Extending the VDM++ formal specification language with type inference and generic classes

Thomas John Hørlyck Christensen

Department of Computer Science (DAIMI)
University of Aarhus
Aarhus, Denmark

tc@daimi.au.dk

Student-ID : 20024436

Supervisors :
Michael I. Schwartzbach (DAIMI)
Peter Gorm Larsen (IHA)

April 30, 2007

English abstract

This thesis is divided into three phases. In the first phase, we convert a VDM-SL specification of a VDM++ static semantics checker to VDM++. Apart from the translation, we introduce a new architecture for the static semantics checker based on automatically generated AST visitors. The second phase is an study of the feasability of extending OML with two new language feature, implicitly-typed functions and Java-style generic classes. Finally in the third phase, we provide a proof of concept of type-inference of implicitly-typed functions for a small subset of OML and a partial implementation of Generic classes.

Danish abstract

Denne afhandling er delt i tre faser. I den første fase konverterer vi en VDM-SL specifikation af en VDM++ type-checker til VDM++. Udover denne oversættelse, introducerer vi en ny arkitektur for type-checkeren baseret på automatisk-genereret AST visitors. I den anden fase undersøger vi hvilke muligheder der er for at introducere to nye sprog-features, implicit-typede funktioner og generiske klasser, kendt fra Java. Til sidst i den tredie fase, laver vi en implementering af type-inferens for implicit-typede funktioner for en lille delmængde af OML, og en delvis implementering af generiske klasser.

Acknowledgements

I would like to thank Michael Schwartzbach for agreeing to be my formal DAIMI supervisor on this project.

I would like to thank Peter Gorm Larsen, who as my primary supervisor took on various roles during the project including VDM guru, career advisor, lecturer, academic counselor and thesis supervisor. Always with infinite patience and a goal-oriented and down-to-earth approach. Thankyou.

I owe a great deal of gratitute to Marcel Verhoef of the Overture project, for hacking the OML grammar and parser into place to suit my thesis work, and for answering all my questions and solving all my grammar problems even when he had his own Ph.D dissertation to worry about.

Thanks to the participants of the Third Overture Workshop, held in Newcastle, England in November 2006, for listening to my presentation and providing discussions and feedback. In particular José Oliveira for an interesting discussion on VDM vs. Haskell and type inference.

Thanks go also to Erik Ernst, for helpful email communications on type theory and VDM.

Latex macros for VDM++ code snippets courtesy of Peter Gorm Larsen. Latex macros for inference rules are publically available from the website of Benjamin C. Pierce.

Contents

Chapter 1

Introduction

Software engineering today is a multi-billion dollar industry. Software is pervasive in modern life, controlling everything from our bank account, car, mobile-phone and washing-machine. Software is also in charge of regulating nuclear power plants, controlling space shuttles, monitoring stock markets and regulating radiation doses in hospital scanners. This last category of applications could well be termed "critical", as the failure of the respective piece of software could result in catastrophic loss of life and resources.

Software practitioners continuously debate as to whether the production of software should be classified as an engineering disclipline on par with the classical engineering areas such as electrical, mechanical and chemical engineering. We will not enter into this discussion in this thesis, but merely note that software engineering differs from the other engineering branches in that the use of mathematical rigor is not widespread amongst its practitioners. This may seem surprising given that software is extensively used in critical domains and that it is particularly prone to a wide range of errors in all phases of its development.

Examples of spectacular software failures are not hard to come by. One of the canonical examples being the first test flight of the Ariane 5 rocket in June 1996, which self-destructed 37 seconds after launch due to a bug in its control software caused by an inappropriate data conversion from a 64-bit floating point number to a 16-bit unsigned integer [Wik].

Conventional wisdom states that there is no "silver bullet" in software development, no method which will ensure bug-free software by applying it consistently [Bro86]. However it is possible to raise our confidence in the correctness of a piece of software by applying a more formal approach starting in the early stages of the development process.

1.1 Formal Methods

Formal methods refers to mathematically based techniques for the specification, development and verification of software and hardware systems [Pre00]. Formal methods may be applied at a number of levels:

- **Formal specification** - A mathematical description of the desired behaviour of a software system specifying in abstract terms *what* a system should do and not *how* to do it.
- **Formal development and verification** - A formal development process involves iteratively *refining* a formal specification to produce the finished system. The process of *proving* or *disproving* properties of the software system against a formal specification is known as formal verification.
- **Automated theorem proving** - Using automatic or semi-automatic theorem provers to undertake fully formal correctness proofs against a specification. This can produce set of integrity constraints or proof obligations which must be discharged.

The purpose of formal specification languages, such as VDM++, is to be able to systematically describe the desired behaviour of a system in abstract terms. Since the actual implementation of the system is not interesting at the specification stage, functions are often specified implicitly, giving only pre- and post-conditions.

```
sqrt (x: rat) r: real
pre x >= 0
post r * r = x
```

Figure 1: The square root function specified implicitly using VDM++

Due to the possibility of specifying functionality implicitly, specification languages are not in general required to be executable. The current toolset supporting VDM++, VDMTools, does however support interpretation of an executable subset of VDM++. However, as mentioned in [HJ89] ,

> *The need to be able to execute the model tends to bias models to a more explicit style in which behaviour is described in a functional or even imperative programming style.*

The discussion of implicit specification of functionality vs. explicit (executable) specification is a long-standing one in the formal methods community.

Some feel that requiring a specification to be written in an executable subset of the specification language, unnecessarily restricts the forms of specification that can be used. Furthermore, executable specifications tend to overspecify the problems and constrain the choice of possible implementations [HJ89].

We can even specify functionality which is not immediately computable. For instance, we can specify Fermat's last theorem as an implicit function. Whether or not this function is computable is unknown. Thus, the issue of computability presents no restriction to what we can specify.

```
Fermat (n:nat) x:nat, y:nat, z:nat
post x**n + y**n = z**n
```

Figure 2: Implicitly defined, uncomputable (?) function

In this thesis however, we *will* restrict ourselves to the executable subset of VDM++. This is due to the fact that we will be directly specifying part of a tool-chain, which will eventually be code-generated to Java, and thus must necessarily be executable.

1.1.1 Formal specification languages

Formal specification languages almost invariably use mathematically based syntax. This allows the user to create a high-level model of the system using mathematical structures such as sets, propositions and mappings. The mathematical notation provides a well-defined and well-known semantics, which help to prevent the vagueness and ambiguities which may arise in a more informal requirements-gathering phase. Another advantage of using mathematical notation is the ability to apply logical inference rules for automatically validating the internal consistency of a model.

The expressiveness of mathematical notation allows a high level of abstraction allowing the user to focus on the core functionality of the system without prematurely worrying about implementation details.

A considerable number of formal specification languages are available [Bowa], some of the more well known include:

- **Z** - Pronounced "zed". Based on **Z**ermelo-Fränkel set theory, lambda calculus and first order predicate logic. Developed in 1977 by Jean-Raymond Abrail, Steve Schuman and Bertrand Meyer it was later further developed at the programming research group at Oxford university. A variant supporting object-oriented modelling exists, Object-Z [Bowc].

- **B** - The B Method is based on Abstract Machine Notation. Developed by Jean-Raymond Abrail it allows easier refinement to code (compared to Z). A tool set is available supporting specification, design, proof and code generation. [Bowb]

- **RAISE** - **R**igorous **A**pproach to **I**ndustrial **S**oftware **E**ngineering. Developed by Dines Bjørner as part of the European ESPRIT II LaCoS project in the 1990s. The methods consists of a set of tools based on RSL, the Raise Specification Language [Ter]

- **Alloy** - Developed in 1997 by the Software Design Group at M.I.T. Alloy is a declarative formal specification language designed specifically for automatic analysis. The toolkit includes a model finder/checker which converts the model to a boolean formula which can then be checked using a SAT solver. [TSDGaM]

1.1.2 The Vienna Development Method

The Vienna Development Method (VDM) is a collection of techniques for the modelling, specification and design of computer-based systems, based on mathematically rigorous, formal specifications [Fit].

History

VDM was a product of a long line of programming language research conducted at IBM's Vienna Laboratory in the 1960's and 1970's. The development of the method was inspired by the need to provide a formal semantics of the PL/I programming language as part of compiler correctness arguments. The methods developed were originally applied to formal specifications of programming languages, but have since been sucessfully applied to a wide range of application domains, especially large-scale mission-critical software projects [DCaAE].

The specification languages

The first specification language developed for use with VDM was VDL (Vienna Definition Language). VDL was used to provide formal semantics for various new languages, including PL/I, ALGOL 60 and BASIC.

An improved version of VDL was named META-IV, and was used to formalize amongst others, the CHILL and ADA languages [Luc81]. The current incarnation of the specification language is VDM-SL (VDM Specification Language). VDM-SL acheived status as an ISO standard in 1996 [PBB$^+$96]. An object-oriented extension to VDM-SL, VDM++ was later developed through the European Union's Afrodite project.

Tool Support

A proprietary toolset supporting VDM-SL and VDM++, VDMTools, was originally developed by the Danish software company IFAD A/S (Institut for Anvendt Datalogi). IFAD has since been declared bankrupt and the intellectual rights to VDMTools has passed to the Japanese company CSK [Cor]. The latest version of VDMTools supports an executable subset of VDM++, and includes a syntax checker, type checker, interpreter and code generation to Java and C++.

The Overture Project

Recently, a group of the original VDMTools developers have teamed up with a number of industry users to create a revised, open-source alternative to the proprietary VDMTools. Together they founded the Overture Project [Proa], with a mission to produce a high-quality, open-source toolset for VDM modelling. This thesis is intended to be a part of the Overture project and will hopefully contribute an efficient static semantics checker, implemented as an extension to an existing Eclipse plugin produced by the Overture project.

The language intended to be supported by the Overture Toolset, is OML (Overture Modelling Language), which is essentially an extended version of VDM++.

1.2 The purpose of the thesis

A static semantics checker already exists for the VDM++ language. The static semantics checker is specified (using VDM-SL) in a proprietary, confidential document owned by CSK Systems [CSK].

1.2.1 Goals

The goals of the thesis can be expressed in a number of phases:

- **Phase 1** - *Convert* a subset of the original specification, written in VDM-SL, to an equivalent specification, written in VDM++.
- **Phase 2** - *Investigate*, for a suitable subset of the VDM++ language the possibilities of adding two new language features to OML :
  - Implicitly typed functions (with no function type annotations).
  - Parameterized classes in the style of Java Generics.
- **Phase 3** - *Implement* an extended statics semantics checker for a subset of VDM++ as a proof of concept of Phase 2.

1.2.2 Constraints

- The static semantics checker is to be specified using VDM++.
- The static semantics checker may, if proven useful, be included in the open-source Overture Project and eventually become part of the Overture OML Eclipse plugin.

1.2.3 Learning objectives

My personal learning objectives for the thesis are:

- Learning more about software engineering using formal methods.
- Expanding my knowledge of programming language implementation, building on my interest for this area developed during the compiler course at DAIMI.
- Acquiring a deeper knowledge of aspects of type theory and applying it in a practical context.
- Helping to spread the word about formal methods, hopefully making the approach more popular in academia as well as in the industry.

1.3 Use of notation

VDM-SL and VDM++ may be printed in two distinct yet equivalent styles.

