The American University in Cairo  
School of Science and Engineering  

PARAMETRIC POLYMORPHISM  
IN THE SIMPL LANGUAGE  

A Thesis Submitted to  

Computer Science Department  
In partial fulfillment of the requirements for  
The degree of Master of Science  

by  

SOUMAIA AHMED AL AYYAT  

B.Sc. Computer Science, AUC  

under the supervision of Dr. Muhammed F. Mudawwar  

November 1998
The American University in Cairo

PARAMETRIC POLYMORPHISM
IN THE SIMPL LANGUAGE

A Thesis Submitted by Soumaia Ahmed Al Ayyat
To Department of Computer Science
November/1998
In partial fulfillment of the requirements for
The degree of Master of Science
has been approved by

Dr.
Thesis Committee Chair / Advisor __________________________
Affiliation __________________________

Dr.
Thesis Committee Reader / Examiner __________________________
Affiliation __________________________

Dr.
Thesis Committee Reader / Examiner __________________________
Affiliation __________________________

__________________  _____________  _____________  ___________
Department Chair/ Date Dean Date
Program Director
Acknowledgements

I would like to thank my supervisor, Dr. Mudawwar, for inspiring and advising me in all the phases of the thesis. Also, I would like to thank all the Computer Science Department’s Professors for the knowledge they conveyed to me and for their helpful critiques.

I thank Bjarne Stroustrup for the helpful discussions and providing me with useful information via email. Many thanks to Luca Cardelli for the long discussions we had by email. I learnt a lot from his valuable published papers.

My Special thanks to my parents, Iman and Mohamed for their endurance and patience and for encouraging me. Thanks to my friend Doha for giving me the spirit to continue.

Thanks to all my colleagues in ACS for their encouragement. Thanks to Dr. Mona Kaddah for her understanding and cooperation, and special thanks to Dr. Sherif El Kassas for the long discussions we had on subtyping and for providing me with useful URLs on compilers.

Finally, many thanks to everyone supported me and endured my moments of frustration.
ABSTRACT

The American University in Cairo

School of Science and Engineering

PARAMETRIC POLYMORPHISM

IN THE SIMPL LANGUAGE

By

Soumaia Ahmed Al Ayyat

Supervised by

Dr. Muhammed F. Mudawwar

This thesis explores parametric polymorphism in the frame of the SIMPL (Simplified Imperative Modular Programming Language) programming language and relates that to the work done in other programming languages. SIMPL introduces hidden parameters inference as a new feature in parametric polymorphism. The thesis discusses new ideas of subtyping by specialization. Moreover, the thesis shows how subtyping by specialization gives power to function overloading and specialization. These ideas are backed with formal proofs. The thesis delves into the scheme of representing and processing the semantic information for every identifier. As a supplement, this research work devises algorithms for subtyping detection, ambiguous overloading rejection, function overloading resolution and hidden parameters inference. Besides, a mathematical representation of SIMPL is portrayed in a type system for this new language.
# TABLE OF CONTENTS

List of Tables 6

List of Figures 7

1. INTRODUCTION 8

   1.1. Classification of Polymorphism 8

   1.2. Problem Definition 12

   1.3. Terminology and Notation 12

      1.3.1. Important Notation 12

      1.3.2. Terminology 16

   1.4. Road Map 19

2. POLYMORPHISM IN OTHER LANGUAGES 21

   2.1. C++ 21

      2.1.1. Templates in C++ 21

      2.1.2. Template Specialization in C++ 22

      2.1.3. Inheritance in C++ 24

      2.1.4. Overloading in C++ 24

      2.1.5. Memory Management and Copy Semantics in C++ 26

   2.2. ADA and ADA95 28

      2.2.1. Subtyping in ADA 28

      2.2.2. Overloading in ADA 29

      2.2.3. Generic Units in ADA 29

   2.3. Eiffel 30

      2.3.1. Genericity 30

      2.3.2. Genericity and Inheritance 31
2.3.3. Function Subtyping 32

2.4. Java 32

2.4.1. Overloading in Java 32

2.4.2. Subtyping in Java 33

2.5. Modula-3 33

2.5.1. Subtyping in Modula-3 33

2.6. Conclusion of The Languages' Survey 34

3. SIMPL LANGUAGE FEATURES 35

3.1. What is SIMPL? 35

3.2. Interfaces and Modules 35

3.3. Parameterized Types 37

3.3.1. Type Interface and Type Implementation 38

3.3.2. Synonym Types 40

3.3.3. Union Types 40

3.3.4. Type Overloading 41

3.3.4.1. Type Signature 42

3.3.5. Type Instantiation 42

3.3.6. Subtyping 43

3.3.7. Processing Types 44

3.4. Functions 45

3.4.1. Function Declaration 45

3.4.2. Hidden Parameters 46

3.4.3. Formal Function Parameters 48

3.4.4. Result Type 50

3.4.5. Function Definition 50

3.4.6. Function Types 50
Appendix B: Unification Algorithm: for types and for function calls 134

B.1. Implementation of Some Auxiliary Functions 134

B.2. The Unify Function 136

B.3. Subtyping Check 139

B.4. Check for Unique Function Signature 142

B.5. Check for the Greatest Common Subtype 143

B.6. Function Specialization Check 144

B.7. Sorting Overloaded Functions in the Symbol Table 145
TABLES

1.1. List of Notation 15

6.1. Tags Used in the Attribute Space and their Usage 98
# FIGURES

1.1. Classification of Polymorphism .......................... 10
3.1. Semantics of Formal Function Parameters .......... 49
4.1. Subtyping by Instantiating Type Instances ........ 56
4.2. Subtype Hierarchy among Type Instances ........ 58
4.3. Subtyping of the Type Instances Listed in the Function Signatures of the Example 60
4.4. Function Signature Subtype Relationship ........ 61
6.1. Representing Objects and Named Values in the Semantic Space 80
6.2. Enumeration Literal Representation in the Semantic Space 81
6.3. Representation of the type stackIT and its Fields in the Semantic Space 84
6.4. Semantic Representation of the Union Types ........ 86
6.5. Representation of Synonym Types in the Semantic Space 87
6.6. Semantic Representation of Functions ............. 88
6.7. Representation of Constraints in the Attribute Space 90
6.8. Semantic Representation of Function Types ........ 91
6.9. Semantic Representation of Synonym Functions .... 92
6.10. Side Effects of the Unify Function on the Semantic Space 95
7.1. Covariance Relation between Function Subtyping and Signature Subtyping 105
7.2. Constraint Integrity among Function Subtypes .... 105
7.3. The Greatest Common Subtype between Two Non-related Function Signatures 107
A.1. Subtyping among Function Types .................. 132
1. INTRODUCTION

In any typed language, the type introduces a set of constraints needed to enforce correctness of the program. A type is viewed as “a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use. It provides a protective covering that hides the underlying representation and constrains the way objects may interact with other objects” [Cardelli85].

Programming languages are classified, according to the uniqueness of the type of objects and methods, into monomorphic languages and polymorphic ones. A monomorphic language is a language whose values and variables can be interpreted to be of one and only one type. Pascal is an example of such languages. In contrast, in a polymorphic language some values and variables may have more than one type [Cardelli85].

Parametric Polymorphism is the purest form of polymorphism. It enables programmers to build parameterized types and polymorphic functions. SIMPL language supports the strong features in parametric polymorphism. It implements hidden parameters inference – a new feature in parametric polymorphism – upon function calls.

1.1. Classification of Polymorphism

Polymorphism mainly means to have many shapes. Strachey informally defined polymorphic operations and routines to be the ones that exhibit several forms based on their arguments [Ditchfield94]. He has classified polymorphism into two main kinds: parametric polymorphism and ad-hoc polymorphism. Parametric polymorphism is noticed in the behavior of a function that works uniformly on a range of types that normally exhibit a common structure. Parametric polymorphic functions constitute the core of ML language [Abadi93], [Ditchfield94].
Ad-hoc polymorphism is obtained when a function works on several different types and may behave in unrelated ways for each type. A language can apply more than one kind of polymorphism.

Cardelli in [Cardelli85] refined Strachey’s classification as follows:

Polymorphism is mainly classified into universal polymorphism and ad-hoc polymorphism. Universal polymorphism contains two subcategories: parametric polymorphism and inclusion polymorphism. Ad-hoc polymorphism is divided into overloading and coercion.

Cardelli in [Cardelli85] introduced inclusion to model subtypes and inheritance and accordingly object-oriented programming. Inclusion polymorphism is coupled with polymorphic data whereas parametric polymorphism is conjoined with polymorphic routines. Through parametric polymorphism, types achieve their uniformity by type parameters. However, universal polymorphism has more ways to achieve uniformity over types. That is why, universal polymorphism includes parametric and inclusion polymorphism.

In parametric polymorphism, the polymorphic function must have a type parameter whether implicit or explicit. Upon substitution of this type parameter with a specific type, the type of the arguments is determined. Generic functions of Ada constitute an example of parametric polymorphism. Cardelli in [Cardelli85] considers value sharing to be a special case of parametric polymorphism. The symbol \textit{nil} is a good representation of value sharing. It is difficult to consider it a case of a heavily overloaded type because it is a valid element of an infinite collection of types and still all its uses indicate the same value. Parametric polymorphism is the purest form of polymorphism since the same object or function can be used uniformly in different type contexts without changes nor special encoding of representation.

The other kind of universal polymorphism is inclusion where an object is viewed as belonging to many different classes that may or may not be disjoint. Subtyping is a form of inclusion polymorphism and exemplifies true polymorphism. It is a useful technique that enables an object of a subtype to be used in a context that allows objects of the supertype. Universal polymorphism is true polymorphism.
Ad-hoc polymorphism is apparent polymorphism. Ad-hoc polymorphic functions may execute different code for each type of argument. Ad-hoc polymorphism includes overloading where the same variable name is used to represent different functions; based on the context, only one function is invoked at a time.

Overloading is a pure syntactic way of using the same name for different semantic objects; the compiler can resolve the ambiguity at compile time. Overloading is not true polymorphism since it allows a symbol to have many types, but the values denoted by this symbol have distinct and possibly incompatible types.

Coercion is a semantic operation of converting an argument to the type expected by a function. Coercion is not true polymorphism since an operator may appear to accept values of many types, but the values must be converted to some representation before the operator can use these values. That is, the actual case is that the operator works on only one type. Furthermore, the return type of the operator is not dependent on the argument type. Actually, coercion enables the user to omit semantically necessary type conversions; the compiler generates the needed type conversion code and inserts it into the program [Cardelli85].

![Classification of Polymorphism](image)

**Figure 1.1. Classification of Polymorphism**
We will focus more on parametric polymorphism. There are two ways of achieving parametric polymorphism: explicit and implicit. Explicit parametric polymorphism is obtained by explicit type parameters in function headers and the explicit application (instantiation) of the type parameters upon function calls. For instance the following function explicitly requires the type to be mentioned.

\[
\text{Function } f (t: \text{type}, a: t) \text{ return } t
\]

Possible function calls to this function can be

- \( f(\text{integer}, 4) \)
- \( f(\text{real}, 3.5) \)

Implicit parametric polymorphism is obtained by not mentioning the type parameter and instead a type variable of unknown type is used as a parameter to the function. For instance, the same function can be rewritten as

\[
\text{Function } f (a: t) \text{ return } t
\]

Where \( t \) is an unknown type variable. Moreover, the function call does not explicitly mention the type; instead it passes a value and the compiler infers the type. Examples of function calls could be

- \( f(4) \)
- \( f(3.5) \)

Implicit parametric polymorphism is considered an abbreviation of explicit parametric polymorphism. Actually, omitting type parameters leaves some identifiers that denote type unbound (the type variables). That requires type inference to recover the lost information. Implicit parametric polymorphism can totally omit type information by keeping all parameters as type variables; they appear to be type-free. Yet rigid type checking must take place. Thus an implicit polymorphic program is powerful but ambiguous; since it may be interpreted into different explicit polymorphic programs [Cardelli87].
1.2. Problem Definition

This thesis explores parametric polymorphism in the frame of the SIMPL programming language and relates that to the work done in other programming languages. SIMPL implements many features in polymorphism in addition to introducing hidden parameters inference that is a new feature in parametric polymorphism. The thesis details the scheme of representing the semantic information for every identifier. Moreover, it lists algorithms that detect subtyping, reject ambiguous overloading, resolves function overloading and infer hidden parameters. Besides, a mathematical representation of SIMPL is portrayed in a type system for this new language.

1.3. Terminology and Notation

1.3.1. Important Notation

The symbols used throughout the thesis are listed below with the meaning of each symbol and an example of its usage. This notation is used in the formal rules representing parametric polymorphism in SIMPL. Moreover, it is used in the judgements and inference rules listed in the Type System for SIMPL.

- \( \forall \) indicates ‘For all’. It is used for type variables.
- \( \exists \) means there is.
- \( \leftrightarrow \) shows that the terms are equivalent in terms of a certain type.
- \( = \) represents that two terms are syntactically identical.
- Any \( A, B \) denotes a type whether it is a type variable or a specific parameterized type or a specific non-parameterized type.
- \( \text{TYPE} \) is the set of all types. It is the universal set whose elements are type sets.
- The function signature type (the aggregate type of the function formal parameters) is denoted by \( \text{Sig} \).
• The function signature (the aggregate of the function actual parameters) is denoted by \( \text{sig} \).

• The function result type is denoted by \( \text{Rtype} \).

• Any function type is represented as a mapping from the signature type (the aggregate of formal arguments) to the result type. E.g. \( \text{Sig} \rightarrow \text{Rtype} \)

• Variables are denoted by \( a, b, x \) … etc.

• Literal values are denoted by \( # \) followed by any variable name.

• Any type instance is denoted by \( s \).

• A type variable is denoted by \( t \).

• A parameterized type declaration consists of the type name followed by a list of its formal type parameters. That is, \( s\{t, n\} \) is a parameterized type whose parameters are the type variable \( t \) and the integer variable \( n \).

• Function implementations are denoted by \( \forall \{n: \text{integer}, t: \text{TYPE}\} f(a: A, \text{val} b: B): \text{Rtype} \). The function name is preceded by the list of hidden parameters - if any – which can be type variables or integer variables. The function name – \( f \) is replaced by the function name – is followed by an aggregate of the formal parameters with their types. Next, the function body, which encloses a set of statements, is listed. Listing the function body is optional. Finally comes the result type – if it is not explicitly stated, it is considered void type. Formal parameters are preceded by their mode of access (\text{obj} or \text{val}). Within the function body, Formal object parameters allow read and write access mode to the actual parameter, whereas \text{val} parameters are pass by value, that is, read-only values.

• SIMPL deals with function types more or less as data types. Accordingly, function types can be parameterized. \( \forall \{n: \text{integer}, t: \text{TYPE}\} f(a: s1 \{t, n\}): s2 \) is a parameterized function of type \( \forall \{n: \text{integer}, t: \text{TYPE}\} (s1\{t, n\}) \rightarrow s2 \) where the signature type of the formal parameters includes the parameterized type \( s1 \) whose parameters are passed as hidden parameters to the
function. By the way, the result type can be parameterized via the same set of hidden parameters (e.g. \( \forall \{n: \text{integer}, t: \text{TYPE}\}(s1 \{t, n\}) \rightarrow s2 \{n, t\} \)). Of course, the access mode of the formal parameters and the result type is still applicable.

- \( f(s1 \leftarrow s2: \text{Sig1}) \) represents a function call with appropriate actual parameters is simple. It is a substitution of the actual parameters for the formal parameters. Actual parameters must be of the same type of the formal parameters or its subtype. The function is called with the actual parameters with no explicit mention of the hidden parameters. The type inference algorithm infers the value of the hidden parameters accordingly.

- \( \forall \{n: \text{integer}\} s1 \{a, A, s2\{A, n\}\} \) is a sample of type instantiation; where a parameterized type is instantiated with actual parameters or a function declaration as an instance of a parametric polymorphic function type. Type instantiation of data types leads to type instances of parameterized types where a type instance is a subtype of its parameterized type.

- The subtyping relationship between types A and B is represented by \( A \leq B \). That is, A can be the same type as B or can strictly be a subtype of B.

- The proper subtyping between two types. \( A <: B \) indicates that A is strictly a subtype of B and cannot be the same type as B.

- The implication rule is represented by \( p \Rightarrow q \) where p implies q.

- \( p \Leftrightarrow q \) indicates that p if and only if q. That is to say, p is a necessary and sufficient condition for q.

- \( \exists!x \) means there is a unique variable x [Barendregt84].
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>∀</td>
<td>For all. Used for type variables</td>
<td>(∀X &lt;: A) B</td>
</tr>
<tr>
<td>∃</td>
<td>There exists</td>
<td>∃</td>
</tr>
<tr>
<td>∃!</td>
<td>There is a unique variable</td>
<td>∃!x</td>
</tr>
<tr>
<td>→</td>
<td>Function mapping</td>
<td>A → B</td>
</tr>
<tr>
<td>≤:</td>
<td>Subtyping i.e. they may be equal types</td>
<td>A ≤: B</td>
</tr>
<tr>
<td>&lt;:</td>
<td>Proper subtyping i.e. they cannot be of the same type</td>
<td>A &lt;: B</td>
</tr>
<tr>
<td>↔</td>
<td>Equivalence between terms</td>
<td>a ↔ b</td>
</tr>
<tr>
<td>⇒</td>
<td>Implies</td>
<td>p⇒q</td>
</tr>
<tr>
<td>⇔</td>
<td>If and only if</td>
<td>p⇔q</td>
</tr>
<tr>
<td>≡</td>
<td>Syntactically identical</td>
<td>Ftype≡ Sig→Rtype</td>
</tr>
<tr>
<td>A, B</td>
<td>type (a monomorphic or a polymorphic type)</td>
<td>a: A</td>
</tr>
<tr>
<td>TYPE</td>
<td>the set of all types</td>
<td>t: TYPE</td>
</tr>
<tr>
<td>Sig</td>
<td>function signature type</td>
<td>f : Sig→Rtype</td>
</tr>
<tr>
<td>sig</td>
<td>function signature</td>
<td>f(sig)</td>
</tr>
<tr>
<td>Rtype</td>
<td>function result type</td>
<td>f : Sig→Rtype</td>
</tr>
<tr>
<td>Sig→Rtype</td>
<td>function type</td>
<td>Ftype≡ Sig→Rtype</td>
</tr>
<tr>
<td>a, b, x, y</td>
<td>Variables</td>
<td>x: integer</td>
</tr>
<tr>
<td>#n</td>
<td>literal</td>
<td>#n = 3</td>
</tr>
<tr>
<td>s</td>
<td>specific type (type instance)</td>
<td>s{n: integer } ≡ array{n: integer }</td>
</tr>
<tr>
<td>t: TYPE</td>
<td>type variable</td>
<td>∀{ t: TYPE} Stack{t}</td>
</tr>
<tr>
<td>∀{n: integer, t: TYPE} s {t, n}</td>
<td>parameterized type declaration</td>
<td>∀{n: integer, t: TYPE} array {t, n}</td>
</tr>
<tr>
<td>f(obj a: A, b: B):Rtype</td>
<td>function declaration</td>
<td>Add(a, b: integer) : integer</td>
</tr>
<tr>
<td>∀{n: integer, t: TYPE} (s{t, n})→Rtype</td>
<td>parameterized polymorphic function type</td>
<td>∀{ t: TYPE} (array{t})→real</td>
</tr>
<tr>
<td>f(s1←s2: Sig1)</td>
<td>Application (a function call)</td>
<td>Add((a, b←(3, 4): (integer, integer))</td>
</tr>
<tr>
<td>∀{n: integer} s1{a, A, s2(A, n)}</td>
<td>type instantiation</td>
<td>∀{n: integer} array{n, array{n, real} }</td>
</tr>
</tbody>
</table>
1.3.2. Terminology

Let’s settle on the following terms:

- **Typed language**: A typed language is one whose variables can be given nontrivial types.

- **Type Sound Language**: A language is type sound if all the program runs are free of execution errors.

- **Type inference**: It is the technique of inferring the type of the arguments in the implicit parameterized polymorphic types.

- **Inference Rules**: This is the notation used to describe type systems. It constitutes of:

  - **Judgements**: the formal assertions about the typing of programs. Here is an example of a judgement

    \[ E \vdash M : A \]

    the term M has a type A in E

  - **Type rules**: the implications between judgements. The general form of a type rule is:

    \[
    \begin{array}{c}
    \text{(Rule name)} \\
    \text{(Annotations)} \\
    E_1 \vdash A_1 \\
    \vdots \\
    E_n \vdash A_n \text{(Annotations)} \\
    \hline
    E \vdash A
    \end{array}
    \]

  - **Derivations**: deductions based on the type rules. A derivation is a tree of judgements with leaves at the top and a root at the bottom. Each judgement is obtained from the ones immediately above it via a certain rule in the type system.

- **Polymorphism**: Generally speaking, polymorphism means to have many forms.

