prove the equivalence of the denotational semantics of *MiniMaple* programs [95] and the operational semantics of Why3ML programs [63].

As discussed in Chapter 6, the denotational semantics of a *MiniMaple* command C is defined as a relationship between a pre and a post-state:

$$[\![C]\!](e)(s, s')$$

such that semantically, in a given type environment e, execution of a command C in a pre-state s yields to a post-state s'.

On the other hand, in [63] a big-step operational semantics of Why3 expression e is defined by a transition:

$$<t, e> \longrightarrow <t', v>$$

which states that in a pre-state t, the execution of a Why3 expression e yields a post-state t' and a value v.

Based on these semantices, we have formulated and proved the soundness statements for the translation of selected constructs of *MiniMaple* to Why3ML, i.e. command sequence, conditional command, assignment statement and a while-loop command. In the following, we sketch the structure and proof strategy for the soundness statement of a command sequence. For the complete proof of the corresponding soundness statements, please see Appendix F and [98].

7.4.1. Soundness of Command Sequence

We illustrate the soundness statement for the translation of a command sequence with the help of the diagram shown in Figure 7.3. Its formal definition is as follows:

$$
\begin{array}{l}
\forall\ Cseq \in Command\_Sequence : \\
\forall\ em \in Environment, cw \in Exprression_w, ew, ew' \in Environment_w, \\
\quad dw, dw' \in Decl_w, tw, tw' \in Theory_w : \\
\quad wellTyped(em, Cseq) \wedge consistent(em, ew, dw, tw) \wedge \\
\quad < cw, ew', dw', tw' > = \mathrm{T}[\![Cseq]\!](em, ew, dw, tw) \\
\quad \Rightarrow \\
\qquad wellTyped(cw, ew', dw', tw') \wedge extendsEnv(ew', cw, ew) \wedge \\
\qquad extendsDecl(dw', cw, dw) \wedge extendsTheory(tw', cw, tw) \wedge \\
\qquad \forall t, t' \in State_w, vw \in Value_w :< t', cw > \longrightarrow < t', vw > \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists s, s' \in State_m : equals(s, t) \wedge [\![Cseq]\!](em)(s, s') \wedge \\
\qquad\qquad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\
\qquad\qquad [\![Cseq]\!](em)(s, s') \wedge dm = infoData(s') \\
\qquad\qquad\quad \Rightarrow equals(s', t') \wedge equals(dm, vw)
\end{array}
$$

This statement says that

- if a command sequence $Cseq$ translates to Why3 expression cw such that various predicates hold for $Cseq$ (e.g. well-typing),
- then various predicates also hold for the translated expression cw (e.g. extension of the declarations *extendsDecl* and theory *extendsTheory*) and

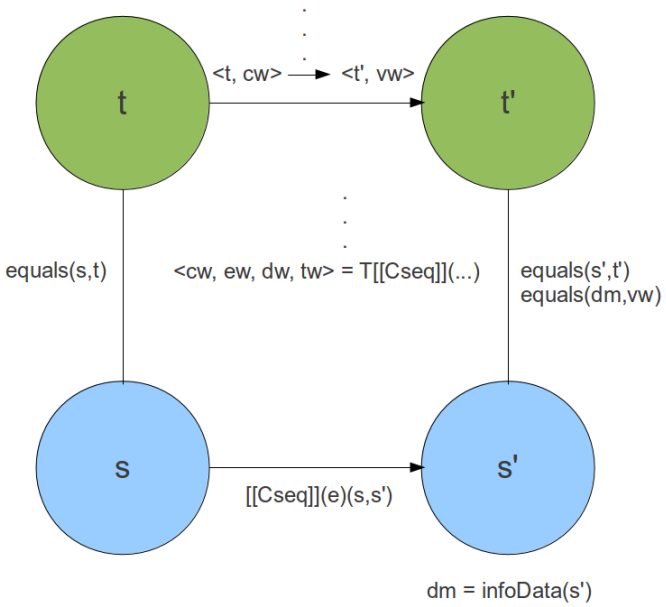

Figure 7.3.: Illustration of Soundness Statement for Command Sequence

- if for arbitrary Why3 states t and t', the execution of the translated expression cw in state t yields post-state t' and value vw,
- then there are corresponding *MiniMaple* states s and s' such that states s and t are equal and the execution of a command sequence $Cseq$ in state s yields state s' and
- if, an arbitrary *MiniMaple* state s is equal to state t; and with a given environment em the execution of $Cseq$ in pre-state s yields post-state s' and also dm is the information of state s',
- then the post-states s' and t' are equal and also the values dm and vw are equal

The formulation of other soundness statements is discussed in Appendix G and [98].

Proof of Soundness (Command Sequence)

In this section, we will discuss the proof of the soundness of the command sequence. The proof of the other selected constructs is discussed in [98]. For further details on definitions, lemmas and auxiliary functions and predicates used in the following proofs, please see the corresponding subsections of the Appendix G.

We prove the goal by structural induction on $Cseq$ which is defined by grammar rule

$Cseq ::= C \mid C;\ Cseq$. Here, we only discuss the interesting case when the command sequence has the form $C;\ Cseq$.

We instantiate the soundness statement with $C;\ Cseq$ to get

$$
\begin{array}{l}
\forall\ em \in Environment, cw \in Exprression_w, ew, ew' \in Environment_w, \\
\quad dw, dw' \in Decl_w, tw, tw' \in Theory_w : \\
\quad wellTyped(em, C;\ Cseq) \wedge consistent(em, ew, dw, tw) \wedge \\
\quad < cw, ew', dw', tw' > = \mathrm{T}[\![C;\ Cseq]\!](em, ew, dw, tw) \\
\quad \Rightarrow \\
\qquad wellTyped(cw, ew', dw', tw') \wedge extendsEnv(ew', cw, ew) \wedge \\
\qquad extendsDecl(dw', cw, dw) \wedge extendsTheory(tw', cw, tw) \wedge \\
\qquad \forall t, t' \in State_w, vw \in Value_w :< t', cw > \longrightarrow < t', vw > \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists s, s' \in State_m : equals(s, t) \wedge [\![C;\ Cseq]\!](e)(s, s') \wedge \\
\qquad\qquad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\
\qquad\qquad [\![C;\ Cseq]\!](e)(s, s') \wedge dm = infoData(s') \\
\qquad\qquad\quad \Rightarrow equals(s', t') \wedge equals(dm, vw)
\end{array}
$$

Let $em, cw, em, ew', dw, dw', tw, tw'$, be arbitrary but fixed.

We assume:

$$wellTyped(em, C;\ Cseq) \tag{7.1}$$

$$consistent(em, ew, dw, tw) \tag{7.2}$$

$$< cw, ew', dw', tw' > = T[\![C;\ Cseq]\!](em, ew, dw, tw) \tag{7.3}$$

We show:

- $wellTyped(cw, ew', dw', tw')$ (a)
- $extendsEnv(ew', cw, ew)$ (b)
- $extendsDecl(dw', cw, dw)$ (c)
- $extendsTheory(tw', cw, tw)$ (d)
- $\forall t, t' \in State_w, vw \in Value_w :< t', cw > \longrightarrow < t', vw >$
  $\Rightarrow$
  $\exists s, s' \in State_m : equals(s, t) \wedge [\![C;\ Cseq]\!](e)(s, s') \wedge$
  $\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$
  $[\![C;\ Cseq]\!](e)(s, s') \wedge dm = infoData(s')$
  $\Rightarrow equals(s', t') \wedge equals(dm, vw)$ (e)

We prove these goals with the help of lemmas which in principle guarantee the absence of internal inconsistencies of typing, environments, theory and global declarations that are used respectively extended by the translation of *MiniMaple* to Why3ML.

The proofs of these lemmas are not very complex and can be proved by the principle of structural induction. In the following, we formulate and describe (informally) these lemmas:

Lemma *L-cseq*1

$$\begin{aligned}
&\forall\ cseq \in Command_Sequence, em \in Environment, e \in Expressionw,\\
&\quad ew, ew' \in Environmentw, dw, dw' \in Declw, tw, tw' \in Theoryw:\\
&\quad wellTyped(em, cseq) \wedge <e, ew', dw', tw'>= \mathrm{T}[\![cseq]\!](em, ew, dw, tw)\\
&\quad \Rightarrow wellTyped(e, ew', dw', tw')
\end{aligned}$$

This lemma says that if a *MiniMaple* command sequence ($cseq$) is well typed then the translated Why3 expression (e) is also well typed in the corresponding environment and declarations.

Lemma *L-cseq*2

$$\begin{aligned}
&\forall\ em \in Environment, C \in Command, Cseq \in Command_Sequence,\\
&\quad ew, ew', ew'' \in Environmentw, e1, e2 \in Expressionw, dw, dw', dw'' \in Declw,\\
&\quad tw, tw', tw'' \in Theoryw :\\
&\quad wellTyped(em, C;\ Cseq) \wedge (e1;\ e2, ew', dw', tw') = T[\![C;\ Cseq]\!](em, ew, dw, tw)\\
&\quad \Rightarrow\\
&\quad [\ extendsEnv(ew'', e1, ew) \wedge extendsEnv(ew', e2, ew'')\\
&\qquad\qquad \Rightarrow extendsEnv(ew', e1;\ e2, ew)\ ] \wedge\\
&\quad [\ extendsDecl(dw'', e1, dw) \wedge extendsDecl(dw', e2, dw'')\\
&\qquad\qquad \Rightarrow extendsDecl(dw', e1;\ e2, dw)\ ] \wedge\\
&\quad [\ extendsTheory(tw'', e1, tw) \wedge extendsTheory(tw', e2, tw'')\\
&\qquad\qquad \Rightarrow extendsTheory(tw', e1;\ e2, tw)\ ]
\end{aligned}$$

This lemma says the fact that if Why3 expressions ($e1$ and $e2$ are the translations of *MiniMaple* command sequences (C and $Cseq$ respectively), then the finally generated Why3 environment, global and theory declarations (ew', dw' and tw') extends the corresponding intermediate Why3 environment, global and theory declarations (ew'', dw'' and tw'').

Lemma *L-cseq*3

$$\begin{aligned}
&\forall\ em, em' \in Environment, C \in Command, Cseq \in Command_Sequence :\\
&\quad wellTyped(em, C;\ Cseq)\\
&\quad \Rightarrow wellTyped(em, C) \wedge em' = Env(em, C) \wedge wellTyped(em', Cseq)
\end{aligned}$$

This lemma is about the well typing of a command sequence $Cseq$ in an intermediate type environment em'.

Lemma *L-cseq*4

$\forall\ em, em' \in Environment, C \in Command, Cseq \in Command\_Sequence,$
$ew, ew', ew'' \in Environmentw, e1, e2 \in Expressionw, dw, dw', dw'' \in Declw,$
$tw, tw', tw'' \in Theoryw :$
$< e1, ew'', dw'', tw'' >= \mathrm{T}[\![C]\!](em, ew, dw, tw) \wedge em' = Env(em, C) \wedge$
$< e2, ew', dw', tw' >= \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \wedge consistent(em, ew, dw, tw)$
$\Rightarrow consistent(em', ew'', dw'', tw'')$

This lemma is about the consistency of an intermediate type environment em' w.r.t. intermediate Why3 environment, global and theory declarations (ew'', dw'' and tw'').

Lemma *L-cseq*5

$\forall\ s \in State, t \in Statew : s = constructs(t) \Rightarrow equals(s, t)$

This lemma says that if we construct a *MiniMaple* state (s) from a Why3 state (t), then the two states are equal.

Lemma *L-cseq*6

$\forall\ v \in Valuew, v' \in InfoData : v' = constructs(v) \Rightarrow equals(v', v)$

This lemma is about the equivalence of state values, i.e. if we construct a *MiniMaple* value (v') from a Why3 value (v), then the two values are equal.

In the following, we prove each of the five goals ($a - e$) above.

Goal (a)

This goal is about the well-typing of the translated Why3 expression cw. To prove, this goal, we instantiate lemma (*L-cseq*1) with *cseq* as C; *Cseq*, *em* as em, e as cw, ew as ew, ew' as ew', dw as dw, dw' as dw', tw as tw, tw' as tw' and get

$wellTyped(em, C;\ Cseq) \wedge < cw, ew', dw', tw' >= \mathrm{T}[\![C;\ Cseq]\!](em, ew, dw, tw)$
$\Rightarrow wellTyped(cw, ew', dw', tw')$

This goal follows from the above formula with assumptions (7.1) and (7.3).

Goals (b,c,d)

The goals (b), (c) and (d) are similar. So for simplicity, we only show here the proof of the sub-goal (b). The proof of this sub-goal (b) requires the expansions of some definitions and some more sub-goals to be proved; because semantically, the execution

of the command sequence C; $Cseq$ produces an intermediate environment e'' by the execution of C and then $Cseq$ is executed in environment e''.

To prove this goal, we proceed as follows: By the definition of the translation function $(D2)$ of $\mathrm{T}[\![C;\ Cseq]\!]$, there are $e1, e2, ew'', dw'', tw''$ for which

$$< cw, ew', dw', tw' >= T[\![C;\ Cseq]\!](em, ew, dw, tw) \tag{7.4}$$

where

$$cw = e1;\ e2 \tag{7.5}$$

$$< e1, ew'', dw''', tw'' >= T[\![C]\!](em, ew, dw, tw) \tag{7.6}$$

$$em' = Env(em, C) \tag{7.7}$$

$$< e2, ew', dw', tw' >= T[\![Cseq]\!](em', ew'', dw'', tw'') \tag{7.8}$$

Here $e1;\ e2$ is a syntactic sugar for the Why3 semantic construct **let** $\_ = e1$ **in** $e2$.

We instantiate lemma $(L\text{-}cseq3)$ with em as em, em' as em', C as C and $Cseq$ as $Cseq$ from which the following holds

$$wellTyped(em, C) \tag{7.9}$$

$$em' = Env(em, C) \tag{7.10}$$

$$wellTyped(em', Cseq) \tag{7.11}$$

In order to show that this intermediate environment e'' preserves the properties, we instantiate the soundness statement for C with em as em, cw as $e1$, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw'' to get

$$\begin{array}{l}
wellTyped(em, C) \wedge consistent(em, ew, dw, tw) \wedge \\
< e1, ew'', dw'', tw'' >= \mathrm{T}[\![C]\!](em, ew, dw, tw) \\
\Rightarrow \\
\quad wellTyped(e1, ew'', dw'', tw'') \wedge extendsEnv(ew'', e1, ew) \wedge \\
\quad extendsDecl(dw'', e1, dw) \wedge extendsTheory(tw'', e1, tw) \wedge \\
\quad \forall t, t' \in State_w, vw \in Value_w, :< t', e1 > \longrightarrow < t', vw > \\
\quad\ \Rightarrow \\
\quad\quad \exists s, s' \in State_m : equals(s, t) \wedge [\![C]\!](e)(s, s') \wedge \\
\quad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\
\quad\quad [\![C]\!](e)(s, s') \wedge dm = infoData(s') \\
\quad\quad\ \Rightarrow equals(s', t') \wedge equals(dm, vw)
\end{array} \tag{A}$$

From (A) and assumptions (7.9), (7.2) and (7.6), it follows that

$$extendsEnv(ew'', e1, ew) \tag{7.12}$$

the intermediate environment preserves extension.

Now, we show that environment e' extends an intermediate environment e'', we proceed as follows: We instantiate lemma (*L-cseq*4) with em as em, em' as em', C as C, $Cseq$ as $Cseq$, ew as ew, ew' as ew', $e1$ as $e1$, $e2$ as $e2$, dw as dw, dw' as dw', tw as tw, tw' as tw', ew'' as ew'', dw'' as dw'', tw'' as tw'' to get

$$
\begin{array}{l}
< e1, ew'', dw'', tw'' >= \mathrm{T}[\![C]\!](em, ew, dw, tw) \wedge em' = Env(em, C) \wedge \\
< e2, ew', dw', tw' >= \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \wedge consistent(em, ew, dw, tw) \\
\Rightarrow consistent(em', dw'', dw'', tw'')
\end{array} \tag{B}
$$

From (B) with assumptions (7.6), (7.7), (7.8) and (7.2), it follows that

$$consistent(em', ew'', dw'', tw'') \tag{7.13}$$

We instantiate the induction assumption for $Cseq$ with em as em', cw as $e2$, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw' to get

$$
\begin{array}{l}
wellTyped(em', Cseq) \wedge consistent(em', ew'', dw'', tw'') \wedge \\
< e2, ew', dw', tw' >= \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \\
\Rightarrow \\
\quad wellTyped(e2, ew', dw', tw') \wedge extendsEnv(ew', e2, ew'') \wedge \\
\quad extendsDecl(dw', e2, dw'') \wedge extendsTheory(tw', e2, tw'') \wedge \\
\quad \forall t, t' \in State_w, vw \in Value_w, :< t', e2 > \longrightarrow < t', vw > \\
\quad\quad \Rightarrow \\
\quad\quad\quad \exists s, s' \in State_m : equals(s, t) \wedge [\![Cseq]\!](e)(s, s') \wedge \\
\quad\quad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\
\quad\quad\quad [\![Cseq]\!](e)(s, s') \wedge dm = infoData(s') \\
\quad\quad\quad\quad \Rightarrow equals(s', t') \wedge equals(dm, vw)
\end{array} \tag{C}
$$

From (C) with assumptions (7.11), (7.13) and (7.8), it follows that

$$extendsEnv(ew', e2, ew'') \tag{7.14}$$

From (7.5), we can re-write the goal (b) as

$$extendsEnv(ew', e1;\ e2, ew)$$

In order to prove this goal, we instantiate lemma (*L-cseq*2) with em as em, C as C, $Cseq$ as $Cseq$, ew as ew, ew' as ew', ew'' as ew'', $e1$ as $e1$, $e2$ as $e2$, dw as dw, dw' as dw', dw'' as dw'', tw as tw, tw' as tw', tw'' as tw'' to get

$$wellTyped(em, C;\ Cseq) \wedge < e1;\ e2, ew', dw', tw' >= \mathrm{T}[\![C;\ Cseq]\!](em, ew, dw, tw)$$

$$
\begin{aligned}
&\Rightarrow \\
&\quad [extendsEnv(ew'', e1, ew) \wedge extendsEnv(ew', e2, ew'') \\
&\qquad\qquad \Rightarrow extendsEnv(ew', e1;\ e2, ew)\ ] \wedge \\
&\quad [extendsDecl(dw'', e1, dw) \wedge extendsDecl(dw', e2, dw'') \\
&\qquad\qquad \Rightarrow extendsDecl(dw', e1;\ e2, dw)\ ] \wedge \\
&\quad [extendsTheory(tw'', e1, tw) \wedge extendsTheory(tw', e2, tw'') \Rightarrow \\
&\qquad\qquad extendsTheory(tw', e1;\ e2, tw)\ ]
\end{aligned} \tag{D}
$$

The goal (b) follows from (D) and assumptions (7.1), (7.4), (7.5), (7.12) and (7.14).