The formal mathematical style:

```
$max : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$
$max\,(v1, v2) \triangleq$
  if $v1 > v2$
  then $v1$
  else $v2$
```

Figure 3: Mathematical syntax

and an ASCII (interchange) style:

To ease the reading of any VDM++ code appearing in this thesis, I will consistently use the ASCII notation.

```
max: int * int -> int
max(v1, v2) ==
        if v1 > v2
        then v1
        else v2
```

Figure 4: ASCII syntax

1.4 Assumptions

The reader is assumed to be familiar with basic terminology related to types and type systems as this will not be covered fully from scratch. The reader is not expected to have detailed knowledge of the VDM++ language as this will be covered to the necessary extent in chapter 3.

Chapter 2

The VDM++ language and type system

This purpose of this chapter is to introduce the reader to the VDM++ language and its associated type system. The coverage of the language is purposely kept as brief as possible. We instead refer the reader to the full VDM++ language reference manual [VDM] and the VDM++ grammar in Appendix 1. The examples in this chapter are taken from [FLM+05].

2.1 Class Definitions

A VDM++ model consists of a collection of class descriptions. A class definition consists of a set of blocks, each containing zero or more definitions. Figure 3.1 shows the structure of a VDM++ class definition.

Object-oriented structure

VDM++ supports inheritance relationships between classes by use of the `is subclass of` keyword. The standard OO access modifiers, `private`, `public` and `protected`, can be applied to influence class member visibility. Static members are also supported, as is dynamic binding and abstract classes.

2.1.1 Invariants

Invariants may be defined on instance variables. An invariant property is required to hold at all times during the lifetime of an instance of the class. Any instance variable invariant must be checked before it is assigned to, to ensure that the invariant is respected. Invariants may also be defined on type declarations to restrict the set of values that are included in that type. Figure 3.2 gives an example of a complex invariant defined on a set of instance variables.

```
class <class name>

values
    <constant value definitions>
types
    <type definitions>
instance variables
    <instance variable definitions>
operations
    <operation definitions>
functions
    <function definitions>
thread
    <thread definitions>
sync
    <synchronization constraints>

end <class name>
```

Figure 5: The structure of a VDM++ class definition

Values

A value definition is similar to a constant definition in other programming languages.

Functions

Computations on data may be performed by either *functions* or *operations*. Both take input and return a result but only operations may act on class instance variables. Functions and operations are defined in seperate blocks in the class definition. Functions may be defined explicity or implicitly.

Implicit function definitions When defining a formal model with the purpose of analysis, it is useful to be able to specify the pre- and post-conditions of a function and omit the implementation details which may unfairly bias subsequent design choices. This is possible using implicit function definitions. A pre- or post-condition is a boolean expression that describes the relationship between the input and the result of the function. Figure 1 shows the square-root function described implicitly.

Explicit function definitions Explicit functions are functions in the traditional sense, in which the body of the function describes the algorithmic procedure

```
class Date

instance variables
       day : nat1;
       month : nat1;
       year : int;

inv day <= 31 and
    month <= 12 and
    if month in set {4, 6, 9, 11}
    then day <= 30
    else (month = 2) => (day <= 29)

end Date
```

Figure 6: A Date class, with an instance variable invariant

required to perform the desired computation.

```
max : int * int -> int
max(v1, v2) ==
       if v1 > v2
       then v1
       else v2
```

Figure 7: An explicit function computing the max. of two integers

Operations may in a similar manner be defined either implicitly or explicitly. Operation post-conditions may refer to "before" and "after" values of instance variables in addition to being able to refer to operation input and result.

Expressions and statements

VDM++ has 44 kinds of expressions [VDM], most of which are known from other programming languages, functional as well as imperative. Aside from the usual conditional expressions and various kinds of loop statements, VDM++ features patterns, bindings, quantified expressions and various comprehension expressions, eg. set comprehension.

```
forall x in set {1, 3, 5, 7} & x**2 < 40
```

Figure 8: An example of a quantified expression

Collections and comprehensions

There are a number of possibilites available when modelling collections of objects, Sets, mappings and sequences.

Sets A set is a collection of items in which order and repetition is not important. Sets may be constructed explicitly by listing their contents, as in Figure 3.4., or implicitly by a set comprehension expression.

```
{x * 3 | x in set {1, 2, 3, 4, 5}}
```

Figure 9: A set comprehension expression evaluating to {3, 6, 9, 12, 15}

Maps A mapping captures the concept of a mathematical map, with a domain and range. A mapping is a set of pairs of domain and range elements called a maplet.

```
{"A" |-> 1, "C" |-> 3, "E" |-> 5}
```

Figure 10: A mapping from characters to their position in the alphabet

Mappings may, like sets, be constructed explicitly by listing its elements, or implicitly by using a map comprehension. Mappings may additionaly be defined as injective.

Sequences Sequences are ordered, arbitrary length collections of items. Unsurprisingly, sequences may also be constructed either implicitly or explicitly.

```
[i*i | i in set {1,..., 10} & i mod 2 = 0]
```

Figure 11: A sequence comprehension expression yielding [4, 16, 36, 64, 100]

2.2 The VDM++ type system

The types in VDM++ are divided into the two categories, Basic types and constructed types.

Basic Types

Basic types represent atomic values that cannot be decomposed further.

bool	true, false
nat1	1, 2, 3, ...
nat	0, 1, 2, ...
int	..., -2, -1, 0, 1, 2, ...
rat	..., -1/7, -1/356, ..., 1/3, ...
real	..., -12.78356, ..., 0, ..., 3478.654, ...
char	'a', 'b', ..., '1', '2', ..., '+', '-', ...
quote	<RED>, <CAR>, <QuoteLiteral>, ...
token	mk_token(...)

Numeric types The numeric types (nat1, nat, int, rat and real) are unbounded, with no maximum value value. This is useful for modelling purposes, but is not supported in any tools. The numeric types are related by the subset relation as follows.

$$\boxed{\textbf{nat1} \subset \textbf{nat} \subset \textbf{int} \subset \textbf{rat} \subset \textbf{real}}$$

Figure 12: The numeric types in VDM++ ordered by subset inclusion

The **char** and **bool** types are defined as expected. The **quote** type is a single named value enclosed in angle brackets. The quote literal is the only value of that type. The **token** type represents values with no internal structure, or a structure which is not relevant to our model. A token may be constructed by the token constructor mk_token(...).

Constructed types

Constructed types are created from basic types using type constructors. Mappings, sequences and sets, as mentioned earlier are also constructed types.

Union types A union type contains all the values from its component types. It is possible to construct union types using non-disjoint types, such as `int` and `nat`.

```
nat | bool | seq of Foo
```

Figure 13: A union type with component types nat and bool and sequence of Foo objects

Product types A product types is a composition of other types into a tuple. The tuple constructor mk_(...) is used to create a tuple.

```
nat1 * (seq of char) * bool
```

Figure 14: A product type consisting of three different component types

Record types A record type is essentially a product type with named components.

```
Address ::
       house : HouseNumber
       street : Street
       town : PostalTown
```

Figure 15: Record type with three fields

Optional types The optional type constructor takes any type as its argument add a the special value **nil** to it. The optional type is typically used to model the possible absence of a value.

```
[bool]
```

Figure 16: Optional type constructor having possible values **true**, **false** and **nil**

Object references An object reference is a reference to an instance of a named class.

Type invariants

Invariants may be specified on type definitions. A type invariant is a boolean predicate which restricts the extent of the type being defined.

```
a_ref : A := new A()
```

Figure 17: `a_ref` is an object reference to an object of class A

```
SmallNat = nat
inv s == s < 100
```

Figure 18: Type invariant restricting SmallNat to values < 100

2.3 Summary

To sum up, VDM++ is a large language with many constructs that may appear exotic compared to normal general purpose programming languages. We must remember however, that the intent of VDM is not to program, but to *specify* a model of a program. The level of abstraction is necessarily higher than a general purpose language. We can specify models using high-level constructs such as sets, maps and implicit functions, without having to worry about how they will be implemented.

Chapter 3

Types and type systems

3.1 Types

A *type* is a set of values that a program variable, of that type, can assume during the execution of the program. For example, a VDM variable **x** of type *nat1* may assume values from the set {1, 2, 3, ...}. A programming language that uses types is referred to as a *typed* programming language.
A typed programming language where the programmer must explicitly annotate variable with a type, for example:

```
x : nat1
b : bool
m : map nat to bool
```

is referred to as an *explicitly typed* language. A typed language where explicit annotations are not required is referred to as *implicitly typed*.

3.2 Type system

A *type system* is collection of rules which collectively describe the conditions which must be met by the programming language constructs in order for the program to be *statically type correct*.

A concise description by Benjamin Pierce [Pie02] states:

> *A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.*

In addition a type system provides a number of advantages.

- Safety - Obvious programming errors such as

```
5/true
```

can be detected at compile-time.

- Optimization - Annotating variables with types provides useful information to the compiler, which may allow it to perform optimizations based on the variables type.

- Documentation - Properly used types may help express the programmers intention about a piece of code.

- Abstraction/Modularity - Types allow a programmer to think about his program at a higher level, eg. in terms of interfaces.

A type system can be seen as a set of constraints imposed on the programmer by the designer of the programming language in order to prevent certain kinds of errors. It is desirable to have a powerful type system which can statically determine the presence of errors, while not overly restricting the programmer.

3.3 Type Inference

Type inference, also known as *type reconstruction* or *type synthesis*, is the process by which a compiler will attempt to infer the type of program phrases. In an implictly typed programming language, type annotations are omitted and the compiler has to recover the missing type information and ensure that the program is statically type correct.

3.3.1 Algorithms for type inference

The Hindley-Milner algorithm

The standard algorithm for perfoming polymorphic type inference was discovered by Robin Milner in 1978 [Mil78]. Later, Luis Damas proved that the system was complete and the algorithm computed the *principal type* of every expression (ie. the most general type). The algorithm proceeds in three phases.

- In phase 1, a *type variable* is assigned to every compound expression in the AST. Constants and operations are assigned the type which is known for them.

- In phase 2, a set of constraints are generated on the types used in the program. The constraints are generated using the structure of the AST.

- In phase 3, the set of constraints are solved using a unification algorithm, which solves the set of constraints in a way which is least constraining for each type.

3.4 Polymorphism

The concept of programming language constructs being able to have multiple types is known as *polymorphism*, meaning to have many forms. In contrast, a *monomorphic* construct has a single unique type. A polymorphic function is a function whose actual parameters can have more than one type. Polymorphic types are types whose operations are applicable to values of more than one type.

3.4.1 Kinds of polymorphism

Cardelli and Wegner [CW85] classify polymorphism variants into a hierarchy, with the two main kinds of polymorphism, *universal* and *ad-hoc* being further specialized into *parametric*, *inclusion*, *overloading* and *coercion* polymorphism.

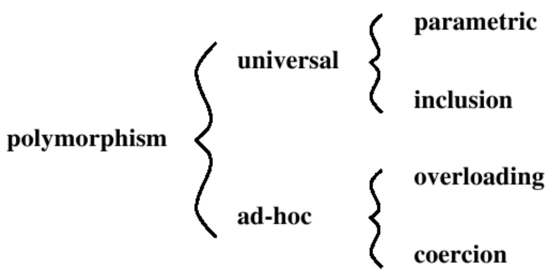


Figure 19: The Cardelli-Wegner polymorphism heirarchy.

Universal polymorphism

A universally polymorphic function can be applied to arguments of an infinite number of types provided that the argument types have some common structure. This common structure can be enforced as parametric polymorphism or inclusion polymorphism.