- **Parametric Polymorphism**: It is a property of programs that are parametric with respect to the type of some of their identifiers. It is mainly one of two ways: explicit or implicit parametric polymorphism.
• **Type Variables**: Strictly speaking, a type variable is not a type, it is a placeholder (or representative) for a type. We assume that all free type variables are (implicitly) universally quantified at an appropriate level. [Graver90]

• **Formal type systems**: These are the mathematical characterization of the informal type systems that are described in manuals. This enables further proof of the **type soundness theorem** (it states that a well typed program is well behaved)

• **Parametric Polymorphic Function**: A polymorphic function is parametric if its behavior does not depend on the type at which it is instantiated. For instance, a function that reverses lists is parametric since it follows the same set of commands despite the knowledge of the elements’ type [Abadi93].

• **Hidden parameters**: Hidden parameters are formal type parameters used in the types of the formal function parameters, and possibly in the result type of a function.

• **Type Overloading**: A type is *overloaded* if a new type with the same identifier has been declared. A type can be overloaded if and only if the new type (with the same identifier) has a different signature.

• **Type Signature**: A *type signature* consists of the type identifier and the signature of the formal parameters, which can be either integers or type variables. The signature of an integer parameter is "I", and the signature of a type variable is "T".

• **Result Type**: It is the type of the result returned from a function. A result type can be void or any valid type.

• **Tuple Type**: Generally speaking a tuple is an ordered list of elements of possibly different types. It is named the **signature type** if it is the aggregate type enclosing the formal parameters of a function or the actual parameters of a function call. It includes the mode of each argument whether it is an object or a value. For instance, the signature type of function
function \( f(n: \text{integer})(\text{obj a: array}\{n, \text{real}\}, m: \text{integer}) : \text{array}\{n, \text{real}\} \)

is

\( \forall\{n: \text{integer}\} \quad (\text{obj array}\ \{n, \text{real}\}, \text{integer}) \)

- **Function Type:** It is the mapping of an aggregate type of parameters to the result type. That is, the function type is a mapping from the signature type to the result type. For instance, the function type of function

  \[
  \text{function } f\{n: \text{integer}\}(a: \text{array}\{n, \text{real}\}, m: \text{integer}) : \text{array}\{n, \text{real}\}
  \]

  can be represented as follows:

  \[
  \forall\{n: \text{integer}\} \quad (\text{array}\ \{n, \text{real}\}, \text{integer}) \rightarrow \text{array}\{n, \text{real}\}
  \]

- **Subtyping by Specialization:** SIMPL defines a subtyping relation among types based on the rule that a subtype must be a type instance of a parameterized type. A type \( s1 \) is more specific than another type \( s2 \) based on two reasons:

  1. **Specialization:** if \( s1 \) is a type instance of \( s2 \) where the type parameters of \( s1 \) are instances or special cases of the type parameters of \( s2 \). Since the parameters of a parameterized type can be integer variables or type variables, their specialization can be integer literals or type instances.

  2. **Constraints:** If \( s1\{a, b\} \) is a type instance and \( s1\{c, b[c]\} \) is another type instance but has more constraints. The type instance with more constraints is more specific.

- **Function Subtyping:** A function type \( f1 \) is a subtype of another function type \( f2 \), if \( f1 \) is a type instance of \( f2 \). This is achieved by three conditions:

  1. If the signature type of \( f1 \) is a subtype of the signature type of \( f2 \)

  2. If the result type of \( f1 \) is a subtype of the result type of \( f2 \). 

18
3. If the hidden parameters of f2 are substituted correctly in f1 and their constraints are preserved.

- **Function Overloading**: it is the declaration of several function interfaces that share the same function name. For instance, the function \( \forall \{t: \text{TYPE}\} f(x: t):t \) can overload the function \( \forall \{t: \text{TYPE}\} f(x: t, y: t):\text{void} \)

- **Function Specialization**: when the overloaded functions share the same function name, same number of arguments, and the type of each argument is a subtype of the corresponding argument’s type in the other signature, it is called specialization. Moreover, the result types experience a subtyping relation. Actually, specialization is a special case of overloading. For instance, the function \( f(x: \text{integer}):\text{integer} \) is a specialization to the function \( \forall \{t: \text{TYPE}\} f(x: t):t \)

- **Ambiguous Overloading**: It is the case when a function call matches several declared overloaded functions. This leads to ambiguity in which function to invoke. The compiler has to settle on the rules of overloading that prevent ambiguity upon function calls.

### 1.4. Road Map

The thesis consists of nine chapters. This chapter provided a flavor of polymorphism and its classification. It covered an illustrative problem definition of parametric polymorphism in SIMPL language. The next chapter guides you in a tour among different languages showing how other languages deal with polymorphism. Chapter 3 is dedicated to highlighting various forms of polymorphism in SIMPL. The concept of “subtyping by specialization” is then explored in chapter 4. The succeeding chapter addresses the formal representation of parameterized polymorphism in SIMPL as encapsulated in the SIMPL Type System. Chapter 6 delves into the semantic representation of the system and illustrates the organization of information in the semantic space for use in type checking and code translation. The new concepts of function overloading and
specialization adopted in SIMPL are investigated in chapter 7 and supplemented with formal proves. The last chapter, chapter 8, encompasses what ideas are concluded from this work and suggestions for further research. Appendix A explains the difference in the subtyping concept between System Fc and SIMPL Type System. The associated algorithms for unification and subtyping checking are listed and explained in Appendix B.
2. POLYMORPHISM IN OTHER LANGUAGES

Polymorphism has been successfully implemented in different languages at various levels. Some languages have introduced new power to polymorphism through adding new features. We will have a tour among some aspects of polymorphism that have been successfully supported by some programming languages.

2.1. C++

C++ provides parametric polymorphism in the form of type templates and function templates. Specialization of templates resembles subtyping by specialization in SIMPL. Moreover, C++ allows function overloading. Inheritance in C++ is another form of polymorphism.

2.1.1. Templates in C++

Parameterized types in SIMPL have got some similarities with Templates in C++, yet templates suffer some deficiencies that SIMPL overcomes. Templates in C++ provide an implementation of generic classes and generic functions. Class templates in C++ are also known as parameterized types. “A template is a description of a type (or a function) that is not complete; it requires one or more parameters (types or constant values) to be provided before the type (or function) is fully defined … When a template is used to name a type, the missing parameters are provided. The result is a ready-to-use type.” [Terribile94]. Templates allow declaration of type variables [Kehoe] the same powerful feature is supported by SIMPL; where SIMPL allows integer and type variables to be type parameters.

```
Template <class T> // “class” indicates that T is a
 class Var_array_T // type parameter whether a
 { … … … … }; // built-in or a user-defined
 Var_array_T < int > indices; // is a class template
```
2.1.2. Template Specialization in C++

Creating the type from a template is called “specializing the template”. It is also sometimes called instantiating the template. Specializing a template takes place through direct instantiation of the template (resembles type instantiation in SIMPL) or by typedef (like type synonyms in SIMPL). The direct instantiation yields a class template (by ‘class’ keyword), struct template (via ‘struct’ keyword), or a union template (using ‘union’ keyword) [Terribile94].

However, template specialization causes access problems. “Specialization can also be considered as opening a protection loophole in C++ because a specialized member function can access a template class’ private data in a way that is not discernible from reading the template definition.” [Stroustrup94]. There has been a suggestion in San Jose meeting that “A specialization must be declared before it is used” [Stroustrup94]. So that if no specialization is in scope at a point of use, the general template definition will be used.

Most compilers implement templates in one of two ways: [Lakos96]

1. CFront-Like\textsuperscript{1}: when templates are encountered in a program, a system-wide repository is created to provide information that is shared among translation units. A “simulated” link is performed to determine what undefined symbols can be resolved via template instantiation. In the normal mode, resolving these symbols can result in new undefined symbols being generated from the bodies of newly instantiated functions. This process repeats until the simulated link does not turn up any new undefined symbols. The number of simulated links could be as large as the depth of

\textsuperscript{1} CFront is a compiler that included the earliest implementation of templates which supported class templates only. It was written by Sam Haradhvala at Object Design Inc. in 1989. [Stroustrup94]
the function-call hierarchy, making template development in this mode very expensive in terms of link time.

2. **MACRO-Like:** the source code for both the header and the implementation of the template component must be made available to clients. This approach does not suffer from excessive link time, but does create a problem with respect to insulation. Having to make implementation source available to clients also means that the code can no longer be treated as proprietary.

Moreover, duplicate object code is being generated for the non-inline functions of each and every instantiated template type. Templates can provide the pointer type safety through inline functions without generating redundant object code. This makes great difference in the size of the executable program. [Lakos96]

The two above-mentioned approaches are not efficient. SIMPL, on the other hand, treats parameterized types as any type but keeps space for their parameters (whether integer variables or type variables). For the storage, it generates size functions to calculate the size information at runtime. This approach saves the waste of code expansion of the macro-like approach and the expense of simulated links in the CFRONT-like approach.

Many people expressed concern that templates relied heavily on the availability of source code. This was seen as having two bad side-effects:

1. You can’t keep your implementation a trade secret.

2. If a template implementation changes, user code must be recompiled.

The trick is deriving template classes from classes that provide a clean interface. [Stroustrup94]

Template mechanism in C++, as well in SIMPL, is not a run-time mechanism. It is merely a compile-time and link-time mechanism. The problem that rises is how can the classes and functions generated (instantiated) from templates get to depend on the information known only at run time. [Stroustrup94] suggests, as ever in C++, to use virtual functions. SIMPL handles the problem differently as discussed in the following chapters.
2.1.3. Inheritance in C++

C++ uses inheritance as a substitute for subtype polymorphism. In fact, C++ offers inheritance to support code reuse and to define a subtype relationship among classes. “The overloading of a single language construct, the class, for defining a type, for implementing a type, as the basis for code reuse, and as the basis for subtyping, not only limits the expressiveness of type abstraction and subtyping, but also the flexibility of inheritance for code reuse.”[Russo95] Russo95 proved this deficiency in the class construct in C++ and introduced a language extension—“signatures”—that enables the definition of an abstract type hierarchy independent of any implementation hierarchies. This language extension uses subtyping based on class interfaces only in connection with signatures. A signature type is similar to the class in its declaration with the keyword signature instead of class or struct. However, a signature declaration contains only interface descriptions. An example of a signature would be:

```cpp
Signature T{
  int * f();
  int g (int *);
  T & h (int *);
};
```

Thus a signature declaration only includes type declarations, constant declarations, member function declarations, operator declarations, and conversion operator declarations [Baumgartner95b].

2.1.4. Overloading in C++

In C++, overloading is powerful and safe. Overloading imposes order on types to suit it. Moreover, it maintains order whenever a user-defined class type displaces a native type. An “overloaded function name” is a function name that represents several functions in the same scope. C++ allows overloading of member and nonmember functions [Terribile94].

In order for C++ to resolve overloading, it uses conversions to match a function signature to the types written in a call. C++ divides conversions into 4 groups: [Terribile94]
1. *Trivial conversions:* These conversions are often “unavoidable”; they are needed to meet C and C++ semantics. They do not affect the selection of an overload match but a given conversion may be illegal, making the selected match illegal. These conversions include:

\[
\begin{align*}
T & \quad \Rightarrow \quad T & \& \\
T & \& & \Rightarrow \quad T \\
T[ ] & \Rightarrow \quad T_\star \\
T(\text{argtypes}) & \Rightarrow \quad (T_\star)(\text{argtypes}) \\
T & \quad \Rightarrow \quad \text{const } T \\
T & \quad \Rightarrow \quad \text{volatile } T \\
T_\star & \quad \Rightarrow \quad \text{const } T_\star \\
T_\star & \quad \Rightarrow \quad \text{volatile } T_\star
\end{align*}
\]

2. *Promotions:* Promotion, or widening, is the conversion from a type with a representation and a width to another type with the same representation but a possibly greater width. For instance, C++ allows promotion from `float` to `double` and from `short int` to `long int`.

3. *Other built-in conversions:* These are conversions among arithmetic types with different representations (signed, unsigned, floating point) from any pointer-to-object to `void*` or vice versa. Also it includes the conversions needed by inheritance and virtualization between the subtype (or derived type) and the supertype (or base type).

4. *User-defined conversions:* These are conversions by constructor or member conversion operator.

In comparison, overloading in SIMPL allows specialization among supertypes and their subtypes. SIMPL does not support promotion, but supports subtyping conversions and user-defined type conversion.
C++ type-checking rules state that function calls must be checked at compile time. The checking of trailing arguments can be suppressed by explicit specification in a function declaration. Among several proposed mechanisms, Stroustrup chose function name overloading and default arguments to give the appearance of a single function taking a variety of argument lists without compromising type safety [Stroustrup94]. SIMPL, on the contrary, checks function calls at compile time and generates extra code inside the most general function to redirect the execution – at run-time - to the appropriate function according to the type of the actual parameters.

2.1.5. Memory Management and Copy Semantics in C++

C++ provides a single declaration for a class that acts as the interface to both users and implementers of the member functions. There is no direct support for the notions of “interface definition” and “implementation module”. This implies that the compiler knows the size of every object. This in turn implies that a C++ program requires recompilation whenever changes are made to the private or protected parts, even though such changes apparently affect only the implementer of a class and not its users. [Ellis90] In comparison, SIMPL supports interface definition and implementation module for types. Moreover, it stores the size information of each type in a function to be called at run-time. If the size is a constant value, it is directly stored in the symbol table. In case of type structures that are dynamically constructed, their size is an expression that is calculated by the size function and returned at run-time. A detailed explanation of this approach is in the semantic information representation chapter. This mechanism solves the problems faced by C++. The approach followed by C++ allows allocating and manipulating objects directly rather than through a pointer. Direct access of objects supports better performance at run-time,
better use of space – using the stack instead of heap – and accordingly saves memory management operations. Had an indirection been required, extra storage is needed for the pointers used by the compiler for “housekeeping”. [Ellis90]

Allocation of objects on the free store can lead not only to inefficiency, but also – in languages where the indirection is implicit – to a noticeable discontinuity in the semantics of assignment i.e. copy semantics are applied on built-in types whereas pointer semantics are used in user-defined types. In C++, copy semantics are used for both built-in and user-defined type. Pointer semantics for user-defined types can be implemented by overloading the assignment operator. In SIMPL, the assignment operator is a polymorphic operator that accepts two parameters of the same type and copies their content byte by byte assuming contiguous space. In case of pointers (called references in SIMPL) the assignment operator is overloaded to accept two parameters of type ref(t) and t – where t is a type variable that can be instantiated with any type.

[Lakos96] mentioned some memory management problems that might rise due to applying templates:

1. While writing a template, we must be aware that the parameter type could be either a fundamental type or a user-defined type that itself manages dynamic memory.
2. It is not possible to derive directly from a fundamental type
3. In C++, it is not in general possible to use a bitwise copy to move a user-defined type.
   SIMPL deals with built-in and user-defined types in the same way.
4. In general, an object cannot be copied or moved using a bitwise copy; this object may contain a pointer or reference to another subordinate object that it owns. After making a bitwise copy, there would be two instances of the object, each thinking that it alone is responsible for deleting the memory of the same subordinate objects. Destroying
both instances would result in deleting the subordinate object twice causing a programming error. It is recommended to avoid bitwise copies of objects when implementing containers using C++ templates. SIMPL states that assignment is a bitwise copy operation. It allows programmers to overload the assignment operator to suit their programs.

5. When implementing memory management for a general, parameterized container template, be careful not to use the assignment operator of the contained type when the target of the assignment is an uninitialized memory. In some cases, the memory allocation system might be corrupted.

2.2. ADA and ADA95

2.2.1. Subtyping in ADA

Subtyping in Ada has two forms: Subtypes and Derived types. Subtype in Ada is a simple user defined type. It is considered an automatic error-checking facility. A subtype is obtained from a scalar type - called base type - and it is always a range between two simple expressions of the base type. The subtype inherits all the operations of its base type. If a subtype parameter passes and at execution there is an attempt to assign a value that is outside the range of the subtype of the parameter, an exception would be raised called (CONSTRAINT_ERROR) and the procedure terminates [Savitch92]. Subtypes are types created from an existing "parent" type which are distinct but compatible with the parent. On the other hand, derived types are new types created from an existing "parent" type which are distinct and separate (incompatible) from the parent [Conn95].

Matching of formal and actual parameters must be of the same base type. Thus, subtypes can be passed as actual parameters of their base type and vice versa. [Barnes89]
2.2.2. Overloading in ADA

For overloading, ADA allows subprogram overloading but does not allow subprogram specialization. That is, subprogram names may be overloaded (i.e., two or more subprograms may have the same names but different types or numbers of parameters) under the condition that they have different profiles (different base types for parameters and for the result type) [Skansholm97].

2.2.3. Generic Units in ADA

Polymorphism in ADA is applied in two ways: static and dynamic polymorphism. Static polymorphism is achieved via the generic parameter mechanism. The generic parameter mechanism is attained when “a generic unit may at compile time be instantiated with any type from a class of types.” [Guerby95] A generic unit is a reusable software component, a special implementation of a subprogram or package that defines a commonly used algorithm in data-independent terms. Ada systems accept the declaration of generic subprograms and packages, which are templates describing general-purpose algorithms that apply to a variety of datatypes. There are three kinds of generic formal parameters: types, objects, and subprograms [Conn95]. This approach has great resemblance to parameterized types in SIMPL that can be instantiated at compile-time.

On the other hand, dynamic polymorphism is obtained by using class-wide types where the distinction is achieved at runtime via the tag’s value. “A class-wide type is declared implicitly whenever a tagged record type is defined. The set of values of the class-wide type is the union of the sets of values of all the types of the class. Values of class-wide types are distinguished at runtime by the value of the tag giving class-wide programming or dynamic polymorphism. The class-wide type associated with a tagged record type T is denoted by the attribute T’Class.” [Guerby95] Ada95 accomplishes inheritance through tagged types. Tagged types are expandable types that include a hidden component stating its exact type, that is its tag [Skansholm97]. Ada95 applies dynamic dispatching by determining the appropriate routine call for any tagged type. Dispatching only
occurs on subprograms that were defined in the same package as the tagged type itself. These subprograms are formally called **primitive subprograms**. This feature has an equivalent in SIMPL: the union type. At run-time, based on the tag of the type, its current type is determined. Moreover, instances of this type can change their type through explicit assignments to values of the new type (this type has to be one of the declared members of the union type). The tag changes automatically to indicate the new type after assignment.

### 2.3. Eiffel

Eiffel is an object-oriented language that supports static typing. Eiffel reconciles dynamic binding with static typing. Dynamic binding guarantees that whenever more than one version of a routine is applicable the right version (the one most directly adapted to the target object) will be selected. Static typing means that the compiler makes sure there is at least one such version [Meyer92]. Polymorphism in Eiffel is illustrated in genericity, inheritance and function subtyping.

#### 2.3.1. Genericity

Eiffel supports type parameterization through **generic classes**. A generic class is a class declared with formal generic parameters. These classes can be constrained or unconstrained. Derivation of types from a generic class yields generically derived types; where actual generic parameters are provided for the formal generic parameters. It supports explicit declaration of type constraints. The declaration of the type constraints is given in a class and enforced upon the generic definition declaration [Kim97]. For unconstrained generic parameters, any type is acceptable as an actual generic parameter. The constrained generic parameters impose a constraint on the actual generic parameters to be descendents of the constraining class. An example of a generic class with both kinds of generic parameters is:

```eiffel
class Hash_table [G, Key-> Hashable]
```
where the G parameter is an unconstrained generic parameter that accepts any type for the table elements. The Key parameter is a constrained generic parameter that is constrained by the class Hashable. In SIMPL, parameterized types are not constrained. However, an extension to SIMPL may allow declaration of constrained type variables and use them in defining constrained parameterized types.

Eiffel has a kernel Library class named ANY that is automatically an ancestor of any class that a programmer may write [Meyer92]. This is similar to the built-in type type in SIMPL.

2.3.2. Genericity and Inheritance

Eiffel combines genericity and inheritance in two ways. The first technique is polymorphic data structures; where a generic class that is parameterized by a certain type can be instantiated - upon declaration of objects - by descendents of that parameter. The second mechanism of combination is constrained genericity; where a class name is indicated after a formal generic parameter. That is, a generic class is constrained by a constrained parameter. The constraint means that the base class of any actual generic parameter used must be a descendant of the constraining class. The effect of this constraint is to restrict allowable actual generic parameters to types that conform to the stated class type. Mainly, the conformance rule states that a type C conforms to a type B if the base class of C is a descendent of the base class of B; also, if C is generically derived, its actual generic parameters must conform to the corresponding ones in B [Meyer92]. Anchored type is another form of providing automatic redefinition in the descendents of the class where it appears. Due to this feature, upon redefinition of the anchor, the anchored type will be automatically redefined following its anchor. For instance, the following anchored declaration of the variable

```eiffel
element in the routine put_element of the class Linked_list
    put_element(element : like first_element)
```
indicates that any redefinition of \texttt{first\_element} in a descendent of \texttt{Linked\_list} does not need a redefinition of \texttt{element}; its type will automatically follow the redeclared type of its anchor, \texttt{first\_element} [Meyer92]

\subsection*{2.3.3. Function Subtyping}

Eiffel favors function subtyping covariantly with argument subtyping: “A routing redefinition may replace the type of a formal argument by a type conforming to the original. This is known as the covariant argument typing policy” [Meyer92], [Abadi96]. It is the same policy applied in SIMPL. Appendix A explains the difference in views between object-oriented type systems and SIMPL type system in terms of function subtyping covariance / contravariance with respect to argument subtyping.