Goal (e)

To prove, this goal, we proceed as follow:

Let t, t', cw, vw be arbitrary but fixed.

We assume:

$$< t, cw > \longrightarrow < t', vw > \tag{7.15}$$

From (7.15), (7.5), and the semantics of Why3 (i.e. $e1;\ e2$ is a syntactic sugar for $\mathbf{let}\ \_ = e1\ \mathbf{in}\ e2$), we know

$$cw = \mathbf{let}\_ =\ \mathrm{e1}\ \mathbf{in}\ \mathrm{e2} \tag{7.16}$$

From Why3 semantics (*com-s*) [63] and Appendix G, we get

$$< t, \mathbf{let}\_ =\ e1\ \mathbf{in}\ e2 > \longrightarrow < t', vw > \tag{7.17}$$

$$< t, e1 > \longrightarrow < t'', vw' > \tag{7.18}$$

for some t'', where vw' is not an exception

$$< t'', e2 > \longrightarrow < t', vw > \tag{7.19}$$

for some t''.

We show:

$$\exists\, s, s' \in State : equals(s, t) \wedge [\![C;\ Cseq]\!](em)(s, s') \tag{e.a}$$

$$\forall\, s, s' \in State, dm \in InfoData : equals(s, t) \wedge [\![C;\ Cseq]\!](em)(s, s') \wedge dm = infoData(s') \Rightarrow equals(s', t') \wedge equals(dm, vw) \tag{e.b}$$

In the following, we prove these two sub-goals (e.a) and (e.b) of goal (e).

Sub-Goal (e.a)

To prove this goal, we define

$$s := constructs(t) \tag{7.20}$$

We split the original goal (e.a) and show the following sub-goals:

$$equals(s, t) \tag{e.a.1}$$

$$[\![C;\ Cseq]\!](em)(s, s') \tag{e.a.2}$$

Now, in the following we prove the two further sub-goals (e.a.1) and (e.a.2).

Sub-Goal (e.a.1)

We instantiate lemma (*L-cseq*5) with s as s and t as t to get

$$s = constructs(t) \Rightarrow equals(s, t) \tag{E}$$

The sub-goal (e.a.1) follows from (E) with assumption (7.20).

Sub-Goal (e.a.2)

We instantiate the soundness statement for C with em as em, cw as $e1$, ew as ew, ew' as ew'', dw as dw, dw' as dw'', tw as tw, tw' as tw'' to get

$$\begin{array}{l} wellTyped(em, C) \wedge consistent(em, ew, dw, tw) \wedge \\ < e1, ew'', dw'', tw'' > = \mathrm{T}[\![C]\!](em, ew, dw, tw) \\ \Rightarrow \\ \quad wellTyped(e1, ew'', dw'', tw'') \wedge extendsEnv(ew'', e1, ew) \wedge \\ \quad extendsDecl(dw'', e1, dw) \wedge extendsTheory(tw'', e1, tw) \wedge \\ \quad \forall t, t' \in State_w, vw \in Value_w, :< t', e1 > \longrightarrow < t', vw > \\ \qquad \Rightarrow \\ \qquad\quad \exists s, s' \in State_m : equals(s, t) \wedge [\![C]\!](e)(s, s') \wedge \\ \qquad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\ \qquad\quad [\![C]\!](e)(s, s') \wedge dm = infoData(s') \\ \qquad\qquad \Rightarrow equals(s', t') \wedge equals(dm, vw) \end{array} \tag{F}$$

From (F) with assumptions (7.9), (7.2), (7.6), we get

$$\begin{array}{l}\forall t, t' \in State_w, vw \in Value_w :< t', e1 > \longrightarrow < t', vw > \\ \Rightarrow \\ \quad \exists s, s' \in State_m : equals(s,t) \wedge [\![C]\!](e)(s,s') \wedge \\ \quad \forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge \\ \quad [\![C]\!](e)(s,s') \wedge dm = infoData(s') \\ \quad\quad \Rightarrow equals(s',t') \wedge equals(dm, vw)\end{array} \tag{F.1}$$

We instantiate the above formula (F.1) with t as t and t' as t'', vw as vw' to get

$$\begin{array}{l}\forall t, t'' \in State_w, vw' \in Value_w :< t', e1 > \longrightarrow < t'', vw' > \\ \Rightarrow \\ \quad \exists s, s' \in State_m : equals(s,t) \wedge [\![C]\!](e)(s,s') \wedge \\ \quad \forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge \\ \quad [\![C]\!](e)(s,s') \wedge dm = infoData(s') \\ \quad\quad \Rightarrow equals(s',t'') \wedge equals(dm, vw')\end{array} \tag{F.2}$$

From (F.2) with assumption (7.18), we know

$$\exists\, s, s' \in State : equals(s,t) \wedge [\![C]\!](em)(s,s') \tag{F.3}$$

By instantiating (F.3) with s as s, s' as s'', we know that there is s, s'' s.t.

$$[C](em)(s,s'') \tag{7.21}$$

We instantiate the induction assumption for $Cseq$ with em as em', cw as $e2$, ew as ew'', ew' as ew', dw as dw'', dw' as dw', tw as tw'', tw' as tw' to get

$$\begin{array}{l}wellTyped(em', Cseq) \wedge consistent(em', ew'', dw'', tw'') \wedge \\ < e2, ew', dw', tw' > = \mathrm{T}[\![Cseq]\!](em', ew'', dw'', tw'') \\ \Rightarrow \\ \quad wellTyped(e2, ew', dw', tw') \wedge extendsEnv(ew', e2, ew'') \wedge \\ \quad extendsDecl(dw', e2, dw'') \wedge extendsTheory(tw', e2, tw'') \wedge \\ \quad \forall t, t' \in State_w, vw \in Value_w, :< t', e2 > \longrightarrow < t', vw > \\ \quad\quad \Rightarrow \\ \quad\quad\quad \exists s, s' \in State_m : equals(s,t) \wedge [\![Cseq]\!](em')(s,s') \wedge \\ \quad\quad\quad \forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge \\ \quad\quad\quad [\![Cseq]\!](em')(s,s') \wedge dm = infoData(s') \\ \quad\quad\quad\quad \Rightarrow equals(s',t') \wedge equals(dm, vw)\end{array} \tag{G}$$

From (G) with assumptions (7.11), (7.13) and (7.8), it follows that

$$\begin{array}{l}\forall t, t' \in State_w, vw \in Value_w :< t, e2 > \longrightarrow < t', vw > \\ \Rightarrow\end{array}$$

$$\begin{aligned}
&\exists s, s' \in State_m : equals(s,t) \wedge [\![Cseq]\!](em')(s,s') \wedge \\
&\forall s, s' \in State_m, dm \in InfoData : equals(s,t) \wedge \\
&[\![Cseq]\!](em')(s,s') \wedge dm = infoData(s') \\
&\quad \Rightarrow equals(s',t'') \wedge equals(dm, vw')
\end{aligned} \tag{G.1}$$

We instantiate the formula (G.1) with t as t'', t' as t', vw as vw to get

$$\begin{aligned}
&\forall t'', t' \in State_w, vw \in Value_w : < t'', e2 > \longrightarrow < t', vw > \\
&\Rightarrow \\
&\quad \exists s, s' \in State_m : equals(s,t) \wedge [\![Cseq]\!](em')(s,s') \wedge \\
&\quad \forall s, s' \in State_m, dm \in InfoData : equals(s,t'') \wedge \\
&\quad [\![C]\!](em')(s,s') \wedge dm = infoData(s') \\
&\quad\quad \Rightarrow equals(s',t') \wedge equals(dm, vw')
\end{aligned} \tag{G.2}$$

From (G.2) and assumption (7.19), we get

$$\exists\, s, s' \in State : equals(s,t'') \wedge [\![Cseq]\!](em')(s,s') \tag{G.3}$$

By instantiating (G.3) with s as s'', s' as s', we know that there is s'', s' s.t.

$$[\![Cseq]\!](em')(s'',s') \tag{7.22}$$

This sub-goal (e.a.2), which is a definition of the semantics of the command sequence $C;\ Cseq$ follows from the assumptions (7.21), (7.22) and (7.7).

Hence sub-goals (e.a.1) and (e.a.2) are proved thus the sub-goal (e.a) is proved.

Sub-Goal (e.b)

This goal is the heart of this proof; as we need to show here the semantic equivalence of the corresponding *MiniMaple* and Why3 states and values.

Let s, s', dm be arbitrary but fixed.

We assume:

$$equals(s,t) \tag{7.23}$$

$$[\![C;\ Cseq]\!](em)(s,s') \tag{7.24}$$

$$dm = infoData(s') \tag{7.25}$$

We define:

$$s' := constructs(t') \tag{7.26}$$

$$vw := constructs(dm) \tag{7.27}$$

We split the original goal (e.b) and show the following sub-goals:

$equals(s', t')$ (e.b.1)

$equals(dm, vw)$ (e.b.2)

In the following, we prove the two further sub-goals (e.b.1) and (e.b.2).

Sub-Goal (e.b.1)

We instantiate lemma (*L-cseq*5) with s as s' and t as t' to get

$s' = constructs(t') \Rightarrow equals(s', t')$ (H)

This sub-goal follows from (H) with assumption (7.26).

Sub-Goal (e.b.2)

We instantiate lemma (*L-cseq*6) with v as vw, v' as dm to get

$vw = constructs(dm) \Rightarrow equals(dm, vw)$ (I)

This sub-goal follows from (I) with assumption (7.27).

7.4.2. Soundness of While-loop

The soundness statement for command C is similar to the soundness statement of command sequence $Cseq$ as formulated in Section 7.4.1. Thus the goal for the soundness statement of a while-loop command is:

$$
\begin{array}{l}
\forall\ E \in Expression, Cseq \in Command\_Sequence : \\
\forall\ em \in Environment, e1, e2 \in Exprression_w, ew, ew' \in Environment_w, \\
\quad dw, dw' \in Decl_w, tw, tw' \in Theory_w : \\
\quad wellTyped(em, \textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end})\ mathit{\wedge} consistent(em, ew, dw, tw) \wedge \\
\quad < \textbf{while}\ e1\ \textbf{do}\ e2, ew', dw', tw' >= \mathrm{T}[\![\textbf{while}\ e1\ \textbf{do}\ e2]\!](em, ew, dw, tw) \\
\quad \Rightarrow \\
\qquad wellTyped(\textbf{while}\ e1\ \textbf{do}\ e_2, ew', dw', tw') \wedge extendsEnv(ew', \textbf{while}\ e1\ \textbf{do}\ e2, ew) \wedge \\
\qquad extendsDecl(dw', \textbf{while}\ e1\ \textbf{do}\ e_2, dw) \wedge extendsTheory(tw', \textbf{while}\ e1\ \textbf{do}\ e2, tw) \wedge \\
\qquad \forall t, t' \in State_w, vw \in Value_w, :< t, \textbf{while}\ e1\ \textbf{do}\ e_2 > \longrightarrow < t', vw > \\
\qquad\quad \Rightarrow \\
\qquad\qquad \exists s, s' \in State_m : equals(s, t) \wedge\ [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge \\
\qquad\qquad \forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge \\
\qquad\qquad [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge dm = infoData(s') \\
\qquad\qquad\quad \Rightarrow equals(s', t') \wedge equals(dm, vw)
\end{array}
$$

The proof of the soundness statement of a while-loop command is not straight forward because the semantics of a Why3 while-loop [63] is defined by a corresponding complex exception handling mechanism of Why3:

$$\begin{array}{l}\textbf{while } e1 \textbf{ do } e2 \equiv \\ \quad \textbf{try} \\ \quad\quad \textbf{loop if } e1 \textbf{ then } e2 \textbf{ else raise } \mathit{Exit} \\ \quad \textbf{with } \mathit{Exit}\ \_ \rightarrow \mathit{void} \textbf{ end}\end{array} \tag{SE}$$

The semantics above involves various other Why3 constructs, e.g. loop and conditional which makes the reasoning complex as the reasoning requires lot of applications respectively unfolding of semantics rules of the other Why3 constructs.

In order to make the proof respectively reasoning simpler, we have derived the following two new rules for the semantics of while-loop (from the above defined semantics of Why3 while-loop):

$$\frac{< t, e1 > \longrightarrow < t', \mathit{false} >}{< t, \textbf{while } e1 \textbf{ do } e2 > \longrightarrow < t', \mathit{void} >} \tag{R.1}$$

$$\frac{\begin{array}{r}< t, e1 > \longrightarrow < t'', \mathit{true} > \\ < t'', e2 > \longrightarrow < t''', \mathit{void} > \\ < t''', \textbf{while } e1 \textbf{ do } e2 > \longrightarrow < t', \mathit{void} >\end{array}}{< t, \textbf{while } e1 \textbf{ do } e2 > \longrightarrow < t', \mathit{void} >} \tag{R.2}$$

These rules operate directly on the level of while-loop (without expansion/unfolding). Based on these rules, we prove the soundness of typical while-loops by the principle of rule induction [158]. We will subsequently prove the soundness of rules (R.1) and (R.2) with the help of the following lemma.

Lemma (*L-a*1)

If there exists a derivation
$< t''', \textbf{try loop if } e1 \textbf{ then } e2 \textbf{ else raise } \mathit{Exit} \textbf{ with } \mathit{Exit}\_ \rightarrow \mathit{void} \textbf{ end} > \longrightarrow < t', \mathit{void} >$
then there also exists a corresponding derivation
$< t''', \textbf{loop if } e1 \textbf{ then } e2 \textbf{ else raise } \mathit{Exit} \textbf{ with } \mathit{Exit}\_ \rightarrow \mathit{void} \textbf{ end} > \longrightarrow < t', \mathit{Exit}\ c >$

Proof

We assume:
$< t''', \textbf{try loop if } e1 \textbf{ then } e2 \textbf{ else raise } \mathit{Exit} \textbf{ with } \mathit{Exit}\_ \rightarrow \mathit{void} \textbf{ end} > \longrightarrow < t', \mathit{void} >$ (7.28)

We show:
$< t''', \textbf{loop if } e1 \textbf{ then } e2 \textbf{ else raise } \mathit{Exit} \textbf{ with } \mathit{Exit}\_ \rightarrow \mathit{void} \textbf{ end} > \longrightarrow < t', \mathit{Exit}\ c >$

In fact, we show here that the goal-derivation from assumption (7.28) is possible by only one semantic rule such that it does not change the respective semantics.

We suppose a derivation (7.28), which is a try-catch construct. There are three semantic rules for Why3 try-catch construct; in the following, we apply case analysis on each of these three rules:

Case 1:

The first rule is *try*-1:

$$\frac{\begin{array}{r} < t, e1 > \longrightarrow < t'', E\ c > \\ < t'', e2[x \leftarrow c] > \longrightarrow < t', vw > \end{array}}{< t, \textbf{try } e1 \textbf{ with } E\ x \rightarrow e2 \textbf{ end} > \longrightarrow < t', vw >}$$

We instantiate rule *try*-1 with t as t''', t' as t', t'' as t'', $e1$ as **loop if** $e1$ **then** $e2$ **else raise** *Exit* _, $e2$ as *void*, E as *Exit*, vw as *void* and x as _; from which the following

$$< t', \mathit{void} > \longrightarrow < t', \mathit{void} > \tag{7.29}$$

$$< t''', \textbf{loop if } e1 \textbf{ then } e2 \textbf{ else raise } \mathit{Exit} \textbf{ with } \mathit{Exit}\_ \rightarrow \mathit{void} \textbf{ end} > \longrightarrow < t', \mathit{Exit}\ c > \tag{7.30}$$

holds. The goal follows from assumption (7.30).

Case 2:

The second rule is *try*-2:

$$\frac{< t, e1 > \longrightarrow < t', vw > \quad \text{where } vw \text{ is not an exception}}{< t, \textbf{try } e1 \textbf{ with } E\ x \rightarrow e2 \textbf{ end} > \longrightarrow < t', vw >}$$

This rule cannot be applied to derive the goal from supposition (7.28). We prove this by induction on the number of iterations. Suppose $n \in \mathbb{N}$ is the number of loop iterations:

Induction Basis

We instantiate rule *try*-2 with t as t''', t' as t', $e1$ as **loop if** $e1$ **then** $e2$ **else raise** *Exit* _, $e2$ as *void*, E as *Exit*, vw as *void* and x as _; from which the following

$$< t, e1 > \longrightarrow < t_n, void > \tag{7.31}$$

$$< t''', \textbf{loop if } e1 \textbf{ then } e2 \textbf{ else raise } Exit \textbf{ with } Exit\_ \rightarrow void \textbf{ end} > \longrightarrow < t', void > \tag{7.32}$$

holds. Thus no derivation is found as neither (7.31) nor (7.32) is the goal.