Parametric polymorphism A parametric polymorphic function may be applied to any arguments whose types match a type expression involving *type variables*. In addition to its actual parameters, the function must be called with a type parameter to indicate which specific type the function will operate on.
The canonical example of this type of polymorphism is a function which appends

an item to a list of items of the same type.

```
sort : ('a * 'a -> bool) * 'a list -> 'a list
```

Figure 20: The type signature of an ML list sorting function

Parametric polymorphism may be *explicit*, requiring the programmer to annotate function declarations with type parameters and correspondingly to supply a type parameter to the function when it is called. Conversely, *implicit* parametric polymorphism does not require either type annotations nor type parameters in function calls. The missing type information is identified and recovered by the compiler using *type inference*.

Inclusion polymorphism Inclusion polymorphism allows an object to belong to multiple different, non-disjoint classes. Also known as *subtype polymorphism*, this is a well-known concept from Object-oriented languages such as Java. Any method `foo(Bar arg1) {...}` accepting an argument of class `Bar` will also accept any argument of a subtype of `Bar`. This allows late binding of argument types, such that the precise class of the argument of eg. `foo()` is not known until runtime.

Ad-hoc polymorphism

Ad-hoc polymorphism variants are not considered *true* polymorphism as are the universal variants. Function overloading is essentially a syntactic shortcut, while coercion allows operations to be polymorphic on the surface, but behind the scene converts operands to the type expected by the operation.

Overloading A function name may be *overloaded* if several different functions (usually distinguished by number of parameters and parameter type) have the same name. The compiler must decide which version of the function to call, based on the context the function was called in. It is also possible to overload operators so that they take operands of different types. In most programming languages the addition '+' operator is overloaded to support both integer and floating-point addition (in Java it is further overloaded to support string concatenation).

Coercion A coercion is an operation performed on an argument type in order to convert it to a different expected type. An addition operation between a real and an integer will cause the integer to be coerced into a real as this is the most general type for both arguments. The overloaded addition operator selected for the job will be the one operating on two reals. Coercing a real to an int is possible, but would lead to an undesirable loss of precision.

3.5 Polymorphism in VDM++

VDM++ supports several of the above-mentioned variants of polymorphism.

3.5.1 Parametric polymorphism in VDM++

The user must supply a type parameter when the function is called, this is therefore explicit parametric polymorphism. More precisely, VDM++ supports first order, explicit parametric polymorphism [CW85, Cardelli&Wegner]. This form of polymorphism does not allow polymorphic functions to take other polymorphic functions as arguments. Furthermore, polymorphic types cannot be introduced by means of a type definition. [FOR98, 1.7.3 VDM-SL formal definition]

3.5.2 Subtype polymorphism

Similarly to Java and other Object-oriented languages, an function argument of type eg. `Foo` may be replaced by any subtype of `Foo`.

```
class foo
end foo

class bar is subclass of foo
end bar

class test

  functions
    test : foo -> int
    test(f) == 42;

end test
```

Figure 21: Function call with subtype parameter

3.5.3 Overloading

VDM++ supports function overloading provided that the function argument types do not overlap. Thus:

```
>> print t.test(new bar())
42
```

Figure 22: VDMTools output demonstrating subtype polymorphism

```
m : foo -> int
m : int -> int
```

can co-exist but

```
m : foo -> int
m : bar -> int
```

cannot, as `bar` is a subtype of `foo`.

3.5.4 Coercion

Since all the numeric types in VDM++ are related to each other by subset inclusion, the addition operator is defined to be of type `real * real -> real`. Any two numbers may therefore be added sucessfully as `real` is the most general numeric type.

3.6 The semantics of VDM++

As described in [PBB$^+$96, VDM-SL standard, chapter 9], any VDM specification conforming to the Concrete syntax may be parsed to an *Outer Abstract Syntax* representation (parse tree). This parse tree may be transformed into an equivalent *Core Abstract Syntax* representation. The dynamic semantics is then defined as a total function mapping a specification conforming to the Core Abstract Syntax to the set of models of this specification. Thus a specification containing syntax or type errors will yield an empty set of models. Consistent specifications are thus those that have a model in the non-empty set of models produced by the dynamic semantics.

3.6.1 The static semantics of VDM++

As described in [PBB$^+$96, VDM-SL standard, ch,9], the static semantics of VDM++ may informally be described as consisting of type checking and a series of related and decidable consistency checks.
In traditional programming languages, statically undetectable type errors may be

detected by dynamic checks performed at run-time. Since VDM specifications are not required to be executable, it is not possible to rely on dynamic checks in general.

3.6.2 The VDM++ type system

As mentioned in [Aic97], the purpose of having a type system is twofold. Types are an important tool for modelling and as an instrument of automatic consistency checking. General purpose programmings languages tend to emphasize the automatic consistency checking and seek to build fast and reliable compilers. Since VDM is a modelling language, the modelling aspect is emphasized. As a result VDM has a very expressive type system. Examples of this include.

- Non-disjoint types. A value having one numeric type does not preclude it from also having another numeric type. The fact that the VDM numeric types are related to each other by subset inclusion, as shown in figure 12, allows us to define functions such as

```
foo : nat -> int
foo(n) == n;
```

Figure 23: Non-disjoint numeric types

- Union types. A *union type*, $T_1 \cup T_2$, as defined in [Pie02] represents the ordinary union of the set of values belonging to the type T_1 and the type T_2. The components of the union may basic or compound types, including another union type. A variable having a union type may be assigned a value which has the type of any of the elements of the union. The union type allows us to write functions such as:

```
Div : int * int -> real | bool
Div(n1, n2) ==
        if n2 <> 0
        then n1 / n2
        else false
```

Figure 24: Safe division using union types

- Type invariants. Subtypes can be defined by restricting the range of values in a type by means of an invariant. The acceptable values of the subtype are those for which the invariant evaluates to true. In figure 25, the subtype `SmallNat` is restricted to natural numbers less than 100.

```
SmallNat = nat
inv n == n < 100
```

Figure 25: Type invariants

3.6.3 Typechecking in VDM

The language feature mentioned in section 3.6.2 are some of the reasons why typechecking VDM specifications is in general undecidable.

Union types An expression like `x + 1` where x has the type `nat | bool`, cannot be statically checked. If `x` is of type `bool` the expression will be type inconsistent and type consistent if `x` is of type `bool`.

Type invariants Checking an type invariant in a given context would require evaluating the invariant, which would move the check out of the arena of static type checking.

The VDM type system is very expressive and pays the price in terms of type-checking being undecidable in general. Type-checking VDM specifications is based on a twofold approach. [DHB91].

- Acceptance of definitely consistent specifications.
- Rejection of definitely inconsistent specifications.

The rejection strategy corresponds to type-checking in traditional programming languages. This is called the *possible*, or `POS`, type-check as it checks whether a specification is possibly consistent or not. If not it is rejected. The acceptance strategy, the *definitely* consistent, or `DEF` check, ensures that a specification is definitely type-consistent. For instance `1 + 2` is definitely type-correct. In the presence of the "difficult" constructs as mentioned above as well as in the presence of partial operators such as division, the `DEF` check will fail.

The diagram in figure 26 shows the set of all VDM specifications that can be written. The middle line separates the consistent and the inconsistent specifications. The areas to the far left and right represent the specifications that can be accepted or rejected by the static semantics checker. The extended rejection/acceptance areas represent specifications that could be rejected or accepted using more sophisticated analyses possibly developed in the future.

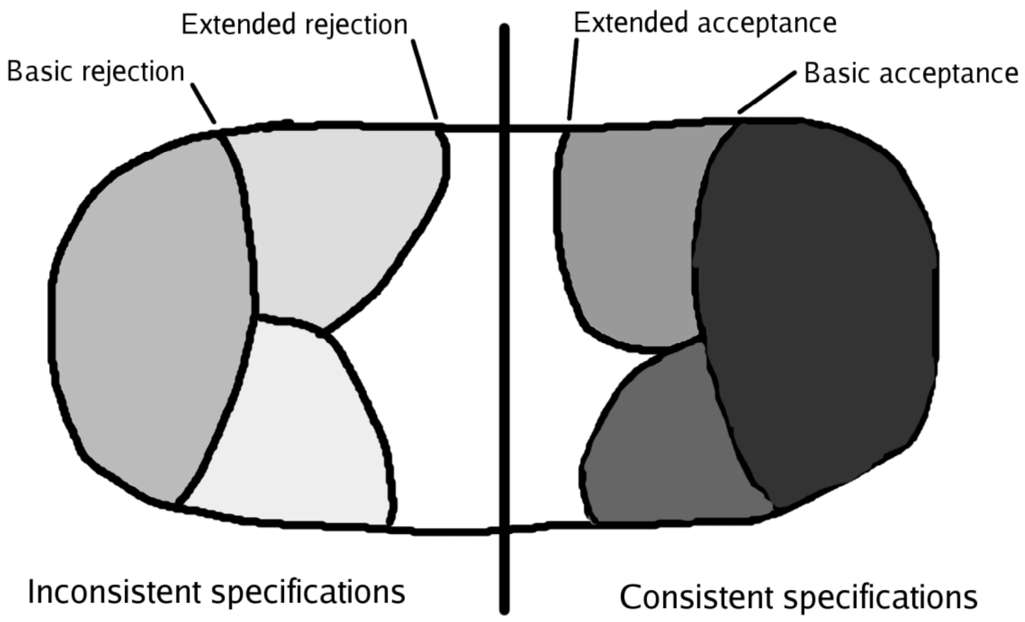


Figure 26: Rejected and accepted subsets of specifications

The current static semantics checker in VDMTools can operate in two modes, `POS` and `DEF`. `DEF` is not used very often as most specifications fail this check. During the conversion process, the `DEF` checks were removed, leaving only `POS` mode. The `DEF` check will be replaced in the future by a proof-obligation generator combined with a automatic theorem-prover, currently under development, as mentioned in section 4.4.2.

Chapter 4

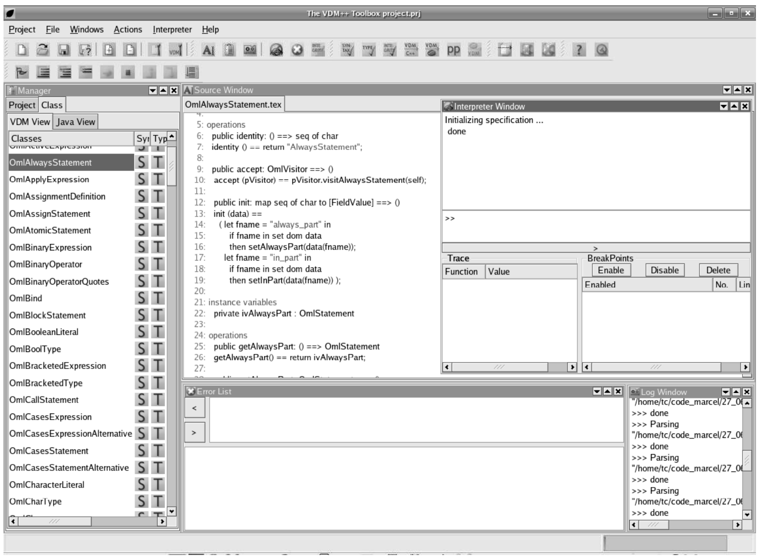
The Overture Tool Chain

One of the purposes of the Overture project was to provide a setup for researchers and students (B.Sc, M.Sc and PhD) to experiment with formal methods in general and VDM++ in particular. Since its inception in 2004, Overture has spawned a number of student projects which have contributed to the overall tool chain. This chapter provides a brief overview of previous work on the project and some insight into the current status of the tools.