\subsection*{2.4. Java}

\subsubsection*{2.4.1. Overloading in Java}

Java was indeed based on C++, but it was also designed to be simple and portable; accordingly many of C++’s features have been removed. Although Java allows method overloading, it does not have the concept of operator overloading. There is no operator overloading because the operator can be defined to mean anything; it makes it very difficult to figure out what any given operator is doing at any one time. This can result in entirely unreadable code. When you use a method, you know it can mean many things to many classes, but when you use an operator you would like to know that it always means the same thing. Given the potential for abuse, the designers of Java felt it was one of the C++ features that was best left out [Lemay97]. Although SIMPL has the intention to be portable, it emphasizes static type checking and binds function calls to the most appropriate function provided. At run time, if the called function is the most general function it might redirect the call to the best matching function (a more specialized function).
2.4.2. Subtyping in Java

Java provides subtyping through inheritance. Yet, Java allows single inheritance only. The reasoning behind preventing multiple inheritance is to avoid ambiguity that might rise from multiple copies of the overloaded method in the superclasses, i.e. multiple inheritance of implementation. However, Java overcomes this restriction by allowing inheritance of the interface without inheriting the implementation. This takes place by declaring interface type instead of class type. Thus, interfaces in the class hierarchy add multiple inheritance to Java [Arnold98].

2.5. Modula-3

Modula-3 is a new modern object-oriented language that is simple and modular. It provides exception handling, concurrency, object-oriented programming, and automatic garbage collection. It is simple and safe and applies a systematic type system [Modula3].

2.5.1. Subtyping in Modula-3

It allows type declarations and revelations - it does not introduce new names in the subtype relationship. It has opaque types - a type whose concrete structure is hidden from clients in an interface. The keyword REVEAL can be used to enable clients to access the details of a type via the subtype relationship. Thus the REVEAL enables revelation for opaque types. Its type system uses structural equivalence instead of name equivalence. Yet some problems might reveal due to the structural equivalence feature; two types with different names but turned out to be of the same structure will be considered by the language to be of the same type even if you don’t intend to have them as such. The keyword BRANDED is used to brand any or both types i.e. to make them unequal even if they are structurally equal. How does this take place? Actually BRANDED adds a bit of virtual structure of the type that guarantees it will be distinct from every other type [Cardelli89].
2.6. Conclusion of the Languages' Survey

Throughout the chapter we have discussed how do different programming languages handle polymorphism and how they provide various aspects of implementing it. Some languages face some difficulties in providing polymorphic types - such as implementing templates in C++ - and whether to consider them types or just templates to generate monomorphic types. On the other hand, generic units in ADA95 and generic classes in Eiffel exemplify a successful implementation of polymorphic types. Moreover, the idea of tagged types in ADA95 and the constrained generic classes in Eiffel add power to generic classes and impose restrictions on the instantiations even at run-time. Another form of polymorphism revealed in subtyping as shown in Java and Modula-3. An interesting feature supported by Modula-3 is revealing opaque types upon explicit request from the client of the type. Finally, we have seen how does Modula-3 overcome the restriction imposed by structural type equivalence via the keyword **BRANDED** to allow differentiating two types even if they share the same structure.
3. SIMPL LANGUAGE FEATURES

3.1. What is SIMPL?

SIMPL is a new programming language, being developed at the American University in Cairo, which offers a set of powerful features such as the ability to program sequentially and concurrently in the same module. SIMPL (stands for Simplified Imperative Modular Programming Language) is designed to be a platform independent language. Moreover, SIMPL supports parameterized polymorphism. It utilizes hidden parameters to add a new dimension to parametric polymorphism. A full specification of the language can be found in [Mudawwar 98b]. The rest of this chapter covers in details polymorphism in SIMPL.

3.2. Interfaces and Modules

SIMPL is a modular language; any SIMPL program consists of several interfaces and modules. A SIMPL program is a collection of interfaces and modules stored in separate files and compiled separately. An interface describes the interface of a module. It consists mainly of type and function interfaces. A module describes the implementation of an interface. It consists mainly of type and function implementations. The module holds the implementation of every public object and function declared in the interface in addition to any private implementation. All interface members are public and exportable to other interfaces and modules. Module members, being not declared in an interface, are private and visible only within the given module. In the following example, the interface of the type complex exports all the public members to any other interfaces or modules.

```
Interface complex is
  type complex is -- Definition of the type complex
    re, im: real  -- All data members of complex are
                  -- public
  end           -- type complex
```

function + (x, y: complex): complex
...

function complex (re: real, im: real): complex
-- Converting two real expressions to a complex
...
end -- interface complex

module complex is
-- A module is the implementation of
-- an interface. It consists of type
-- and function implementation

function complex (re: real, im: real): complex is

result.re := re
result.im := im
end -- function complex

function complex (r: real): complex is

function complex (r, 0.0) -- Function Synonym

function := (obj lhs: complex, rhs: real) is
lhs := complex(rhs)
end

function + (x, y: complex): complex is

function complex (x.re + y.re, x.im + y.im)
-- Overloading of arithmetic operators
...
end -- module complex

Any module or interface can use this module. The accessible functions and types are the
ones exported through the interface. The following testcomplex module uses complex
interface.

module testcomplex is
import complex -- use the interface complex
x, y: complex -- static allocation

x := complex (3.5, -1.2)
y := 1.2 -- real can be assigned to complex
x := x * y - x /y -- overloading of arithmetic
-- operators
end -- module testcomplex

SIMPL provides two interface types: a functional interface and an object-oriented interface. The
above complex interface is functional where the functions manipulating the type complex are
encapsulated separately. An object-oriented interface encapsulates the member functions within the type declaration itself. In such a case the type declaration with its public member functions’ declarations is called a type interface. The module has to follow its interface structure (a functional module defines the functions’ bodies outside the type complex’s definition, whereas the object-oriented module defines the functions’ implementation inside the type’s definition). As long as the member function is declared inside the type interface, it has an implicit obj parameter that represents an object of the type enclosing the function. As illustrated in the following object-oriented interface of type list, the public member function length has no explicit formal parameters. Yet it has one implicit formal parameter, which is obj l:list(t).

Interface list is
   type list{t: type} is
      first: ref{list{t}}
      function length(): integer
   end
   end
   -- Interface of a linked list type
   -- pointer to the 1st element
   -- length of the list
   -- type list{t}
   -- interface list{t}

3.3. Parameterized Types

SIMPL provides polymorphism via parameterization of types and overloading of types and methods. Types in SIMPL can be parameterized. Parameterized types are called polymorphic because their shape and size change according to the values of their parameters. This type parameterization allows the development of polymorphic functions with hidden parameters as will be discussed later. A type declaration informs the compiler of the existence of a new type. This type may or may not be parameterized. A non-parameterized type is simply an identifier, while a parameterized type should include at least one formal type parameter surrounded by braces. Here are some examples of type declarations:

   type complex
      -- Not parameterized
   type list{t: type}
      -- Parameterized
type array{n: integer, t: type}  -- Parameterized

A formal type parameter must be either an integer parameter or a type variable. A type variable is a formal type parameter of type type. The keyword type denotes the set of all types declared by a programmer. Thus, a type variable is an unspecified element of type. Type variables are not themselves types but rather placeholders to any type. In the above examples, the formal type parameter t is a type variable. Limiting a formal type parameter to an integer or to a type variable is not really a restriction. To declare a parameterized type, we need to generalize the size and the type of its elements. The size is generalized using one or more integer parameters. The type is generalized using one or more type variables [Mudawwar98a].

3.3.1. Type Interface and Type Implementation

For a type to be used by other programs, it has to be declared first. This is achieved through the type interface. A type interface contains the public members of a type. A type interface must be specified in an interface. It cannot be specified in a module. An example of a type interface is shown below. It specifies the interface of the parameterized type stack. There are three public member functions, but no public field in this type interface.

```plaintext
interface stack is
type stack{n: integer, t: type} is
  function push(x: t)
  function pop(): t
  function items(): integer
end

-- interface of type stack

-- interface stack
```

Member functions declared in type interfaces are processed exactly like non-member ones. Consider the interface of the type stack with member functions declared within the type. This interface can be written differently as shown below. The two interfaces are semantically equivalent. The difference is syntactic only. Functions encapsulated in a parameterized type use the formal type parameters as hidden parameters. These hidden parameters are manifested when the function
is declared outside the type interface. The first formal parameter of a function is omitted when that
function is encapsulated in a type interface. The first formal parameter of an encapsulated function
is always an obj parameter whose type is the encapsulating type [Mudawwar98a].

```
interface stack is
  type stack(n: integer, t: type) is
    function push(x: t)
    function pop(): t
    function items(): integer
  end
-- interface of type stack
end
-- interface stack
```

A type implementation contains the private fields and the implementation of the public functions
of a type. It may also contain additional private functions. A type implementation must be specified
in a module. It cannot be specified in an interface. The implementation of the parameterized type
stack is shown below. It contains two private fields: top and storage, and the implementation
of the three public functions: push, pop, and items.

```
module stack is
  type stack(n: integer, t: type) is
    top :integer := 0
    storage :array{n, t}

    function push(x: t) is ... end -- implementation of push()
    function pop(): t is ... end  -- implementation of pop()
    function items(): integer is
      ... end
-- implementation of items()
end
-- module stack
```

The implementation of the parameterized type stack uses an array of size n and element type t
for internal storage, where n and t are the formal type parameters. Furthermore, n and t can be
used in the implementation of the member functions. The actual values of \( n \) and \( t \) are specified when \texttt{stack} objects are declared. The type \texttt{stack} is an example of a \textit{polymorphic type} whose size and shape vary according to the actual values of \( n \) and \( t \). The member functions of \texttt{stack} are also \textit{polymorphic}, because they can be called with \texttt{stack} objects of any size \( n \) and element type \( t \) [Mudawwar98a].

\subsection*{3.3.2. Synonym Types}

SIMPL provides type equivalence via synonym types. A synonym type introduces a new equivalent type name, signature, or instance of a given type. An example of synonym types can be as such:

\begin{verbatim}
  type matrix \{n, m: integer, t: type\} is
    type array \{n, array\{m, t\}\}

  type matrix \{n: integer, t: type\} is
    type matrix \{n, n, t\}
\end{verbatim}

The above synonym types define a general matrix with \( n \) rows and \( m \) columns and a square matrix with \( n \) rows and \( n \) columns. Both declarations of \texttt{matrix} are acceptable because they have unique signatures.

\subsection*{3.3.3. Union Types}

The union type is a form of dynamic typing. It is a declaration of a new type identifier or a new type signature – if it is parameterized - and associating it to more than one type. The syntax of the union type declaration is the keyword \texttt{type} followed by the new type name with its type parameters – if any - and the keyword \texttt{is}. This is followed by the keyword \texttt{type} and a list of the alternative types to be assigned separated by the keyword \texttt{or}. This is illustrated in the following union type declaration.

\begin{verbatim}
  Type student is
\end{verbatim}
-- freshman, sophomore, junior and senior
-- are already declared types.

```
type freshman or sophomore or junior or senior
```

The structure of this type in data space is a tag that indicates which of the types is selected at current. In addition, its size is the largest size needed by any of these types. For instance, if the type `freshman` needs 2 bytes, type `sophomore` requires 3 bytes, type `junior` needs 1 byte and `senior` requires 4 bytes of space. Then type `student` needs 4 bytes of space in order to be able to hold any type of them at a time. The tag size depends on the number of types it selects among.

In this example, the tag needs two bits to distinguish among 4 types. Accordingly, the type `student` becomes `freshman` if the tag is 0 and `senior` if the tag is 3. At run-time, based on the tag of the union type, its current type is determined. The tag is a read-only element to programmers, i.e. they cannot change its value through assignment. However, instances of this type can change their type through explicit assignments to values of the new type (this type has to be one of the declared members of the union type). The tag changes automatically to indicate the new type after assignment. This indicates that more processing takes place within the assignment function.

### 3.3.4. Type Overloading

A type is *overloaded* if a new type with the same identifier has been declared. A type can be overloaded if and only if the new type (with the same identifier) has a different signature. A type signature consists of the type identifier followed by the types of the formal parameters. Accordingly the following types can be overloaded without any ambiguity.

```
type stack                -- 1. a nonparameterized stack

(type stack{n: integer}) -- 2. stack{integer}

(type stack{t: type})    -- 3. stack{type}

(type stack{n: integer, t: type}) -- 4. stack{integer, type}
```
If we declare a variable \( s \) of type \( \text{stack} \), the stack signature determines which one of the above types \( s \) is an instance of.

\[
S: \text{stack} \quad -- \quad 1^{\text{st}} \text{ stack type}
\]
\[
S: \text{stack}\{3\} \quad -- \quad 2^{\text{nd}} \text{ stack type}
\]
\[
S: \text{stack}\{m\} \quad -- \quad 2^{\text{nd}} \text{ stack type}
\]
\[
S: \text{stack}\{\text{real}\} \quad -- \quad 3^{\text{rd}} \text{ stack type}
\]
\[
S: \text{stack}\{\text{array}\{4, \text{char}\}\} \quad -- \quad 3^{\text{rd}} \text{ stack type}
\]
\[
S: \text{stack}\{4, \text{integer}\} \quad -- \quad 4^{\text{th}} \text{ stack type}
\]

### 3.3.4.1. Type Signature

A *type signature* consists of the type identifier and the signature of the formal parameters, which can be either integers or type variables. The signature of an integer parameter is "I", and the signature of a type variable is "T". The following stack types have different signatures.

<table>
<thead>
<tr>
<th>Declared Type:</th>
<th>Type Signature:</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{type stack}</td>
<td>stack</td>
</tr>
<tr>
<td>\text{type stack}{t: \text{type}}</td>
<td>stackT</td>
</tr>
<tr>
<td>\text{type stack}{n: \text{integer}, t: \text{type}}</td>
<td>stackIT</td>
</tr>
<tr>
<td>\text{type stack}{t: \text{type}, n: \text{integer}}</td>
<td>stackTI</td>
</tr>
</tbody>
</table>

### 3.3.5. Type Instantiation

Given a parameterized type, we can derive instances of that type. A *parameterized type instance* is a type identifier followed by a list of actual type parameters. An *actual type parameter* can be an integer identifier, an integer literal, a type variable, a type identifier, or a parameterized type instance. If the formal type parameter is an integer parameter then the actual parameter should be an integer identifier or an integer literal. If the formal type parameter is a type variable then the
actual type parameter can be either a type variable, a type identifier, or a parameterized type instance. This leads to four kinds of actual type parameters. These kinds are:

**Integer literal:** such as `4` in `stack{4, real}`

**Integer identifier:** such as `m` in `stack{m, real}`

**Type variable:** such as `t` in `stack{5, t}`

**Type identifier:** such as `real` in `stack{4, real}`

**Parameterized Type instance:** such as `matrix{3, 4, integer}` in `stack{6, matrix{3, 4, integer} }`

Different instances of a parameterized type are considered different types. For example, the types `array{100, real}, array{200, real}, and array{100, array{100, real}}` are distinct types. The types `array{m, real}` and `array{n, real}` are also considered different even when the values of `m` and `n` turn out to be identical at runtime. This is because type checking is done statically at compile time and the compiler cannot, in general, verify that two integer identifiers have identical values. Finally, integer identifiers used as actual type parameters should not be modifiable objects. They can be integer constants declared in a function, read-only value parameters or hidden parameters of a function as will be clarified later [Mudawwar98a].

### 3.3.6. Subtyping

A type is said to be **polymorphic** if it contains at least one type variable or an integer parameter. Otherwise, it is called **monomorphic**. For example, the types `t, array{n, array{n, real}}`, and `stack{5, t}` are polymorphic, while the types `integer` and `array{10, real}` are monomorphic. Types are also related by instantiation. A subtyping relation is defined between a polymorphic type and its instances. For example, the types `array{100, t}, array{n, real}, and array{100, array{200, real}}` are all subtypes (because they are instances) of `array{n, t}`. However, the types `array{100, t}` and `array{n, real}` are not related (neither one is a subtype of the other), but have a common subtype which is `array{100, real}` [Mudawwar98a].
Subtyping is a reflexive, transitive and antisymmetric binary relation over types, which satisfies subsumption; it asserts the inclusion of collections of values – where types are viewed as collections of values. Subsumption states that a variable of a certain type is also considered as an element of its supertypes, and accordingly it is used flexibly in many different typed contexts. [Cardelli97] SIMPL supports subtyping via type instantiation. By declaring a parameterized type, we declare a generic base type. The declaration of a type instance provides a new subtype of the generic base type. In the following example, these type instances constitute subtypes of their base types.

```
stack{100}      -- instance of stack{n: integer}
stack{50, real} -- instance of stack{n: integer, -- t : type}
stack{stack{real}} -- instance of stack{t : type}
stack{n}        -- instance of stack{n: integer} -- where n is a constant integer
```

In general, different instances of a parameterized type are considered different types. For example, the types `stack{100, integer}` and `stack{200, real}`, which are instances of `stack(integer, type)`, are different. The Unification Algorithm, detailed in appendix B, unifies type instances based on their signatures (i.e. matches their names, number of parameters and the parameter list). The subtyping relation is explained in details in a separate chapter.

### 3.3.7. Processing Types

A parameterized type in SIMPL is not a type generator as is the case with the class template of C++. A parameterized type is itself a type and occupies a unique place in a symbol table when processed. To process a parameterized type, we need to record its size, its fields, and its member functions. This is explained in details in the semantic information representation chapter.
3.4. Functions

A function denotes a computation that may or may not have a result value. In SIMPL, we do not distinguish between a function and a procedure. All methods are functions. A function with no result value is equivalent to a procedure in other languages, but the keyword procedure is not used in this language. Functions in SIMPL can be monomorphic or polymorphic. A function is monomorphic, if its actual parameters can assume only one type. A function is polymorphic, if the formal function parameters may assume different types. A polymorphic function is parametric if its behavior does not depend on the type at which it is instantiated. For instance, a function that reverses lists is parametric since it follows the same set of commands despite the knowledge of the elements’ type [Abadi93]. Polymorphic functions have hidden parameters; a feature, I believe, is unique to this programming language. The following sections explain in details the function declaration in terms of the hidden parameters, the formal parameters and the result type. The function definition is detailed in a separate section. The function types and the function synonyms are explored in another sections.

3.4.1. Function Declaration

A function declaration is the interface of a function. It informs the compiler about the formal parameters, the hidden parameters, and the result type of a function. A function declaration is also a useful notation for the programmer. It is the means of conveying information about a function without showing its implementation.

The syntax of a function declaration starts with the function keyword. This is followed by a function name. The function name can be a function identifier, or an operator. Following the function name is an optional list of hidden parameters. Following the hidden parameters is an optional list of formal function parameters. According to the syntax, the braces of the hidden parameters are not included if there are no hidden parameters. However, the parentheses of the
formal parameters are always included, even when no formal parameter is listed. A function declaration is optionally ended with a colon and a result type (which is a type instance). If a result type is declared, it means that the function has a result value, and a function call is an expression. Otherwise, the function would be equivalent to a procedure in other programming languages, and a function call is a statement [Mudawwar98b].

In SIMPL, a function declaration (or interface) can be separated from its definition (or implementation). A function can be declared in an interface or in a module. However, it cannot be declared twice. A function declared in an interface is public and exportable to other compilation units. A function declared in a module is private and visible only inside the module. Here are some examples of function declarations:

```plaintext
function add (x, y: complex): complex

function add {n: integer}(x, y: array{n, real}): array{n, real}

function reverse {n: integer, t: type}(obj x: array{n, t})
```

The first function defines the addition of two complex numbers. This function is monomorphic because its actual parameters can assume only one type. The second function is polymorphic and uses two lists of parameters. The first list is the hidden function parameters, while the second list is the formal function parameters.

### 3.4.2. Hidden Parameters

Hidden parameters are formal type parameters used in the parameterized types of the formal function parameters - if there is any - and possibly in the result type of a function. With hidden parameters, a function has two sets of parameters: the hidden parameters and the formal function parameters. Hidden parameters are enclosed by braces and are listed first. Formal parameters are enclosed by parenthesis. For example, the function

```plaintext
function add {n: integer}(x, y: array{n, real}):
```
declares n as a hidden parameter, and uses it in the types of the formal parameters x and y as well as in the result type. Moreover, the function

```plaintext
function reverse {n: integer, t: type}(obj x: array{n, t})
```

has two hidden parameters, n and t, and one formal parameter x. A hidden parameter must be used in the type of at least one formal function parameter, or else it cannot be inferred. Furthermore, a hidden parameter may be used in the body of a function as if it were a formal value parameter. Hidden and formal value parameters are considered constants inside the body of the function and can be used as actual parameters to parameterized types [Mudawwar98a].