Induction Step

Here, we assume that by the application of rule *try*-2 the required derivation is not possible for iteration n and prove that also the goal cannot be derived for loop iteration $n + 1$.

To prove, we instantiate rule *try*-2 with t as t''', t' as t_n, $e1$ as **loop if** $e1$ **then** $e2$ **else raise** *Exit* _, $e2$ as *void*, E as *Exit*, vw as *void* and x as _; from which the following

$$< t, e1 > \longrightarrow < t', void > \tag{7.33}$$

$$< t''', \textbf{loop if } e1 \textbf{ then } e2 \textbf{ else raise } Exit \textbf{ with } Exit\_ \rightarrow void \textbf{ end} > \longrightarrow < t_n, void > \tag{7.34}$$

holds. Thus again no derivation is found as neither (7.33) nor (7.34) is the goal.

Case 3:

The third rule is *try*-3:

$$\frac{< t, e1 > \longrightarrow < t', E'\ c > \quad E <> E'}{< t, \textbf{try } e1 \textbf{ with } E\ x \rightarrow e2 \textbf{ end} > \longrightarrow < t', E'\ c >}$$

The structure of this rule does not allow its application to derive the required goal. Because the consequence of the transition of this rule ($<t'$, E' $c>$) has an exception value (E' c), while the corresponding consequence of the transition of our assumption (7.28) ($<t'$, *void*$>$) has a non-exception value (*void*). □

In the following, we show that the aforementioned two new semantic rules (R.1) and (R.2) follow from the basic rule calculus, i.e. adding these rules does not change the semantics of Why3ML.

Derivation of Rule (R.1)

In order to derive rule (R.1), first we get the following derivation based on the application of various semantics rules of Why3, e.g. *try*-1, *loop*-*e*. For the definition of these semantics rules, please see Appendix G.

$$\frac{}{< t', \_ > \longrightarrow < t', c >} \quad where\ c = \_ \quad (const)$$

$$\frac{< t, e1 > \longrightarrow < t', false > \quad < t', \textbf{raise}\ Exit\ > \longrightarrow < t', Exit\ c >}{< t, \textbf{if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit > \longrightarrow < t', Exit\ c >} (raise)$$

$$\frac{< t, \textbf{if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit > \longrightarrow < t', Exit\ c >}{< t, \textbf{loop if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit > \longrightarrow < t', Exit\ c >} (cond\text{-}f)$$

$$(loop\text{-}e)$$

$$\frac{}{< t', void > \longrightarrow < t', void >} (const)$$

$$\frac{}{< t', void[\_ \leftarrow c] > \longrightarrow < t', void >} (rewriting)$$

$$(try\text{-}1)$$

$$\frac{}{< t, \textbf{try loop if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit\ \textbf{with}\ Exit\_ \rightarrow void\ \textbf{end} > \longrightarrow < t', void >}$$

This derivation is only possible, if the following $(d1)$ holds:

$$< t, e1 > \longrightarrow < t', false > \quad (d1)$$

$$\frac{}{< t, \textbf{try loop if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit\ \textbf{with}\ Exit\_ \rightarrow void\ \textbf{end} > \longrightarrow < t', void >}$$

From (SE), we can rewrite (d1) as:

$$\frac{< t, e1 > \longrightarrow < t', false >}{< t, \textbf{while}\ e1\ \textbf{do}\ e2 > \longrightarrow < t', void >}$$

which is the required rule (R.1).

Derivation of Rule (R.2)

The derivation of rule (R.2) is similar to the derivation of (R.1). By the application of various semantics rules, we get the following derivation:

$$\frac{\dfrac{< t', e1 > \longrightarrow < t'', true > \quad < t'', e2 > \longrightarrow < t''', void >}{< t, \textbf{if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit > \longrightarrow < t''', void >} (cond\text{-}t) \quad < t''', \textbf{loop if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit > \longrightarrow < t', Exit\ c >}{< t, \textbf{loop if}\ e1\ \textbf{then}\ e2\ \textbf{else raise}\ Exit > \longrightarrow < t', Exit\ c >} (loop\text{-}n)$$

$$\frac{}{< t', void >\longrightarrow< t', void >}(const)$$

$$\frac{}{< t', void[\_ \leftarrow c] >\longrightarrow< t', void >}(rewriting)$$

$$\frac{}{< t, \textbf{try loop if } e1 \textbf{ then } e2 \textbf{ else raise } Exit \textbf{ with } Exit\_ \rightarrow void \textbf{ end} >\longrightarrow< t', void >}(try\text{-}1)$$

This derivation is only possible, when the following $(d2)$ holds:

$$\frac{< t, e1 >\longrightarrow< t'', true > \quad < t'', e2 >\longrightarrow< t''', void > \quad < t''', \textbf{loop if } e1 \textbf{ then } e2 \textbf{ else raise } Exit >\longrightarrow< t', Exit\ c >}{< t, \textbf{try loop if } e1 \textbf{ then } e2 \textbf{ else raise } Exit \textbf{ with } Exit\_ \rightarrow void \textbf{ end} >\longrightarrow< t', void >}$$

Based on lemma *L-a*1, we can derive the following derivation (d3) from derivation (d2) above.

$$\frac{< t, e1 >\longrightarrow< t'', true > \quad < t'', e2 >\longrightarrow< t''', void > \quad < t''', \textbf{try loop if } e1 \textbf{ then } e2 \textbf{ else raise } Exit \textbf{ with } Exit\_ \rightarrow void \textbf{ end} >\longrightarrow< t', void >}{< t, \textbf{try loop if } e1 \textbf{ then } e2 \textbf{ else raise } Exit \textbf{ with } Exit\_ \rightarrow void \textbf{ end} >\longrightarrow< t', void >}$$

From (SE), we can rewrite (d3) as:

$$\frac{< t, e1 >\longrightarrow< t'', true > \quad < t'', e2 >\longrightarrow< t''', void > \quad < t''', \textbf{while } e1 \textbf{ do } e2 >\longrightarrow< t', void >}{< t, \textbf{while } e1 \textbf{ do } e2 >\longrightarrow< t', void >}$$

which is the required rule (R.2).

Proof of Soundness (While-loop)

In this section, we sketch the structure and strategy for the proof of soundness of while-loops. For the complete proof, please see Appendix G.

In the following, we discuss the proof of goal (SE) as formulated in Section 7.4.2.

Let $em, e1, e2, ew, ew', dw, dw', tw, tw', dm$ and vw be arbitrary but fixed.

We assume:

$$wellTyped(em, \textbf{while } E \textbf{ do } Cseq \textbf{ end}) \tag{7.35}$$

$$consistent(em, ew, dw, tw) \tag{7.36}$$

$$< \textbf{while } e1 \textbf{ do } e2, ew', dw', tw' >= T[\![\textbf{while } E \textbf{ do } Cseq \textbf{ end}]\!](em, ew, dw, tw) \tag{7.37}$$

By expanding the definition of (7.37), we know

$$< e1, ew'', dw'', tw'' >= T[\![E]\!](em, ew, dw, tw) \tag{7.38}$$

$$em' = Env(em, E) \tag{7.39}$$

$$< e2, ew', dw', tw' >= T[\![Cseq]\!](em', ew'', dw'', tw'') \tag{7.40}$$

We show:

- $wellTyped(\textbf{while}\ e1\ \textbf{do}\ e2, ew', dw', tw')$ (a)
- $extendsEnv(ew', \textbf{while}\ e1\ \textbf{do}\ e2, ew)$ (b)
- $extendsDecl(dw', \textbf{while}\ e1\ \textbf{do}\ e2, dw)$ (c)
- $extendsTheory(tw', \textbf{while}\ e1\ \textbf{do}\ e2, tw)$ (d)
- $\forall t, t' \in State_w, vw \in Value_w :< t', \textbf{while}\ e1\ \textbf{do}\ e2 > \longrightarrow < t', vw >$
  $\Rightarrow$
  $\exists s, s' \in State_m : equals(s, t) \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq \textbf{end}]\!](em)(s, s') \wedge$
  $\forall s, s' \in State_m, dm \in InfoData : equals(s, t) \wedge$
  $[\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](e)(s, s') \wedge dm = infoData(s')$
  $\Rightarrow equals(s', t') \wedge equals(dm, vw)$ (e)

The goals (a), (b), (c), (d) above are comparatively simple and can be proved based on the strategy similar to the corresponding goals of the proof for the command sequence as discussed in Section 7.4.1. In the following, we only sketch the configurations required for the proof of the crucial goal (e).

We prove the goal (e) by rule induction [158] on the operational semantics of while-loop which is defined above by the two derivation rules (R.1) and (R.2). By the principle of rule induction, the goal (e) for a property P can be re-formulated as:

$$\forall\, t, t' \in Statew, vw \in Valuew :< t, \textbf{while}\ e1\ \textbf{do}\ e2 > \longrightarrow < t', vw > \Rightarrow P(t, t', vw) \quad \text{(e')}$$

where

$$P(t, t', vw) \Leftrightarrow$$
$$[\exists\, s, s' \in State : equals(s, t) \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s')] \wedge$$
$$[\forall s, s' \in State, dm \in InfoData :$$
$$equals(s', t') \wedge [\![\textbf{while}\ E\ \textbf{do}\ Cseq\ \textbf{end}]\!](em)(s, s') \wedge dm = infoData(s')$$
$$\Rightarrow equals(s', t') \wedge equals(dm, vw)] \tag{D-p}$$

where E, $Cseq$ and em are fixed as defined above.

To show goal (e'), based on the principle of rule induction it suffices to show the following sub-goals for while-loop for the corresponding derivation rules (R.1) and (R.2) respectively:

$$\forall\ t, t' \in Statew, vw \in Valuew, e1 \in Expressionw :$$
$$< t, e1 > \longrightarrow < t', false > \Rightarrow P(t, t', vw) \tag{e.a}$$

$$\forall\ t, t', t'', t''' \in \mathit{Statew}, vw \in \mathit{Valuew}, e1, e2 \in \mathit{Expressionw}:$$
$$< t, e1 > \longrightarrow < t'', \mathit{true} > \wedge < t'', e2 > \longrightarrow < t''', \mathit{void} > \wedge$$
$$< t''', \textbf{while}\ e1\ \textbf{do}\ e2 > \longrightarrow < t', \mathit{void} > \wedge P(t''', t', \mathit{void})$$
$$\Rightarrow P(t, t', vw) \qquad \text{(e.b)}$$

With all of the above settings, now the proof of goal (e') gets simpler. The sub-goals (e.a) and (e.b) can be proved now with the help of the derivation rules (R.1) and (R.2) and the corresponding definition (D-p). For the complete proof of the soundness of while-loops and related definitions, please see [98] and Appendix G.

8. Application

In this chapter we discuss the application of our verification framework to the Maple package *DifferenceDifferential*. The rest of the chapter is organized as follows: Section 8.1 gives an overview of the package while Section 8.2 describes the application of our type system to it. In Section 8.3 we sketch the formal specification of the package, in particular, the abstract data type based specification of it's high level procedures. Finally, in Section 8.4 we first demonstrate the verification of an abstract specification example and then discuss the verification of *DifferenceDifferential*.

8.1. The Package "DifferenceDifferential"

The Maple package *DifferenceDifferential* [42] was developed by Christian Dönch to compute difference-differential dimension polynomials in two variables; the computation is based on the concept of relative Gröbner bases which employs the method of Franz Winkler and Meng Zhou as discussed in [162].

The implementation of the method is based on the definition of a difference differential operator (which we call 'ddo'): a 'ddo' $s \in K[\Delta, \Sigma]E$ depends on a differential field K, a set of derivatives Δ, a set of automorphisms Σ and the generators E of the operator respectively field. A 'ddo' can be modeled as a list of tuples each of which is a difference-differential term represented by a quadruple $<c, d, s, e>$, where

- c is an element of K,
- d is a list of integers whose elements have non negative values and whose size equals the size of the derivative set Δ,
- s is a list of integers whose size equals the size of the automorphism set Σ and
- e is an element in E.

The package requires Δ, Σ, E and list of differential operators as parameters. Additionally, the computation takes optional values such as names of the variables, symbols for automorphism and derivatives; if not given, these parameters get default values. The package implements the method in a stepwise fashion which first computes the relative Gröbner bases and then, based on this bases, computes the resulting difference-differential dimension polynomials. For further details of the algorithm and other related mathematical notions and properties, please see [162]. Moreover, the applications of such polynomials can be derived from Einstein [6] notion of a systems' strength as introduced in his theory of relativity.

The package *Difference-Differential* contains both low-level and high-level procedures. The sixteen low-level procedures are standalone, while the eighteen high-level procedures call the low-level procedures. The low-level procedures implement the basic operations on difference-differential dimension operators, e.g. "gleicheterme" (comparing two difference-differential dimension terms), "sigmamax" (computing a difference-differential dimension term with given constraints), "ddsub" and "ddprod" (computing the difference and product of given difference-differential operators); the high-level procedures implement abstract mathematical notions of the algorithm, e.g. "SP" (computing s-polynomials for the given difference-differential operators) and "relGB" (computing relative Gröbner bases).

8.2. Type Checking the Package

The application of the type checker to the Maple package *DifferenceDifferential* required some pre-processing which included:

1. adding type annotations to the package and
2. translating those parts of the package which were not supported in *MiniMaple* into corresponding logically equivalent *MiniMaple* constructs.

Type annotating the package was a challenging task because type information and comments were missing. Therefore, based on discussions with author of the package, we first identified the intentions of various parts/procedures of the package. Then, based on the intentions, we incrementally added type annotations:

- we first annotated the procedure headers, i.e. parameters and their respective return types;
- we then annotated the local (identifier) declarations of the corresponding procedures;
- finally we annotated the other expressions used in the procedures. These expressions included variables which were directly used but not declared (locally or globally).

In general, the appropriate procedure headers (parameters and return types) were (manually) inferred from the corresponding procedure applications. Similarly, type annotations of the local variable declarations were inferred from the use of these variables.

In particular, most of the expressions representing mathematical objects were modeled as nested lists by the author of the package. We tried to annotate the parameters of the package with appropriate types; e.g. we annotated the differential polynomial expression with the list type

$$\textbf{list}([\textbf{Or(integer, symbol)}, \textbf{list}(\textbf{integer}), \textbf{list}(\textbf{integer}), \textbf{symbol}])$$

where each tuple in the list represents a term of the differential polynomial (see Section 8.3).

In general, *MiniMaple* supported most of the expressions appearing in the package. We translated the few unsupported expressions manually to semantically equivalent *MiniMaple* constructs. Such an unsupported expression is the Maple expression `NULL` which can be used to delete an element of the list. In the following example, the third element of the list l is assigned a `NULL` value and which removes this element from l.

```
> l:=[12,43,321,54,4];
                        l := [12, 43, 321, 54, 4]

> l:=subsop(3=NULL, l);
                        l := [12, 43, 54, 4]
```

This construction can be translated into the following equivalent form which reconstructs l from the elements before and after the third element; and which is supported by *MiniMaple*:

```
> l:=[12,43,321,54,4];
                        l := [12, 43, 321, 54, 4]

> l:=[op(1..2, l), op(4..nops(l),l)];
                        l := [12, 43, 54, 4]
```

The translation of nested lists that involve `NULL` expressions was more complicated and involved our auxiliary procedure which deletes a required element.

After the addition of type annotations and the translation of unsupported expressions, the type checker was applied by executing the command:

```
java fmrisc/typechecker/MiniMapleTypeChecker -typecheck DDDP76.m
```

While no crucial errors were found by the type checker, some bad code parts were identified that could cause problems. These code parts were

- declarations of variables that were not used and correspondingly could not be type-checked
- duplicate declarations of the same variables by global and local constructs.

After fixing these errors, the program could be correctly type-checked.

8.3. Specifying the Package

In this section, we discuss the formal specification of the package *DifferenceDifferential*. In particular, we demonstrate the expressiveness of our specification language by for-

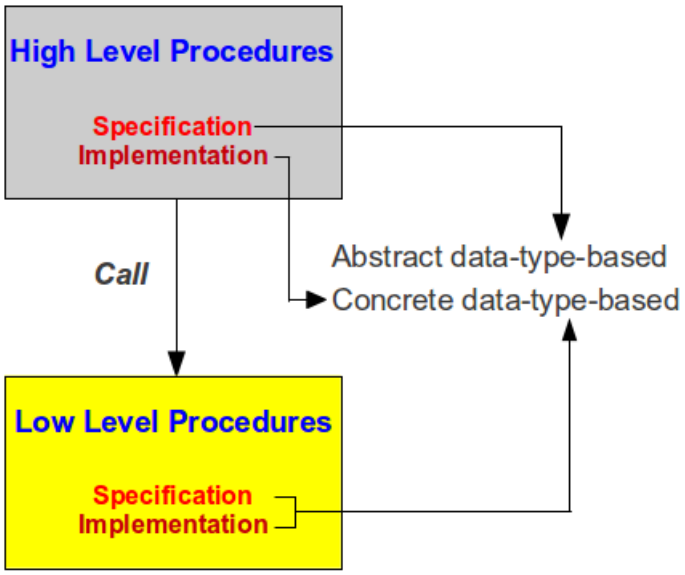

Figure 8.1.: Overview of Specification of the Package *DifferenceDifferential*

mally specifying different mathematical concepts that were used in the package.