4.1 Previous work

4.1.1 VDMTools

During the period 1992-95, the object-oriented extension to the formal specification language, VDM-SL was developed. As mentioned in chapter 1, this led to the development of VDMTools, an industrial-strength, commercial tool.



```
OmlAlwaysStatement
OmlApplyExpression
OmlAssignmentDefinition
OmlAssignStatement
OmlAtomicStatement
OmlBinaryExpression
OmlBinaryOperator
OmlBinaryOperatorQuotes
OmlBind
OmlBlockStatement
OmlBooleanLiteral
OmlBoolType
OmlBracketedExpression
OmlBracketedType
OmlCallStatement
OmlCasesExpression
OmlCasesExpressionAlternative
OmlCasesStatement
OmlCasesStatementAlternative
OmlCharacterLiteral
OmlCharType
```

```
operations
  public identity: () ==> seq of char
  identity () == return "AlwaysStatement";

  public accept: OmlVisitor ==> ()
  accept (pVisitor) == pVisitor.visitAlwaysStatement(self);

  public init: map seq of char to [FieldValue] ==> ()
  init (data) ==
    ( let fname = "always_part" in
        if fname in set dom data
        then setAlwaysPart(data(fname));
      let fname = "in_part" in
        if fname in set dom data
        then setInPart(data(fname)) );

instance variables
  private ivAlwaysPart : OmlStatement

operations
  public getAlwaysPart: () ==> OmlStatement
  getAlwaysPart() == return ivAlwaysPart;
```

```
Initializing specification ...
 done
>>
```

```
"/home/tc/code_marcel/27_0
>>> done
>>> Parsing
"/home/tc/code_marcel/27_0
>>> done
>>> Parsing
"/home/tc/code_marcel/27_0
>>> done
>>> Parsing
"/home/tc/code_marcel/27_0
>>> done
```

Figure 27: Screenshot of the VDMTools IDE

VDMTools is still under active, though limited, development and is currently being promoted to the automobile and consumer electronics industries in Japan.

VDMTools consists of an IDE (see figure 23) with a parser, type-checker, interpreter, code-generator (to C++ and Java) and a proof-obligation generator (PO-generator based on [Aic97]). At the recent third Overture Workshop, the full version of VDMTools was released for academic use [Prob] .

Being a commercial product, VDMTools is not a part of the Overture toolset. However it provides a baseline for the development of the Overture toolset and the ambition is to create an equivalent or better tool based on the Eclipse platform.

4.2 Parser implementations

Two early implementations of an OML parser were conducted as student projects [Proa].

- In 2004, Pieter van der Spek, an M.Sc student from Technical University of Delft created a parser and a pretty-printer, which included experiments on improved error handling in the parser generator [vdS04]. The project was based on an Eclipse plugin and was published in ACM SigPlan notices. The parser did not support XML representations of the AST and was based on direct manipulation of a concrete syntax tree.
- In 2005 Jacob Porsborg Nielsen and Jens Kielsgaard Hansen, two M.Sc students from DTU (Technical University of Denmark) re-implemented the parser using the ANTLR framework [NH05] . The project was again based on an Eclipse plugin and featured XML support for reading and writing AST instances. The hand-coded AST (!) (200+ classes), was however quite error-prone and not easily maintainable and was eventually discarded.

4.3 The new parser

in 2006, it was decided to look into ways of automating as many steps as possible in the tool-creation process. In 2006, Marcel Verhoef [Ver], a software developer, PhD-student and member of the Overture team, started development of a new, robust and fully automated AST generation tool. To de-couple the OML AST specification and the concrete parser implementation, the OML AST is specified in a subset of VDM-SL. This requires any future parser implementations to implement the interface definitions.

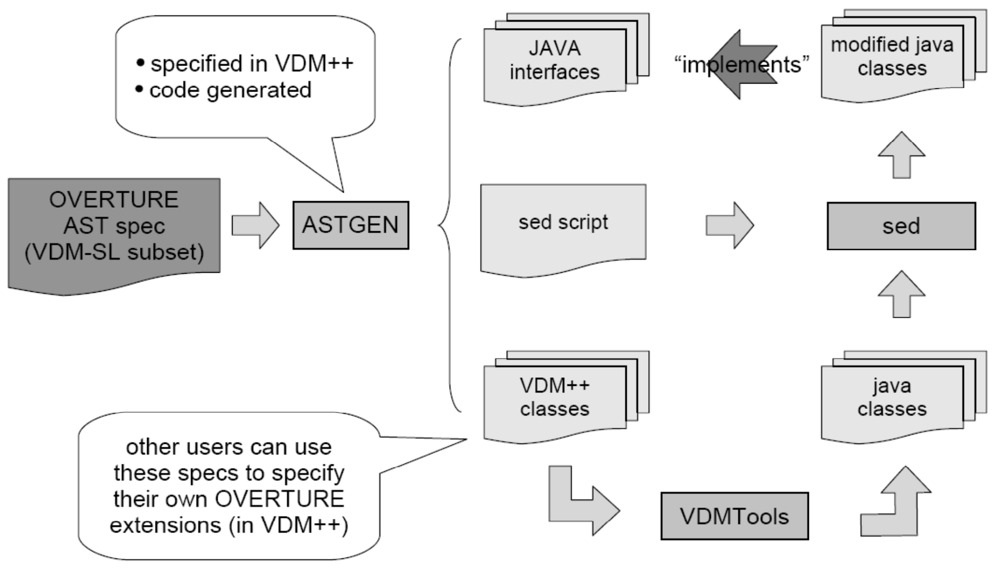


Figure 28: The new parser, automatic AST generation

The tool generates a set of VDM++ interface classes specifying the public methods available for each type of AST node. A corresponding "implementation" class is also generated for each node type.

```
class IOmlClass is subclass of IOmlNode

operations

public getIdentifier: () ==> seq of char
getIdentifier() == is subclass responsibility;

public getGenericTypes: () ==> seq of IOmlType
getGenericTypes() == is subclass responsibility;

public getInheritanceClause: () ==> IOmlInheritanceClause
getInheritanceClause() == is subclass responsibility;

public hasInheritanceClause: () ==> bool
hasInheritanceClause () == is subclass responsibility;

public getClassBody: () ==> seq of IOmlDefinitionBlock
getClassBody() == is subclass responsibility;

public getSystemSpec: () ==> bool
getSystemSpec() == is subclass responsibility;

end IOmlClass
```

Figure 29: Automatically generated "Interface" class for the IOmlClass AST node

The tools generates a default AST visitor class implementing the "Visitor" design pattern [GHJV95] , which can be subclassed to perform customized AST traversals in later phases. A Java interface is also generated for each node type.

The parser outputs a VDM++ or VDM-SL expression representing the AST of the source code. The AST expression can be used as input to later phases in the tool chain.

The parser tool incorporates the JFLEX lexical analyzer generator [JFl] and the BYACCJ parser generator [BYA]. Since the first release in late October 2006, the parser tool has proven robust and equally important, easily extendible. The extension to the grammar of typeless explicit functions and parameterized classes to be used in this thesis, were added to the OML grammar in less than a day.

```
class IOmlVisitor

operations

public visitSpecifications: IOmlSpecifications ==> ()
visitSpecifications (-) == is subclass responsibility;

public visitClass: IOmlClass ==> ()
visitClass (-) == is subclass responsibility;

public visitInheritanceClause: IOmlInheritanceClause ==> ()
visitInheritanceClause (-) == is subclass responsibility;

public visitDefinitionBlock: IOmlDefinitionBlock ==> ()
visitDefinitionBlock (-) == is subclass responsibility;

... operations for each AST element

end IOmlVisitor
```

Figure 30: Automatically generated visitor class

4.4 Work in progress

4.4.1 Dynamic Semantics

As of January 2007, Hugo Macedo, a M.Sc student from the University of Minho, Portugal, started work on the dynamic semantics of OML. This project involves designing and implementing an interpreter for OML. Currently Hugo is investigating whether or not it is feasible to use the existing Java JVM as a runtime platform. The Overture interpreter would thus have to generate Java bytecode and leverage the JVM instead of building a virtual machine from scratch and (re)using the proprietary stack machine instruction set from VDMTools.

4.4.2 Proof support using HOL

Sander Vermolen, M.Sc student from Radboud University in Nijmegen, Holland, started his master thesis work in January 2007 on the topic of adding proof support to the Overture tool set. This work involves transforming a VDM++ model and its proof obligations to HOL 4 (Higher Order Logic, an interactive theorem prover) [HOL] and attempting to automatically prove as many as possible of the proof obligations.

```
new OmlLetBeExpression(
   new OmlSetBind(
     [
       new OmlPatternIdentifier("a")
     ],
     new OmlSetRangeExpression(
       new OmlSymbolicLiteralExpression(new OmlNumericLiteral(1)),
       new OmlSymbolicLiteralExpression(new OmlNumericLiteral(10))
     )
   ),
   nil,
   new OmlBinaryExpression(
     new OmlName(
       nil,
       "a"
     ),
     new OmlBinaryOperator(28),
     new OmlSymbolicLiteralExpression(new OmlNumericLiteral(1))
   )
)
```

Figure 31: VDM++ expression output from parser

4.5 Future Work

Future plans for the Overture Tool set include

- Full integration with Eclipse, all tools should be created as Eclipse plug-ins to allow a modular and flexible architecture.
- Support for automated testing and test generation.
- Model checking and interactive proof support.
- Model visualization using UML.
- Code generation to C#, Java or C++
- Translation from OML to JML (Java Modelling Language)
- Refactoring support and in general incorporating more of Eclipse's feature set.

Given that one of the purposes of the Overture project is to allow language experiments on OML and student participation in the project, there is enough to keep many students busy for quite a while.

Chapter 5

Part 1 - The Conversion

As outlined in section 1.2.1, the first part of this thesis project involves a conversion of a subset of an existing static semantics checker written in VDM-SL to an equivalent version written in VDM++. This chapter will provide a more fine-grained specification of this goal, and detail the work done towards acheiving it.

5.1 Initial setup

This work originated in a description of a project idea proposed by Peter Gorm Larsen in connection with the Master Thesis Preparation course held each semester at DAIMI [Ern]. The original project involved developing an extensible static semantics checker on top of the existing Overture platform, which at the start of the project was the re-implemented ANTLR parser in an Eclipse plugin.

As a starting point, the VDM-SL specification of the existing static semantics checker as implemented in the commercial VDMTools [CSK], was made available for this work. The specification is a 300-page document composed of about 600 functions and operations specifying well-formedness of types, functions, operations, instance variables, threads, expressions, statements and patterns as well as the environment and scope rules. The source specification of the existing static semanctics checker has been released by CSK to the Overture project, however it is still governed by a non-disclosure agreement.

The course *Model driven development using VDM++ and UML 1* at Århus Engineering College (IHA) [Lar] in the fourth quarter of 2006 provided a solid introduction to the VDM++ core language, and VDMTools.

5.2 The existing static semantics checker

5.2.1 Why convert from VDM-SL ?

There are a number of reasons why the existing static semantics specification would benefit from being refactored into VDM++.