Any function call submits actual parameters for the formal parameters but not for the hidden parameters. The compiler, from the actual parameters, must induce values for the hidden parameters. In other words, hidden parameters are not passed explicitly when a function is called. Instead they are inferred by the type system upon the unification of the types of the actual parameters against the types of the formal parameters. For example, the function

```plaintext
function reverse {n: integer, t: type}
   (obj m: array{n, array{n, t}})
```

when invoked with the following calls

```plaintext
obj a: array{10, array{10, real}}
obj b: array{10, array{20, real}}
reverse(a)
reverse(b)
```

One can notice that the number of actual function parameters must match the number of formal function parameters, and that the actual values of the hidden parameters are not passed explicitly. Upon the function call with the array a, the compiler unifies the actual parameters against the formal parameters and infers the value of n to be 10 and t to be real. The second function call results in a compile-time error since the type system is unable to unify the type of the actual
parameter against the type of the formal parameter. The cause is that array \texttt{b} is not a square matrix since it has different numbers for the two dimensions to its elements whereas \texttt{reverse} function requires its parameter to be a square matrix.

The feature of hidden parameters adds more power to SIMPL and more checks by the compiler on behalf of the programmer. The hidden parameters in the matrix multiplication function

\begin{verbatim}
function multiply \{r1, c1, c2 : integer\} \{m1: matrix\{r1, c1\},
                m2: matrix\{c1, c2\}\}: matrix\{r1,c2\}
\end{verbatim}

enforce implicit constraints on the formal parameters (matrices' dimensions); the result matrix's rows match the number of rows in the first matrix and its columns match those of the second matrix. Furthermore, the number of columns of the first matrix must match the number of rows in the second matrix. If hidden parameters were not allowed then additional formal parameters would be required. However, SIMPL does not permit the use of type parameters as formal function parameters. Thus, SIMPL forces you to use hidden parameters when declaring or implementing polymorphic functions [Mudawwar98b].

\subsection{3.4.3. Formal Function Parameters}

A formal function parameter is used to carry data into and/or out of a function. A formal function parameter has a mode and a type. The \textit{mode} can be \textit{object} or \textit{value}. The presence of the \texttt{obj} keyword means an \textit{object} parameter, while its absence means a \textit{value} parameter. The semantics of formal function parameters are summarized in Figure 3.1. If the formal parameter is a \textit{value} parameter, the actual parameter can be an expression (including an object) whose value is passed at call time. If the formal parameter is an \textit{object} parameter, the actual parameter should be an object whose value is passed initially at call time and then updated at exit time. A function body can read \textit{value} and \textit{object} parameters but can write only \textit{object} parameters. A \textit{value} parameter can be passed only to a \textit{value} parameter, while an \textit{object} parameter can be passed either to a \textit{value} or to an \textit{object} parameter.
The modes of formal function parameters in SIMPL are similar but simpler than the modes of Ada. The \textit{value} parameter is equivalent to the \textbf{in} parameter of Ada and the \textit{object} parameter is equivalent to the \textbf{in out} parameter. Ada also makes the distinction between an \textbf{out} and an \textbf{in out} parameter. However, SIMPL does not make this distinction for simplicity.

The mode separates the meaning of a formal function parameter from its implementation. A trivial implementation may pass \textit{value} parameters by copy and \textit{object} parameters by reference. An optimized implementation may pass both \textit{value} and \textit{object} parameters by reference to avoid copying of large data structures. This is because \textit{value} parameters are never modified inside the body of a function. However, copying may sometimes be necessary and can be handled transparently by the compiler (e.g., when a function call passes the same object to a \textit{value} and to an \textit{object} formal parameter) [Mudawwar98a].
3.4.4. Result Type

It is the type of the result returned from a function. If the function returns a value, the result type is explicitly declared in the function declaration part. A result type can be void or any valid type. The mode of the result type must be value. The result type can be a parameterized type; in such a case, its type parameters must be listed in the hidden parameters of the function declaration.