Both high-level and low-level procedures of the package have concrete-data type based implementation, however,

- low-level procedures have concrete-data type based specifications while
- the high-level procedures have abstract-data type based specifications,

see Figure 8.1.

The high-level procedures specify and implement some abstract computer algebraic concepts. As an example, we have chosen the high-level procedure *SP* to demonstrate the results on our specification language. This procedure computes the s-polynomial of two difference-differential polynomials with the help of two low-level procedures, *ddsub* and *ddprod* which compute the difference and product of differential polynomials, respectively. As an application of our specification language, we have formally specified all low-level procedures and approx. 70% of the high-level procedures. In the following subsections, we discuss these specifications in more detail.

8.3.1. Concrete Data Type-based Specifications

As a starting point, based on a concrete data type, we formally specify the main mathematical notion used in the package, namely the concept of a "difference-differential operator/polynomial (ddo)". Then we specify the low-level procedures *ddsub* and

ddprod based on the (concrete data type) specification of 'ddo'.

Concretely, we declare a type 'ddo' and its related types as follows:

```
‘type/ddo‘:=list(ddo_term);
‘type/ddo_term‘:=[Or(integer,symbol),list(integer),
                                    list(integer),symbol];
‘type/ddo_data‘:=[integer, integer, list(symbol)];
```

Here the type `ddo_data` represents the corresponding constraint elements of the 'ddo', i.e. Δ, Σ and E respectively. Now, we formally specify the aforementioned mathematical properties of a 'ddo' as follows:

```
define(isddo(a:: ddo, d:: ddo_data)::boolean,
       isddo(a:: ddo, d:: ddo_data) =
         forall(n:: integer, 1 <= n and n <= nops(a) implies
                              isddo_term(d, a[n])
      );
define(isddo_term(d:: ddo_data, t:: ddo_term)::boolean,
       isddo_term(d:: ddo_data, t:: ddo_term) =
          inField(t[1], d) = true and
          forall(j:: integer, 1<=j and j <= nops(t[2]) implies
                0 <= t[2][j]) and
          nops(t[2]) = d[1] and nops(t[3]) = d[2] and
          exists(n::integer,1<=n and n <= nops(d[3]) and t[4]=d[3][n])
      );
define(inField(s:: Or(integer, symbol), d:: addo_data)::boolean);
define(sub_ddo(a:: ddo, b:: ddo, d:: ddo_data, m:: integer)::ddo, ...);
define(mul_ddo(a:: [symbol,list(integer),list(integer)],
               b:: ddo, d:: ddo_data, m:: integer)::ddo, ...);
```

In detail, the specification function `isddo` says that an object `a` of type 'ddo' is a well-formed 'ddo', if each element of `a` is a differential term as specified by the corresponding predicate `isddo_term`. The function `isddo_term` is a logical conjunction of four formulas where each formula specifies the corresponding mathematical property of each element of the term, respectively.

Based on this specification of 'ddo', we sketch the (concrete data type based) specification of the low-level procedure *ddsub* which computes the difference of the two given difference-differential operators:

```
ddsub := proc(c:: ddo, b::ddo)::ddo;
(*@
requires isddo(c, [anzdelta, anzsigma, generators]) = true and
```

```
         isddo(b, [anzdelta, anzsigma, generators]) = true;
global EMPTY;
ensures isddo(RESULT, [anzdelta, anzsigma, generators]) and
        RESULT = sub_ddo(c, b, [anzdelta, anzsigma, generators], 0);
@*)
...
end proc;
```

The specification states

- as a pre-condition, that both of the procedure arguments c and b are well-formed 'ddo's
- that the body of the procedure does not modify any global variable and
- as a post-condition, that the result of the procedure is a well-formed 'ddo' whose value is defined by the application of the specification function `sub_ddo`.

Here, `anzdelta, anzsigma` and `generators` are global variables which form the `addo_data` as discussed above.

Similarly, the corresponding procedure *ddprod* that computes the differential product is specified as follows:

```
ddprod := proc (u::ddo, v::ddo)::ddo;
(*@
requires isddo(v, [anzdelta, anzsigma, generators]) = true and
         isddo(u, [anzdelta, anzsigma, generators]) = true;
global EMPTY;
ensures isddo(RESULT, [anzdelta, anzsigma, generators]) and
        RESULT = mul_ddo(u, v, [anzdelta, anzsigma, generators]);
@*)
...
end proc;
```

The specification of the procedure *ddprod* is the same as explained above for the procedure *ddsub*. However, the value of the result of the procedure *ddprod* is defined by the application of the specification function `mul_ddo`.

8.3.2. Abstract Data Type-based Specifications

To specify the high-level procedure *SP*, we first formally specify the notion of an abstract 'ddo' with the help of an abstract data type; the support of abstract data types is a key feature of our formal specification language. The specification of procedures operating on concrete representations of the abstract data type follows the strategy pioneered by Tony Hoare [81] which is based on an "abstraction function" that maps the concrete data type into the corresponding abstract data type.

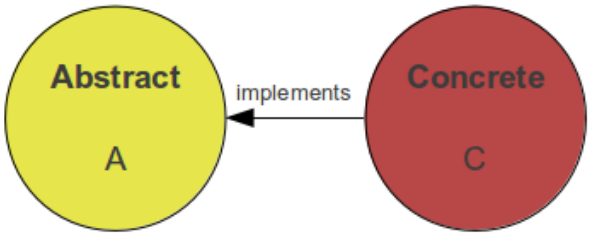

Figure 8.2.: Formulation for Abstract Specification and Verification

In more detail, if there is an abstract object A which has an underlying implementation based on a concrete object C, then our formulation for the specification of such an object consists of the following elements (see Figure 8.2):

1. an abstract data type A (the *model* type),
2. a concrete data type C (the *representation* type),
3. an *abstraction* function $abstract : C \to A$.

Subsequently, the contract of a method operating on a concrete object $x \in C$ is then specified in terms of its abstract counterpart $abstract(x) \in A$ (i.e. the specification refers to x only in the form $abstract(x)$). In the following, we specify step-wise the notion of a 'ddo' based on this strategy.

Model

To model 'ddo', we first declare an abstract 'ddo' (which we call 'addo') and its corresponding constructors as follows:

```
(* addo type declarations *)
‘type/addo‘;

(* addo constructors *)
define(create_addo()::addo);
define(add_term_addo(d:: addo_data, a:: addo, t:: ddo_term)::addo);
```

The first constructor `create_addo` returns an empty 'addo' while the second constructor `add_term_addo` adds a new term `t` to the given 'addo' `a` such that the term `t` respects the data `d`. In fact, here we abstract away the concrete data type 'list' (representing a 'ddo') to 'addo'. Therefore, the two constructors have one-to-one correspondence to the list constructors, i.e. to *empty* and *cons*. Then we specify the other properties of abstract 'ddo'.

```
(* addo operations/properties *)
define(isAddo(a:: addo)::boolean,
       isAddo(a:: addo) =
        forall(n:: integer, 1 <= n and n <= length_addo(a) implies
          isAddo_term(get_addo_data(a), get_addo_term(a))
      );
define(isAddo_term(d:: addo_data, t:: ddo_term)::boolean,
       isAddo_term(d:: addo_data, t:: ddo_term) =
          inField(t[1], d) = true and
          forall(j:: integer, 1<=j and j <= nops(t[2]) implies
                 0 <= t[2][j]) and
          nops(t[2]) = d[1] and nops(t[3]) = d[2] and
          exists(n::integer,1<=n and n <= nops(d[3]) and t[4]=d[3][n])
      );
define(length_addo(a:: addo)::integer);
define(get_addo_data(a:: addo)::addo_data);
define(get_addo_term(a:: addo)::ddo_term);
define(remove_term_addo(a:: addo)::addo);
define(is_empty_addo(a:: addo)::boolean,
       is_empty_addo(a:: addo) = 'if'(a = create_addo(),true,false));
define(equals_addo(a:: addo, b:: addo)::boolean, ...);
define(equals_addo_term(t1:: ddo_term, t2:: ddo_term)::boolean, ...);
define(sub_addo(a:: addo, b:: addo):: addo);
define(mul_addo(a:: addo, b:: addo):: addo);
...
```

One of the main properties of interest for 'addo' is specified by predicate `isAddo` which says that an 'addo' `a` is an abstract 'ddo', if each of its terms is an abstract differential term as specified by predicate `isAddo_term`.

Representation

The underlying *representation* as an implementation of 'addo' is a concrete type `addo_rep`:

```
`type/addo_rep`:=list(ddo_term);
```

In fact, this type is identical to the type 'ddo' declared before.

Abstraction

Based on the previously defined concrete representation type, we define the *abstraction* of concrete type `addo_rep` to an abstract type `addo` by the function `to_abstract_addo`.

In addition to representation `m`, the function also requires a parameter `d` which is used to specify various constraints on the corresponding abstract 'ddo' respectively terms.

```
define(to_abstract_ddo(d:: addo_data, m:: addo_rep)::addo,
       to_abstract_addo(d:: addo_data, m:: addo_rep) =
         'if'(nops(m) = 0,
               create_addo(),
               add_addo_term(d, [op(1..nops(m)-1, m)], m[nops(m)]) )
      );
```

In detail, the function says that for a given 'ddo' `m`

- if the length of `m` is zero, then create an empty 'addo' by the corresponding constructor `create_addo()` , otherwise
- construct an 'addo' by the constructor `add_addo_term` which adds the last element of list `m` as an 'addo' term to the remaining list.

In the following, we sketch the formal specification of the high-level procedure *SP* based on the above specification (model type and abstraction function) of an abstract 'ddo'.

Procedure Specification

The procedure *SP* computes the s-polynomial of two given 'ddo's (`s` and `t`) and other auxiliary arguments. The procedure has arguments

- z $\in \mathbb{Z}$, z ≥ 0
- s, t $\in K[\Delta, \Sigma]E$
- v $\in [\Delta, \Sigma]E$
- s1, t1 $\in [\Delta, \Sigma]$

and returns a 'ddo'.

Based on above description, we specify the procedure with the help of the abstract data type as sketched below:

```
SP := proc (z::integer, s::ddo, t::ddo,
            v::[list(integer), list(integer), list(symbol)],
            s1::[list(integer),list(integer)],
            t1::[list(integer),list(integer)])::ddo;
(*@
requires
    1 <= z and z <= power(2, anzsigma) and
    isAddo(to_abstract_addo([anzdelta,anzsigma,generators],s))=true and
    isAddo(to_abstract_addo([anzdelta,anzsigma,generators],t))=true and
    ...
```

```
global EMPTY;
ensures
    LET ad=to_abstract_addo([anzdelta,anzsigma,generators],RESULT) IN
        isAddo(ad) = true and
        ad=sPol(z,
                to_abstract_addo([anzdelta, anzsigma, generators], s),
                to_abstract_addo([anzdelta, anzsigma, generators], t),
                v , s1, t1);
@*)
...
...
c1 := ddprod(d1, f);
c2 := ddprod(d2, g);
sp := ddsub(c1, c2);
return sp;
end proc;
```

In detail, the specification is structured as follows:

- In the pre-condition, we first construct abstract 'ddo's by the application of function `to_abstract_ddo` to the actual arguments of the procedure `s` and `t`: we then apply the predicate `isAddo` to the constructed abstract 'ddo' in order to specify the mathematical constraints on an abstract 'ddo'. Corresponding constraints are also specified on the other other auxiliary arguments of the procedure.
- Similarly, in the post-condition, we first construct an abstract 'ddo' from the `RESULT` of the procedure and then test the abstract 'ddo' if it respects the predicate `isAddo`. Furthermore, this `RESULT` equals the value of the s-polynomial defined with the help of the specification function `sPol`.

In the following, we discuss the verification of the implementation of *SP* (which calls the low-level procedures *ddsub* and *ddprod*) with respect to this specification.

8.4. Verifying the Package

In this section, we discuss the verification of selected procedures of the package *DifferenceDifferential*. In Subsection 8.4.1, we describe the verification of low-level procedures of the package. In Subsection 8.4.2, we describe the verification of an abstract data type based example program (written in Why3ML) from which we have derived strategies for the appropriate verification of high-level procedures. Finally, in Subsection 8.4.3, we use this strategy to verify the high-level procedure *SP*.

8.4.1. Verification of Low-level Procedures

We have verified all the sixteen low-level procedures of *DifferenceDifferential* which includes approx. 80% automatic and 20% interactive proofs. The automatic proofs were performed using the SMT (satisfiability modulo theories [140]) solvers Z3 [51], CVC3 [43] and Alt-Ergo [45] while the interactive proofs were performed with the help of the proving assistant Coq [20]. Most of the proofs could be performed automatically because the low-level procedures of the package *DifferenceDifferential* have both specifications and implementations based on concrete-data types as discussed in Section 8.3.

The verification of the procedures also involved the definition and verification of loop invariants and termination terms (aka variants) in addition to the procedure specifications (pre- and post-conditions). One of the challenges of the verification of low-level procedures was the adequacy of the definition of loop invariants and corresponding termination terms. Here, the definitions of loop invariants and termination terms were refined incrementally to make them directly amenable for verification after translating them into corresponding Why3ML constructs.

In the following we sketch the *MiniMaple* specification of a simple loop from the low-level procedure *ddsub* which is the result of quite a few refinements.

```
...
l := nops(f);
...
while (i0 < nops(g)) do
(*@
invariant ( forall(j::integer, i0 < j and j < nops(g) implies
                               g[j] = OLD(g[j])) ) and
          ( forall(k::integer, 0 <= k and k < i0 implies
                               g[k] = OLD(-g[k])) ) and
          ( forall(j0::integer, 0 <= j0 and j0 < l implies
                                f[j0] = OLD(f[j0])) ) and
          ( forall(k0::integer, k0 >= l and k0 < nops(f) implies
                                f[k0] = OLD(-g[k0 - nops(g)])) ) and
          ( nops(f) = l + i0 and nops(g) = nops(OLD(g)) ) and
          ...;
decreases ( nops(g) - i0 );
@*)
...
g[i0][1] := -g[i0][1];
f := [nops(f), g[i0]];
...
i0 := i0 + 1;
```

```
end do;
```

The loop invariant is specified with an `invariant` construct. In principle, the invariant specifies the relationship between the elements of the two lists f and g. Furthermore, the invariant describes some additional constraints about the corresponding elements of the lists. The termination term of the loop is specified with the corresponding `decreases` construct; it denotes a nonnegative value which is decreased after each loop iteration.

However, the proofs of the verification conditions (including procedure and loop specifications) of the low-level procedures required some lemmas to be added manually because the definitions of some specification functions generated by the translator were not adequate. For instance, as discussed in Section 8.3.1, the specification of the procedure *ddsub* involves a specification function *sub_ddo* which computes the difference of the 'ddo's. However, the function *sub_ddo* which is generated by the translator (for the corresponding specification function *ddsub*) is not directly amenable for verification. Therefore, to make the definition of the translated function *sub_ddo* adequate for proving, we manually introduced the lemmas:

```
(* the following two lemmas are about "sub_ddo" function *)
lemma ddo_sub0:
  forall b: list ddo_term, c: list ddo_term, d: ddo_data, j: int, k: int.
  0 <= j < length c /\ 0 <= k < length b ->
      equals_ddo_term (nth j c) (nth k b) (d) ->
              sub_ddo b c d (k+1) = sub_ddo b c d k

lemma ddo_sub1:
  forall b: list ddo_term, c: list ddo_term, d: ddo_data, j: int, k: int.
  0 <= j < length c /\ 0 <= k < length b ->
    forall s: symbol, s1: symbol, s2: symbol.
    not (equals_ddo_term (nth j c) (nth k b) (d)) ->
      sub_ddo b c d (k+1) =
      sub_ddo b c d k ++ Cons (sub_ddo_term (nth j c) (nth k b) (d)) Nil
```

The above lemmas state the facts that the specification function *sub_ddo* (over lists of 'ddo' terms) correctly computes the result. These lemmas were proved manually using the interactive prover Coq based on the strategy of structural induction. In future, we plan to generate lemmas of this kind automatically.

8.4.2. Verification of High-level Procedures

The verification of the high-level procedures of *DifferenceDifferential* included eight proofs, which were all interactive. However, as discussed in Section 8.3, we have devised a logical formulation for the verification of abstract data type based high-level procedures. The challenge here was to prove the correctness of procedures, which

- on the one hand have an implementation based on concrete data types (e.g. the difference-differential operator is implemented as a list of its terms as tuples) and
- on the other hand are specified by abstract data types (e.g. the difference-differential operator is specified by an abstract data type "addo" with corresponding operations and mathematical properties).

In this section, we discuss the appropriate strategy for the verification of this kind of procedures based on the example of a typical"stack"; we have used this example to experiment with strategies to derive successful proofs.

As discussed in Section 8.3.2 a concrete *representation* type C can be specified with the help of an abstract type A (the *model* type) and an *abstraction* function $abstract : C \to A$.

Our formulation for the verification of a procedure that is specified in terms of A and operating on objects of type C consists of:

1. a *concretization-relationship* between C and A defined as "$concrete \subseteq C \times A$",
2. an *invariant* predicate "$invariant \subseteq C$" and
3. a *lemma*

$$\forall\, c : C, a : A, invariant(c) \Rightarrow (a = abstract(c) \Leftrightarrow concrete(a, c)).$$

The concretization relation and the associated lemma make knowledge about the derivation of a concrete representation from an abstract model directly available in the proof such that the reasoning gets simpler.

To demonstrate the adequacy of our formulation with respect to the underlying tools used in our verification framework (Why3 and the supported back-end provers), we specify in the following the abstract data type "stack" and verify its mutable array based implementation.