- The basic unit of code in VDM-SL is a module. VDM++ is object-oriented and provides a more natural mapping from VDM++ to Java. The VDM++ specification of the static semantics checker will eventually be code-generated to Java and integrated into the Eclipse platform.
- The new AST generation tool provides a default visitor class for free. Subclassing the OMLvisitor class allows us to split the momolithic structure of the old specification into a set of well-defined visitors for use in different passes of the AST.
- The Eclipse architecture is based on plug-ins and the idea of a hierarchy of extension points. By sufficiently de-coupling the individual phases of the static semantics checker, it would be possible to have a one-to-one mapping beteen phases and Eclipse extensions, allowing for a fine-grained, modular and maintainable architecture
- Although VDM-SL is not obsolete, VDM++ is currently the language of choice when writing specifications using the VDM method. Given this, VDM++ will probably be the primary specification language for VDM-related activities, both in academia and in the industry, for the forseeable future.

5.2.2 The module structure

The diagram in Figure 32 shows the relationships between the various VDM-SL modules that comprise the VDMTools static semantics checker. The boxes represent modules and the arrows represent import dependencies between modules (Thus the DEF module imports constructs from the STMT, EXPR, PAT, TYPE and ENV modules). The following provides a brief introduction to the role of each module.

DEF: The DEF module checks the well-formedness of definitions that can appear at the top-level of a specification, eg. a class definition.

STMT: Well-formedness of statements.

EXPR: Well-formedness of expressions.

PAT: Well-formedness of patterns and bindings.

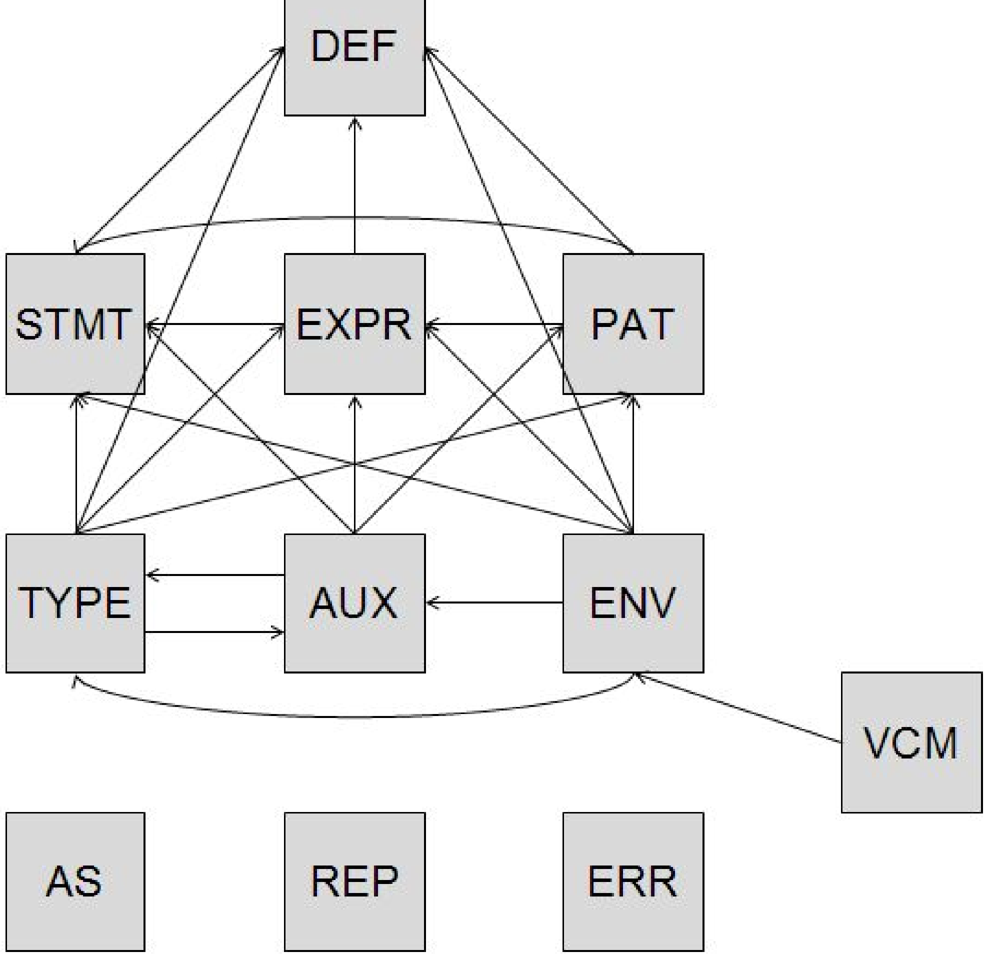


Figure 32: Module structure of the VDMTools static semantics checker

TYPE: Checks compatability between types, depending on the level of type-checking performed (POS or DEF).

AUX: Auxillary functions, mainly responsible for composing collections of types into a new type according to the rules of the given construct.

ENV: The ENV module describes the structures necessary to maintain the state of the static semantics. It contains lookup operations, scope manipulation operations and state initialization functions

REP: The internal representation of types.

ERR: The ERR module tracks errors found during the static semantics check.

VCM: Version Control Manager. This is not part of the static semantics checker as such, but provides an interface to VDMTools.

AS: Contains the abstract syntax for VDM++.

Modules AS, REP and ERR are imported by all other modules (not shown on the diagram).

5.2.3 How it works

```
wf_Class : TYPE`Ind * AS`Class ==> bool
wf_Class (i, class) ==
  ( dcl reswf : bool := ExpandClass(i, class);
    -- check that super classes exists
    for cls in class.supercls do
      if not CheckClassName(cls) then
        ----------------------------------------
        -- Error message #1
        -- The class "%1" has not been pre-checked
        ----------------------------------------
      ( ERR`GenErr(cls,<ERROR>, 1, [PrintName(cls)]);
        reswf := false
      );
    if IsLocalSuperSuper(class.nm) then
        ----------------------------------------
        -- Error message #2
        -- Circular inheritance dependecy detected
        ----------------------------------------
      ( ERR`GenErr(class.nm, <ERROR>, 2, []);
        reswf := false
      );

    if class.defs <> nil then
      let mk_AS`Definitions(typem,-,fnm,opm,instvars,-,
                            syncs,thread,-) = class.defs in
      ( reswf := wf_TypeDefs (i, typem) and reswf; --OK
        reswf := wf_InstanceVars(i, instvars) and reswf;
        reswf := wf_Functions (i, fnm) and reswf; --OK
        reswf := wf_Operations(i, opm) and reswf;
        reswf := wf_Sync (i, syncs) and reswf;
        reswf := wf_Thread (i, thread) and reswf;
      );
    InitEnv();
    return reswf
  );
```

Figure 33: The wf_Class operation

The `wf_Class` operation is the entry point into the static semantics checker. The operation is called with two arguments, a class definition (`AS‘Class`) and a union type (`TYPE‘Ind`) indicating the level of static checking to perform (POS or DEF). The operation declares a boolean `reswf`, to track the success of the operation.

The `ExpandClass` operation extracts information known at parse time, to the `ParseTypeEnv` structure.

```
(From the VCM Module)
ParseTypeEnv  = map AS‘Name to ENV‘ParseTypeInfo

(From the ENV Module)
ParseTypeInfo :: nm   : AS‘Name
                 super : set of AS‘Name
                 tps   : map AS‘Name to AccessTypeRep
                 vals  : map AS‘Name to AccessTypeRep
                 insts : map AS‘Name to AccessTypeRep
                 fcts  : map AS‘Name to AccessFnTypeRep
                 polys : map AS‘Name to AccessPolyTypeRep
                 ops   : map AS‘Name to AccessOpTypeRep
                 tags  : map AS‘Name to AccessFieldRep
                 overloaded : map AS‘Name to
                              set of (AccessOpTypeRep |
                                      AccessFnTypeRep |
                                      AccessPolyTypeRep)
                 thread : <NONE> | <PER> | <DECL>
                 constructors : set of AccessOpTypeRep
                 expanded : bool;
```

Figure 34: The parse type environment

The `ParseTypeEnv` (PTE) maps a class name to a `ParseTypeInfo` (PTI) record. The PTI contains:

- The name of the current class and the transitive closure of its superclasses.
- The name, type, access modifier and static flag of all user-defined types, instance variables, values, functions and operations.
- The name and access modifier of all record fields.

- A map of overloaded functions and operations.
- Status of the current thread (None, periodic or declared)
- Set of constructors.
- A boolean indicating if the class has been sucessfully expanded.

After the definitions in the class have been expanded to the PTE, the set of superclasses is checked to determine if they have been expanded. If this is the case, a check for circular inheritance is performed and finally the definitions comprising the class are recursively checked for well-formedness. The functions and operations defined in the other modules shown in figure 32 are invoked as necessary during the traversal of the AST.

5.3 Converting the necessary parts

Due to time constraints, converting the entire static semantics checker for the full VDM++ language was not feasible. In order to allow the desired experiments in phase 2, the conversion will be restricted the parts of the static semantics checker that are necessary to support testing of the new language features. As will be described in more detail in a later chapter, the OML grammar was extended with two additional constructs. The first, a *typeless explicit function*, which is essentially an explicit function with the type declaration stripped off. The second, class definitions with *type parameters* similar to the style of Generic Java [GW98b, GW98a].

A subset of the static semantics checker had to be converted to allow for testing of simple expressions and the type inference mechanism that would be necessary to support the two new language features.

5.3.1 The new structure

It was convenient to make use of the automatically generated AST visitor class and divide the old `wf_Class` into a number of visitors.

ParseEnvUpdater : Adds definitions to the `ParseTypeEnv` structure. Replaces the first part of the `ExpandXXX` operations (Where XXX is a definition type, eg. Types, instance variables etc.). Information is extracted from each definition type as the ParseEnvUpdater visits the relevant node.

Expander : The expander installs each definition type into its respective environment, ie. functions are installed in the `FunctionEnv`, constants (values) in the `ConstEnv`. Composite types (records) are analysed and their fields added to the TagEnv. The signatures of function pre- and post-conditions are also added to the `FunctionEnv`.

WellFormednessChecker : Performs the well-formedness checks necessary to declare the specification statically correct. The WellFormednessChecker checks structural properties such as the number of actual parameters matching the formal parameters of a function, as well as semantic checks such as typechecking.

TypeInferenceVisitor : Traverses the AST and assigns type variables to identifiers and expressions. Generates constraints involving expressions and their subexpressions.

Unification : The generated constraints are attempted solved using a unification algorithm. If the constraints are sucessfully solved the related AST nodes are annotated with their inferred type

Type Propagation: Gathers information from subexpressions and creates a resulting type for functions.

The TypeInferenceVisitor is activated if implicitly typed functions are detected in the AST. The function is annotated with its inferred type and is then treated as a normal explicitly typed function.

5.4 Conversion results

5.4.1 Supported language features

To be able to create interesting examples, we needed to convert a sufficiently large subset of VDM++. Table 35 summarizes the language constructs that are supported in the various phases of the static semantics checker.

Feature / Phase	ParseEnvUpdater	Expander	WellFormedness	Type inference
Explicit functions	•	•	•	-
Typeless explicit functions	•	•	•	•
Basic types	•	•	•	•
User-defined types	•	•	•	-
Instance variables	•	•	•	-
Values	•	•	•	-
Class parameters	•	•	•	-
Operations	-	-	-	-
Threads	-	-	-	-
Synchronization	-	-	-	-
Implicit functions	-	-	-	-

Figure 35: Supported language features

The main goal of allowing implicitly typed functions is achieved using the type inference engine implemented for this thesis. However, there exist many possibilities for extending the range and usefullness of the type inference machinery to include more of the language features.

5.4.2 A new design

The module structure of the previous static semantics checker, shown in figure 32, has been replaced by a new design leveraging the AST visitor class, generated by the parser.