3.4.5. Function Definition

A function definition is the implementation of a function. It consists of a function declaration (or header) and a function body. The keyword is follows the function declaration. Next comes the function body; the function body consists of local object declarations and statements. The statements are succeeded by the end keyword [Mudawwar98b]. The result value is assigned to the reserved variable result. The type of result is declared in the result type part of the function declaration. This is illustrated in the following function definition

```plaintext
function identity_matrix {n: integer} (obj m: array{n, integer}):
    boolean is -- checks if the matrix m is an identity matrix

    result := 'true'
    for i := 1 to n do
        for j := 1 to n do
            result := result and ((i = j and m[i][j] = 1) or (i <> j and m[i][j] = 0))
            end
        end
    end

3.4.6. Function Types

Function type is the mapping of an aggregate of parameter types to the result type. That is, the function type is a mapping from the tuple type to the result type.

SIMPL deals with function types more or less as data types. Accordingly, function types can be parameterized. A function type declaration starts with the keyword type followed by the function type name. If this function type is parameterized, a list of the hidden parameters follows. The mode
of the hidden parameters is value by default. The keywords is and function proceed. Next come the tuple type and the result type. The function type declaration

\[
\text{Type functype } \{n: \text{integer}, \ t: \text{type}\} \text{ is } \\
\quad \quad \quad \text{function } (\text{obj array}\{n,\ t\}, \text{integer}): \text{array}\{n,\ t\}
\]

declares a type of functions that receive two formal parameters of type \text{array}\{n,\ t\} and integer and return \text{array}\{n,\ t\}. The needed hidden parameters are listed next to the function type name.

In the formal representation, the function type is a list of hidden parameters – if any – followed by the tuple type that is mapped to the result type. For instance, the function type of function

\[
\text{function } f \{n: \text{integer}\} (a: \text{array} \{n, \text{real}\}, m: \text{integer}): \text{array}\{n, \text{real}\}
\]

can be represented as follows:

\[
\forall\{n: \text{integer}\} \ (\text{array} \{n, \text{real}\}, \text{integer}) \rightarrow \text{array}\{n, \text{real}\}
\]

The result type can be parameterized via the same set of hidden parameters (e.g. \(\forall\{n: \text{integer}, \ t: \text{TYPE}\}(s1 \{t, n\}) \rightarrow s2 \{n, t\}\)). The access mode of the formal parameters and the result type is still applicable.

### 3.4.7. Synonym Functions

A synonym function does not introduce a function body. It simply specifies how to call another function. Synonym functions are useful for defining variations of a general-purpose function. Synonym functions can be handled efficiently; the compiler replaces a call to a synonym function by an equivalent call to the synonymous function.

A \textit{synonym function} definition declares a new function name or signature and makes it synonymous to a call to an existing function name. For instance, the summation of two complex numbers can be a function call to the function that generates a new complex number given its real part and the
imaginary part. It is noticeable that the summation of real numbers is used in the parameters passed to the function complex.

\[
\text{function } + (x, y: \text{ complex}): \text{ complex is}
\]
\[
\text{function } \text{ complex}(x.\text{re} + y.\text{re}, x.\text{im} + y.\text{im})
\]

3.5. Basic Polymorphic Types, Operators, and Functions

Every programming language needs some basic types, operators, and functions upon which other types and functions can be constructed. The SIMPL language defines four scalar non-parameterized core types and three parameterized ones.

The four scalar types are:

1. integer (32-bit signed). In practice, we may need more than one integer type for space efficiency reasons.

2. real (64-bit IEEE 754-1985 double-precision floating-point standard). In practice, we may need more than one floating-point type for space efficiency reasons.

3. char. It is one of the enumeration types.

4. boolean. It is one of the enumeration types. The enumeration literals for the boolean type are 'true' and 'false' (enclosed by single quotes\(^2\)).

In addition to the four basic scalar types, SIMPL defines three parameterized core types:

1. The reference type ref\{t\}. The reference type has a single parameter that specifies the type of the target object being referenced. A reference object carries either the null address, or the address of a dynamically allocated object.

\(^2\) All enumeration literals are enclosed by single quotes to distinguish them from identifiers. String literals are enclosed by double quotes. Enumeration and string literals are case sensitive, while identifiers and keywords are not.
2. The statically sized array type $\text{array}\{n, t\}$ has two formal parameters that specify the length of the array and the type of its elements. All array objects start at index zero. Array objects of length zero are allowed. Array objects of a negative length are rounded to zero.

3. The dynamically sized array type $\text{array}\{t\}$ has a single parameter that specifies the array element type. The length is not specified. The new function is used to allocate a dynamic array object of a given length.

Polymorphic functions require the existence of some basic polymorphic operators. SIMPL offers the following polymorphic operators:

1. The assignment operator is polymorphic and can be applied to any type $t$. Its interface is

   $$\text{function} := \{t: \text{type}\}(\text{obj} \text{ object: } t, \text{ expression: } t)$$

   The assignment operator copies all the bytes of an expression to an object of the same type $t$.

2. The equality operators are polymorphic and can be applied to any type $t$. Their interfaces are

   $$\text{function} = \{t: \text{type}\}(\text{expr1, expr2: } t): \text{ boolean}$$

   $$\text{function} <> \{t: \text{type}\}(\text{expr1, expr2: } t): \text{ boolean}$$

   The equality operators (= and <>) compare two expressions byte by byte as long as their types are identical.

SIMPL provides some core polymorphic functions. The new functions shown below are also examples of core polymorphic functions.
1. The first `new` function is used to allocate a dynamic object of type `t` and returns its address of type `ref{t}` through the object parameter `r`. Its interface is

   ```
   function new {t: type}{obj r: ref{t}}
   ```

2. The second `new` function is used to allocate a dynamic array of length `n`. A reference to the dynamic array as well as its length `n` are then stored in the object parameter `a` [Mudawwar98a].

   ```
   function new {t: type}{obj a: array{t}, n: integer}
   ```
4. SUBTYPING BY SPECIALIZATION

SIMPL provides parametric polymorphism and subtyping via instantiating parameterized types. That is, any type instance is a subtype of its base type. Subtyping by instantiation - or we may call it subtyping by specialization - is applicable to types and functions. However, the subtyping hierarchy arranges the type instances with respect to each other. It is the fact that a parameterized type instance holds instances of its formal type parameters; these instances undergo subtyping too. The same idea applies to functions; a function is a mapping among types. Thus, the subtyping relation between the formal parameters of any two functions and the subtyping relation between the result types of the same functions determine the subtyping relation between these functions. The following sections illustrate the concept of subtyping by specialization, the rules of subtyping among types and among functions.

4.1. The Concept of Subtyping by Specialization

As long as we consider types to be sets of values, subtypes can be regarded as subsets. All types are members of TYPE where TYPE is the set of all types.

\[ \forall s \text{ is a specific type, } s \in \text{TYPE} \]

That is, TYPE is a set of sets. Any type variable \( t \) is a member of TYPE.

\[ \forall t \text{ is a specific type, } t \in \text{TYPE} \]

Since type variables can be any type, a type variable is considered a family of types i.e. an unlimited union of types. However, the set of values that purely belong to a type variable and not to any other type is the intersection of all types. Since any type is a set of values, it can be a subset of the type variable set. That is, any specific type \( s \) is a subtype of any type variable \( t \).

\[ \forall s \text{ is a specific type, } s \subseteq t \]
Moreover, parameterized types are actually a family of some types. The values held only by the parameterized type are the intersection of all types that can be derived from it. For instance, $\forall \{n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, t\}$ is a set that includes all the elements of the set $\forall \{t: \text{TYPE}\} \text{array} \{3, t\}$ and those of the set $\forall \{n: \text{integer}\} \text{array} \{n, \text{real}\}$ and much more. However, all the arrays of real elements belong to the set $\forall \{n: \text{integer}\} \text{array} \{n, \text{real}\}$, and all arrays that hold three elements belong to the set $\forall \{t: \text{TYPE}\} \text{array} \{3, t\}$. That is,

$$\forall \{n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, t\} = \bigcap_{i=0}^{\text{Maxinteger}} \forall \{t: \text{TYPE}\} \text{array} \{i, t\}$$

Accordingly, $\forall \{t: \text{TYPE}\} \text{array} \{3, t\} : \forall \{n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, t\}$ and $\forall \{n: \text{integer}\} \text{array} \{n, \text{real}\} : \forall \{n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, t\}$. Hence, any type instance is a subtype of its parameterized type. The subtyping hierarchy is constructed according to sorting the types based on their parameters and the hierarchy of instantiation of these parameters. That is, a parameterized type $s \{3, \text{real}\}$ is subtype of $\forall \{t: \text{TYPE}\} s \{3, t\}$ which is a subtype of $\forall \{n: \text{integer}, t: \text{TYPE}\} s\{n, t\}$ and at the top of this hierarchy resides any type variable $t$.

![Subtyping by Instantiating Type Instances](Image)

Figure 4.1. Subtyping by Instantiating Type Instances

56
4.2. Subtyping by Specialization

SIMPL defines a subtyping relation among types based on the rule that a subtype must be a type instance of a parameterized type. A type \( s_1 \) is more specific than another type \( s_2 \) based on two reasons:

1. **Specialization**: \( s_1 \) is a subtype of \( s_2 \), if \( s_1 \) is a type instance of \( s_2 \) where the type parameters of \( s_1 \) are instances or special cases of the type parameters of \( s_2 \). Since the parameters of a parameterized type can be integer variables or type variables, their specialization can be integer literals or type instances. As an example, \( \text{array} \{4, \text{complex}\} \) is a type instance of \( \forall\{t: \text{TYPE}\} \text{array} \{4, t\} \) because \text{complex} is an instantiation of the type variable \( t \), and \( \text{array}\{4, \text{complex}\} \) is a type instance of \( \forall\{n: \text{integer}\} \text{array} \{n, \text{complex}\} \) due to the specialization of the integer variable \( n \). In formal notation, specialization can be expressed as

\[
\begin{array}{c}
E \vdash \#n: \text{integer} & E \vdash t_1: \text{TYPE} & E \vdash m: \text{integer} & E \vdash t_2: \text{TYPE} \\
\hline
E \vdash \forall\{t_1: \text{TYPE}\} s_1 \{#n, t_1\} <: \forall\{m: \text{integer}, t_2: \text{TYPE}\} s_1 \{m, t_2\}
\end{array}
\]

\[
\begin{array}{c}
\text{(integer specialization)} \\
\text{(type instantiation)}
\end{array}
\]

2. **Constraints**: If \( s_1\{a, b\} \) is a type instance and \( s_1\{c, b\{c\}\} \) is another type instance that has more constraints, \( s_1\{c, b\{c\}\} \) is a subtype of \( s_1\{a, b\} \). The type instance with more constraints is more specific. For instance, \( \forall\{n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, n, t\} \) is more specific than \( \forall\{m: \text{integer}, n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, m, t\} \). Accordingly, the type instance with more constraints is a subtype of the type instance with less number of constraints. Formal notation represents constraints as follows:

\[
\begin{array}{c}
E \vdash s_3 \{\ldots\} & E \vdash n: \text{integer} & E \vdash t: \text{TYPE} \\
\hline
E \vdash \forall\{n: \text{integer}\} s_1 \{n, s_3 \{\ldots\}\} <: \forall\{n: \text{integer}, t: \text{TYPE}\} s_1 \{n, t\}
\end{array}
\]

\[
\begin{array}{c}
\text{(integer specialization)} \\
\text{(type instantiation)}
\end{array}
\]
Figure 4.2. illustrates a general subtyping hierarchy among type instances based on instantiating types and/or imposing more constraints on the formal type parameters.
4.3. Function Subtyping

SIMPL deals with function types more or less as data types. Thus, function types undergo subtyping. A function type f1 is a subtype of another function type f2, if f1 is a type instance of f2. This is achieved by three conditions:

1. If the signature type of f1 is a subtype of the signature type of f2
2. If the result type of f1 is a subtype of the result type of f2.
3. If the hidden parameters of f2 are substituted correctly in f1 and their constraints are preserved.

That is, function subtyping relation is covariant (in the same direction) with the signatures’ subtyping and the results’ subtyping. Moreover, function subtyping is another form of subtyping by instantiation. For instance, let f1 be \( \forall \{t: \text{TYPE}\} \ (x: \text{array}\ \{t\}) \rightarrow \text{array}\ \{t\} \) and f2 be \( \forall \{t: \text{TYPE}\} \ (x: t) \rightarrow t \) and \( f1 \prec \prec f2 \) because the signature types experience subtyping i.e. \( \forall \{t: \text{TYPE}\} \ (\text{array}\ \{t\}) \prec \prec \forall \{t: \text{TYPE}\} \ (t) \), and the result types undergo subtyping. Moreover, the constraint enforced on f2 - the same t must be in the formal parameter and the result type - is preserved in f1. That is, f1 becomes the instantiation of f2 with substituting certain values for the hidden parameters of f2.

This can be expressed in the form of the following type rule.

\[
\begin{align*}
E &\vdash \text{Sig1} \prec \prec \text{Sig2} \\
E &\vdash \text{Rtype1} \prec \prec \text{Rtype2} \\
E &\vdash \text{ftype2} \ {n: \text{integer, t2: \text{TYPE}}} \Rightarrow \text{Sig2} \Rightarrow \text{Rtype2} \\
E &\vdash \text{ftype1} \ {m: \text{integer, t1: \text{TYPE}}} \Rightarrow \text{Sig1} \Rightarrow \text{Rtype1} \Rightarrow \text{ftype2} \ {n \leftarrow m: \text{integer, t2 \leftarrow t1: \text{TYPE}}} \\
\hline
E &\vdash \text{ftype1} \ {m: \text{integer, t1: \text{type}}} \prec \prec \text{ftype2} \ {n: \text{integer, t2: \text{type}}}
\end{align*}
\]

Function subtyping is applicable to functions. That is, a function is a subtype of another function if they follow the above conditions. This is similar to subtyping among type instances. An important point to mention is that function subtyping does not require the use of the same function name. As long as the two functions experience subtyping among their signature types, and among their result types in addition to preserving the constraints on the hidden parameters, they are subtypes.
In order to check for function subtyping, there must be a clear procedure to detect the above mentioned three conditions. Let us have a closer look at each of these conditions.

### 4.3.1. Signature Types Subtyping

The signature type is an aggregate of parameter types; thus, if the type of each parameter in a signature type $\text{Sig1}$ is a subtype of its corresponding parameter’s type in the other signature type $\text{Sig2}$, then $\text{Sig1}$ is a subtype of $\text{Sig2}$.

$$
E \vdash \forall \{t_1, n_1, \ldots, t_n\}, \{t_1, n_1, \ldots, t_n\} : \text{Sig1} \ E \vdash \forall \{t_2, n_2, \ldots, t_{2n}\}, \{t_2, n_2, \ldots, t_{2n}\} : \text{Sig2} \\
E \vdash t_1 \leq t_2 \ E \vdash n_1 \leq n_2 \ \ldots \ E \vdash t_n \leq t_{2n} \\
---------------------------------------- \\
E \vdash \text{Sig1} \leq \text{Sig2}
$$

![Figure 4.3. Subtyping of the Type Instances Listed in the Function Signatures of the Example](image-url)
Using the subtype relationship illustrated in figure 4.3, the signature subtype relationship of these function types will be as follows:

\[
\forall \{n: \text{integer}, t: \text{TYPE}\}(\text{array}\{n, t\}, \text{integer}, \text{matrix}\{n, t\}, t) \rightarrow t
\]

\[
\forall \{n: \text{integer}\}(\text{array}\{n, \text{real}\}, \text{integer}, \text{matrix}\{n, \text{real}\}, \text{real}) \rightarrow \text{integer}
\]

\[
\forall \{t: \text{TYPE}\} (\text{array}\{3, t\}, \text{integer}, \text{matrix}\{3, t\}, t) \rightarrow \text{array}\{3, t\}
\]

(array\{3, real\}, integer, matrix\{3, real\}, real) \rightarrow real

(array\{5, real\}, integer, matrix\{5, real\}, real) \rightarrow real

\[
\forall \{n: \text{integer}, t: \text{TYPE}\} (\text{array}\{n, t\}, \text{integer}, \text{matrix}\{n, t\}, t)
\]

\[
\forall \{t: \text{TYPE}\} (\text{array}\{3, t\}, \text{integer}, \text{matrix}\{3, t\}, t)
\]

\[
\text{array}\{3, \text{real}\}, \text{integer}, \text{matrix}\{3, \text{real}\}, \text{real}
\]

\[
\text{array}\{5, \text{real}\}, \text{integer}, \text{matrix}\{5, \text{real}\}, \text{real}
\]

**Figure 4.4. Function Signature Subtype Relationship**

### 4.3.2. Result Types Subtyping

The result type of any function can be any type instance whether parameterized or not. The subtyping relation among result types follows the subtyping rules for type instances. This takes into consideration that constraints on hidden parameters must be applicable in the type parameters of the result type – if it is a parameterized type.
4.3.3. Preserving Hidden Parameters Constraints

It is worth mentioning that function subtyping does not require the same number of hidden parameters but rather an appropriate substitution for each hidden parameter in the supertype function type. This can take place through specialization of the formal function parameters using the rules discussed in the previous section. The following type rule encompasses a formal representation of hidden parameters' specialization.

\[
\begin{align*}
\text{(Hidden parameter specialization)} & \quad (n\leftarrow\#n) \\
E \vdash \text{Sig1} \llt \text{Sig2} & \quad E \vdash \text{Rtype1} \llt \text{Rtype2} \\
E \vdash \text{ftype2} \{n: \text{integer, t2: TYPE}\} = \text{Sig2} \rightarrow \text{Rtype2} & \\
E \vdash \text{ftype1} \{t1: \text{TYPE}\} = \text{Sig1} \rightarrow \text{Rtype1} = \text{ftype2} \{n\leftarrow\#n: \text{integer, t2}\leftarrow\text{t1}: \text{TYPE}\}
\end{align*}
\]

\[
\begin{align*}
E \vdash \text{ftype1} \{t1: \text{type}\} \llt \text{ftype2} \{n: \text{integer, t2: type}\}
\end{align*}
\]

\[
\begin{align*}
\text{(Hidden parameter specialization)} & \quad (t2\leftarrow s\{\ldots\}) \\
E \vdash \text{Sig1} \llt \text{Sig2} & \quad E \vdash \text{Rtype1} \llt \text{Rtype2} \\
E \vdash \text{ftype2} \{n: \text{integer, t2: TYPE}\} = \text{Sig2} \rightarrow \text{Rtype2} & \\
E \vdash \text{ftype1} \{m: \text{integer}\} = \text{Sig1} \rightarrow \text{Rtype1} = \text{ftype2} \{n\leftarrow m: \text{integer, t2}\leftarrow s\{\ldots\}: \text{TYPE}\}
\end{align*}
\]

\[
\begin{align*}
E \vdash \text{ftype1} \{m: \text{integer}\} \llt \text{ftype2} \{n: \text{integer, t2: type}\}
\end{align*}
\]
5. TYPE SYSTEM FOR SIMPL LANGUAGE

The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program. Thus, formal type systems should be part of the definition of all typed programming languages. The so-called second-order type systems can model parametric polymorphism and subtyping by instantiation in SIMPL. Accordingly, the type system of SIMPL follows the second-order type systems. Such systems extend first-order type systems with the notion of type parameters. A type variable is represented by \( X \) to indicate any arbitrary type. A new kind of terms \( \lambda(X)b \) denotes that a program \( b \) is parameterized with respect to a type variable \( X \) [Cardelli96]. For instance, the parametric function

\[
\text{function } F(\ t: \text{type}): \text{integer}
\]

can be represented as

\[
F \triangleleft \forall X. \lambda(x: X)b: B
\]

where \( b \) represents the function body and its type \( B \) stands for the type integer (i.e. the return type). The instantiation of this parametric function with a certain type \( A \) (called type instantiation)

\[
F(A),
\]

produces

\[
\lambda(x: A)b: B
\]

However, SIMPL is based on statements rather than expressions. Hence, a function body in SIMPL has no type. This leads to a modification of the function representation into a function name, list of parameters and the result type. For instance, \( \lambda(x: A): B \) where \( \lambda \) is the function name, \( x \) is the formal parameter of type \( A \) and \( B \) is the result type.

System \( F \) is a well-known typed \( \lambda \)-calculus with polymorphic types, which provides a basis for polymorphic programming languages. System \( F_2 \) is a pure second-order type system. It extends

---

\( ^3 \) Representing the type of the function body is optional.
system F with type variables, polymorphic abstractions and quantified types. \( F_{2<:} \) (pronounced \( ef2\)-sub) is an extension of \( F_2 \) that combines parametric polymorphism with subtyping introduced by [Cardelli96]. \( F_{2<:} \) is closely related to the language \( F_\leq \) identified by Curien. \( F_\leq \) is a fragment of the language Fun. SIMPL Type System relies on the subtyping features in \( F_{2<:} \) system.

In the rest of this chapter, we will introduce the syntax of the Type System for SIMPL: judgements, inference rules, and equivalence rules.

First of all, let us settle on the following notation:

\( A \leq B \) indicates a subtyping relationship between \( A \) and \( B \). That is, \( A \) can be the same type as \( B \) or can be strictly a subtype of \( B \).

\( A <: B \) indicates proper subtyping. That is, \( A \) is strictly a subtype of \( B \) and cannot be the same type as \( B \).

### 5.1. Syntax of the Type System for SIMPL

- **S, A, B**: type declaration
- **TYPE**: the set of all types
- (obj A, val B): a tuple type
- (A)→B: function type
- \( \forall \{n: \text{integer}, t: \text{TYPE}\} \ S\{t, n\} \): universally quantified type (parameterized type)
- a, b, x, y: variables
- n, m: integer variable
- #n: literal
- s: type instance
- t: type variable
- (obj a: A, val b: B): a tuple
- f(obj a: A, val b: B): C: function
∀{n: integer, t: TYPE} f (a: A{t, n}): B  
function with hidden parameters

f(a)  
application

∀{n: integer} s(a, A, c{A, n})  
type instantiation (a parameterized type instance)

- As the syntax illustrates, the symbol S or A or B denotes a specific type whether it is parameterized or not.

- TYPE is the set of all types. It is the universal set whose elements are type sets.

- A tuple type is the type of an aggregate of parameters. This type includes the mode of each parameter whether obj or val.

- Any function type is represented as a mapping from the signature type, which is an aggregate type, to the result type.

- A universally quantified type is a parameterized type. It is followed by a list of its formal type parameters. That is, a type variable and an integer variable parameterize the universally quantified type S. SIMPL allows type parameterization through type variables and/or integer variables only.

On the other hand, values include the following:

- variables denoted by a, b, x etc.

- Integer variables are denoted by n or m.

- Literal values are denoted by # followed by a variable name.

- Any type instance is denoted by s.

- A type variable is denoted by t.

- A tuple is the aggregate of parameters including their mode of access.
Function implementations constitute another form of values, where the list of hidden parameters - if any - can be type variables or integer variables. The hidden parameters (in case of parametric polymorphic functions) are followed by the function name – \( f \) is replaced by the function name – then comes an aggregate of the formal parameters with their types. Finally comes the result type – if it is not explicitly stated, it is considered void type. Formal parameters and result type are preceded by their mode of access (\( \text{obj} \) or \( \text{val} \)). Object parameters allow read and write access to this parameter within the function, whereas \( \text{val} \) parameters are pass by value, that is, read-only values. By the way, the result type can be parameterized via the same set of hidden parameters (e.g. \( \forall \{n: \text{integer}, t: \text{TYPE}\} f\{n, t\}(a: S1\{t, n\}): S2\{n, t\} \)).

SIMPL deals with function types more or less as data types. Accordingly, function types can be parameterized. \( \forall \{n: \text{integer}, t: \text{TYPE}\} f\{n, t\}(a: S1\{t, n\}): S2 \) is a parameterized function of type \( \forall \{n: \text{integer}, t: \text{TYPE}\} f\text{type}\{n, t\} (S1\{t, n\}) \rightarrow S2 \) where \( S1 \) is a parameterized type whose parameters are passed as hidden parameters to the function. Mentioning the function type name is optional. Of course, the access mode of the formal parameters and the result type is still applicable.

Another form of values is the application of a variable to a function, i.e. a function call with appropriate actual parameters. Actual parameters must be of the same type of the formal parameters or its subtype. Moreover, the type inference algorithm must check conformance of the actual parameters to the access mode of the formal parameters. The access mode rules state that an \( \text{obj} \) formal parameter must be called with an object actual parameter whereas a \( \text{val} \) formal parameter can be instantiated by an object or a constant actual parameter. The function is called with the actual parameters with no explicit mention of the hidden parameters. The type inference algorithm infers the value of the hidden parameters accordingly.
The final form of values is type instantiation where a parameterized type is instantiated with actual parameters. Type instantiation of data types leads to type instances of parameterized types where a type instance is a subtype of its parameterized type.

### 5.2. Judgements for SIMPL Type System

1. \( E \vdash \diamond \)  
   \( E \) is a well-formed environment

2. \( E \vdash A \text{ type} \)  
   \( A \) is a well-formed type in \( E \)

3. \( E \vdash \forall \{n: \text{integer}, t: \text{TYPE}\} S\{n, t\} \)  
   \( S \) is a well-formed parameterized type in \( E \)

4. \( E \vdash a: A \)  
   variable \( a \) has type \( A \)

5. \( E \vdash s1\{m, s2\}: \forall \{n: \text{integer}, t: \text{TYPE}\} S\{n \leftarrow m, t \leftarrow s2\} \)  
   \( s1 \) is type instance of type \( S \)

6. **Equality Judgement**
   \( E \vdash a \leftrightarrow b: A \)  
   \( a \) and \( b \) are equal members of type \( A \)

7. \( E \vdash A \in \text{TYPE} \)  
   any type is a member in the set \( \text{TYPE} \)

8. \( E \vdash t \in \text{TYPE} \)  
   a type variable is a member in the set \( \text{TYPE} \)

9. \( E \vdash A \leq: A \)  
   any type is a subtype of itself

10. \( E \vdash A <: t \)  
    any type is a subtype of any type variable

11. **Subtyping Judgement**
    \( E \vdash s1\{\ldots\} \leq: S\{\ldots\} \)  
    subtyping by specialization

Let’s explain each of the above judgements in details.

1. \( E \) is a well-formed environment i.e. it has been properly constructed and all its contents follow the typing rules.

2. \( A \) is a well-formed type in \( E \). This judgement declares a new type named \( A \).

3. \( S \) is a well-formed parameterized type in \( E \). The judgement states that the allowed type parameters can be integer variables or type variables.
4. The variable $a$ has type $A$. This judgement declares a new value (a variable, a function, an application, a bounded type function, or a type instantiation) of type $A$.

5. $s1$ is type instance of type $S$. This judgement instantiates a parameterized type.

6. Equality Judgement states that $a$ and $b$ are equal members of type $A$. Equality Judgement is important since values in $F<-$ can have many types; accordingly two values may or may not be equivalent depending on the type that those values are considered as processing.

7. Any type is a member of the set $\text{TYPE}$.

8. Any type variable is a member of the set $\text{TYPE}$.

9. Any type is a subtype of itself. This judgement illustrates the reflexive property of subtyping. This applies to parameterized types and type variables too i.e. any type variable is a subtype of itself.

10. Any type is a subtype of any type variable. This judgement sets the top of the subtyping hierarchy.

11. Subtyping Judgement states that any type instance is a subtype of its parameterized type. Type instantiation states that the parameters of the type instance are subtypes or instances of their corresponding parameters in the parameterized type. Subtyping Judgement can deduce the other judgements of subtyping. It illustrates a reflexive and transitive relation on types with a subsumption property [Cardelli91].
5.3. Inference Rules

Note that \( \text{dom}(E) \) is the set of variables defined by an environment \( E \).

\[ B[t \leftarrow A] \] indicates that \( A \) substitutes \( t \) in \( B \). It is a means of instantiation.

**Environments**

\[(\text{Env } \emptyset)\]

\[\emptyset \vdash \diamond\]

\[(\text{Env } x)\]

\[x \notin \text{dom}(E)\]

\[\begin{array}{c}
E, x: A \vdash \diamond
\end{array}\]

\[(\text{Env } t)\]

\[E \vdash A \text{ type } \ t \notin \text{dom}(E)\]

\[\begin{array}{c}
E, t \in \text{TYPE} \vdash \diamond
\end{array}\]

\[(\text{Env } \emptyset)\text{ states that } \emptyset \text{ is a well-formed environment.}\]

\[(\text{Env } x)\text{ declares a new well-formed environment constituted of the variable } x \text{ of type } A \text{ (that is not declared in environment } E) \text{ and the well-formed environment } E.\]
(Env t) declares a new well-formed environment constitutes the type variable t - that is a member of TYPE (t is not declared in environment E) - and the well-formed environment E.

Types

(Type t)
E, t ∈ TYPE, E' ⊢ ⊤

________________________________________
E, t ∈ TYPE, E' ⊢ t

(Type TYPE)
E ⊢ ⊤

____________________
E ⊢ TYPE type

(Type →)
E ⊢ A type  E ⊢ B type

____________________________________
E ⊢ (A)→B

(Type ∀)
E, t, n: integer ⊢ B type

____________________________________
E ⊢ ∀{n: integer, t: TYPE} B(t, n)

(Type t) states that as long as the environment E, t ∈ TYPE, E' is a well-formed environment, we can declare a type variable t in this environment.

(Type TYPE) asserts that in any well-formed environment E, there is the type TYPE which is not a subtype of any other type.
(Type $\rightarrow$) declares a function type $(A) \rightarrow B$ as long as the environment includes the declared types $A$ and $B$. It is the set of all functions mapping tuple type $(A)$ to type $B$.

(Type $\forall$) declares a parameterized type $B$ whose type parameters - the type variable $t$ and the integer variable $n$ – are already declared in the environment.

Subtypes

(Sub reflexive)

\[ E \vdash A \text{ type} \]

\[ \]

\[ E \vdash A \leq: A \]

(Sub transitive)

\[ E \vdash s_1\{\ldots\} \leq: s_2\{\ldots\} \quad E \vdash s_2\{\ldots\} \leq: s_3\{\ldots\} \]

\[ \]

\[ E \vdash s_1\{\ldots\} \leq: s_3\{\ldots\} \]

(Sub $\subseteq$)

\[ E \vdash A \text{ type} \]

\[ \]

\[ E \vdash A \leq: t \]

(Sub $\rightarrow$)

\[ E \vdash S_1\{\ldots\} \leq: S_2\{\ldots\} \quad E \vdash S_3\{\ldots\} \leq: S_4\{\ldots\} \]

\[ \]

\[ E \vdash (S_1\{\ldots\}) \rightarrow S_3\{\ldots\} \leq: (S_2\{\ldots\}) \rightarrow S_4\{\ldots\} \]

(Sub $\forall$)

\[ E \vdash s(s_1, m): \forall\{n: \text{integer}, t: \text{TYPE}\} S\{t \leftarrow s_1, n \leftarrow m\} \]

\[ \]

\[ E \vdash s(s_1, m) \leq: \forall\{n: \text{integer}, t: \text{TYPE}\} S\{t, n\} \]
(Sub \( \forall t \))

\[
E \vdash s_1\{\ldots\} \leq s_2\{\ldots\} \quad E, t: \text{TYPE} \vdash b(t): B(t)
\]

\[
E \vdash b(t \leftarrow s_1\{\ldots\}) \leq b(t \leftarrow s_2\{\ldots\})
\]

(Sub \( \forall \text{l literal} \))

\[
E \vdash \#n: \text{integer} \quad E \vdash b[n]: \forall\{n: \text{integer}\} B[n] \quad E \vdash m: \text{integer}
\]

\[
E \vdash b[n \leftarrow \#n] \leq b[n \leftarrow m]
\]

(Sub reflexive) states that subtyping is reflexive. That is, every type is a subtype of itself.

(Sub transitive) states that subtyping is transitive. That is, every type is a subtype of the supertype of its supertype. These types have to be parameterized types.

(Sub \( \leq:\)) states that any type is a subtype of any type variable, since all type variables are not restricted by subtyping relations. Actually, type variables are equivalent types.

(Sub \( \rightarrow \)) declares the subtyping of function types according to the following conditions. For the function type \( f_1 \) to be a subtype of another function type \( f_2 \), the signature type of \( f_1 \) must be a subtype of \( f_2 \)'s signature and the return type of \( f_1 \) must be a subtype of the return type of \( f_2 \). For instance, if \( \text{array}\{\text{integer}\} \leq \forall\{t: \text{TYPE}\} \text{array}\{t\} \) and \( \text{matrix}\{ \text{b}\{\text{real}\} \} \leq \text{matrix}\{ \forall\{t: \text{TYPE}\} \text{b}\{t\} \} \) then \( \text{array}\{\text{integer}\} \rightarrow \text{matrix}\{ \text{b}\{\text{real}\} \} \leq \forall\{t: \text{TYPE}\} \text{array}\{t\} \rightarrow \text{matrix}\{ \forall\{t: \text{TYPE}\} \text{b}\{t\} \} \).

(Sub \( \forall \)) declares the subtyping relation between a type instance and its parameterized type.

(Sub \( \forall t \)) states subtyping between type instances based on the subtyping relation between the actual parameters. For instance, if \( \text{array}\{3, \text{real}\} \leq \forall\{n: \text{integer}, t: \text{TYPE}\} \text{array}\{n, t\} \) then \( \text{matrix}\{3, \text{array}\{3, \text{real}\}\} \leq \forall\{n: \text{integer}, t: \text{TYPE}\} \text{matrix}\{n, \text{array}\{n, t\}\} \).

(Sub \( \forall \text{l literal} \)) mentions the subtyping of type instances whose type parameter is an integer. The subtyping is based on the rule that any integer literal is an instance of the type integer. This is called subtyping by specialization, since an integer literal is a specialization of an integer variable.
Accordingly, any integer literal is a subtype of any integer variable. For instance, \texttt{stack}(3) <: \forall \{n: \text{integer}\} \texttt{stack}(n)$ since 3 is an instance of \texttt{integer}. Actually, 3 and $n$ are instances of \texttt{integer} but 3 is more specific than $n$.

**Values**

(Val $x$)

\[
E, x: A \vdash \hat{\diamond}
\]

\[
E, x: A \vdash x: A
\]

(Subsumption)

\[
E \vdash a: A \quad E \vdash A \leq B
\]

\[
E \vdash a: B
\]

(Val $t$)

\[
E \vdash \hat{\diamond}
\]

\[
E \vdash t: \text{TYPE}
\]

(Val function)

\[
E, x: A \vdash B \quad \text{type}
\]

\[
E \vdash f (x: A): B \quad : (A) \rightarrow B
\]

(Val application)

\[
E \vdash b: (A) \rightarrow B \quad E \vdash a: A
\]

\[
E \vdash b (a): B
\]
\[(\text{Val function2})\]
\[
E \mid B \; \text{type} \quad E, t, n: \text{integer} \vdash \forall\{t: \text{TYPE}, n: \text{integer}\} \; S\{t, n\} \; \text{type}
\]

\[
E \mid \forall\{t: \text{TYPE}, n: \text{integer}\} \; f\{t, n\}(S\{t, n\}) : B : (S\{t, n\}) \rightarrow B
\]

\[(\text{Val application2})\]
\[
E \mid b : \forall\{t: \text{TYPE}, n: \text{integer}\} \; B\{t, n\} 
E \mid A' \leq t 
E \mid m: \text{integer}
\]

\[
E \vdash b \{A', m\} : B\{t \leftarrow A', n \leftarrow m\}
\]

\[(\text{Val } x)\) declares a variable \(x\) of type \(A\)

\[(\text{Subsumption})\) states that a member of a type is also a member of any supertype of that type. For instance, if \(m: \text{array}\{3, \text{real}\}\) where \(\forall\{n: \text{integer}, t: \text{TYPE}\} \; \text{array}\{n, t\}\) then \(m: \forall\{n: \text{integer}, t: \text{TYPE}\} \; \text{array}\{n, t\}\). Moreover, it proves that every type is a subtype of \(t\). A type power is hidden in the subsumption rule, where a function can take an argument of a supertype of its input type. For instance, the function \(\forall\{t: \text{TYPE}\} \; f\{t\}(x: \text{array}\{t\})\) can be called as \(f(a)\) where \(a: \text{array}\{\text{integer}\}\) since \(\text{integer} \prec t\) according to \((\text{Sub } t)\) rule.

\[(\text{Val type})\) declares a type variable \(t\) of type \(\text{TYPE}\)

\[(\text{Val function})\) declares the function implementation of a function \(f\) whose formal parameter is \(x\) of type \(A\) and its return type is \(B\). Thus, the function type is \((A) \rightarrow B\)

\[(\text{Val Application})\) states the application of a function \(b\) of type \((A) \rightarrow B\) with an actual parameter \(a\) of type \(A\). Accordingly, the result of this function call is \(B\).

\[(\text{Val function2})\) declares the function implementation of a parameterized polymorphic function \(f\{t, n\}\) whose hidden parameters are the type variable \(t\) and the integer variable \(n\). The formal parameter \(\forall\{t: \text{TYPE}, n: \text{integer}\} \; S\{t, n\}\) is a parameterized type whose parameters are already declared through the hidden parameters. The result type is \(B\). Accordingly, the function type is a mapping from the tuple of the parameterized type \(S\{t, n\}\) to type \(B\).
(Val Application2) declares the type instantiation of a parametric type. $b(A')$ is the type instance of the parametric type $\forall t: \text{TYPE} \ B(t)$ where all occurrences of $t$ in type $B$ is substituted with $A'$.

**Equivalence**

(Eq symmetric)

$E \vdash a \leftrightarrow b: A$

\[\]

$E \vdash b \leftrightarrow a: A$

(Eq transitive)

$E \vdash a \leftrightarrow b: A$ \quad $E \vdash b \leftrightarrow c: A$

\[\]

$E \vdash a \leftrightarrow c: A$

(Eq reflexive)

$E \vdash x: A$

\[\]

$E \vdash x \leftrightarrow x: A$

(Eq collapse)

$E \vdash t1: \text{TYPE}$ \quad $E \vdash t2: \text{TYPE}$

\[\]

$E \vdash t1 \leftrightarrow t2: \text{TYPE}$

(Eq name)

$E \vdash s1: S$ \quad $E \vdash s2: S$ \quad $E \vdash \text{name}(s1) = \text{name}(s2)$

\[\]

$E \vdash s1 \leftrightarrow s2: S$
(Eq structural)
\[ E \vdash f_1: (A) \rightarrow B \quad E \vdash f_2: (A) \rightarrow B \]

\[ \text{---------------------------------------------} \]
\[ E \vdash f_1 \leftrightarrow f_2: (A) \rightarrow B \]

(Eq symmetric) states that equivalence of two values of the same type is symmetric.

(Eq transitive) states that the transitivity of equivalence; if \( a \) and \( b \) are equal members of type \( A \), and \( b \) and \( c \) are equal members of type \( A \) then \( a \) and \( c \) are equivalent.

(Eq reflexive) illustrates that any member is equivalent to itself in terms of its type.

(Eq collapse) states that any two values are equivalent when “seen” at type TYPE [Cardelli91].

(Eq name) the name equivalence rule states that two type instances are type equivalent if they share the same name and same type.

(Eq structural) the structural type equivalence rule states that two functions are type equivalent if they share the mapping from the same tuple type to the same result type.

These rules illustrate the typing rules in SIMPL. They are heavily used in proving subtyping, overloading and specialization theorems that show up in the following chapters.
6. SEMANTIC INFORMATION REPRESENTATION

6.1. General Structure

Each programming language uses a symbol table to handle all the symbols mentioned within a program. The symbol table provides a correlation between all the different occurrences of each name throughout the program and hence provides a link between each name and its declaration, whether implicit or explicit, in the source text [Bornat79]. The way of representing the semantic information affects the efficiency of the type-checking algorithm. The SIMPL parser stores the extracted semantic information in the symbol table and the attribute space. The symbol table indexes the declared symbols in the program. Each symbol points to its associated attributes in the attribute space.

6.1.1. Symbol Table Structure

The symbol table is an indexed array of symbols. Each symbol is an entry of two fields: the name field and the attribute field. The name field is a pointer to the string space that holds the names of the symbols. The attribute field is a pointer to the attribute space that holds all the needed attributes of the corresponding symbol. During compilation, the program produces a separate symbol table for each scope. For instance, the module has a symbol table, while each function has its symbol table that is created upon compilation of the function body. Once the function is compiled its symbol table is deleted and we return back to the global symbol table (the module symbol table). Moreover, to speed up the lookup for types we can create a separate symbol table to store these types called the type symbol table. Function types are dealt as data types; accordingly they are stored in the type symbol table. Furthermore, function interfaces are gathered in one symbol table that is named the function symbol table. This symbol table includes an extra field called the next field. This field is a pointer to another entry holding a function sharing the same name of the current function entry. This field is needed for function overloading and specialization.
After full compilation of the module, the symbol tables and the attached attribute space are deleted except for the symbols exported to other programs through the interface. The interface for the symbol table structure as represented in SIMPL syntax is:

```simpl
type anysymbol is
  name: string  -- pointer to the name
  attribute: attref  -- reference to an
  -- attribute entry
end

-- type anysymbol


type fsymbol is
  next: ref{fsymbol}  -- pointer to the next
  name: string  -- pointer to the name
  -- stored in the string space
  attribute: attref  -- reference to an
  -- attribute entry
end

-- type fsymbol


type symbol is type anysymbol or fsymbol


type symtable{s: symbol}

function new(obj table: symtable{symbol})
  -- creates a new symbol table

function delete(obj table: symtable{symbol})
  -- deletes a symbol table

function lookup(table: symtable{symbol}, s: string): ref{symbol}
  -- searches for the symbol whose name matches the passed string
  -- It returns a pointer to the first matching symbol, the
  -- remaining matching symbols are linked to the 1st symbol

function enter(table: symtable{symbol}, s: string): ref{symbol}
  -- enters a new symbol in the symbol table

function name(s: symbol): string
  -- returns the name of the symbol pointed to by s

function attribute(s: symbol): attref
  -- returns a pointer to the attribute entry of the symbol pointed
  -- to by s

function next(s: ref{fsymbol}): ref{fsymbol}
  -- returns a pointer to the symbol entry linked next to the
  -- symbol pointed to by s
```
6.1.2. Attribute Space Structure

Each symbol in the symbol table (any of the above mentioned tables) has associated information or attributes needed for type checking. These attributes are stored in the attribute space. The attribute space is an array of entries. Each entry consists of two fields: the tag field and the value field. The tag field is an enumerated type field indicating the kind of the symbol. The value field holds the needed value for each kind of symbols. Table 6.1. - at the end of the chapter - summarizes the different tags that can be used and their corresponding value fields. These tags are explained in details in the following sections. It is worth mentioning that the ref tag can be used in various places, yet it has one meaning; it holds a pointer to another entry in the attribute space. The meaning of this pointer differs according to its position within a set of entries. Its specific uses will appear during the explanation of the other tags. Note that the notation @integer that is used in the attribute space represents a pointer to the attribute entry for the built-in integer type. The same applies to any other type whether it is parameterized or not.

The following code represents the needed interface of the global attribute space.

```plaintext
type tagtype is 'alttypes', 'binding', 'call', 'type', 'eliteral', 'field', 'fliteral', 'function', 'ftype', 'hidden', 'iliteral', 'intpar', 'obj', 'offset', 'ref', 'result', 'size', 'sliteral', 'synfun', 'syntype', 'typeinst', 'tuple', 'typevar', 'uniontype', 'val'}

type valtype is type integer or ref{attribute}

type attribute is
    tag: tagtype
    value: valtype
end -- attribute

type attref is type ref{attribute}

type semantic_space

function grow(num_entries: integer): attref
    -- allocates num_entries of entries in the global semantic space

function shrink(num_entries: integer): attref
    -- deallocates num_entries of entries from the global semantic space
```
function tag(a: attref): tagtype
-- returns the tag of the passed attribute entry pointed to by a

function value(a: attref): valtype
-- gets the value of the passed attribute entry pointed to by a

function setvalue(a: attref, v: valtype)
-- sets the value of the attribute entry, pointed to by a, to
-- the value v

function settag(a: attref, v: tagtype)
-- sets the tag of the attribute entry, pointed to by a, to
-- the tag value v

function +(a: attref, offset: integer): attref
-- returns the address of the attribute entry that follows
-- the entry pointed to by a by offset

6.2. Objects and Named Values Representation

SIMPL declare objects and named values. Objects are modifiable (mutable) locations in memory, whereas named values are single-assigned or immutable locations in memory. This is expressed via the keywords obj and val that indicate the mode of the identifier – it can be a parameter, a field or a local variable. The val tag in the attribute space represents these named values. This entry points to the type of the identifier. The obj tag represents object variables and points to their types. For instance, the following declared value x and the object a can be represented in the semantic space as follows.

val x: real
obj a: real

<table>
<thead>
<tr>
<th>Name</th>
<th>Attribute</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>val</td>
<td>@real</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>obj</td>
<td>@real</td>
</tr>
</tbody>
</table>

Figure 6.1. Representing Objects and Named Values in the Semantic Space
6.3. Literals Representation

SIMPL provides four kinds of literals: integer literals, floating point literals, enumeration literals and string literals. To represent them in the attribute space, we use four different tags:

1. **iliteral**: this tag indicates an integer literal. The associated value field holds the literal value.

2. **fliteral**: this tag indicates a floating-point literal. The associated value field is a pointer to where the floating-point value is stored.

3. **eliteral**: this tag indicates an enumeration literal. The associated value field is a pointer to its type. The following entry in the attribute space holds an **iliteral** tag and the value field holds the integer value associated to this enumeration literal in its type. SIMPL allows more than one enumeration literal, of the same type, to hold the same integer value; accordingly more than enumeration literal can point to the same **eliteral** tag in the attribute space to achieve the same **iliteral** value. For instance, the declaration of the enumeration type boolean may include both 'true' of value 0 and 'True' of value 0 as shown in figure 6.2.

   Type boolean is {'true' is 0, 'false' is 1, 'True' is 0, 'False' is 1}

   

<table>
<thead>
<tr>
<th>Name</th>
<th>Attribute</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'true'</td>
<td></td>
<td>eliteral</td>
<td>@boolean</td>
</tr>
<tr>
<td>'True'</td>
<td></td>
<td>iliteral</td>
<td>0</td>
</tr>
<tr>
<td>'false'</td>
<td></td>
<td>eliteral</td>
<td>@boolean</td>
</tr>
<tr>
<td>'False'</td>
<td></td>
<td>iliteral</td>
<td>1</td>
</tr>
</tbody>
</table>

   **Figure 6.2. Enumeration Literal Representation in the Semantic Space**

4. **sliteral**: this tag indicates string literals.
6.4. Types Representation

Types are represented in the semantic space with a separate entry for each type in the symbol table and a pointer to an entry in the attribute space. The attribute entry consists of the tag `type` and the value field that holds the number of parameters (zero if the type is non-parameterized). The entry that follows holds the size of this type. The tag of this entry is `size` and the value field either holds a constant value to indicate a constant-valued size or an index to the size function. As shown in table 6.1, each type (built-in or user-defined) has a unique place in the attribute space, whose tag is `type` and whose value is the number of formal type parameters, where all types are stored in the types symbol table.

6.4.1. Parameterized Types

A parameterized type is itself a type and occupies a unique place in a symbol table when processed. To process a parameterized type, we need to record its size, its parameters, its fields, and its member functions. The storage of the parameterized types in the symbol table requires a unique type signature for each new declaration of a parameterized type. This type signature consists of the type identifier and the signature of the formal parameters, which can be either integers or type variables. The signature of an integer parameter is "I", and the signature of a type variable is "T". For example, the following stack types can coexist because their signatures are different. The programmer can implement each stack type differently.

<table>
<thead>
<tr>
<th>Declared Type:</th>
<th>Type Signature:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type stack</code></td>
<td><code>stack</code></td>
</tr>
<tr>
<td><code>type stack{t: type}</code></td>
<td><code>stackT</code></td>
</tr>
<tr>
<td><code>type stack{n: integer, t: type}</code></td>
<td><code>stackIT</code></td>
</tr>
<tr>
<td><code>type stack{t: type, n: integer}</code></td>
<td><code>stackTI</code></td>
</tr>
</tbody>
</table>
Accordingly, type signatures are entered in the symbol table rather than type names because they are unique and encode the signature of the formal parameters (using the I and T letters to denote an integer formal type parameter and a type variable respectively). SIMPL identifiers are not case sensitive. They are converted to lowercase when entered in the symbol table [Mudawwar98a]. Thus, they are not confused with the uppercase I and T representing the formal type parameters. For example, the type arrayIT has two parameters; the next entries in the attribute space capture the formal parameters. For example, the first parameter of arrayIT is an intpar (i.e., an integer parameter), and the second formal parameter is a typevar (i.e., a type variable). Each formal type parameter stores the index of its constraint if it is referenced by another entries; otherwise it holds the value –1 (indication that it is unconstrained). The address (or index) of a type in an attribute space uniquely identifies the type and serves the purpose of specifying the types of objects, fields, parameters, and function results. Figure 6.3. shows a symbol table with entries for the integer type, the arrayIT type, and the stackIT type of the following type declaration [Mudawwar98a].

```plaintext
type stack{n: integer, t: type} is
    top: integer := 0
    storage: array{n, t}
    function push(x: t)
    function pop(): t
    function items(): integer
end -- interface of type stack
```

6.4.2. Fields of a Class Type

Field names are postfixed when entered in a symbol table. The postfix @stackIT represents the address (or index) of the type stackIT in the attribute space. This encoding will make a field name unique and will simplify its search. A field entry in the attribute space holds the address of its type. It is followed by an offset entry that holds the offset of the filed within the class type. If the type of a field entry is a parameterized type instance, new entries in the attribute space are allocated for this type instance. An entry with the tag typeinst holds the address of a parameterized type. The entries that follow specify the actual parameters of the type instance. The
actual parameters to the \texttt{array\{n, t\}} type of the field \texttt{storage} are references to the formal parameters of the type \texttt{stackIT} as indicated by the \texttt{ref} entries in the attribute space.

<table>
<thead>
<tr>
<th>Name</th>
<th>Attribute</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td></td>
<td>type</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size</td>
<td>4</td>
</tr>
<tr>
<td>arrayIT</td>
<td></td>
<td>type</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size</td>
<td>-123</td>
</tr>
<tr>
<td></td>
<td></td>
<td>intpar</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>typevar</td>
<td>-1</td>
</tr>
<tr>
<td>stackIT</td>
<td></td>
<td>type</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>size</td>
<td>-124</td>
</tr>
<tr>
<td></td>
<td></td>
<td>intval</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>typevar</td>
<td>-1</td>
</tr>
<tr>
<td>top@stackIT</td>
<td></td>
<td>field</td>
<td>@integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>offset</td>
<td>0</td>
</tr>
<tr>
<td>storage@stackIT</td>
<td></td>
<td>field</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>offset</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>typeinst</td>
<td>@arrayIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ref</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ref</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.3. Representation of the type \texttt{stackIT} and its Fields in the Semantic Space

6.4.3. Processing the Size of Parameterized Types

The size of a parameterized type and the offset (relative address) of a field need not be constant values, but are in general functions of the formal type parameters. Furthermore, even when a type is not parameterized and its size is a constant value, the size is not known, in general, when a type is imported because a type interface does not list the private members. Therefore, for each type, the compiler generates a function to calculate the non-constant size instead of recording the size in the attribute space. These functions can be expanded inline whenever possible to avoid the overhead of calling them. The name of a generated size function should be encoded in a special way to distinguish it from other functions. Thus, \texttt{SIZE\_123} and \texttt{SIZE\_124} are the names of the functions that return the size of the types \texttt{arrayIT} and \texttt{stackIT} respectively. The address of a type is encoded in the \texttt{SIZE} function name. This address is kept in the \texttt{size} entry next to the
type entry in the attribute space. In such a case, the stored value is stored as a negative integer value to differentiate it from types of constant size. The parameters of a size function are the formal parameters of the corresponding type. The size of a type can be related to the size of other types and can have only addition and multiplication operators when generated. Knowing the size of types, we can determine the size of objects, fields, and parameters, as well as the relative offsets of fields and local declarations. However, the size value is not known in general until runtime and cannot be assumed at compile time [Mudawwar98a].

```plaintext
function SIZE_123 (n, SIZEt: integer)
    if n <= 0 then result := 0
    else result := n * SIZEt end
end -- SIZE_123

function SIZE_124 (n, SIZEt: integer)
    result := value(@integer + 1) + SIZE_123(n, SIZEt)
end -- SIZE_124
```

6.4.4. Union Types

Each union type has its unique entry in the symbol table. The associated attribute holds a set of entries: the first entry is tagged with the uniontype tag and its value field holds the number of type parameters - if any - of this union type. The size of the type and the list of these type parameters are stored in the succeeding entries. Next comes the list of references to the type instances acceptable by this union type. This list is preceded by alttypes entry that keeps the number of types accepted by this union type. Figure 6.4. illustrates the semantic information of the declared union type

```
Type one {n: integer, t: type} is
    type two {3, t} or three {n, real}
```

An instance of a union type is a typeinst. By default, it is assigned the type of the first alternative type. Upon any assignment operation, the tag of this instance is automatically set to the new type and accordingly, its type parameters – if any – are assigned values. For instance, \( x \) is a declared union instance and \( y \) is an instance of type three. When \( y \) is assigned to \( x \), the tag of \( x \) – in memory
space- is set to the type three (the second type in the list assigned to the union type) and its actual type parameters are unified with their correspondence in y as represented in figure 6.4.

```plaintext
x: one{n, t}
y: three{5, real}
x := y
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Attribute</th>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>oneIT</td>
<td>uniontype</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>size</td>
<td>-125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>intval</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>typevar</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>alttypes</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ref</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ref</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeinst</td>
<td>@twoIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iliteral</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ref</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeinst</td>
<td>@threeIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ref</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeinst</td>
<td>@real</td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeinst</td>
<td>@threeIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iliteral</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeinst</td>
<td>@real</td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeinst</td>
<td>@oneIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>iliteral</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>typeinst</td>
<td>@real</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4. Semantic Representation of the Union Types

6.4.5. Synonym Types

A synonymous type has its unique entry in the symbol table. It is represented by the syntype entry in the attribute space. The value stored in this entry is the number of type parameters of this type. The entries that follow are its type parameters - if it is parameterized. This is followed by entries for its synonymous type instance. Figure 6.5. illustrates the semantic representation of the synonym type declaration

```plaintext
type matrix{n: integer, t: type} is type array{n, array{n, t}}
```
Functions constitute great power in SIMPL. In order for the compiler to check for function overloading and specialization, there has to be an efficient way of representing function declarations in the semantic space. Each function reserves an entry in the symbol table and the associated attribute entries. The name field in the symbol table holds the function name concatenated to the number of formal parameters this function has. Functions that share the same name (as stored in the name field) are linked together through the next field. The function entry holds the name of its label in the code area. This information is needed for the translation phase. The following entry is tagged with hidden and stores the number of hidden parameters in this function. If the function has hidden parameters, they are listed in the succeeding entries. The formal parameters are collectively called a tuple. A tuple is an ordered list of elements of possibly different types. A tuple entry in an attribute space specifies the number of elements. The formal parameters are listed in the form of an entry that is tagged with the tuple tag and holds the number of formal parameters that follow. The result entry in an attribute space specifies the result type of a function.
If a function does not have a result type, a null pointer is used as the value of the result entry. The following function implementation is parsed into the symbol table as shown in figure 6.6.

```plaintext
function swap {n: integer, t: type} (i: integer,
    obj m: array{n, array{n, t}}, j: integer) is
    obj row: array{n, t} -- swap row i with row j

    row := m[i]
    m[i] := m[j]
    m[j] := row
end -- function swap
```

Upon a function call to this function, the unification algorithm binds the unbound variables to their correspondence in the actual parameters. Accordingly, the binding entries, in the constraint area, hold the new values or point to the variables they are bound to. The unification algorithm is detailed in appendix B.

![Diagram of Functions Symbol Table and Attribute Space]

Figure 6.6. Semantic Representation of Functions
6.5.1. Polymorphic Functions

Parameterized polymorphic functions in SIMPL are not function generators such as the template function of C++. Each function (whether monomorphic or polymorphic) has a unique translation and a unique place in the functions symbol table and attribute space. Consider the above mentioned swap function. As the above figure shows, the internal representation of the swap function header goes as follows. The function entry in the attribute space specifies the number of hidden parameters to be two hidden parameters. The first hidden parameter is an intpar (integer parameter) while the second one is a typevar (type variable). The formal function parameters follow. In this example there are three formal parameters. The first and third formal parameters are val (value parameters) while the second parameter is an obj (object parameter). Each parameter entry specifies its type. If the type of a formal parameter is an instance of a parameterized type then a typeinst entry is allocated in the attribute space. The typeinst entry points to a parameterized type. The entries that follow the typeinst entry specify the actual type parameters. At the end comes the result entry that holds 'null' since the function does not return any result.

The symbol table of the swap function is also shown in Figure 6.6. Identifiers local to a function (hidden parameters, formal parameters, local object and constant declarations) are placed in the function's symbol table. In the above figure, the hidden parameters n and t, the formal parameters m, i, and j, and the local object row of the function swap are placed in this function symbol table. A function symbol table and its allocated entries in the attribute space are recycled when the compiler finishes the translation of a function [Mudawwar98a].