Model

A stack of integer values is modeled as an algebraic type as follows:

```
type stack = Create | Push (stack) (int)
```

The two constructors correspond to

1. the construction of an empty stack and
2. the construction of a stack by pushing a given integer (element) to the given stack respectively.

The other typical operations of the stack are defined by pattern matching respectively structural induction as follows:

```
function is_empty_stack (s: stack) : bool=
match s with
```

```
| Create  -> True
| Push s1 e -> False
end

function top (s: stack) : int=
match s with
| Create -> 0
| Push s1 e -> e
end

function pop (s: stack) : stack=
match s with
| Create -> s
| Push s1 e -> s1
end

function length_stack (s: stack) : int=
match s with
| Create -> 0
| Push s1 e -> 1 + length_stack (s1)
end
```

Representation

An abstract stack can be represented by a tuple of two elements as follows:

```
type stack_rep = {| mutable size: int; data: mutable array int |}
```

The first element `size` represents the number of elements in the stack and the second element `data` represents the actual stack elements as a mutable array of integers. Here, the `mutable` array was later used in an imperative code part of the implementation of the stack function `push_array`.

Abstraction

The function `to_abstract_stack` defines the mapping of a stack-representation of type `stack_rep` to a corresponding abstract type `stack` as follows:

```
function to_abstract_stack (r: stack_rep) : stack

axiom to_abstract_stack0:
 forall r: stack_rep. r.size = 0 -> to_abstract_stack (r) = Create
```

```
axiom to_abstract_stack1:
 forall r: stack_rep, r1: stack_rep.
  let r1 = {| size=r.size-1; data=r.data |} in
   r.size > 0 ->
   to_abstract_stack(r) = Push(to_abstract_stack (r1))(r.data[r.size-1])
```

In detail, the two axioms say that

1. if the size of the stack representation `r` is zero, then the abstraction of `r` is an empty stack and
2. if the size of the stack-representation `r` is greater than zero, then the abstraction of `r` is the result of the application of the push operation to a stack `r1` (derived from `r` by removing the last element).

In the following, we discuss the corresponding formulation for the verification of the implementation specified above.

Concretization Relation

The concretization relationship specifies the other direction of the *abstraction* function. We thus specify the relation of the stack-representation `r` to an abstract stack `s` as defined below:

```
predicate concrete_stack (s: stack) (r: stack_rep)=
match s with
| Create -> r.size = 0
| Push s1 e ->
   r.size > 0 /\ r.size < r.data.length ->
     r.data[r.size-1] = e /\
     exists a1: array int. r.data.length = a1.length /\
     s1 = to_abstract_stack ({|size=r.size-1; data=a1 |}) /\
     forall j: int. 0 <= j < r.size-1 -> r.data[j] = a1[j]
end
```

The relation `concrete_stack` says that

- if abstract stack `s` is empty, then the size of the stack representation `r` is zero
- if the abstract stack is non-empty, i.e. of form `Push s1 e` and if the size of `r` is greater than zero and less than the length of the stack representation, then the element `e` is the last element of the stack representation.

The formulation is based on an array `a1` whose abstraction is equal to the stack `s1`.

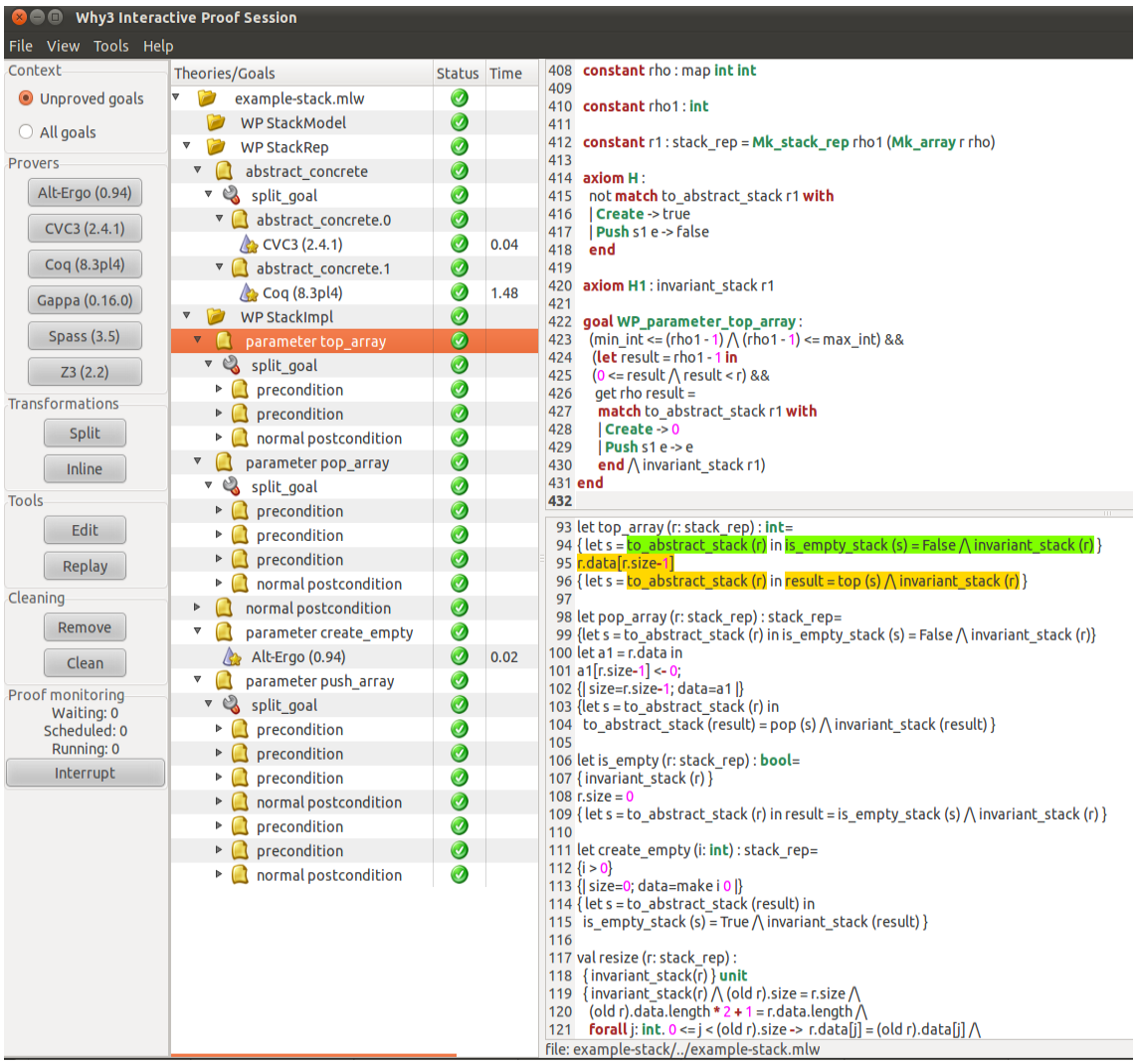

Figure 8.3.: Verification of a Stack Example

Invariant

There can be various such combinations of the elements (`size` and `data`) of the stack-representation from which we cannot construct a legal abstract stack. To avoid such instances of the stack-representation we define the following invariant:

```
predicate invariant_stack (r: stack_rep) = 0 <= r.size <= r.data.length
```

which prevents illegal stack representations.

Specified Stack Implementation

In the following, we give the complete Why3ML code for the stack example presumed in the previous subsections:

```
(* stack model (abstract type stack and its operations) *)
module StackModel

  use export int.Int
  use export list.List
  use export bool.Bool
  use export module ref.Ref
  use export module array.Array

type stack = Create | Push (stack) (int)

function length_stack (s: stack) : int=
match s with
| Create -> 0
| Push s1 e -> 1 + length_stack (s1)
end

function is_empty_stack (s: stack) : bool=
match s with
| Create  -> True
| Push s1 e -> False
end

function top (s: stack) : int=
match s with
| Create -> 0
| Push s1 e -> e
end

function pop (s: stack) : stack=
match s with
| Create -> s
| Push s1 e -> s1
end
```

```
function pop0 (s: stack) : int=
match s with
| Create -> 0
| Push s1 e -> e
end

function get_stack (s: stack) (i: int) : int

axiom get_stack0: forall s: stack, i: int. i = 0 -> get_stack (s) (i) = top (s)
axiom get_stack1: forall s: stack, i: int.
                        i > 0 -> get_stack (s) (i) = get_stack (pop (s))  (i-1)

end

(* stack representation (mapping, concretization and invariant) *)
module StackRep

  use import module StackModel
  use export module arith.Int32

type stack_rep = {| mutable size: int; data: array int |}

function to_abstract_stack (r: stack_rep) : stack

axiom to_abstract_stack0: forall r: stack_rep.
                r.size = 0 -> to_abstract_stack (r) = Create

axiom to_abstract_stack1:
        forall r: stack_rep, r1: stack_rep.
        let r1 = {| size=r.size-1; data=r.data |} in
                r.size > 0 ->
        to_abstract_stack (r) = Push (to_abstract_stack (r1)) (r.data[r.size-1])

predicate concrete_stack (s: stack) (r: stack_rep)=
match s with
| Create -> r.size = 0
| Push s1 e -> r.size > 0 /\ r.size < r.data.length -> r.data[r.size-1] = e /\
                exists a1: array int. r.data.length = a1.length /\
                s1 = to_abstract_stack ({|size=r.size-1; data=a1 |}) /\
                forall j: int. 0 <= j < r.size-1 -> r.data[j] = a1[j]
end

predicate invariant_stack (r: stack_rep)=
0 <= r.size <= r.data.length

lemma abstract_concrete:
        forall r: stack_rep, s: stack.
        invariant_stack (r) -> ( s = to_abstract_stack (r) <-> concrete_stack (s) (r))

end
```

```
(* stack implementation (with abstract specifications) *)
module StackImpl

  use import module StackModel
  use import module StackRep

let top_array (r: stack_rep) : int=
{ let s = to_abstract_stack (r) in is_empty_stack (s) = False /\ invariant_stack (r) }
r.data[r.size-1]
{ let s = to_abstract_stack (r) in result = top (s) /\ invariant_stack (r) }

let pop_array (r: stack_rep) : stack_rep=
{let s = to_abstract_stack (r) in is_empty_stack (s) = False /\ invariant_stack (r)}
let a1 = r.data in
a1[r.size-1] <- 0;
{| size=r.size-1; data=a1 |}
{let s = to_abstract_stack (r) in
        to_abstract_stack (result) = pop (s) /\ invariant_stack (result) }

let is_empty (r: stack_rep) : bool=
{ invariant_stack (r) }
r.size = 0
{ let s = to_abstract_stack (r) in result = is_empty_stack (s) /\ invariant_stack (r) }

let create_empty (i: int) : stack_rep=
{i > 0}
{| size=0; data=make i 0 |}
{ let s = to_abstract_stack (result) in
        is_empty_stack (s) = True /\ invariant_stack (result) }

val resize (r: stack_rep) :
        { invariant_stack(r) } unit
        { invariant_stack(r) /\ (old r).size = r.size /\
                (old r).data.length * 2 + 1 = r.data.length /\
                forall j: int. 0 <= j < (old r).size ->  r.data[j] = (old r).data[j] /\
                forall k: int. (old r).size <= k < r.data.length -> r.data[k] = 0 }

let push_array (r: stack_rep) (e: int) =
{ invariant_stack (r) }
if r.size = r.data.length then
begin
resize(r)
end;
r.data[r.size] <- e;
r.size <- r.size+1
{ invariant_stack(r) /\
        let s = to_abstract_stack (old r) in
        let s1 = to_abstract_stack (r) in
                s1 = Push s e }
```

```
end
```

The example above contains three modules where

1. module `StackModel` specifies the stack *model*,
2. module `StackRep` specifies the stack *representation*, *abstraction*, *concretization relation* and *invariant* and
3. module `StackImpl` contains the actual stack implementation (based on representation) annotated with abstract specifications.

In the following, we discuss the verification of the above example.

Verification

Figure 8.3 shows a screen shot of the Why3-GUI for the verification of the stack example. In fact, the verification conditions generated by Why3 are proved with the help of the following lemma which is part of our logical formulation for verification:

```
lemma abstract_concrete:
  forall r: stack_rep, s: stack.
    invariant_stack (r) ->
      ( s = to_abstract_stack (r) <-> concrete_stack (s) (r) )
```

This lemma makes the knowledge of stack representation `r` and abstract stack `s` directly available for use in proving; it was proved in Coq by structural induction on stack `s`.

With the introduction of the lemma, the proofs of most of the implementation functions were fully automatic. Only the proof of the implementation function `push_array` was partially interactive, because the push operation needs to `resize` the stack-representation array. Here, the interactive proof was a mix of case analysis, induction on the size of the stack and some other tactics of Coq.

8.4.3. Verification of the High-level Procedure "SP"

We approached the goal of verification of the high-level procedure *SP* based on our experiment with the verification of the stack example discussed in the previous section. The verification of the implementation of the procedure requires to define a concretization relation and an invariant as discussed in the following.

Concretization Relation

The concretization function specifies the relation between an abstract 'ddo' `a` and its corresponding concrete representation `m` as defined below:

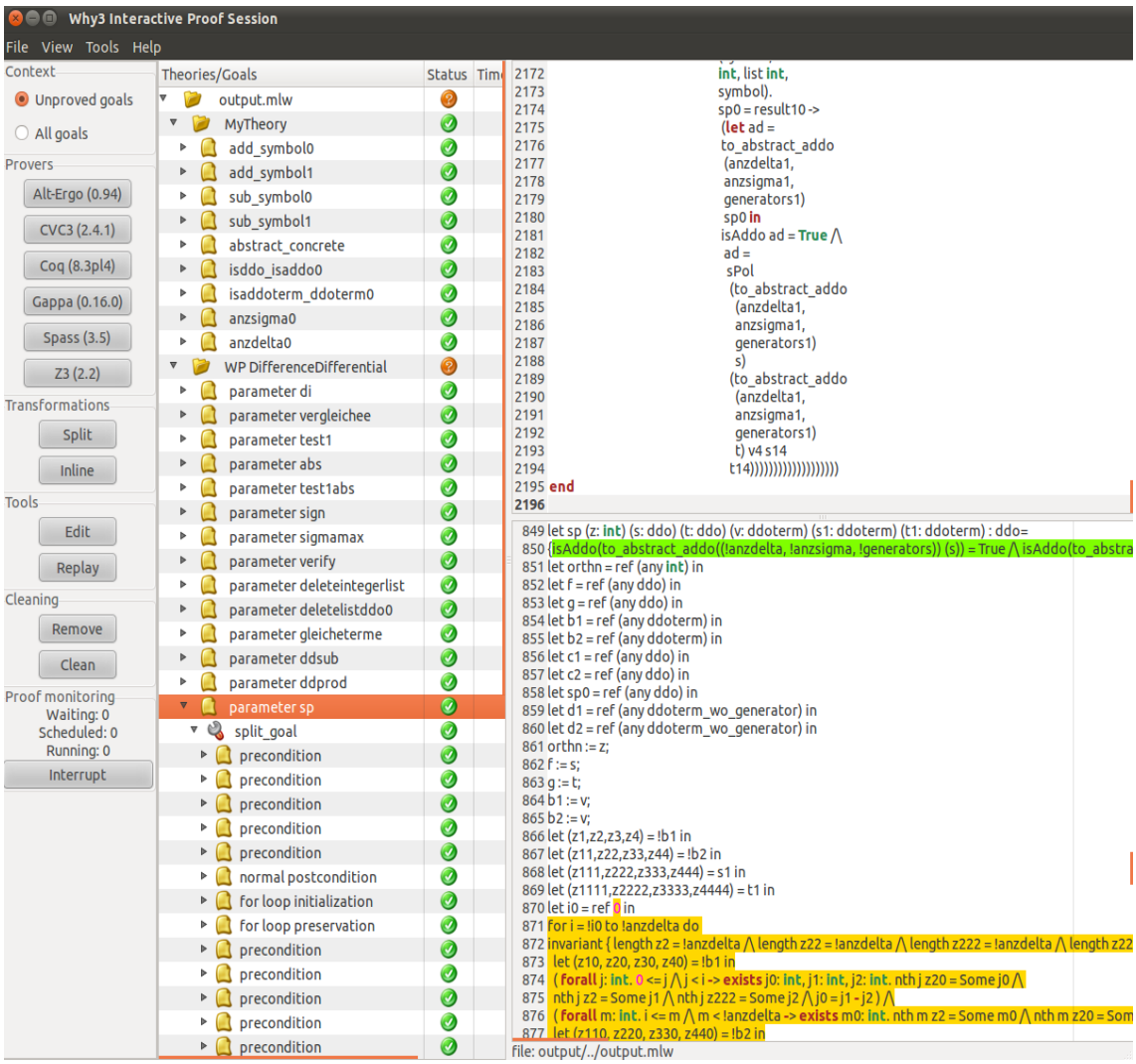

Figure 8.4.: Verification of DifferenceDifferential

```
define(concrete_addo(d:: addo_data, m:: addo_rep, a:: addo)::boolean,
       concrete_addo(d:: addo_data, m:: addo_rep, a:: addo) =
         'if'(nops(m) > 0 ,
            nops(m) > 0 implies
              exists(a1:: list(ddo_term), t:: ddo_term,
                     a = add_addo_term(d,to_abstract_addo(d, a1),t) and
                     t = m[nops(m)] and nops(a1) = nops(m) - 1 and
                     forall(i:: integer, 1 <= i and i <= nops(m)-1
                            implies a1[i] = m[i])
                    )
            , a = create_addo() )
      );
```

In detail, the concretization relation says that if the concrete representation of 'ddo' `m` is non-empty, then the abstract 'ddo' `a` is constructed with the help of a corresponding constructor `add_addo_term` by adding some term `t` to some abstract 'ddo' (that is based on some concrete representation `a1`). Here, the term `t` is the last element of the concrete 'ddo' `m` and the elements of `a1` are the same as of `m` such that the last element of `m` is missing in `a1`. Furthermore, if the concrete representation `m` is empty, then the given abstract 'ddo' `a` is equal to the corresponding constructor `create_addo`.