The main modules implementing the well-formedness checks, `DEF`, `EXPR`, `PAT` and `STMT`, have been translated from VDM-SL to VDM++ and merged into a

Old Structure(s)	**New structure(s)**
EXPR STMT DEF PAT	WellFormednessChecker
ExpandClass	ParseEnvUpdater Expander
AS	New AST classes
REP	REP
AUX	AUXIL
ENV	ENV
ERR	ERR
VCM	VCM
TYPE	TYPE

Figure 36: Old and new code modules/classes

single class, `WellFormednessChecker.vpp`. The previous `ExpandClass` operation and its related sub-procedures have been translated and split into `ParseEnvUpdater.vpp` and `Expander.vpp`. The old `AS` module, representing the old AST, is naturally replaced by the 204 new generated AST classes. The remaining modules, REP through TYPE mainly containing supporting functions, have been converted from VDM-SL modules to VDM++ classes and the necessary functions translated.

5.4.3 Conversion summary

In total, approximately 5100 lines of VDM-SL code were converted to VDM++. Approximately 4500 lines of new code was added to support the conversion and the experiments performed in this thesis.

Chapter 6

Part 2 - An investigation of two new language features

At this point it is not a question as to whether or not it is possible to allow implicitly typed functions into OML. By choosing a sufficiently small and well-known subset of the language, we can add type inference without gambling with the soundness of the type system. However, OML is a large language with many unusual constructs compared to ordinary, general purpose programming languages. Furthermore, OML is specification language, and as such has many constructs that operate at a higher level of abstraction than ordinary programming languages.

The use of type and function invariants as well as untagged union types, are issues that need to be investigated with respect to their interoperability with type inference. This chapter will investigate the theoretical aspects of the above mentioned issues and attempt to clarify any problems and limitations.

We also take a look at generic classes and investigate whether this is a feasible extension to OML.

6.1 Implicitly vs. explicitly typed functions

As was mentioned in section 1.2.1, one of the goals of this thesis is to investigate the possibilities of introducing implicitly-typed functions. In Figure 37, the polymorphic `AppendToSeq` function is explicitly instantiated in line 10, using a type parameter, `[int]`. In Figure 38, the `AppendToSeq` function is not instantiated prior to being called with a different arguments. Here, the omitted type parameter and the function type of the `AppendToSeq` function must be reconstructed by the type-checker.

```
class PolyFunctionTest1

functions
  AppendToSeq[@param] : seq of @param * @param -> seq of @param
  AppendToSeq (sq, elem) == sq ^ [elem];

  doTest : () -> int
  doTest() ==
    let s = [1, 2, 3],
        a = AppendToSeq[int]
    in
      a(s, 42);

end PolyFunctionTest1
```

Figure 37: Explicit instantiantion of a polymorphic function in OML

```
functions

  AppendToSeq (sq, elem) == sq ^ [elem]

  doTest : () -> int
  doTest() == AppendToSeq([1, 2, 3], 42);

  doTest : () -> bool
  doTest() == AppendToSeq([true, false, false], true);

  doTest : () -> char
  doTest() == AppendToSeq(["a", "b", "c"], "d");
```

Figure 38: Example from Figure 37 as an implicitly typed polymorphic function

6.1.1 New language construct

To support experimentation with implicitly-typed functions, a new language construct, *typeless explicit functions* (TEF) were added to the OML grammar.

```
Foo (x, y, z) == x + y + z
```

Figure 39: An untyped explicit function

As the name implies, it is an explicit function with the type annotations removed. Figure 40 shows the differences in the specifications of the AST nodes in the OML grammar.

```
ExplicitFunction ::
  identifier : Identifier
  type_variable_list : seq of TypeVariable
  type : Type
  parameter_list : seq of Parameter
  body : FunctionBody
  trailer : FunctionTrailer;

TypelessExplicitFunction ::
  identifier : Identifier
  parameter_list : seq of Parameter
  body : FunctionBody
  trailer : FunctionTrailer;
```

Figure 40: AST specification for ExplicitFunction and TypelessExplicitFunction

The typeless explicit function (TEF), is identical to a normal explicit function except it lacks a type variable list and a function type. The type of the function is determined by type inference.

6.2 Language issues

A number of OML language constructs present us with some issues when considering type inference.

6.2.1 Union Types

A union type represents the ordinary union of the set of values belonging to type T_1 and the set of values belonging to type T_2, with no tag to identify the origin of a given element [Pie02].

$$T_1 \cup T_2 \qquad \text{(Type Union)}$$

6.2.2 Issues with union types

```
f[@p] : seq of @p -> @p
f(x) == if len x = 1
        then x(1) + 1
        else x(2) or false;

let a = [true, 87]
   in f(a)
```

Figure 41: Type inference with union types

Solving the type constraints in figure 41 gives us the result that the type of `x` can be either `seq of nat` or `seq of bool`. Normally, a type inference algorithm would stop at this point an report a type error as `x` cannot be assigned a unique type.

However, in the presence of union types, we are given a little more freedom. If, somewhere in our specification, we had defined a union type as in Figure 42, we would be able to infer that `x` could be of type `natbool`. This is not unsound, as we infer an existing, user-defined type.

```
types
  natbool = bool | nat
```

Figure 42: A union type definition

Ethical aspects The "ethical" aspects of doing this are questionable however. A user-defined union type may be intended for use in a specific modelling context. Using this type in another, possibly unrelated context may make our modelling intentions unclear. To mitigate this concern, we can have user access modifiers (private, public, protected) to control whether the inference algorithm may use the predefined type in another context.

Nominal vs. structural type systems Another issue related to this is that if we have defined two different union or record types consisting of the same sub-elements as in figure 43, the inference algorithm may infer that a variable has type `bool|char|real`, but will be unable to choose which precise type `unionFoo` or `unionBar` is the correct one. The name difference is significant in a *nominal* type system where types can be compared by their name alone. In a *structural* type system, the names are merely cosmetic, and `unionFoo` and `unionBar` would be just two aliases for the same underlying type.[Pie02]. Similarly for record types eg. `recordBaz` and `recordBoo`.

```
types
      unionFoo = bool | char | real

      unionBar = bool | char | real

      recordBaz ::
            field_a : nat
            field_b : bool

      recordBoo ::
            field_a : nat
            field_b : bool
```

Figure 43: Union and record with the same sub-elements

Retro-fitting If there is no suitable, previously defined type eg. `natbool`, the inference algorithm can create one for us by simply defining a new union type consisting of the incompatible types. The consequence of allowing this is that all specifications are guaranteed to be statically type correct since we can create suitable union types on the fly whenever necessary. Obviously, this would lead to a diluted and essentially useless type system and we should report a type error instead.

Non-disjoint union types Another potential issue with union types, is the question as to whether we can permit ourselves to merge inferred non-disjoint union types to the most general type.

```
class PolyFunctionTest6

types
  intreal = int | real

functions
  f[@p] : seq of @p -> @p
  f(x) == if len x = 1
          then x(1) mod 2
          else x(2) + 76;

  doTest : () -> int
  doTest() == let a = [42.1 , 87 ]
              in f[intreal](a);

end PolyFunctionTest6
```

Figure 44: Merging disjoint union types

In figure 44 the two numeric operators used in function `f` are

- `mod : int * int -> int`
- `+ : real * real -> real`

We may infer that `x` is a `seq of int` or a `seq of real`. If we allow it, we may also infer that `x` is of the user-defined type `seq of intreal`. However, since the type `int` is a proper subset of the type `real`, we could also merge the inferred types and declare that `x` is a `seq of real`. In the above example, this eliminates a union type and simplifies the inferred type of `x`. It also discards potentially useful type information which may be applied by the inference algorithm elsewhere in the code.

On the safe side To be completely safe, we can allow only operations on union types which are valid for **all** member types in the union. This may be overly restrictive in practice. A better solution would be to just allow the programmer to keep track of this information himself. On the unsafe side we have the related untagged `union` types in the `C` language, which are a potential source of safety violations and "implementation dependant" results because they allow any operations on an element of $T_1 \cup T_2$ that makes sense for *either* T_1 or T_2. [Pie02, KR99]

$$\begin{tikzcd}[row sep=small]
\top \arrow[d, dash] \\
\text{real} \arrow[d, dash] \\
\text{rat} \arrow[d, dash] \\
\text{int} \arrow[d, dash] \\
\text{nat} \arrow[d, dash] \\
\text{nat1} \arrow[d, dash] \\
\bot
\end{tikzcd}$$

Figure 45: Partial type lattice for OML numeric types

6.2.3 Invariants

Type invariants

As shown in figure 25 user-defined types may have arbitrarily complex invariants imposed on them. We are in general unable to evaluate these type invariants statically. The question arises, is it possible to infer that a variable belongs to a user-defined type with a type invariant ? In the example in figure 25, the type `SmallNat` is a user-defined subtype of the type `nat`. Since we cannot statically evaluate the predicate that restricts the type, we can in general only consider `SmallNat` to be a nat.

Given the subtype `NegativeInt` shown in figure 46 and a partial type lattice for OML, shown in figure 45, showing only the numeric types ordered by subset inclusion, where can we place `NegativeInt` ? We know that it is a subtype of `int`, but since we can't evaluate the restriction predicate, we cannot just place it below `int` and above `nat`, since `nat` is not a subset of `NegativeInt`.

```
NegativeInt = int
inv n == n < 0
```

Figure 46: Type invariant

Instead it would be more correct to place `NegativeInt` between bottom(⊥) and `int`. Indicating that it is a subtype of `int`, but that we don't have (and can't get) enough information to place it anywhere else.

This poses a problem during type inference. Given that we are in a situation where we can unify a `nat` with a `NegativeInt`, how should we proceed ? The inference algorithm will always infer the principal type of any expression, ie. the most general solution. The most general solution in this case, is the least upper bound in the type lattice of `nat` and `NegativeInt`, which is `int`. This amounts to ignoring completely the type invariant, and just treating it as its base type, in this case `int`. Imagine if we tried to unify `EvenInt` and `OddInt`, only to get the most general type `int`.

In a modelling laguage such as VDM, type invariants are created for a reason, namely to define constraints and model invariant properties of a system. Ignoring these constraints undermines the entire formal modelling paradigm and is not an option.

Function invariants

Functions and operation can also have invariants in the form of pre- and post-conditions.

```
instance variables
  balance : int
  limit : int

operations
  Withdraw : nat ==> int
  Withdraw(amount) ==
    (balance := balance - amount;
     return balance
    )
pre balance - amount > limit
```

Figure 47: Operation with a pre-condition

In figure 47 [FLM+05], the operation `Withdraw` has an invariant defined on it. Again, we cannot evaluate these invariants statically. However we might be able to extract some useful information from the invariant predicate to help us during type inference. For instance we could extract the constraint that `balance` and `amount` must be of type `real` (we already know that balance is of type `int`). Likewise, `limit` must also be of type `real` since the operator `>` is of type `real * real -> real`. In this case we could choose to respect the programmer's annotations and let `balance` remain an `int` or we could infer the most general type of the function to be `real -> real`. In any case, we can use the information in the function invariant, since it mentions some of the variables used in the function. Thus, we can use the pre- and post-conditions to extract additional constraints that could help during type inference.