6.5.2. Constraint Representation

The tag binding is vital for constraint representation. Constraints indicate that this value is referenced by other entries. Accordingly, the constraint entry enforces integrity among all references to this value. These entries are tagged binding and are associated with hidden parameters of a function that are referenced in some formal/actual function parameters. A constrained variable
can be tagged \texttt{intpar} or \texttt{typevar}. Each of these tags holds the index of the constraint in the constraint array. If the variable is not constrained, it holds the value $-1$. This feature enables the detection of hidden parameters that hadn't been referenced by a formal parameter. Moreover, this representation of constraints is significant in unifying function headers against function calls. Initially, the hidden parameter holds $-1$ indicating that it is not referenced by any formal parameters. Once this variable is referenced by at least one formal parameter, its value field holds its index in the bindings area. Upon unification of this variable with another entry, the \texttt{binding} entry points to this entry. Figure 6.7. illustrates the representation of the constraints of the following function in the attribute space.

\begin{verbatim}
function f {n: integer, t: type} (val m: integer, 
  obj a: array{n, array{n, t}): array{n ,t}
\end{verbatim}

\hspace{1cm}

![Figure 6.7. Representation of Constraints in the Attribute Space](image-url)
6.5.3. Function Types

SIMPL allows the declaration of function types to facilitate higher-order functions. Function types are represented in the semantic space. Each function type has a unique entry in the symbol table. The associated attribute entry has the tag `ftype`. If this function type is parameterized, the entry holds the number of hidden parameters; otherwise, the value zero is stored. The succeeding entries embody the hidden parameters’ information. Next come the formal function parameters and the result type in the same structure mentioned in the previous section. Figure 6.8 shows the semantic representation of the function type

```
Type functype {n: integer, t: type} is
  function (obj array{n, t}, integer): array{n, t}
```

![Figure 6.8. Semantic Representation of Function Types](image)

6.5.4. Synonym Functions

Since synonym functions are more or less an interface to call another function. Such information can be represented as a `synfun` tagged entry in the semantic space. This entry keeps the number of hidden parameters - if any - and entries for the hidden parameters. This is followed by the tuple type of the formal parameters and the result type. The succeeding entry is a `call` entry
relating it to its synonymous function. The tuple type of the actual parameters to the synonymous function supplements this. Figure 6.9. portrays the semantic representation of the synonym function declaration

\[
\text{function } + (x, y: \text{complex}): \text{complex is}
\]

\[
\text{function complex}(x.\text{re} + y.\text{re}, x.\text{im} + y.\text{im})
\]

![Figure 6.9. Semantic Representation of Synonym Functions](image)

### 6.5.5. Static Type-Checking and Inferring Hidden Parameters

The body of a polymorphic function consists of object declarations, statements, and expressions. When translated, local object declarations allocate storage on the run-time stack. Non-control statements and expressions are calls to their respective functions and operators. For
example, the translation of the following assignment statement involves two calls to the index
operator and one call to the assignment operator.

\[
\text{m}[j] := \text{m}[i] \\
\text{Temp1} \leftarrow \text{INDEX(m, i)} \\
\text{Temp2} \leftarrow \text{INDEX(m, j)} \\
\text{ASSIGN(Temp2, Temp1)}
\]

The static type-check of a polymorphic function body means that all the calls generated by the
statements and expressions are valid calls. A function call is valid if we can unify the types of the
actual parameters against the types of the formal parameters. Therefore, the translation of a
polymorphic function boils down to calling a polymorphic function and inferring the hidden
parameters.

The unification algorithm used to check the validity of a function call and inferring the hidden
parameters is listed in appendix B. The **unify** function has three parameters: **formal** and
**actual** which represent the formal and actual parameter types of a function and **bind** which
represents the list of constraints on hidden parameters. The type **attref** (**attribute reference**)
represents the address (or index) of an attribute entry in an attribute space. Thus, **formal** and
**actual** are the addresses of attribute entries.

**function** unify (formal, actual, bind: attref):boolean

The **unify** function returns a boolean value. If it returns 'true' then successful unification was
achieved. Otherwise, unification is unsuccessful and the call cannot be made. As a side effect, the
**unify** function binds the hidden parameters with references to attribute entries that define the
actual values of hidden parameters for a given function call [Mudawwar98a].

The following example illustrates the operation of the unification algorithm. The headers of
functions **f** and **g** are first processed and entered in the attribute space as shown in Figure 6.10.
The local identifiers of function **g** are entered in **g**'s symbol table. The attributes of local objects **b**
and **c** are entered in the attribute space. To apply function **f** on the tuple \((a, b)\), we first enter
this tuple in the attribute space. The attribute values of the binding entries corresponding to the

hidden parameters of function $f$ are initialized to 'null' by the initialize function defined in appendix B.

```plaintext
function f{n: integer, t: type}(obj x: array{n, t}, y: t): t
function g{n: integer, t: type}(obj a: array{n, array{n, t}}) is
    obj b: array{n, t}
    obj c: t
    b := f(a, b) -- valid call
    c := f(a, c) -- invalid call
end -- function g
```

The `unify` function is called by passing the address of the formal tuple of function $f$ to `formal` and the address of the actual tuple to `actual` as shown in Figure 6.10. It starts by unifying the type of the first actual parameter against the type of the first formal parameter. As a result, the attribute value of the binding entry associated with the first hidden parameter for $f$ becomes the address of the first hidden parameter of $g$. On the other hand, the attribute value of the binding entry associated with the second hidden parameter of $f$ becomes the address of the `typeinst` entry `array{n, t}` as illustrated in the figure below.

The `unify` function then unifies the type of the second actual parameter against the type of the second formal parameter and validates that both are `typeinst array{n, t}`. This is required because of the constraint set in function $f$ that the type of $y$ and the element type of the array $x$ should be identical. Therefore, the `unify` function returns 'true' for the first call. However, it returns 'false' for the second call because it is unable to unify the type of the actual parameter $c$ - `typevar t` - against the type of the formal parameter $y$ - `typeinst array{n, t}`.

After a successful unification, we obtain the attribute values of the hidden parameters of the called function. These values are important for translation. We also get the result type for further use. For example, we need the result type of $f(a, b)$ (which is `typeinst array{n, t}`) to validate the call to the assignment operator – where a function call to the unify function with the result type of $f(a, b)$ and the type of $b$ takes place - as well as for translation [Mudawwar98a].
Figure 6.10. Side Effects of the Unify Function on the Semantic Space
6.5.6. Translating Polymorphic Functions with Hidden Parameters

To have a unique translation for a polymorphic function with hidden parameters, we must have a unique translation for the basic polymorphic operators. The SIMPL compiler has a unique translation for the basic polymorphic operators.

Let \( \text{SIZE}_t \) be the size - in bytes - of type \( t \). If \( \text{SIZE}_t \) is passed as a hidden parameter, we can achieve a unique translation for each of the basic polymorphic operators and functions. The assignment operator requires \( \text{SIZE}_t \) to determine the number of bytes to be copied. The equality operators (= and <>) need \( \text{SIZE}_t \) to decide on the number of bytes to be compared. The new function needs \( \text{SIZE}_t \) to settle on the number of bytes to be allocated. The index operator \( [ \ ] \) needs \( \text{SIZE}_t \) to calculate the address of the indexed element. Observe that the actual type of \( t \) is not important in this discussion. \( \text{SIZE}_t \) is all that is required. In general, a polymorphic function with a type variable as a hidden parameter needs to know the size of the type variable for each function call. This size, which is equal to the size of the actual parameter, can be obtained for each function call after inferring the hidden parameters [Mudawar98a].

We can have a closer look at the translation of the polymorphic assignment and the array index operators. The translated functions ASSIGN and INDEX are listed below. The whole idea is converting the hidden parameters into formal parameters of the translated functions. \( \text{SIZE}_t \) is passed as an integer parameter. The type address means a virtual address in the address space of a process. The parameters of a translated function are normally passed through registers. Integer and address parameters use integer registers. User-defined types vanish when a function is translated.

```plaintext
function ASSIGN (SIZE: integer, object: address, 
    expression: address) is
    -- copy SIZE bytes from expression to object keeping account
    -- for possible overlaps between object and expression
    end -- assign

function INDEX (n: integer, SIZE: integer, a: address, 
    i: integer): address is
```

96
\begin{verbatim}
if i >= 0 and i < n then
  result := a + i * SIZEt
else
  Index is Out of Range
end -- if
end -- index

Let us translate the \texttt{swap} function of section 6.5. This function has 2 hidden parameters \texttt{n} and \texttt{t} and 3 formal parameters \texttt{m}, \texttt{i}, and \texttt{j}. These 5 parameters become formal parameters in the translated function. The body of the \texttt{swap} function had one local object \texttt{row}. The size of \texttt{row} is calculated by calling the \texttt{SIZE_123} function with actual parameters \texttt{n} and \texttt{SIZEt}. This is because \texttt{row} is of type \texttt{array\{n, t\}}. We then allocate space for \texttt{row} on the stack by modifying the stack pointer. This modification to the stack pointer is done after the function \texttt{swap} is called (i.e. after allocating a frame for the temporaries, saved registers, return address, and other known fixed size local objects). The indexing and assignment operations are then translated into calls to the polymorphic \texttt{INDEX} and \texttt{ASSIGN} functions. These functions require hidden parameters, which are inferred by the unification algorithm. Observe that actual integer parameters are passed for hidden parameters (which are no longer hidden) in the assignment and the indexing operator calls \cite{Mudawwar98a}.

\textbf{function} swap(n: integer, SIZEt: integer, m: address, 
  i: integer, j: integer) \textbf{is}
  row : address
  Temp1: integer
  Temp2: address
  Temp3: address
  Temp1   <- SIZE_123(n, SIZEt)    -- size of array\{n, t\}
  row     <- PUSH(Temp1)          -- modify the stack pointer
  Temp2   <- INDEX(n, Temp1, m, i) -- compute address of \texttt{m[i]}
  ASSIGN(Temp1, row, Temp2)       -- \texttt{row := m[i]; copy Temp1}
                                 -- bytes
  Temp3   <- INDEX(n, Temp1, m, j) -- compute address of \texttt{m[j]}
  ASSIGN(Temp1, Temp2, Temp3)     -- \texttt{m[i] := m[j]; copy Temp1}
                                 -- bytes
  ASSIGN(Temp1, Temp3, row)       -- \texttt{m[j] := row; copy Temp1}
                                 -- bytes
  POP(Temp1)                      -- restore the previous
                                 -- stack pointer
\end{verbatim}
<table>
<thead>
<tr>
<th>Tag</th>
<th>Meaning of the tag field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alttypes</td>
<td>The alternatives types acceptable by a union type</td>
<td>The number of the alternative types</td>
</tr>
<tr>
<td>Binding</td>
<td>binds the constrained variable to a value</td>
<td>If unbound, it holds 'null' for type variables and 0 for integer variables. If bound, it holds the value (an integer literal or a pointer to an another variable in the attribute space )</td>
</tr>
<tr>
<td>Call</td>
<td>A function call through a synonymous function</td>
<td>Pointer to the function entry</td>
</tr>
<tr>
<td>Eliteral</td>
<td>An enumerated type literal</td>
<td>Pointer to its enumeration type. The following entry holds the integer value of this literal</td>
</tr>
<tr>
<td>Field</td>
<td>A field of a declared type</td>
<td>Pointer to the type or the subtype of this field</td>
</tr>
<tr>
<td>Fliteral</td>
<td>A floating-point literal</td>
<td>A pointer to the value of the literal</td>
</tr>
<tr>
<td>Function</td>
<td>The function header</td>
<td>Label to the function in code space. The following entry holds the number of hidden parameters. The succeeding entries to this entry are the hidden parameters, formal parameters and the result</td>
</tr>
<tr>
<td>Ftype</td>
<td>A declared function type</td>
<td>The number of hidden parameters. The succeeding entries to this entry are the hidden parameters, formal parameters and the result</td>
</tr>
<tr>
<td>Hidden</td>
<td>The number of function hidden parameters</td>
<td>A constant value represents the number of function hidden parameters</td>
</tr>
<tr>
<td>Iliteral</td>
<td>An integer literal</td>
<td>The integer value of this literal</td>
</tr>
<tr>
<td>Intpar</td>
<td>An integer parameter (its mode is val)</td>
<td>Its index in the set of bindings if it is bound to a value or another variable otherwise it contains the value -1</td>
</tr>
<tr>
<td>Obj</td>
<td>An object variable declaration</td>
<td>Pointer to its type</td>
</tr>
<tr>
<td>Offset</td>
<td>The offset of a field within the class type</td>
<td>An integer represents the relative address of the field in the class type</td>
</tr>
<tr>
<td>Ref</td>
<td>Reference to another attribute entry</td>
<td>Pointer to an entry in the attribute space</td>
</tr>
<tr>
<td>Result</td>
<td>The result type of a function</td>
<td>Pointer to the type of the result</td>
</tr>
<tr>
<td>Size</td>
<td>The size of the declared type</td>
<td>Label to the size function in code space or a constant value</td>
</tr>
<tr>
<td>Sliteral</td>
<td>String literal</td>
<td>Pointer to the string space where the string is stored</td>
</tr>
<tr>
<td>Synfun</td>
<td>A synonym function</td>
<td>Number of hidden parameters (if any). It is followed by entries for the hidden parameters, the tuple type of the formal parameters, the result type and a reference entry to its synonymous function followed by the tuple type of the actual parameters</td>
</tr>
</tbody>
</table>
Table 6.1. Tags Used in the Attribute Space and their Usage (cont.)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntype</td>
<td>A synonym type</td>
<td>The number of type parameters of this type. The entries that follow this entry are its type parameters - if it is parameterized - and entries for its synonymous type instance.</td>
</tr>
<tr>
<td>Type</td>
<td>A class type (built-in/user-defined)</td>
<td>The number of type parameters of this type. The entries that follow this entry are its size and its type parameters - if it is parameterized.</td>
</tr>
<tr>
<td>Typeinst</td>
<td>A type instance</td>
<td>Pointer to the base type of this instance. The entries that follow it, are the actual parameters of the type instance - if it is parameterized.</td>
</tr>
<tr>
<td>Tuple</td>
<td>The tuple of the formal parameters or the actual parameters</td>
<td>The number of parameters. The attributes of the formal/actual parameters follow in the subsequent entries.</td>
</tr>
<tr>
<td>Typevar</td>
<td>A type variable parameter</td>
<td>Its index in the set of bindings if it is bound to another type variable or to a type, otherwise it contains the value $-1$.</td>
</tr>
<tr>
<td>Uniontype</td>
<td>A union type</td>
<td>The number of formal type parameters it has (if it is parameterized) followed by a size entry stores its size. This is followed by entries for the type parameters (if any). An entry with the number of alternative types it holds. The next entries represent references to the type instances it accepts.</td>
</tr>
<tr>
<td>Val</td>
<td>A constant item</td>
<td>Pointer to its type</td>
</tr>
</tbody>
</table>
7. FUNCTION OVERLOADING AND SPECIALIZATION

Overloading of polymorphic functions adds a second dimension to a polymorphic language; the power of SIMPL lies mainly in function overloading and specialization. With overloading, one can overload the assignment operator to allow the assignment of a real to a complex, or a value of type t to all the elements of an array of type array{n, t} as shown below. The type system does not allow the overloading of a function (or type) if the new function (or type) has the same signature.

\[
\text{function} := (\text{obj } x: \text{ complex}, y: \text{ real}) \\
\text{function} := \{n: \text{ integer}, t: \text{ type}\}(\text{obj } x: \text{ array}\{n, t\}, y: t)
\]

A special case of function overloading is called function specialization. An overloading function is said to be a specialization of the overloaded function, if the tuple type of the overloading function parameters is a subtype of that of the overloaded function parameters [Mudawwar98a]. For example, one may wish to overload the assignment operator for list{t} parameters to handle the appropriate copying of lists. This overloading is called specialization because the tuple type of the specialized function parameters (list{t}, list{t}) is a subtype of the tuple type of the overloaded function parameters (t, t). Similarly, the third function is a specialization of both the second and first functions.

\[
\text{function} := \{t: \text{ type}\}(\text{obj } x: t, y: t) \\
\text{function} := \{t: \text{ type}\}(\text{obj } x: \text{ list}\{t\}, y: \text{ list}\{t\}) \\
\text{function} := (\text{obj } x: \text{ list}\{\text{real}\}, y: \text{ list}\{\text{real}\})
\]

Overloading with specialization is more powerful in a programming language but much more difficult to implement than overloading without specialization. There are essentially two problems to solve:

1. We need to identify and implement the rules for unambiguous overloading.
2. We need to bind function calls to function addresses.
7.1. Function Overloading

*Function overloading* is the declaration of several function interfaces that share the same function name. For instance, the function \( \forall \{t: \text{TYPE}\} \ f (x: t): t \) can overload the function \( \forall \{t: \text{TYPE}\} \ f (x: t, y: t): \text{void} \). This can be expressed formally by the following type rule.

\[
\frac{\text{E} \vdash \text{sig1}: \text{Sig1} \quad \text{E} \vdash \text{f(sig1):Rtype1} : \text{Sig1} \rightarrow \text{Rtype1} \quad \text{E} \vdash \text{sig2: Sig2}}{\text{E}, \text{f(sig1):Rtype1} : \text{Sig1} \rightarrow \text{Rtype1} \vdash \text{f(sig2):Rtype2} : \text{Sig2} \rightarrow \text{Rtype2}}
\]

Programming languages allow overloading as long as it does not lead to ambiguity in function calls. Accordingly, there must be a clear-cut definition of function overloading and a definitive procedure to guarantee its prevention. First of all, let us define ambiguous overloading.

**Ambiguous Overloading** is the case when a function call matches several declared overloaded functions. This leads to ambiguity in which function to call. The compiler has to settle on the rules of overloading that prevent ambiguity upon function calls.

The following type rule illustrates ambiguous overloading.

\[
\frac{\text{E} \vdash \text{f(s1):Rtype1} : \text{Sig1} \rightarrow \text{Rtype1} \quad \text{E} \vdash \text{f(s2):Rtype1} : \text{Sig2} \rightarrow \text{Rtype1}}{\text{E}, \text{f(s2}\leftarrow\text{s4:Sig2}) \vdash \text{f(s1}\leftarrow\text{s4:Sig1)}}
\]

An example of ambiguous overloading is given below. If function \( f \) is called with an actual parameter of type \( \text{array\{100, real\}} \) then both functions can apply. This is because the types \( \text{array\{n, real\}} \) and \( \text{array\{100, t\}} \) are not related but have the common subtype \( \text{array\{100, real\}} \). This ambiguous overloading can be eliminated by either dropping one of the two functions, or by introducing a third function declaration \( f(\text{obj x: array\{100, real\}}) \) with parameter type \( \text{array\{100,real\}} \).

```plaintext
function f \{n: integer\}(\text{obj x: array\{n, real\}})
function f \{t: type\} \quad (\text{obj x: array\{100, t\}})
```
7.2. Function Specialization

An overloading function is said to be a specialization of an overloaded function, if the tuple type of the overloading function parameters is a subtype of the tuple type of the overloaded function parameters. When the overloaded functions share the same function name, same number of arguments, and the type of each argument is a subtype of the corresponding argument’s type in the other signature, it is called specialization. Moreover, the result types experience a subtyping relation. Actually, specialization is a special case of overloading. For instance, the function \( f(x: \text{integer}): \text{integer} \) is a specialization of the function \( \forall \{t: \text{TYPE}\} f(x: t): t \). It can be formalized as follows:

\[
\begin{align*}
E \vdash \text{ftype1} \leq \text{ftype2} & \quad E \vdash f(\text{Sig1}):\text{Rtype1} : \text{ftype1} & \quad E \vdash f(\text{Sig2}):\text{Rtype2} : \text{ftype2} \\
\hline
E \vdash f(\text{Sig1}):\text{Rtype1} \leq f(\text{Sig2}):\text{Rtype2}
\end{align*}
\]

7.3. Non-Ambiguous Overloading Rule

In order to achieve non-ambiguous overloading, there are some checks to be performed.

The overloading rule for ambiguity prevention is:

A necessary and sufficient condition: to achieve non-ambiguous overloading, the set of overloaded functions must satisfy all of these conditions:

1. Each function must have a unique signature type.

2. If the signature types are subtypes of one another, the function types must be subtypes of one another in the same order. That is, the subtype relation for function types is covariant (in the same direction) with the arguments’ subtyping.

3. No function call with a signature type \( \text{Sig3} \) where there is no overloaded function with this signature type, and \( \text{Sig3} \) is a subtype of more than one
declared signature type – i.e. these signatures exist in some declared overloaded functions.

Let’s explain these conditions in details.

7.3.1. Unique Signature Type

The first condition: Each function must have a unique signature type i.e. if more than one function share the same name, their signatures must differ either in the number of parameters or the aggregate type of parameters. The difference between signature types does not count on the difference in mode of access or the difference in the order of listing the hidden parameters.

\[
(\forall f \exists !\text{Sig}) \\
E \vdash f_1 : \text{Sig}_1 \rightarrow \text{Rtype}_1 \ldots E \vdash f_m : \text{Sig}_m \rightarrow \text{Rtype}_m \quad E \vdash \text{name}(f_1) = \ldots = \text{name}(f_m) \\
\hline \\
E \vdash \text{Sig}_1 \neq \text{Sig}_2 \neq \ldots \neq \text{Sig}_m
\]

For example, these function declarations are allowed.

```plaintext
function f {t: type} (x: t): t
function f {t: type} (x, y: t)
function f (x: integer): integer
function f {n: integer, t: type}(x: array{n, t}): array{n, t}
```

They have unique signatures and accordingly they satisfy the necessary condition for overloading.