Invariant

Now, we define the invariant for the concrete representation of the 'ddo' that ensures the legal construction of abstract 'ddo' for a given concrete representation.

```
define(invariant_addo(d:: addo_data, m:: addo_rep)::boolean,
       invariant_addo(d:: addo_data, m:: addo_rep) =
       'if'(d[1] >= 0 and
            nops(d[3]) > 0 and nops(m) >= 0 and
            isddo(m, d) = true, true, false)
      );
```

In principle, the function specifies some constraint on `add_data` which is used to specify the difference-differential dimension terms of the given representation `add_rep` of a 'ddo'. The function returns true only if the first element of the data is a positive integer and the list of generators is not empty as per the definition of 'ddo'. Furthermore, the given `m` is a well-formed 'ddo' and hence respects `isddo`.

Verification

To prove the verification conditions generated by Why3 for the procedure *SP*, we introduced the following lemmas:

```
lemma abstract_concrete:
  forall a: addo, m: addo_rep, d: addo_data.
    invariant_addo(d) (m) ->
      a = to_abstract_addo(d) (m) <-> concrete_addo (d) (m) (a)

lemma isddo_isaddo0:
  forall d: addo_data, m: addo_rep, a: addo.
      a = to_abstract_addo (d) (m) ->
       isddo (m) (d) = isAddo (to_abstract_addo (d) (m))
```

The first lemma is the part of our verification strategy which says that for any abstract 'ddo' `a`, underlying concrete representation `m` and constraint data `d`, if the invariant holds for the concrete 'ddo' `m`, then `a` is the abstraction of the concrete 'ddo' if and only if the concretization relation between `m` and `a` holds. The second lemma is introduced to remove the redundancy of the proof of various verification conditions which include similar goals. The lemma essentially says that if we abstract any concrete representation `m` to `a` then the definition of the concrete 'ddo' predicate `isddo` is equal to the definition of the abstract 'ddo' predicate `isAddo`. In fact, as mentioned above the procedure *SP* includes the call to two other procedures which have concrete data type specifications of 'ddo'. Both of the lemmas were proved in Coq by the principle of structural induction on the list `m` and the corresponding constructor of an abstract 'ddo' `a`.

With the introduction of the lemmas, most of the proofs of the verification conditions were automatic. However, the proofs of some conditions that involved loop-invariants and procedure calls were also interactive. The interactive proofs were mainly based on the structural induction along-with other Coq tactics, e.g. destruction of definition of abstract 'ddo', case analysis and the expansion of the lemma. The task of verification here was simpler as compared to the verification of our stack-example because the constructors of abstract 'ddo' correspond to the constructors of the underlying representation (i.e. list). The verification of the high-level procedures in general and of *SP* in particular is presented by a screenshot of the Why3-GUI in Figure 8.4.

9. Conclusions and Future Work

In this chapter, we first review the design and development of our verification framework for the specification and verification of computer algebra software; then we sketch the lessons learned followed by a discussion on possible future extensions.

In this thesis, we have first formalized the syntax and semantics of *MiniMaple* and its specification language. We have then developed the type systems for *MiniMaple* and its specification language. We have elaborated the translation of a *MiniMaple* program into a Why3ML program, and have proved the soundness of the translation based on the denotational semantics of *MiniMaple* and the operational semantics of Why3ML. Finally, we have applied our framework to verify the main parts of the non-trivial Maple package *DifferenceDifferential*. The verification of the high-level procedures of the package was performed with the help of a strategy for specifying and verifying such procedures which on one hand have concrete data type based implementations and on the other hand have specifications based on abstract data types.

During the entire course of the design and development of this framework, we have gained fundamental knowledge in the various issues of designing and formalizing programming and specification languages in general and computer algebra languages in particular. We have learned to formalize the type system for *MiniMaple* and its specification language based on the specific notations of type systems. The subsequent definition and formalization of the formal semantics of *MiniMaple* and its specification language gave us a deep insight to understand the behavioral principles of these languages and how to apply a variety of mathematical formalisms and notions to model the corresponding semantics. Also we have gained the capabilities to use the intermediate verification tool Why3 and some of the automated and interactive theorem provers (e.g. Z3 and Coq) that it supports as back-ends. We have used the general principles (tactics and proof strategies) of automated theorem proving and also have gained experienced with a handy approach for the specification and verification of high-level procedures by modeling mathematical notions with the help of abstract data types. During the application of the framework to the package we have realized the importance of formal annotations (types and specifications) and informal ones (comments) of programs in order to understand the correct behavior of the programs; this becomes extremely important when sometime later a refactoring of the program is required. The non-trivial proof for the soundness of the translation has enriched our experience in proving by practicing a variety of proof strategies, e.g. rule-based

induction.

As a next step, based on our strategy for the specification and verification of abstract data type based specifications, we may complete the specification and verification of all the high-level procedures of the package *DifferenceDifferential*. Furthermore, we may extend our verification framework in various directions:

- The syntax of *MiniMaple* (and its specification language) can be extended to support more expressions of Maple that are used for general purpose computing.
- We may use the specification language for *MiniMaple* programs to generate executable assertions that are embedded in such programs and can check the validity of pre/post conditions at runtime.
- As an alternative to Why3 (developed by LRI, France), we may use Boogie [13] (developed by Microsoft, USA) as an intermediate verification framework. Boogie also supports as back-ends various automated theorem provers (i.e. Simplify [53] and Z3) and interactive ones (i.e. Isabelle/HOL [124]). Moreover, the framework is also used by various front-end tools for some programming languages, e.g. C# and C [22].
- The results of our verification framework can be also applied to other computer algebra systems, in particular Mathematica. Mathematica shares with Maple many concepts such as the basic kinds of runtime objects; thus the type system of *MiniMaple* in principle can also be applied to Mathematica. However, the task of verification of Mathematica programs is more complex as the programming language is rule-based; here we would need to investigate the exhaustiveness of rules and possible contradictions among rules. In principle, we could then translate a Mathematica program into an equivalent procedural program such that our verification tool can be applied.

Furthermore, the application of our framework to verify other Maple packages developed at our institute will further demonstrate the expressiveness and limitations of our framework and will also show directions for future extension.

Appendices

A. Syntax of MiniMaple

In this appendix we give the formal abstract syntax (language grammar) of *MiniMaple*.

Prog ∈ Program
Cseq ∈ Command_Sequence
C ∈ Command
Elif ∈ ElseIf
Catch ∈ Catch
Eseq ∈ Expression_Sequence
E ∈ Expression
S ∈ Sequence
R ∈ Recurrence
Pseq ∈ Parameter_Sequence
P ∈ Parameter
M ∈ Modifiers
Iseq ∈ Identifier_Sequence
I ∈ Identifier
Itseq ∈ Identifier_Typed_Sequence
It ∈ Identifier_Typed
Bop ∈ Boolean_Operators
Uop ∈ Unary_Operators
Esop ∈ Especial_Operators
Tseq ∈ Type_Sequence
T ∈ Type
N ∈ Numeral

Prog ::= Cseq
Cseq ::= EMPTY | (C; | E;)Cseq
C ::= **if** E **then** Cseq Elif **end if**; | **if** E **then** Cseq Elif **else** Cseq **end if**;
| **while** E **do** Cseq **end do**;
| **for** I **in** E **do** Cseq **end do**;
| **for** I **in** E **while** E **do** Cseq **end do;**
| **for** I **from** E **by** E **to** E **do** Cseq **end do**;
| **for** I **from** E **by** E **to** E **while** E **do** Cseq **end do;**
| **return** E; | **return**; | **error;** | **error** I,Eseq;
| **try** Cseq Catch **end;** | **try** Cseq Catch **finally** Cseq **end;**

| I,Iseq := E,Eseq; | E(Eseq); | **‘type/I‘** := T; | **print**(e);
Elif ::= EMPTY | **elif** E **then** Cseq;Elif
Catch ::= EMPTY | **catch “I” :**Cseq, Catch
Eseq ::= EMPTY | E,Eseq
E ::= I | N | **module()** S;R **end module**;
| **proc**(Pseq) S;R **end proc**;| **proc**(Pseq)::T; S;R **end proc**;
| E1 Bop E2 | Uop E | Esop | E1 **and** E2 | E1 **or** E2 | E(Eseq)
| I1:-I2 | E,E,Eseq | **type(** I,T **)** | E1 = E2 | E1 <> E2
S ::= EMPTY | **local** It,Itseq;S | **global** I,Iseq;S | **uses** I,Iseq;S
| **export** It,Itseq;S
R ::= Cseq | Cseq;E
Pseq ::= EMPTY | P,Pseq
P ::= I | I :: M
M ::= **seq(** T **)** | T
Iseq :: = EMPTY | I, Iseq
I ::= any valid Maple name
Itseq :: = EMPTY | It, Itseq
It ::= I | I :: T | I := E | I::T:=E
Bop ::= + | − | / | ∗ | **mod** |<|>|≤|≥
Uop ::= **not** | − | +
Esop ::= **op(** E1, E2 **)** | **op(** E **)** | **op(** E..E, E **)** | **nops(** E **)**
| **subsop(** E1=E2, E3 **)** | **subs(** I=E1, E2 **)** | “ E “ | [Eseq]
| I[Eseq] | **seq(**E, I = E..E **)** | **seq(**E, I **in** E**)** | **eval(** I,1 **)**
Tseq ::= EMPTY | T,Tseq
T ::= **integer** | **boolean** | **string** | **float** | **rational** | **anything** | **{** T **}**
| **list(** T **)** | [Tseq] | **procedure**[T](Tseq)
| I(Tseq) | **Or(** Tseq **)** | **symbol** | **void** | **uneval** | I
N ::= a sequence of decimal digits

B. Syntax of the Specification Language for MiniMaple

In this appendix we give the formal abstract syntax (language grammar) of a specification language for *MiniMaple*.

decl ∈ Declaration
proc-spec ∈ Procedure_Specification
loop-spec ∈ Loop_Specification
asrt ∈ Assertion
rules ∈ Rules
excep-clause ∈ Exception_Clause
eseq ∈ Specification_Expression_Sequence
spec-expr ∈ Specification_Expression
binding ∈ Binding
Itseq ∈ Identifier_Typed_Sequence
It ∈ Identifier_Typed
Iseq ∈ Identifier_Sequence
I ∈ Identifier
Bop ∈ Binary_Operator
Uop ∈ Unary_Operator
it-op ∈ Iteration_Operator
esop ∈ Especial_Operator
sel-op ∈ Selection_Operator
Tseq ∈ Type_Sequence
T ∈ Type
N ∈ Numeral

decl ::= EMPTY | (**define**(I(Itseq)::T,rules);
| **‘type/**I‘;
| **‘type/**I‘:=T;
| **assume**(spec-expr);) decl
proc-spec ::= **requires** spec-expr; **global** Iseq; **ensures** spec-expr; excep-clause
loop-spec := **invariant** spec-expr; **decreases** spec-expr;
asrt := **ASSERT**(spec-expr, (EMPTY | “I“));
rules ::= EMPTY | I(Itseq) = spec-expr, rules

excep-clause ::= EMPTY | **exceptions** "I" spec-expr; excep-clause
eseq ::= EMPTY | spec-expr, eseq
spec-expr ::= I **(** eseq **)** | **type**(spec-expr,T)
| spec-expr **and** spec-expr | spec-expr **or** spec-expr
| spec-expr **equivalent** spec-expr | spec-expr **implies** spec-expr
| **forall**(Itseq, spec-expr) | **exists**(Itseq, spec-expr)
| **(**spec-expr**)** | spec-expr Bop spec-expr | Uop spec-expr | esop
| it-op(spec-expr, binding, (EMPTY | spec-expr))
| **true** | **false** | **LET** Iseq=eseq **IN** spec-expr | **RESULT**
| **'if'**(spec-expr1, spec-expr2, spec-expr3) | I | I1:-I2 | **OLD** I | N
| spec-expr1 = spec-expr2 | spec-expr1 <> spec-expr2
binding ::= I = spec-expr1...spec-expr2 | I **in** spec-expr
Itseq ::= EMPTY | It, Itseq
It ::= I::T
Iseq :: = EMPTY | I, Iseq
I ::= any valid Maple name
Bop ::= + | − | / | ∗ | **mod** |<|>|≤|≥|=|<>
Uop ::= **not** | − | +
it-op ::= **add** | **mul** | **max** | **min** | **seq**
esop ::= **op(** spec-expr1, spec-expr2 **)** | **op(** spec-expr **)**
| **op(** spec-expr..spec-expr, spec-expr **)** | **nops(** spec-expr **)**
| **subsop(** spec-expr1=spec-expr2, spec-expr3 **)**
| **subs(** I=spec-expr1, spec-expr2 **)** | **‘** spec-expr **‘**
| I sel-op | [eseq] | { eseq } | I(eseq) | **eval(** I,1 **)**
sel-op ::= EMPTY | **[** eseq **]** sel-op
Tseq ::= EMPTY | T,Tseq
T ::= **integer** | **boolean** | **string** | **float** | **rational** | **anything** | **{** T **}**
| **list(** T **)** | [Tseq] | **procedure**[T](Tseq)
| I(Tseq) | **Or(** Tseq **)** | **symbol** | **void** | **unevaluated** | I
N ::= a sequence of decimal digits

C. Type System of MiniMaple

In this appendix we give the logical rules to derive the typing judgments. We also give the auxiliary functions and predicates that are used in the rules. The contents of the following sections are not shown in this printout but in the supplementary electronic version of this thesis.

C.1. Logical Rules

In this section, we list the logical rules for each phrase/alternative of syntactic domain, which are used to derive typing judgments. The rules state the conditions under which the syntactic phrases are well typed.

C.2. Auxiliary Functions

In this section we define the auxiliary functions used in logical rules to derive typing judgments. The auxiliary functions are defined over type environments and the syntactic domains “type”, “identifier”, “typed identifier”, “parameter” and “expression”. Some additional utility functions are defined over return flag and the sequences of various syntactic domains.

C.3. Auxiliary Predicates

In this section we give the auxiliary predicates used in logical rules to derive typing judgments. The auxiliary predicates are defined over type environments and the syntactic domains “type”, “identifier”, “typed identifier” and “parameter”.

D. Formal Semantics of MiniMaple

In this appendix, we give the formalization and definitions of the denotational semantics of *MiniMaple*. The contents of the following sections are not shown in this printout but in the supplementary electronic version of this thesis.

D.1. Semantic Algebras

In this section, we formalize the semantic domains of values and their corresponding operations.

D.2. Signatures of Valuation Functions

In this section, we give the signatures of the valuation functions for the semantics of each *MiniMaple* syntactic domain.

D.3. Auxiliary Functions and Predicates

In this section, we define auxiliary functions and predicates that are later used in the definition of the semantic functions.

D.4. Semantics

In this section, we give the definition of the valuation functions for *MiniMaple* syntactic domains.

E. Formal Semantics of the Specification Language for MiniMaple

In this appendix, we give the formalization and definitions of the denotational semantics of the core formula language and the annotations of the specification language for *MiniMaple*. The contents of the following sections are not shown in this printout but in the supplementary electronic version of this thesis.

E.1. Semantic Algebras

In this section, we formalize the semantic domains of values and their corresponding operations.

E.2. Signatures of Valuation Functions of Formula Language

In this section, we give the signatures of the valuation functions for the semantics of the syntactic domains of the formula language.

E.3. Auxiliary Functions and Predicates

In this section, we declare and define the auxiliary functions and predicates that are later used in the semantic functions definitions.

E.4. Semantics of Formula Language

In this section, we define the semantic functions of the formula language of the specification language.

E.5. Signatures of Valuation Functions for Specification Annotations

In this section, we give the signatures of the valuation functions for the semantics of the syntactic domains of the elements of the specification language.

E.6. Semantics of Specification Annotations

In this section, we define the semantic functions of the elements of the specification language.

F. Translation of MiniMaple into Why3ML

In this appendix, we give the formalization and definitions of the translation of *MiniMaple* and its specification language into Why3ML constructs. The contents of the following sections are not shown in this printout but in the supplementary electronic version of this thesis.

F.1. Semantic Algebras

In this section, we formalize the semantic domains of *MiniMaple* and Why3ML and also declare/define their corresponding operations.

F.2. Signatures of Translation Functions

In this section, we give the signatures of translation functions for the syntactic domains of *MiniMaple* and its specification language.

F.3. Auxiliary Functions and Predicates

In this section, we declare/define the auxiliary functions and predicates which are later used in the definition of the translation functions.

F.4. Definition of Translation Functions

In this section, we define the translation functions.

G. Proof of the Soundness of the Translation

In this appendix, we formulate the proof-settings and then discuss the proof of the soundness of the translation for selected constructs of *MiniMaple*. The contents of the following sections are not shown in this printout but in the supplementary electronic version of this thesis.

G.1. Semantic Algebras

In this section, we formalize the semantic domains of values of *MiniMaple* and Why3ML and also define their corresponding operations.

G.2. Auxiliary Functions and Predicates

In this section, we declare/define the auxiliary functions and predicates which are used in the proof.

G.3. Soundness Statements

In this section, we formulate the soundness statements for the syntactic domains of command sequences, commands, expressions and identifiers of *MiniMaple*.