6.3 Generic classes

6.3.1 Example : generic classes in OML

To support generic classes the grammar was extended as shown in figure 53. A example of the usage of a generic clas is shown in figure 48.

```
class GenericClassTest8 <[@E]>

  instance variable
    ivList : seq of @E := [ ];

end GenericClassTest8

class tester

instance variables
    iv1 : GenericClassTest8 <[ int ]>;
    iv2 : GenericClassTest8 <[ char ]>;
    iv3 : GenericClassTest8 <[ seq of bool ]>;

end tester
```

Figure 48: Usage of generic class in OML

In Java, generics classes are commonly used to support collection classes, eg. generic lists, sets and maps.

Chapter 7

Part 3 - Implementation

The third goal of this thesis involves creating a proof of concept of the ideas presented in chapter 6. This involves implementing a type inference engine for a subset of OML and an infrastructure to suport parameterized classes. This chapter will detail the implementation of both of these.

7.1 The type inference engine

The purpose of creating a type inference engine was to experiment with implicitly typed functions as a new language feature of OML. This has been implemented for a small subset of OML. Equally important is that it can be extended to a larger subset of the language at a later stage. The type inference engine is implemented as three VDM++ classes, `TI`, `TypeInferenceVisitor` and `TypePropagation`.

7.1.1 Grammar additions

For the purpose of testing type inference in this thesis, a new function shape, `TypelessExplicitFunction` was added to the OML grammar, the grammar addition is described in section 6.1.1.

7.1.2 Type variable annotation

The first stage of the type inference procedure is to annotate functions, identifiers and expressions with a type variable. This is the first task performed by the `TypeInferenceVisitor`. Type variables are generated by a call to the `getNewTypeVar()` function which increments a global, static variable and returns a unique `nat1`. This number is recorded in a `TypeInfo` class which is then associated with the particular AST node. The association between the type variable, the AST node and later, its inferred type is recorded in a map, (`TI‘tyVarMap`) which is used later during the type propagation phase.

7.1.3 Constraint generation

Having annotated expressions with type variables, the next task is to generate constraints using the type variables. This is also done by the `TypeInferenceVisitor`. For each language construct, an operation is defined in the `TI` class, taking the AST node as an argument and producing a set of constraints.

For example, for the sequence length expression, the operation gcSeqLenExp() (gc = generate constraints), adds the necessary constraints to the constraint set.

```
public static
gcSeqLenExp : IOmlUnaryExpression ==> ()
gcSeqLenExp(unexpr) ==
  ( let unOp : IOmlUnaryOperator = unexpr.getOperator(),
        unOpStr : seq of char = unOp.getStringValue(),
        exp : IOmlExpression = unexpr.getExpression()
    in
      updateConstraintSet({
        mk_Constraint(mk_tyVar(getTypeVar(unexpr)),mk_tyIdx(6)),
        mk_Constraint(mk_tyVar(getTypeVar(exp)),mk_tyIdx(7))
      })
  );
```

Figure 49: Operation generating constraints for the sequence length expression

The sequence length expression has the signature `seq -> nat`, and has a the type rule:

$$\frac{\vdash s : \texttt{seq of } \alpha}{\vdash \texttt{len } s : \text{nat}} \qquad (\text{Sequence length})$$

The first constraint generated by the gcSeqLenExp() operation constrains the type of the whole expression (unexpr) to be of type nat (represented by type index 6). The second constraint states that the type of the expression must be a sequence (type index 7). This corresponds closely with the corresponding type rule.

Constraints are represented as a record with a left field and a right field, both of type `Term`. A `Term` can be either a `tyVar`, representing a type variable or a `tyIdx` representing a type. To make comparisons easier during unification, it was decided to represent types internally as numbers (`nat1`), this may also prove more efficient when interpreted compared to say, string comparisons.

Thus, for each language construct, a constraint generation operation is invoked which adds constraints to the constraint set. A special case is the apply expression, which in the case of the expression being the name of a function, adds constraints for each corresponding pair of formal and actual parameters.

$$\mathrm{C}(X_1, ..., X_n) = \mathrm{C}(Y_1, ..., Y_n)$$

Where C is the name of a function and X_1 to X_n are the formal parameters and Y_1 to Y_n are the actual parameters. Thus, n constraints are added of the form $X_i = Y_i$.

```
constraintSet : set of Constraint;

Constraint ::
       lhs : Term
       rhs : Term;

Term = tyIdx | tyVar;

tyVar :: var : nat1;

tyIdx :: idx : nat1;
```

Figure 50: Representation of constraints

7.1.4 Unification

After all the constraints have been generated for the program, the `Unify()` operation is invoked. The `Unify()` operation transforms the constraint set into a sequence and calls the `UnifyAux()` operation, where the actual unification algorithm is applied to the constraint sequence.

```
if true
then len [1, 2, 3]    -- [nat]
else 3 * 42           -- [real]
```

Figure 51: Merging numeric types

The `UnifyAux` operaration as shown in figure 52, is an implementation of a unification algorithm by Martelli and Montanari [MM82]. The algorithm proceeds by getting the first constraint from the sequence. If the constraint has a `tyIdx` on both the left and right sides, corresponding to a comparison of two types, eg. `[real] = [nat]`, the `MergeOverlappingNumericTypes()` operation is called. This operation checks if the two types are numeric, if so, the most general type wins and replaces the more restrictive type. Thus `[real] = [nat]` would be replaced by `[real] = [real]`.

```
public static
UnifyAux : seq of Constraint ==> bool * [Term] * [Term]
UnifyAux(conseq) ==
    if len conseq = 0 then
      return mk_( true, nil, nil )
    else
      let con : Constraint = hd conseq,
          left : Term = con.lhs,
          right : Term = con.rhs
      in
        let mk_(left, right) =
            MergeOverlappingNumericTypes(left, right)
        in
          if left = right then       -- Case 1
            UnifyAux( tl conseq)
          elseif is_tyVar(left) then -- Case 2
            ( let conSeq' = ApplyStackReplacement(conseq, left,
                right)
              in
                (ApplySubstitutionReplacement(left, right);
                 UnifyAux(conSeq'))
            )
          elseif is_tyVar(right) then -- Case 3
            ( let conSeq' = ApplyStackReplacement(conseq, right,
                left)
              in
                (ApplySubstitutionReplacement(right, left);
                 UnifyAux(conSeq'))
            )
          else
            return mk_( false, left, right);
```

Figure 52: The `UnifyAux` operation

If this was not done, the expression in figure 51 would not be typeable as the types of the two branches of the *if* expression would be considered different. This is consistent with the current static semantics checker in VDMTools, where the example in figure 51 is considerd well-typed both in `POS` and `DEF` mode.

The algorithm then proceeds by case analysis of the two terms. In case one, the two terms are equal. In this case the equations are implicitly unifyable, both in the case of two types and in the case of two type variables. Two equal terms do not however, provide any new information and are therefore deleted without changing the substitution or stack.

In case two and three, either one or both of the terms is a type variable. In case two, the left term is substituted for the right term both in the remaining constraint sequence and in the substitution set. Symmetrically for case three. In both cases the terms are added to the substitution set.

If none of rules one to three can be applied, we have a situation where we are attempting to unify two different, incompatible types, eg. `[bool] = [real]`. In this case, the algorithm halts with failure and reports the constraint that could not be unified.

If the unification is sucessful, the `TypeInfo` structure associated with each AST node is updated with the inferred type of the expression represented by the node.

7.1.5 Type propagation

In a final pass of the type-annotated AST, the `TypePropagation` visitor creates the resulting function type of the typeless explicit functions, by gathering the inferred types of its parameters and the inferred type of the function body. The function's `TypeInfo` is updated and the previously implicitly-typed function can now be safely converted to a corresponding explicitly-typed function, in preparation for future phases.

7.1.6 Supported subset of OML

Currently is is possible to write specifications in OML and test them with the type-inferencer using these language features:

- Typeless Explicit functions.
- Binary expressions (+, -, *, /, and, or, =).
- Unary expressions (len, hd, tl).
- If expressions.
- Function application expressions.

7.2 Generic classes

Supporting generic classes in OML is a sub-goal of this thesis, and has been partially implemented. However it is our belief that generic classes are a viable language feature which has the potential to be useful to OML users, as it is to Java users today.

7.2.1 Grammar additions

To support experimentation with generic classes, the existing grammer definition of a class was extended with type parameters as seen in figure 53.

```
Class ::
  identifier : Identifier
  generic_types: seq of Type         -- Added
  inheritance_clause : [InheritanceClause]
  class_body : seq of DefinitionBlock
  system_spec : bool;
```

Figure 53: Extended class definition in the OML grammar

7.2.2 Static checks

To check that generic classes are used in the correct fashion, a number of static checks are performed by the `WellformednessChecker` on various parts of the code related to generic classes. The following list of error messages illustrate which checks are currently performed.

Formal class parameters

- Class formal parameter name must be different from an existing class name.
- Class formal parameter name may not be the name of a user-defined type.
- Class formal parameter name may not be the name of an OML type.
- Class formal parameter name must be a type variable, prefixed with a @.
- Class formal parameter name must not be a previously used identifier.

Class type instantiation expressions

- Class instantiation expression is attempting to instantiate a non-existing class.

- The number of actual parameters does not match the number of formal parameters in the class instantiation expression.
- Function or operation types cannot be used as class type parameters.
- The empty type or optional types cannot be used as class type parameters.
- Nested parameterized classes are not permitted.

Class member static access

- Type definitions are static and may not use class type parameters.
- Static function definitions may not use class type parameters.
- The names of the type parameters of polymorphic function must be different from the names of the class formal parameters.
- Static instance variable definition may not use class type parameters.
- Static value definition may not use class type parameters.

The formal class parameter checks, verify that the name of each formal class parameter is a valid, unique and previously unused identifier. The class type instantiation expressions checks verify that the actual type parameters supplied to the class are valid and are of a permitted type. The static access checks basically check that class type parameters are not used in a static context. Initializing a static class member with a class type parameter would allow static members to be different across instances of the class, thus conflicting with their static status.

Chapter 8

Discussion and Conclusion

We will look at the results of our investigation, and discuss them. We will evaluate the project as a whole and look at what experiences we have gained during the project. We will suggest directions for future work and provide a conclusion to the report as a whole.

8.1 Summary and discussion of results

8.1.1 Phase 1 - Converting the original specification

In phase one, the goal was to convert a subset of the original VDM++ static semantics checker written in VDM-SL, to an equivalent static semantics checker written in VDM++.

This phase was by far the most time-consuming activity of the project. At least 50% of the time spent has been used in converting the old specification to a new one. The outcome of this phase, a static semantics checker written in VDM++, is also the output from this thesis that potentially carries most benefit for the Overture community. The static semantics checker can, when code-generated to Java, be integrated into the Overture Eclipse plugin and become part of the new Overture tool chain.

Design

The old static semantics checker was, despite its rather monolithic nature, not badly structured (albeit the comment to code ratio was quite low). We believe however, that the new design, described in section 5.4.2 using AST visitors provides a more intuitive approach to traversing the AST, as well as a being more maintainable.

We also believe that breaking up the monolithic structure, will be an advantage when using the Eclipse plugin architecture. Smaller, decoupled sub-plugins are easier to extend and maintain than a single plugin encompassing the entire static semantics checker.