G.4. Proof

In this section, we give the proof of the soundness of command sequences, assignment, conditional and while-loop commands respectively.

G.5. Lemmas

In this section, we formulate and discuss the lemmas used in the proof.

G.6. Definitions

In this section, we give the various definitions of the Why3ML and *MiniMaple* constructs.

G.7. Why3 Semantics

In this section, we formalize the definitions of Why3ML operational semantics as given in [63] in order to make this document standalone.

G.8. Derivations

In this section, we give the derivation rules for the operational semantics of Why3 while-loop. These derivations are already used in the proof of the soundness of while-loop command.

Bibliography

[1] Abramsky, Samson and Jung, Achim. *Domain Theory*, volume 3. Oxford University Press, Oxford, UK, 1994.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge, UK, 1996.

[3] A. A. Adams, H. Gottliebsen, S. A. Linton, and U. Martin. Automated Theorem Proving in Support of Computer Algebra: Symbolic Definite Integration as a Case Study. In *ISSAC '99: International Symposium on Symbolic and Algebraic Computation*, pages 253–260, Vancouver, British Columbia, Canada, 1999. ACM Press, New York.

[4] A. A. Adams, Hanne Gottliebsen, Steve Linton, and Ursula Martin. VSDITLU: A Verifiable Symbolic Definite Integral Table Look-Up. In Harald Ganzinger, editor, *CADE-16, 16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Computer Science*, pages 112–126, Trento, Italy, July 7–10, 1999. Springer.

[5] Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer Algebra Meets Automated Theorem Proving: Integrating Maple and PVS. In Richard J. Boulton and Paul B. Jackson, editors, *TPHOLs 2001: 14th International Conference on Theorem Proving in Higher Order Logics*, volume 2152 of *Lecture Notes in Computer Science*, pages 27–42, Edinburgh, Scotland, UK, September 3–6, 2001. Springer.

[6] Albert Einstein. *The Meaning of Relativity*. Methuen and Co. Ltd., London, UK, fourth edition, 1950.

[7] Aldor. `http://www.aldor.org/`.

[8] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards Type Inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP 2005)*, LNCS 3586, pages 428–453. Springer, 2005.

[9] Anthony Narkawicz and César Muñoz and Gilles Dowek. Formal Verification of Air Traffic Prevention Bands Algorithms. Technical Memorandum NASA/TM-2010-216706, NASA, Langley Research Center, Hampton VA 23681-2199, USA, June 2010.

[10] Anthony Narkawicz and César Muñoz and Jeffrey Maddalon. A Mathematical Analysis of Air Traffic Priority Rules. In *Proceedings of the 12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference, AIAA-2012-5544*,

Indianapolis, Indiana, USA, September 2012.

[11] ANTLR v3. `http://www.antlr.org/`.

[12] Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and Algorithms: an Interface between Isabelle and Maple. In *ISSAC '95: International Symposium on Symbolic and Algebraic Computation*, pages 150–157, Montreal, Quebec, Canada, July 10–12, 1995. ACM Press, New York.

[13] Mike Barnett, Boryuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[14] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *CASSIS'2004: International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, Marseille, France, March 10–13, 2004. Springer, Berlin.

[15] Clark Barrett and Sergey Berezin. CVC Lite: A New Implementation of the Cooperating Validity Checker. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13–17, 2004*, volume 3114 of *LNCS*, pages 515–518. Springer, 2004.

[16] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, and Simão Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In David Sands, editor, *Programming Languages and Systems*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer Berlin Heidelberg, 2001.

[17] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.

[18] Andrej Bauer, Edmund Clarke, and Xudong Zhao. Analytica — An Experiment in Combining Theorem Proving and Symbolic Computation. *Journal of Automated Reasoning*, 21(3):295–325, 1998.

[19] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.

[20] Bertot, Yves and Castéran, Pierre and Huet, Gérard (informaticien) and Paulin-Mohring, Christine. *Interactive Theorem Proving and Program Development : Coq'Art : The Calculus of Inductive Constructions*. Texts in theoretical computer science. Springer, Berlin, New York, 2004.

[21] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wrocław, Poland, August 2011.

[22] Sascha Böhme, Michal Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie – An Interactive Prover-Backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 44(1-2):111–144, 2010.

[23] Egon Börger. High Level System Design and Analysis Using Abstract State Machines. In *Proceedings of the International Workshop on Current Trends in Applied Formal Method: Applied Formal Methods*, FM-Trends 98, pages 1–43, London, UK, 1999. Springer-Verlag.

[24] Egon Börger and Wolfram Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In *Formal Syntax and Semantics of Java*, pages 353–404, London, UK, 1999. Springer-Verlag.

[25] Wieb Bosma, John Cannon, and Graham Matthews. Programming with Algebraic Structures: Design of the MAGMA Language. In *ISSAC '94: International Symposium on Symbolic and Algebraic Computation*, pages 52–57, Oxford, UK, July 20–22, 1994. ACM Press, NY.

[26] Wieb Bosma, John Cannon, and Catherine Playoust. The Magma Algebra System I: The User Language. *Journal of Symbolic Computation*, 24(3–4):235 – 265, 1997.

[27] D Bruns. *Formal Semantics for the Java Modeling Language*. PhD thesis, Universität Karlsruhe, Germany, 2009.

[28] B. Buchberger. Theorema: A Proving System Based on Mathematica. *The Mathematica Journal*, 8(2):247–252, 2001.

[29] Bruno Buchberger, Adrian Craciun, Tudor Jebelean, et al. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[30] Bruno Buchberger, Tudor Jebelean, et al. A Survey of the Theorema Project. In Wolfgang Küchlin, editor, *ISSAC'97 International Symposium on Symbolic and Algebraic Computat ion*, pages 384–391, Maui, Hawaii, July 21–23, 1997. ACM Press, New York.

[31] Bundgaard, Jørgen and Schultz, Lennart. A Denotational (Static) Semantics Method for Defining Ada Context Conditions. In *Towards a Formal Description of Ada*, pages 21–212, London, UK, UK, 1980. Springer-Verlag.

[32] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[33] C. Marché and C. Paulin-Mohring and X. Urbain. The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs Annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1–2):89 – 106, 2004.

[34] Luca Cardelli. Type Systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[35] J. Carette and S. Forrest. Property Inference for Maple: an Application of

Abstract Interpretation. In *Calculemus*, pages 5–19, 2007.

[36] Jacques Carette and Stephen Forrest. Mining Maple Code for Contracts. In Silvio Ranise and Anna Bigatti, editors, *Calculemus*, ENTCS. Elsevier, 2006.

[37] Jacques Carette and Michael Kucera. Partial Evaluation of Maple. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '07, pages 41–50. ACM Press, 2007.

[38] Gareth Carter, Rosemary Monahan, and Joseph M. Morris. Software Refinement with Perfect Developer. In *SEFM'05: Third IEEE International Conference on Software Engineerin g and Formal Methods*, pages 363–373, Koblenz, Germany, September 5–9, 2005. IEEE Computer Society.

[39] Kalisyk Cezary and Freek Wiedijk. Certified Computer Algebra on Top of an Interactive Theorem Prover. In *Calculemus 2007 — 14th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, LNAI, Hagenberg, Austria, June 27–30, 2007. Springer.

[40] Charles Wallace. The Semantics of the C++ Programming Language. In *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1993.

[41] Yoonsik Cheon. Inheritance in Larch Interface Specification Languages, its Semantic Foundation and Formal Semantics. In J. Grundy, M. Schwenke, and T. Vickers, editors, *Proceedings of International Refinement Workshop & Formal Methods Pacific (IRW/FMP) '98*, pages 81–99, Canberra, Australia, 1998. Springer-Verlag.

[42] Christian Dönch. Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University (JKU), Linz, 2009.

[43] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editor, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[44] E. M. Clarke, A. S. Gavlovski, K. Sutner, and W. Windsteiger. Analytica V: Towards the Mordell-Weil Theorem. In A. Bigatti and S. Ranise, editors, *Calculemus'06, 13th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, Genova, Italy, July 2–6, 2006.

[45] Sylvain Conchon. *SMT Techniques and their Applications: from Alt-Ergo to Cubicle*. Thèse d'habilitation, Université Paris-Sud, December 2012. In English, `http://www.lri.fr/~conchon/publis/conchonHDR.pdf`.

[46] Cuoq, Pascal and Kirchner, Florent and Kosmatov, Nikolai and Prevosto, Virgile and Signoles, Julien and Yakobowski, Boris. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[47] D. Bjørner and C.B. Jones. Algol 60*, Formal Specification and Software Development*, volume Chapter 6. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[48] D. Bjørner and C.B. Jones. Pascal*, Formal Specification and Software Development*, volume Chapter 7. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[49] D. Bjørner, C.B. Jones editors. *Formal Specification & Software Development*. Prentice-Hall, 1982.

[50] Schmidt David A. *The Structure of Typed Programming Languages*. MIT Press, Cambridge, MA, USA, 1994.

[51] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] Decker, Wolfram and Greuel, Gert-Martin and Pfister, Gerhard and Schönemann, Hans. SINGULAR 3-1-6 — A Computer Algebra System for Polynomial Computations. `http://www.singular.uni-kl.de`, 2012.

[53] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A Theorem Prover for Program Checking. *J. ACM*, 52(3):365–473, May 2005.

[54] Andreas Dolzmann and Thomas Sturn. REDLOG: Computer Algebra Meets Computer Logic. *SIGSAM Bulletin*, 31(2):2–9, 1997.

[55] Dubois, Catherine and Hardin, Thérése and Donzeau-Gouge, Véronique. Building Certified Components within FOCAL. In Loidl, Hans-Wolfgang, editor, *Trends in Functional Programming*, volume 5 of *Trends in Functional Programming*, pages 33–48. Intellect, 2004.

[56] Martin Dunstan, Tom Kelsey, Steve Linton, and Ursula Martin. Lightweight Formal Methods for Computer Algebra Systems. In Oliver Gloor, editor, *ISSAC 1998: International Symposium on Symbolic and Algebraic Computation*, pages 80–87, Rostock, Germany, August 13–15, 1998. ACM Press.

[57] Martin Dunstan, Tom Kelsey, Ursula Martin, and Steve Linton. Formal Methods for Extensions to CAS. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 - World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 1758–1777, Toulouse, France, September 20-24, 1999. Springer.

[58] Egon Börger and Dean Rosenzweig. A Mathematical Definition of Full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

[59] Erik Poll and Simon Thompson. The Type System of Aldor. Technical Report 11-99, Computing Laboratory, University of Kent at Canterbury, Kent CT2 7NF, UK, July 1999.

[60] Fantechi, Alessandro and Fokkink, Wan and Morzenti, Angelo. *Some Trends in Formal Methods Applications to Railway Signaling*, pages 61–84. John Wiley &

Sons, Inc., 2012.

[61] Stéphane Fechter. An Object-Oriented Model for the Certified Computer Algebra Library. In *FMOODS 2002, Formal Methods for Open Object-Based Distributed Systems, PhD workshop*, Twente, The Netherlands, March 20–22, 2002.

[62] Fechter, Stéphane. *Sémantique des traits orientés objet de Focal*. PhD thesis, Université Pierre et Marie Curie (UPMC), 2005. Type : Thèse de Doctorat – Soutenue le : 2005-07-18 – Dirigée par : Hardin, Thérèse – Encadrée par : DUBOIS Catherine.

[63] Jean-Christophe Filliâtre. Why: an Intermediate Language for Program Verification. A Tutorial Lecture at Summer School, 2007. `https://www.lri.fr/~filliatr/types-summer-school-2007/notes.pdf`.

[64] Jean-Christophe Filliâtre. Deductive Software Verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.

[65] Jean-Christophe Filliâtre. Verifying Two Lines of C with Why3: an Exercise in Program Verification. In *Verified Software: Theories, Tools and Experiments (VSTTE)*, Philadelphia, USA, January 2012.

[66] Jean-Christophe Filliâtre. One Logic to Use Them All. In Bonacina, Maria Paola, editor, *Automated Deduction – CADE-24*, volume 7898 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2013.

[67] Robert W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19 of *Mathematical Aspects of Computer Science*, pages 19–31, Providence, 1967. American Mathematical Society.

[68] FriCAS. `http://fricas.sourceforge.net/`.

[69] Peter Fritzson. Static and Strong Typing for Extended Mathematica. In *Innovation in Mathematics: Proceedings of the Second International Mathematica Symposium*, IMS-97, pages 153–160, 1997.

[70] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static Type Inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1859–1866, New York, NY, USA, 2009. ACM.

[71] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. A Tutorial, 2006. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`.

[72] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[73] Hanne Gottliebsen, Tom Kelsey, and Ursula Martin. Hidden Verification for Computational Mathematics. *Journal of Symbolic Computation*, 39(5):539–567, 2005.

[74] Benjamin Grégoire and Assia Mahboubi. Proving Equalities in a Commutative Ring Done Right in Coq. In Joe Hurd and Thomas F. Melham, editors, *TPHOls*

2005, Theorem Proving in Higher Order Logics, 18th International Conference, volume 3603 of *Lecture Notes in Computer Science*, Oxford, UK, August 22–25, 2005. Springer.

[75] Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Hybrid Checking for Flexible Specifications. In R. Findler, editor, *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[76] J. V. Guttag, J. J. Horning, Withs. J. Garl, K. D. Jones, A. Modet, and J. M. Wing. Larch: Languages and Tools for Formal Specification. In *Texts and Monographs in Computer Science*. Springer-Verlag, 1993.

[77] John Harrison and Laurent Théry. Extending the HOL Theorem Prover with a Computer Algebra System to Reason about the Reals. In Jeffrey J. Joyce and Carl Seger, editors, *1993 International Workshop on the HOL Theorem Proving System and its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 174–184, Vancouver, Canada, August 1993. Springer.

[78] John Harrison and Laurent Théry. Reasoning About the Reals: the Marriage of HOL and Maple. In Andrei Voronkov, editor, *LPAR '93: 4th International Conference on Logic Programming and Automated Reasoning*, volume 698 of *Lecture Notes in Computer Science*, St. Petersburg, Russia, July 13–20, 1993. Springer.

[79] Phillip Heidegger and Peter Thiemann. Recency Types for Analyzing Scripting Languages. In Theo D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 200–224. Springer Berlin Heidelberg, 2010.

[80] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communication of ACM*, 12(10):576–580, October 1969.

[81] C.A.R. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4):271–281, 1972.

[82] Hudak, Paul. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, June 2000.

[83] Andrew Ireland, Bill J. Ellis, et al. An Integrated Approach to High Integrity Software Verification. *Journal of Automated Reasoning*, 36(4):379–410, 2006.

[84] Isabelle Attali and Denis Caromel and Marjorie Russo. A Formal and Executable Semantics for Java. Technical Report , Proceedings of Formal Underpinnings of Java, an OOPSLA'98 Workshop, Vancouver, CA, 1998.

[85] Daniel Jackson. *Software Abstractions — Logic, Language, and Analysis*. MIT Press, Cambridge, MA, 2006.

[86] John Rushby. Formal Methods and the Certification of Critical Systems. Technical Report SRI-CSL-93-7, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Also issued under the title *Formal Methods and Digital Systems Validation for Airborne Systems* as NASA Contractor

Report 4551, December 1993.

[87] John Rushby. New Challenges In Certification For Aircraft Software. In Sanjoy Baruah and Sebastian Fischmeister, editors, *Proceedings of the Ninth ACM International Conference On Embedded Software: EMSOFT*, pages 211–218, Taipei, Taiwan, 2011. ACM.

[88] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, Upper Saddle River, NJ, 1990.

[89] Tom Kelsey. *Formal Methods and Computer Algebra: A Larch Specification of AXIOM Categories and Functors*. PhD thesis, School of Mathematical and Computational Sciences, University of St Andrews, December 1999.

[90] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. SAGE: Unified Hybrid Checking for Firstclass Types, General Refinement Types, and Dynamics (extended report). Technical report, University of California, Santa Cruz, USA, `http://sage.soe.ucsc.edu/sage-tr.pdf`, 2006.

[91] Muhammad Taimoor Khan. A Type Checker for *MiniMaple*. RISC Technical Report 11-05, also DK Technical Report 2011-05, Research Institute for Symbolic Computation, Linz, 2011.

[92] Muhammad Taimoor Khan. Towards a Behavioral Analysis of Computer Algebra Programs. RISC Technical Report 11-13, also DK Technical Report 2011-13, Research Institute for Symbolic Computation, University of Linz, 2011.

[93] Muhammad Taimoor Khan. Formal Semantics of a Specification Language for *MiniMaple*. DK Technical Report 2012-06, Research Institute for Symbolic Computation, University of Linz, April 2012.

[94] Muhammad Taimoor Khan. Formal Semantics of *MiniMaple*. DK Technical Report 2012-01, Research Institute for Symbolic Computation, University of Linz, January 2012.

[95] Muhammad Taimoor Khan. On the Formal Semantics of *MiniMaple* and its Specification Language. In *Proceedings of Frontiers of Information Technology*, pages 169–174. IEEE Computer Society, 2012.

[96] Muhammad Taimoor Khan. On the Formal Verification of Maple Programs. DK Technical Report 2013-06, Doktoratskolleg, Linz, July 2013.

[97] Muhammad Taimoor Khan. Translation of *MiniMaple* to Why3ML. DK Technical Report 2013-02, Doktoratskolleg, Linz, February 2013.

[98] Muhammad Taimoor Khan. On the Soundness of the Translation of *MiniMaple* to Why3ML. DK Technical Report 2014-03, Research Institute for Symbolic Computation, University of Linz, February 2014.

[99] Muhammad Taimoor Khan and Wolfgang Schreiner. Towards a Behavioral Analysis of Computer Algebra Programs (Extended Abstract). In Paul Pettersson and Cristina Seceleanu, editors, *Proceedings of the 23rd Nordic Workshop*

on Programming Theory (NWPT'11), pages 42–44, Vasteras, Sweden, October 2011.

[100] Muhammad Taimoor Khan and Wolfgang Schreiner. On the Formal Specification of Maple Programs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, volume 7362 of *LNCS*, pages 443–447. Springer, 2012.

[101] Muhammad Taimoor Khan and Wolfgang Schreiner. Towards the Formal Specification and Verification of Maple Programs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics*, volume 7362 of *LNCS*, pages 231–247. Springer, 2012.

[102] Muhammad Taimoor Khan and Wolfgang Schreiner. A Verification Framework for Minimaple Programs (Extended Abstract). *ACM Communication in Computer Algebra*, 47(3/4):98–99, January 2014.

[103] Anneke Kleppe and Jos Warmer. An Introduction to the Object Constraint Language (OCL). In *TOOLS 2000: 33rd International Conference on Technology of Object-Oriented Languages and Systems*, page 456, St. Malo, France, June 5–8, 2000. IEEE Computer Society.

[104] Christoph Koutschan. Holonomic Functions (User's Guide). RISC Report Series 10-01, Research Institute for Symbolic Computation, University of Linz, January 2010.

[105] Gary T. Leavens and Yoonsik Cheon. Preliminary Design of Larch/C++. In U. Martin and J. Wing, editors, *First First International Workshop on Larch*, Workshops in Computing Science, pages 159–184, Deadham, MA, July 13–15, 1992. Springer, Berlin.

[106] Sergio Maffeis, JohnC. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 307–325. Springer Berlin Heidelberg, 2008.

[107] Dan Maharry. *TypeScript Revealed*. Apress, Berkely, CA, USA, 1st edition, 2013.

[108] Assia Mahboubi. Programming and Certifying a CAD Algorithm in the Coq System. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, number 05021 in Dagstuhl Seminar Proceedings. IBFI, Germany, 2005.

[109] Assia Mahboubi. Proving Formally the Implementation of an Efficient gcd Algorithm for Polynomials. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR 2006, Third International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 438–452, Seattle,

WA, USA, August 17–20, 2006. Springer.

[110] Assia Mahboubi. Implementing the Cylindrical Algebraic Decomposition within the Coq System. *Mathematical Structures in Computer Science*, 17(1):99–127, 2007.

[111] MapleSoft. `http://www.maplesoft.com/`.

[112] Marc Vale. The Evolving Algebra Semantics of COBOL – Part 1: Programs and Control. Technical Report CSE-TR-162-93, University of Michigan, EECS Department, Ann Arbor, MI, 1993.

[113] Maxima, A Computer Algebra System. `http://maxima.sourceforge.net/`, 2011.

[114] Meertens, L. On Static Scope Checking in ALGOL 68. *ALGOL Bulletin*, pages 45–58, March 1973.

[115] Mike Spivey. Towards a Formal Semantics for the Z Notation. Technical Report PRG41, OUCL, October 1984.

[116] Michael B. Monagan. Gauss: A Parameterized Domain of Computation System with Support for Signature Functions. In *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, DISCO '93, pages 81–94. Springer-Verlag, 1993.

[117] Peter D. Mosses. Modular Structural Operational Semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.

[118] Peter D. Mosses. Formal Semantics of Programming Languages: – An Overview. *Electronic Notes in Theoretical Computer Science*, 148(1):41 – 73, 2006. Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004) Foundations of Visual Modelling Techniques 2004.

[119] Robert Muller. M-LISP: A Representation-independent Dialect of LISP with Reduction Semantics. *ACM Transactions on Programming Languages and Systems*, 14(4):589–616, October 1992.

[120] OpenAXIOM. `http://www.open-axiom.org/`.

[121] William S. Page. AXIOM: Open Source Computer Algebra System. *ACM Communincations in Computer Algebra*, 41(3):114–114, September 2007.

[122] Papaspyrou, Nikolaos S. Denotational Sematics of ANSI C. *Computer Standards and Interfaces*, 23(3):169–185, July 2001.

[123] PARI Library. `http://pari.math.u-bordeaux.fr`.

[124] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[125] Henrik Persson. Certified Computer Algebra. In *Types summer school'99: Theory and practice of formal proofs*, Giens, France, August 30 – September 10, 1999. INRIA.

[126] Peter Gorm Larsen and Michael Meincke Arentoft and Brian Q. Monahan and Stephen Bear. Towards A Formal Semantics Of The BSI/VDM Specification Language. In *In Information Processing 89, North-Holland*, pages 95–100. IFIP, North-Holland, 1989.

[127] Plotkin, G. D. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000, Aarhus C, Denmark, 1981.

[128] Erik Poll and Simon Thompson. Adding the Axioms to AXIOM: Towards a System of Automated Reasoning in Aldor. In *Calculemus and Types '98*, Eindhoven, The Netherlands, July 13–15, 1998.

[129] Erik Poll and Simon Thompson. Integrating Computer Algebra and Reasoning through the Type System of Aldor. In Helene Kirchner and Christophe Ringeissen, editors, *Frocos 2000, Frontiers of Combining Systems*, volume 1794 of *Lecture Notes in Computer Science*, pages 136–150, Nancy, France, March 22–24, March 2000. Springer.

[130] Virgile Prevosto. Certified Mathematical Hierarchies: The FoCal System. In Thierry Coquand, Henri Lombardi, and Marie-Françoise Roy, editors, *Mathematics, Algorithms, Proofs*, volume 05021 of *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl, Germany, January 9–14, 2005. IBFI, Schloss Dagstuhl, Germany.

[131] Prevosto, Virgile and Boulmé, Sylvain. Proof Contexts with Late Binding. In Urzyczyn, Pawel, editor, *Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, pages 324–338. Springer Berlin Heidelberg, 2005.

[132] R. Milner, M. Tofte, R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, 1990.

[133] R.D. Tennent. A Denotational Definition of the Programming Language Pascal. Technical Report , Programming Research Group, Oxford University, 1978.

[134] REDUCE. `http://reduce-algebra.sourceforge.net/`.

[135] Rees, J and Clinger, W. Revised Report on the Algorithmic Language Scheme. *SIGPLAN Not.*, 21(12):37–79, December 1986.

[136] Schmidt, David A. *Denotational Semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[137] Wolfgang Schreiner. A Program Calculus. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, Austria, September 2008.

[138] Wolfgang Schreiner. Computer-Assisted Program Reasoning Based on a Relational Semantics of Programs. In Pedro Quaresma and Ralph-Johan Back, editors, *Proceedings First Workshop on CTP Components for Educational Software (THedu'11)*, number 79 in Electronic Proceedings in Theoretical Computer

Science (EPTCS), pages 124–142, Wroclaw, Poland, July 31, 2011, February 2012.

[139] Shoup's Library. `http://www.shoup.net/ntl/`.

[140] SMT-LIB — The Satisfiability Modulo Theories Library, 2006. University of Iowa, Iowa City, IA, `http://combination.cs.uiowa.edu/smtlib`.

[141] Volker Sorge. Non-Trivial Symbolic Computations in Proof Planning. In Hélène Kirchner and Christophe Ringeissen, editors, *FroCoS 2000: Third International Workshop on Frontiers of Combining Systems*, volume 1794 of *Lecture Notes in Computer Science*, pages 121–135, Nancy, France, March 22-24, 2000, 2000. Springer.

[142] SPAD. `http://www.euclideanspace.com/maths/standards/program/spad/`.

[143] Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

[144] Sylvain Boulmé. *Spécification d'un environnement dédié á la programmation certifiée de bibliothéques de calcul formel*. PhD thesis, Université Paris 6, December 2000.

[145] Sylvain Boulmé and Thérèse Hardin and Daniel Hirschkoff and Valérie Ménissier-Morain and Renaud Rioboo. On the Way to Certify Computer Algebra Systems. In *Proceedings of the Calculemus workshop of FLOC'99 (Federated Logic Conference, Trento, Italy)*, volume 23(3) of *ENTCS*, pages 370–385. Elsevier, 1999.

[146] Gregory Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical Report 02-3, National Institute of Standards and Technology, USA, May 2002.

[147] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.7.4*, 2014.

[148] Overview of Theorema, 2006. Research Institute for Symbolic Compuation (RISC), Johannes Kepler Unive rsity, Linz, Austria, `http://www.risc.uni-linz.ac.at/research/theorema`.

[149] Laurent Théry. A Certified Version of Buchberger's Algorithm. In *CADE-15: 15th International Conference on Automated Deduction*, number 1421 in LNAI, pages 349–364, Lindau, Germany, July 5–10, 1998. Springer-Verlag.

[150] Peter Thiemann. Towards a Type System for Analyzing Javascript Programs. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, pages 408–422, Berlin, Heidelberg, 2005. Springer-Verlag.

[151] Simon Thompson. Logic and Dependent Types in the Aldor Computer Algebra System. In Manfred Kerber and Michael Kohlhase, editors, *Calculemus 2000: Symbolic Computation and Automated Reasoning*, pages 205–219, St. Andrews, Scotland, August 6–7, 2000. A. K. Peters, Natick, MA.

[152] Simon Thompson, John Shackell, James Beaumont, and Leonid Timochouk.

Atypical: Integrating Computer Algebra and Reasoning, 2003. `http://www.cs.kent.ac.uk/people/staff/sjt/Atypical`.

[153] Tim Lambert and Peter Lindsay and Ken Robinson. Using Miranda as a First Programming Language. *Journal of Functional Programming*, 3(1):5–34, 1993.

[154] Virgile Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés*. PhD thesis, Université Paris 6, Thése de doctorat, 2003.

[155] Frèdèric Vogels, Bart Jacobs, and Frank Piessens. A Machine Checked Soundness Proof for an Intermediate Verification Language. In Mogens Nielsen, Antonìn Kuĉera, PeterBro Miltersen, Catuscia Palamidessi, Petr Tùma, and Frank Valencia, editors, *SOFSEM 2009: Theory and Practice of Computer Science*, volume 5404 of *Lecture Notes in Computer Science*, pages 570–581. Springer Berlin Heidelberg, 2009.

[156] William Stein and David Joyner. SAGE: System for Algebra and Geometry Experimentation. *ACM SIGSAM Bulletin*, 39(2):61–64, 2005.

[157] Wolfgang Windsteiger. A Set Theory Prover in Theorema: Implementation and Practical Applications. PhD Thesis, May 2001.

[158] Winskel, Glynn. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.

[159] Wolfram, Stephen. *The Mathematica Book*. Wolfram Media, Incorporated, 5 edition, 2003.

[160] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

[161] Tian Zhao. Polymorphic Type Inference for Scripting Languages with Object Extensions. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 37–50, New York, NY, USA, 2011. ACM.

[162] M. Zhou and F. Winkler. Groebner bases in Difference-Differential Modules and Difference-Differential Dimension Polynomials. *Science in China Series A: Mathematics*, 51(9):1732–1752, 2008.

[163] Meng Zhou and Franz Winkler. Computing Difference-Differential Dimension Polynomials by Relative Gröbner Bases in Difference-Differential Modules. *Journal of Symbolic Computation*, 43(10):726–745, October 2008.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, April 2014 Muhammad Taimoor Khan

Curriculum Vitae

Contact Information

Research Institute for Symbolic Computation (RISC)	Office/Cell: +43 732 2468 9927/+43 680 237 9670
Altenbergerstraße 69	E-mail: muhammad.khan@dk-compmath.jku.at
A-4040 Linz, Austria	WWW: http://www.risc.jku.at/people/mtkhan/

Personal Data

Name:	Muhammad Taimoor Khan
Date of Birth:	5^{th} of April 1978
Citizenship:	Pakistani

Education

2009 – 2014	Doctoral Studies in "Doctoral Program: Computational Mathematics" (DK) Johannes Kepler University (JKU), Linz, Austria *Thesis:* Formal Specification and Verification of Computer Algebra Software *Supervisor:* Prof. Wolfgang Schreiner
2007 – 2008	M.Sc. in Advanced Distributed Systems (Distinction) University of Leicester, UK *Project:* Space Link Extension Service Management. *Supervisor:* Prof. Reiko Heckel
1998 – 2000	M.Sc. in Computer Science (First Class) The University of Bahawalpur, Pakistan *Project:* WAP-based trading system. *Supervisor:* Prof. Waqar Aslam

Publications

Refereed Papers

- Muhammad Taimoor Khan, Wolfgang Schreiner. *A Verification Framework for MiniMaple Programs (Extended Abstract*. In: ACM Communications in Computer Algebra, 47(3):98–99, ACM, September 2013, 38$^{\text{th}}$ International Symposium on Symbolic and Algebraic Computation (ISSAC).

- Muhammad Taimoor Khan. *On the Formal Semantics of MiniMaple and its Specification Language*. In: Proceedings of the 10th International Conference on Frontiers of Information Technology (FIT 2012), IEEE Digital Library, December 2012, pp. 169-174, ISBN 978-0-7695-4927-9/125.
- Muhammad Taimoor Khan, Wolfgang Schreiner. *Towards the Formal Specification and Verification of Maple Programs*. In: Intelligent Computer Mathematics, Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, Volker Sorge (ed.), Lecture Notes in Artificial Intelligence (LNAI) 7362, pp. 231-247. July 2012. Springer Berlin/Heidelberg, ISBN 978-3-642-31373-8, **Awarded with Best Student Paper Award**.
- Muhammad Taimoor Khan, Wolfgang Schreiner. *On the Formal Specification of Maple Programs*. In: Intelligent Computer Mathematics, Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, Volker Sorge (ed.), Lecture Notes in Artificial Intelligence (LNAI) 7362, pp. 442-446. July 2012. Springer Berlin/Heidelberg, ISBN 978-3-642-31373-8.
- Muhammad Taimoor Khan, Wolfgang Schreiner. *Towards a Behavioral Analysis of Computer Algebra Programs (Extended Abstract)*. In: Proceedings of the 23rd Nordic Workshop on Programming Theory (NWPT'11), Paul Pettersson and Cristina Seceleanu (ed.), pp. 42-44. October 2011. Vasteras, Sweden.
- Muhammad Taimoor Khan, Kashif Zia. *Future Context-aware Pervasive Learning Environment: Smart Campus*. Proc. of the Integration of Information Technology in Science, Gazimagusa, Turkish Republic of Northern Cyprus, January 16–18, 2007.
- Muhammad Taimoor Khan, Kashif Zia, Nadeem Daudpota, S.A. Hussain, Najma Taimoor. *Integrating Context-aware Pervasive Environments.* Proc. of the 2nd IEEE International Conference on Emerging Technologies, Peshawar, Pakistan, pp. 683-688, IEEE, 2006.
- Babar Nazir, Muhammad Taimoor Khan. *Fault Tolerant Job Scheduling in Computational Grid.* Proc. of the 2nd IEEE International Conference on Emerging Technologies, Peshawar, Pakistan, pp. 708-713, IEEE, 2006.
- M.A. Pasha, S.A. Hussain, Muhammad Akhlaq, Muhammad Taimoor Khan. *Using Bayesian Neural Network for Modeling Users in Location Tracking Pervasive Applications.* Proc. of the International Conference on Information Technology and Applications, Quetta, Pakistan, 2005.

Technical Reports

- Muhammad Taimoor Khan, *On the Soundness of the Translation of MiniMaple to Why3ML*. DK Report 2014-03, Johannes Kepler University, Linz-Austria.
- Muhammad Taimoor Khan, *On the Verification of Maple Programs*. DK Report 2013-06, Johannes Kepler University, Linz-Austria.
- Muhammad Taimoor Khan, *Translation of MiniMaple to WhyML*. DK Report 2013-02, Johannes Kepler University, Linz-Austria.

- Muhammad Taimoor Khan, *Formal Semantics of a Specification Language for MiniMaple*. DK Report 2012-06, Johannes Kepler University, Linz-Austria.
- Muhammad Taimoor Khan, *Formal Verification of Space Missions Communication Protocols*, ISBN 978-3-659-25299-0, LAP Lambert Academic Publishing, 2012 (Master's Thesis).
- Muhammad Taimoor Khan, *Formal Semantics of MiniMaple*. DK Report 2012-01, Johannes Kepler University, Linz-Austria.
- Muhammad Taimoor Khan, *Towards a Behavioral Analysis of Computer Algebra Programs*. DK Report 2011-13, Johannes Kepler University, Linz-Austria.
- Muhammad Taimoor Khan, *A Type Checker for MiniMaple*. DK Report 2011-05, Johannes Kepler University, Linz-Austria.

Posters

- Muhammad Taimoor Khan, Wolfgang Schreiner. *A Verification Framework for MiniMaple Programs*. In: 38^{th} International Symposium on Symbolic and Algebraic Computation, June 2013.

National Conferences

- S.A. Hussain, Kashif Zia, Muhammad Taimoor Khan, Sajjad Ahmad, Umar Farooq. *Dynamic Contention Window for Quality of Service in IEEE 802.11 Networks*. Proc. of the National Conference on Emerging Technologies, Karachi, Pakistan, 2004.

Foreign Languages

English, fluent
German, intermediate level
French, basic level.

References

- *Prof. Dr. Wolfgang Schreiner.* Research Institute for Symbolic Computation, Altenbergerstraße 69, 4040, Linz, Austria. Email: Wolfgang.Schreiner@risc.jku.at
- *Prof. Dr. Renaud Rioboo.* Computer Science Professor at ENSIIE and CNAM, France. Email: renaud.rioboo@ensiie.fr.
- *Prof. Dr. Reiko Heckel.* Professor at School of Mathematics and Computer Science, University of Leicester, UK. Email: reiko@mcs.le.ac.uk.