Technical issues

Technically, converting the non-object-oriented VDM-SL code to object-oriented VDM++, at times made us wonder if it would be quicker to write a VDM-SL to VDM++ converter instead. The VDM-SL code made extensive use of pattern-matching to, for instance, quickly dissect an AST node into its sub-expressions, as shown in figure 54. The `AS‘IfExpr` record is matched to `mk_AS‘IfExpr(test,cons,elsif,altn,cid)`. Since pattern matching cannot be used on objects, the corresponding VDM++ operation requires "manually" extracting the subexpression using get operations, as shown in figure 55.

```
wf_IfExpr : TYPE‘Ind * AS‘IfExpr * REP‘TypeRep ==> bool *
    REP‘TypeRep
wf_IfExpr (i, mk_AS‘IfExpr(test,cons,elsif,altn,cid),tp) ==
```

Figure 54: Beginning of the VDM-SL `wf_IfExpr` operation

```
wf_IfExpression: IOmlIfExpression * REP‘TypeRep ==> bool *
    REP‘TypeRep
wf_IfExpression (ifExp, tp) ==
    ( let test : IOmlExpression = ifExp.getIfExpression(),
          cons : IOmlExpression = ifExp.getThenExpression(),
          elsif : seq of IOmlElseIfExpression =
              ifExp.getElseifExpressionList(),
          altn : IOmlExpression = ifExp.getElseExpression()
      in
```

Figure 55: Beginning of the VDM++ `wf_IfExpr` operation

Although this is not difficult, it is less elegant than pattern matching and it produces a lot of boilerplate code. The initial part of the VDM++ operation could probably be automatically generated as a visitor with the subexpressions "pre-extracted". In general the task of translating VDM-SL code to VDM++ code could be automated. This would also be consistent with the implementation strategy of the Overture group, ie. automate as much of the tool-chain as possible. We defer this task to the "Future work" section.

Although the old `AS` module representing the AST, and the new AST classes were to a large extent the same, there were a number of changes to the new AST. To be entirely correct, the old VDM-SL specification was a specification of a VDM++ static semantics checker, while the new specification is a specification of an OML static semantics checker. While OML is 95% identical to VDM++, there are some differences in the grammar and the AST. These differences had to be handled and reconciled with the old specification.

8.1.2 Phase 2 - Investigating two new language features

In phase two, the task was to investigate the possibilities for

- Adding implicitly typed functions to OML.
- Adding parameterized classes in the style of Generic Java.

Implicitly typed functions

The idea of adding type-inference to OML was presented to the Overture community at the Newcastle Overture workshop. From the discussion we had with the participants, there were mixed responses as to the usefulness of this feature. OML is a large language to start off with, do we really need more language features ? Explicit typing is a valuable tool in model specification and is an important source of documentation of the specifier's intention. What will happen when the specifier annotates a variable with the type he wants it to have and the type inferencer alters it to a more general type ? Will the constraints imposed by the explicit types be weakened if we allow the type-inferencer to generalize at will ? All these questions and more, need to be settled by the Overture community.

Given the discussion in chapter 6, the restrictions that may have to be imposed on the user in order to allow type-inference, may be too restrictive to make implicitly-typed functions useful in practice. Union types and type invariants are commonly used in VDM specifications. Requiring the user to restrict his use of these constructs or omit them entirely, would definitely hinder the adoption of this feature if it were to be developed further.

Generic classes

We believe that generic classes in OML would provide enough benefit to justify working on implementing them fully. VDMTools supports code generation to Java. It is also the intention that Overture will support this as well. Generic classes in OML would allow a direct mapping to a corresponding, generic Java class.

8.1.3 Phase 3 - Proof of concept implementation

We implemented a type-inference engine and constraint-generation operations for a small subset of OML as described in section 7.1.6. We would have liked to implement a larger subset but this was not realistic due to time constraints. However, we regard the implementation implementation as being sufficiently general as to easily support extensions with more language constructs at a later time.

For the generic classes, we implemented a series of static checks to check the well-formedness of specifications using generic classes. More static checks are needed as well as extensions the code-generator and possibly also the interpreter. Theses extensions are out of scope of this thesis. As mentioned above, we believe that this feature is worth implementing fully and that it will confer similar benefits to OML users as it currently does to Java users.

8.2 Personal view

8.2.1 Choice of project

I chose this project because it included elements from all the subjects I found interesting during my studies at DAIMI, programming languages, semantics, compiler design and a touch of static analysis. Also because I had never heard of VDM or formal methods before I read a project description as part of the Master thesis preparation course.

8.2.2 Overture Workshop

As I was working on a part of the Overture project, I was invited to present my work at the third Overture Workshop held at The University of Newcastle during the period 27th - 28th of November 2006 [Prob]. This was a great learning experience, with the possibility to discuss programmings languages, VDM semantics and related topics with academic and private researchers working on formal methods and/or VDM. Also it was interesting to get a look at how an academic workshop is held.

Working with VDM has also been given me valuable insight into the area of formal methods and software engineering using these methods. The jump from the type-checker we made in the compilation course at DAIMI, which covered at most 10 A4 pages, to the industrial-strength type-checker used in VDMTools/Overture, covering about 600 A4 pages was quite awe-inspiring. Despite the fact that the future might bring automatic conversion of the old static semantics checker, the feeling of having gotten my hands dirty under the hood of a powerful, commercial language-tool, remains.

8.3 Future Work

Future work along the lines of this project could be.

Finish the conversion The conversion done in phase 1 of this thesis, was focused on the language structures that are most commonly used, such as types, instance variables, values and explicit functions. What remains is operations, implicitly-defined functions, thread definitions and synchronization constraints. If the remaining conversion is not done manually, it would be a good idea to build a reliable source-to-source translator for VDM++/VDM-SK/OML.

Generic types If Generic types are to be fully implemented in OML, there are issues to be resolved with respect to code-generation. VDMTools currently supports code-generation to Java, the addition of generics will require the code-generator to be updated to handle this.

Bibliography

[Aic97] Bernhard Aichernig. A Proof Obligation Generator for the IFAD VDM-SL Toolbox. Master's thesis, Technischen Universität Graz, 1997. [cited at p. 21, 25]

[Bowa] Jonathan Bowen. URL to list of Formal methods at Formal Methods Europe. Website. `http://vl.fmnet.info/`. [cited at p. 3]

[Bowb] Jonathan Bowen. URL to The B Method homepage. Website. `http://vl.fmnet.info/b/`. [cited at p. 4]

[Bowc] Jonathan Bowen. URL to The Z Notation webpage. Website. `http://vl.zuser.org/`. [cited at p. 3]

[Bro86] Fred P. Brooks. No silver bullet - essence and accident in software engineering. In *Proceedings of the IFIP Tenth World Computing Conference*, pages 1069–1076, 1986. [cited at p. 1]

[BYA] BYACCJ. URL to BYACCJ webpage. Website. `http://byaccj.sourceforge.net/`. [cited at p. 27]

[Cor] CSK Corporation. Csk website. Website. `http://www.csk.com/`. [cited at p. 5]

[CSK] VDMTools Group CSK. Specification of the VDM++ Static Semantics Checker. Confidential commercial document, not generally available. [cited at p. 5, 30]

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985. [cited at p. 17, 19]

[DCaAE] D.Bjørner, C.B.Jones, M.Mac an Airchinnigh, and E.J.Neuhold, editors. Number 252 in Lecture Notes in Computer Science. Springer. [cited at p. 4]

[DHB91] Flemming M. Damm, Bo Stig Hansen, and Hans Bruun. On Type Checking in VDM and related Consistency Issues. Technical Report 1991-88, Technical University of Denmark, 1991. [cited at p. 22]

[Ern] Erik Ernst. URL to the Master Thesis Preparation course at DAIMI. Website. `http://www.daimi.au.dk/specialeforberedelse/`. [cited at p. 30]

[Fit] John. S. Fitzgerald. The Typed Logic of Partial Functions and the Vienna Development Method. To be published. [cited at p. 4]

[FLM$^+$05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, first edition, 2005. [cited at p. 8, 46]

[FOR98] A Formal Definition of VDM-SL. Technical report, IFAD and Technical University of Delft and Technical University of Denmark and The University of Leicester, 1998. [cited at p. 19]

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pattens - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. [cited at p. 27]

[GW98a] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. GJ: Extending the Java Programming Language with type parameters. 1998. [cited at p. 35]

[GW98b] Gilad Bracha, Martin Odersky, David Stoutamire and Philip Wadler. GJ Specification. 1998. [cited at p. 35]

[HJ89] Ian J. Hayes and Cliff B. Jones. Specifications are not (necessarily) executable. Technical Report UMCS-89-12-1, Department of Computer Science, University of Manchester, 1989. [cited at p. 2, 3]

[HOL] HOL. URL to HOL webpage. Website. `http://www.cl.cam.ac.uk/research/hvg/HOL/`. [cited at p. 28]

[JFl] JFlex. URL to JFlex webpage. Website. `http://jflex.de/`. [cited at p. 27]

[KR99] Kernighan and Richie. *The C Programming Language*. ???, 9999. [cited at p. 44]

[Lar] Peter Gorm Larsen. URL to the Model driven development using VDM++ and UML 1 course at IHA. Website. `http://kurser.iha.dk/eit/tivdm1/`. [cited at p. 30]

[Luc81] Peter Lucas. Formal semantics of programming languages. *IBM Journal of Research and Development*, 25(5):549–561, September 1981. [cited at p. 4]

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science 17*, pages 348–375, 1978. [cited at p. 16]

[MM82] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258 – 282, April 1982. ISSN:0164-0925. [cited at p. 50]

[NH05] Jacob Porsborg Nielsen and Jens Kielsgaard Hansen. Development of an Overture/VDM Tool Set for Eclipse. Master's thesis, Technical University of Denmark, August 2005. [cited at p. 25]

[PBB$^+$96] P.G.Larsen, B.S.Hansen, H. Bruun, N.Plat, H.Toetenel, D.J.Andrews, J.Dawes, and G.Parkin et.al. *ISO/IEC 13817-1:1996 - Information technology – Programming languages, their environments and system software interfaces – Vienna Development Method – Specification Language – Part 1: Base language*, December 1996. [cited at p. 4, 20]

[Pie02] Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002. [cited at p. 15, 21, 41, 43, 44]

[Pre00] Roger S. Pressman. *Software Engineering - A Practitioner's Approach.* McGraw Hill, 5th edition, 2000. [cited at p. 2]

[Proa] The Overture Project. Overture project website. `http://www.overturetool.org/`. [cited at p. 5, 25]

[Prob] The Overture Project. URL to the third Overture Workshop in Newcastle, November 2006. Website. `http://www.overturetool.org/twiki/bin/view/Main/Workshop3`. [cited at p. 25, 58]

[Ter] Terma. URL to The Raise Homepage. Website. `http://spd-web.terma.com/Projects/RAISE/`. [cited at p. 4]

[TSDGaM] The Software Design Group at M.I.T. URL to The Alloy Analyzer webpage. Website. `http://alloy.mit.edu/`. [cited at p. 4]

[VDM] *The VDM++ language reference manual.* [cited at p. 8, 10]

[vdS04] Pieter van der Spek. The Overture Project; Designing an open source tool set. Master's thesis, Delft University of Technology, 2004. [cited at p. 25]

[Ver] Marcel Verhoef. URL to the personal website of Marcel Verhoef. Website. `http://www.marcelverhoef.nl/`. [cited at p. 25]

[Wik] Wikipedia. URL to Wikipedia article on Ariane 5. Website. `http://en.wikipedia.org/wiki/Ariane_5#Launch_history`. [cited at p. 1]

Appendices

Appendix 1 - CD

The attached CD contains

- The VDM++ specification of the work described in this thesis.
- The Overture parser.
- A Windows and Linux version of VDMTools. These are free to use for academic purposes.
- A collection of VDM++ manuals including the VDM++ language reference and the VDMTools user manual.