On the contrary, this set of function declarations violates the mentioned rule.

```plaintext
(1) function f {n: integer, t: type} (obj a: array{n, t})
(2) function f {n: integer, t: type} (a: array{n, t})
(3) function f {t: type , n: integer} (obj a: array{n, t})
(4) function f {n: integer, t: type} (obj a: array{n, t}): t
```

They lead to ambiguity because:

- Functions (1) and (2) differ only in the mode of the formal parameter. Both functions are appropriate to the function call \( f(a1) \) where \( a1 \) is declared as \( \text{obj a1: array\{n,t\} } \). It
is due to the fact that a value formal parameter accepts object actual parameters. Accordingly, the mode of access cannot be a discriminator between functions.

- Functions (1) and (3) differ only in the order of listing the hidden parameters. A function call does not state the hidden parameters; they are inferred from the actual parameters. Accordingly, upon a function call to any of the two functions, the compiler will not be able to pick the more eligible function. That causes ambiguity.

- Functions (1) and (4) differ in their result type. Thus, the context of the function call will be the factor upon which one of the two functions is selected. SIMPL has a context free grammar rather than a context-sensitive grammar.

The algorithm that checks for unique signature is expressed in the function `unique_signature` listed in appendix B. The function checks if the function under check has a unique signature compared to the overloaded functions stored in the functions symbol table. The function header is:

```latex
function unique_signature(f: ref{fsymbol},
      table: symtable{fsymbol}): boolean
      -- checks if the function f has a unique signature
      -- in the symbol table
```

### 7.3.2. Function Subtyping

*The second condition*: If the signature type $\text{Sig1}$ of a function $f$ of function type $\text{ftype1}$ is a subtype of the signature type $\text{Sig2}$ of another overloaded function $f$ of function type $\text{ftype2}$, then $\text{ftype1}$ must be a subtype of $\text{ftype2}$, to achieve unambiguous overloading.

```
(Sub function)
E | Sig1 <: Sig2  E | ftype2 = Sig2 -> Rtype2  E | ftype1 = Sig1 -> Rtype1
  E, f (sig1): Rtype1 : ftype1 | f (sig2): Rtype2 : ftype2
-----------------------------------------------------------------
E | ftype1 <: ftype2
```

For instance, since the signature type $(\text{integer}, \text{real})$ is a subtype of the signature type $(\forall t: \text{TYPE}) (t, \text{real})$ then the first function type (e.g. $(\text{integer}, \text{real}) \rightarrow \text{integer}$) must be a function subtype of the second function type (e.g. $(\forall t: \text{TYPE}) (t, \text{real}) \rightarrow t$). This is illustrated in figure 7.1.
Since the following function overloading does not satisfy the condition, it is forbidden.

```latex
function f \{t: \text{type}\} (x: t, y: \text{integer}): t
```

```latex
function f (x: \text{real}, y: \text{integer}): \text{integer}
```

---

Figure 7.1. Covariance Relation between Function Subtyping and Signature Subtyping

The algorithm that checks if the overloaded functions satisfy this condition is expressed in the `subtype` function listed in the subtyping check section of appendix B. This function receives the

---

Figure 7.2. Constraint Integrity among Function Subtypes
attribute entries of the two functions under test, and returns a true/false result in addition to the set of bindings inferred from the check.

```cpp
function subtype(t1, t2, c1, c2: attref): boolean
-- checks if t1 is a subtype of t2 and stores inferred values
-- for both types in the corresponding constraint areas c1 and c2
```

### 7.3.3. Common Function Subtype

The third condition rejects the existence of a function call with a signature type Sig3 such that Sig3 is a subtype of at least two signature types – Sig1 and Sig2 – of some existing overloaded functions, whereas there is no overloaded function with the signature type Sig3. Otherwise, ambiguity exists due to the existence of several function signatures that match the function call.

This ambiguous overloading can be formally stated as:

```
(Sub gcs)
E ⊢ f(s1): Rtype1 : Sig1 → Rtype1  E ⊢ f(s2): Rtype2 : Sig2 → Rtype2
E ⊢ Sig3 <: Sig1  E ⊢ Sig3 <: Sig2  E ⊢ f: Sig3 → Rtype ∉ dom(E)
E ⊢ s3: Sig3  E ⊢ Sig1 ⊆ Sig2  E ⊢ Sig2 ⊆ Sig1
________________________________________________________
E, f(s1←s3: Sig1) ⊢ f(s2←s3: Sig2)
```

This phenomenon occurs when there is a function call to a non-existing function that is a common function subtype to other functions as shown in figure 7.3. The following functions satisfy the first two conditions of overloading

```cpp
function f {n: integer, t: type} (obj a: array{n, array{n, t}, b: one{4}}): array{n, t}
function f {n, m: integer} (obj a: array{n, array{n, real}, b: one{m}}): array{n, real}
```

but they share an undeclared common function subtype which is

```cpp
function f {n: integer} (obj a: array{n, array{n, real}, b: one{4}}): array{n, real}
```

Thus, a function call `f(a1, b1)` where `a1` is declared as `obj a1: array{n, array{n, real}}` and `b1` is declared as `b1: one{4}` unifies with both of the above declared overloaded functions, leading to ambiguity.
To prevent such ambiguity, the compiler checks that each overloading function does not share a common function subtype with another overloaded functions. This check is achieved by the \texttt{gcs} function listed in appendix B. This function checks for the greatest common function subtype (gcs) between two functions. It receives the attribute entries of both functions and returns the attribute entry of the constructed gcs – if any – or a null pointer.

\textbf{function} \texttt{gcs}\ (x, y, h\textsubscript{1}, h\textsubscript{2} : attref) : attref

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{figure7.3.png}
\caption{The Greatest Common Subtype between Two Non-related Function Signatures}
\end{figure}

\subsection*{7.4. Formal Proofs}

This section formally proves that the combination of the three conditions is a necessary and sufficient condition for unambiguous overloading.

\textbf{Given}

Let \(c1\) be the condition that each function must have a unique signature type.

\begin{align*}
E \vdash f\textsubscript{1} : \text{Sig}_{1} \rightarrow \text{Rtype}_{1} & \quad \ldots \quad E \vdash f\textsubscript{m} : \text{Sig}_{m} \rightarrow \text{Rtype}_{m} \\
E \vdash \text{name}(f\textsubscript{1}) = \ldots = \text{name}(f\textsubscript{m}) \quad & \\
C1 = \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ Quad
Let $c_2$ be the condition that the function subtyping is covariant with their signature types subtyping.

$$
\begin{align*}
E \models \text{Sig1} &\leq \text{Sig2} & E \models \text{ftype2} = \text{Sig2} \rightarrow \text{Rtype2} & E \models \text{ftype1} = \text{Sig1} \rightarrow \text{Rtype1} \\
E, f(\text{sig1}): \text{Rtype1} & : \text{ftype1} & E \models f(\text{sig2}): \text{Rtype2} : \text{ftype2} \\
C_2 = & \text{__________________________________________________________} \\
& E \models \text{ftype1} \leq \text{ftype2}
\end{align*}
$$

Let $c_3$ be the condition of rejecting any function call with a signature type $\text{Sig3}$ where there is no overloaded function with this signature type, and $\text{Sig3}$ is a subtype of more than one declared signature types – i.e. these signatures belong to declared overloaded functions.

$$
\begin{align*}
E \models \text{Sig3} &\leq \text{Sig1} & E \models \text{Sig3} &\leq \text{Sig2} & E \models \text{Sig1} &\leq \text{Sig2} \\
E \models \text{Sig2} &\leq \text{Sig1} & E \models \text{sig3}: \text{Sig3} & E \models f(\text{sig1}): \text{Rtype1} : \text{Sig1} \rightarrow \text{Rtype1} \\
E \models f(\text{sig2}): \text{Rtype2} : \text{Sig2} \rightarrow \text{Rtype2} & E \models f(\text{sig3}): \text{Rtype3} : \text{Sig3} \rightarrow \text{Rtype3} \notin \text{dom}(E) \\
c_3 = & \text{__________________________________________________________} \\
& E, f(\text{sig1}\leftarrow\text{sig3}: \text{Sig1}) \models f(\text{sig2}\leftarrow\text{sig3}: \text{Sig2})
\end{align*}
$$

Finally let $\text{AO}$ represent ambiguous overloading i.e. an ambiguous function call.

$$
\text{AO} \equiv E, f(\text{s2}\leftarrow\text{s4}: \text{Sig2}) \models f(\text{s1}\leftarrow\text{s4}: \text{Sig1})
$$

**Theorem:** $(c_1 \land c_2 \land c_3) \iff \neg \text{AO}$

**Proof:**

Since the rule $(c_1 \land c_2 \land c_3) \Rightarrow \neg \text{AO}$ means that

1. $(c_1 \land c_2 \land c_3)$ is a sufficient condition for $\neg \text{AO}$

2. $\neg \text{AO}$ is a necessary condition for $(c_1 \land c_2 \land c_3)$

3. and that $\neg(\neg \text{AO}) \Rightarrow \neg(c_1 \land c_2 \land c_3) \equiv \text{AO} \Rightarrow \neg(c_1 \land c_2 \land c_3) \equiv \text{AO} \Rightarrow \neg(c_1) \lor \neg(c_2) \lor \neg(c_3)$
In order to prove that \((c_1 \land c_2 \land c_3)\) is a necessary condition for \(\neg AO\), we need to prove that \(\neg AO \Rightarrow (c_1 \land c_2 \land c_3)\) \hspace{1cm} (1)

And to prove that \((c_1 \land c_2 \land c_3)\) is a sufficient condition for \(\neg AO\), we need to prove that \((c_1 \land c_2 \land c_3) \Rightarrow \neg AO\) \hspace{1cm} (2)

### 7.4.1. Proof of the Necessary Condition

**Theorem:** \(\neg AO \Rightarrow (c_1 \land c_2 \land c_3)\)

**Proof:** Since \(\neg AO \Rightarrow (c_1 \land c_2 \land c_3) \equiv (\neg AO \Rightarrow c_1) \land (\neg AO \Rightarrow c_2) \land (\neg AO \Rightarrow c_3)\). Then we need to prove \((\neg AO \Rightarrow c_1) \land (\neg AO \Rightarrow c_2) \land (\neg AO \Rightarrow c_3)\). That is, we need to prove that \(c_1\) is in itself a necessary condition for \(\neg AO\), the same applies to condition \(c_2\) and condition \(c_3\).

**a. Proof of \((\neg AO \Rightarrow c_1)\)**

Given

\[
E \vdash f_1: \text{Sig}_1 \rightarrow \text{Rtype}_1 \quad \ldots \quad E \vdash f_m: \text{Sig}_m \rightarrow \text{Rtype}_m \quad E \vdash \text{name}(f_1) = \ldots = \text{name}(f_m)
\]

\[
\text{C1} = \begin{align*}
E \vdash & \text{Sig}_1 \neq \text{Sig}_2 \neq \ldots \neq \text{Sig}_m
\end{align*}
\]

and there is a function call \(f(s_1: \text{Sig}_1)\) i.e. \(E \vdash f(s_1: \text{Sig}_1)\)

**Theorem:** \((\neg AO \Rightarrow c_1)\) i.e. \(f(s_1: \text{Sig}_1)\) calls \(f_1\) only.

**Proof:**

\((\neg AO \Rightarrow c_1) \equiv (\neg c_1 \Rightarrow AO)\)

\[
E \vdash f_1: \text{Sig}_1 \rightarrow \text{Rtype}_1 \quad \ldots \quad E \vdash f_m: \text{Sig}_m \rightarrow \text{Rtype}_m \quad E \vdash \text{name}(f_1) = \ldots = \text{name}(f_m)
\]

\[
\neg c_1 = \begin{align*}
E & \vdash \text{Sig}_1 = \text{Sig}_h \\
E & \vdash \text{Sig}_2 \neq \ldots \neq \text{Sig}_g \neq \text{Sig}_i \neq \ldots \neq \text{Sig}_m
\end{align*}
\]

\[
E \vdash f_1(\text{sig}_1): \text{Rtype}_1 \rightarrow \text{Rtype}_1 \quad E \vdash f(s_1: \text{Sig}_1)
\]

since

\[
E \vdash f(\text{sig}_1 \leftarrow s_1: \text{Sig}_1)
\]
generates \( E \vdash f(s_1 \leftarrow s_1: \text{Sig}_1) \)

\[
E \vdash f_{h}(\text{sig}_h): \text{Rtype}_h: \text{Sig}_h \rightarrow \text{Rtype}_h \quad E \vdash f(s_1: \text{Sig}_1) \quad E \vdash \text{Sig}_1 = \text{Sig}_h
\]

and \( E \vdash f(\text{sig}_h \leftarrow s_1: \text{Sig}_h) \)

generates \( E \vdash f(\text{sig}_h \leftarrow s_1: \text{Sig}_h) \)

This leads to \( E, f(\text{sig}_1 \leftarrow s_1: \text{Sig}_1) \vdash f(\text{sig}_h \leftarrow s_1: \text{Sig}_h) \) where

\[
E, f(\text{sig}_1 \leftarrow s_1: \text{Sig}_1) \vdash f(\text{sig}_h \leftarrow s_1: \text{Sig}_h) \equiv \text{AO} \quad \text{(Q.E.D.)}
\]

From the above proof we deduce that ambiguity takes place if more than one signature are equivalent.

**b. Proof of \((\neg \text{AO} \Rightarrow c2)\)**

**Given**

\[
E \vdash \text{Sig}_1 \ll \text{Sig}_2 \quad E \vdash \text{ftype}_2 = \text{Sig}_2 \rightarrow \text{Rtype}_2 \quad E \vdash \text{ftype}_1 = \text{Sig}_1 \rightarrow \text{Rtype}_1
\]

\[
E, f(\text{sig}_1): \text{Rtype}_1: \text{ftype}_1 \vdash f(\text{sig}_2): \text{Rtype}_2: \text{ftype}_2
\]

\( C2 \equiv \quad \text{________________________________________________________________} \)

\[
E \vdash \text{ftype}_1 \ll: \text{ftype}_2
\]

and there is a function call \( f(s_1: \text{Sig}_1) \) i.e. \( E \vdash f(s_1: \text{Sig}_1) \)

**Theorem:** \((\neg \text{AO} \Rightarrow c2)\) i.e. \( f(s_1: \text{Sig}_1) \) calls \( f(\text{sig}_1): \text{Rtype}_1: \text{ftype}_1 \) only.

**Proof:**

\((\neg \text{AO} \Rightarrow c2) \equiv (\neg c2 \Rightarrow \text{AO})\)

\[
E \vdash \text{Sig}_1 \ll \text{Sig}_2 \quad E \vdash \text{ftype}_2 = \text{Sig}_2 \rightarrow \text{Rtype}_2 \quad E \vdash \text{ftype}_1 = \text{Sig}_1 \rightarrow \text{Rtype}_1
\]

\[
E, f(\text{sig}_1): \text{Rtype}_1: \text{ftype}_1 \vdash f(\text{sig}_2): \text{Rtype}_2: \text{ftype}_2
\]

\( \sim c2 \equiv \quad \text{________________________________________________________________} \)

\[
E \vdash \text{ftype}_1 \ll: \text{ftype}_2
\]
since the function call \( f(s1) \) unifies with \( f(sig1):Rtype1 \) as the following rule proves

\[
E \vdash f(sig1):Rtype1: Sig1 \rightarrow Rtype1 \\
E \vdash f(s1: Sig1) \\
\hline
E \vdash f(sig1 \leftarrow s1: Sig1)
\]

Moreover, since the function call \( f(s1) \) unifies with \( f(sig2):Rtype2 \) as the following rule proves

\[
E \vdash f(sig2):Rtype2: Sig2 \rightarrow Rtype2 \\
E \vdash Sig1 <: Sig2 \\
E \vdash f(s1: Sig1) \\
\hline
E \vdash f(sig2 \leftarrow s1: Sig2)
\]

And since function subtyping requires three conditions to be satisfied:

1. Signature subtyping \( E \vdash Sig1 <: Sig2 \)
2. Result subtyping \( E \vdash Rtype1 <: Rtype2 \)
3. Hidden parameter constraint satisfaction

\[
E \vdash \{ \forall t: TYPE, n: integer \} \, Sig1{n, t} \rightarrow Rtype1{n, t} = \\
\{ \forall m: integer, t2: TYPE \} \, Sig2{m \leftarrow n, t2 \leftarrow t} \rightarrow Rtype2{m \leftarrow n, t2 \leftarrow t}
\]

Accordingly, \( E \vdash ftype1 \leftarrow ftype2 \) whereas \( E \vdash Sig1 <: Sig2 \) can be a result of either the result types do not experience subtyping as shown by the following rule

\[
E \vdash Rtype1 <: Rtype2 \\
E \vdash Sig1 <: Sig2 \\
\hline
E \vdash ftype1 \leftarrow ftype2
\]

Which means the compiler has to check the context of the function call to choose the appropriate function; SIMPL compiler applies context free grammar. Accordingly, this condition is not allowed in overloading. The other reason can be a consequence of not preserving the hidden parameter constraints as shown in the rule.
From the above proof we deduce that ambiguity takes place if function subtyping does not
follow their signature subtyping.

c. Proof of \((\neg AO \Rightarrow c3)\)

Given

\[
\begin{align*}
E \vdash \text{Sig3} & \leftarrow \text{Sig1} & E \vdash \text{Sig3} & \leftarrow \text{Sig2} & E \vdash \text{Sig1} & \leftarrow \text{Sig2} & E \vdash \text{sig3} : \text{Sig3} \\
E \vdash \text{Sig2} & \leftarrow \text{Sig1} & E \vdash \text{f(sig1)} : \text{Rtype1} \rightarrow \text{Rtype1} \\
E \vdash \text{f(sig2)} : \text{Rtype2} \rightarrow \text{Rtype2} & & E \vdash \text{f(sig3)} : \text{Rtype3} \rightarrow \text{Rtype3} & \notin \text{dom}(E)
\end{align*}
\]

\(c3 = \sim\)

Theorem: \((\neg AO \Rightarrow c3)\) i.e. \(f(sig3 : \text{Sig3})\) calls \(f(sig3) : \text{Rtype3} \rightarrow \text{Rtype3}\) only, where this
function must have been already declared in the program.
Proof:

\((\sim AO \Rightarrow c3) \equiv (\sim c3 \Rightarrow AO)\)

\[
\begin{align*}
E \vdash \text{Sig3} <: \text{Sig1} & \quad E \vdash \text{Sig2} <: \text{Sig1} & \quad E \vdash \text{Sig1} \triangleleft \text{Sig2} & \quad E \vdash \text{sig3} : \text{Sig3} \\
E \vdash \text{Sig2} \triangleleft \text{Sig1} & \quad E \vdash \text{f(sig1)}: \text{Rtype1} : \text{Sig1} \Rightarrow \text{Rtype1} & \quad E \vdash \text{f(sig3)}: \text{Rtype3} : \text{Sig3} \Rightarrow \text{Rtype3} \\& \not\in \text{dom}(E)
\end{align*}
\]

\(~ c3 \equiv \)

\[
E, f(\text{sig1} \leftarrow \text{sig3}: \text{Sig1}) \vdash f(\text{sig2} \leftarrow \text{sig3}: \text{Sig2})
\]

This condition states that a function call with the signature \(\text{sig3}\) can unify with \(f(\text{sig1}):\text{Rtype1}\) and can unify with \(f(\text{sig2}):\text{Rtype2}\) due to the signature subtype relation between these signatures.

Accordingly, ambiguity on function call is encountered.

\[
E, f(\text{sig1} \leftarrow \text{sig3}: \text{Sig1}) \vdash f(\text{sig2} \leftarrow \text{sig3}: \text{Sig2}) \equiv AO \quad (Q.E.D.)
\]

From the above proof we deduce that ambiguity takes place if there is a function call to an undeclared function yet the signature of the function call is a subtype of several declared functions’ signatures.

### 7.4.2. Proof of the Sufficient Condition

**Theorem:** \((c1 \land c2 \land c3) \Rightarrow \sim AO\)

**Proof:** Since \((c1 \land c2 \land c3) \Rightarrow \sim AO \equiv AO \Rightarrow \sim(c1 \land c2 \land c3)\)

\[
\equiv AO \Rightarrow \sim(c1) \lor \sim(c2) \lor \sim(c3) \equiv (AO \Rightarrow \sim(c1)) \lor (AO \Rightarrow \sim(c2)) \lor (AO \Rightarrow \sim(c3)).
\]

That is, we need to prove that ambiguous overloading may be due to violating any one of the three conditions or even their combinations.

**a. Proof of \((AO \Rightarrow \sim(c1))\)**

Ambiguity, upon a function call, may rise due to the equality in both signature types \(\text{Sig1}\) and \(\text{Sig2}\) which is a violation to condition \(c1\).
Ambiguity, upon a function call, may rise due to overloaded functions whose signature types undergo subtyping whereas the functions themselves don't follow a subtyping relationship. This is a violation to condition c2.

The following rule shows that if the signature types undergo subtyping and no ambiguous overloading takes place then it is due to function subtyping.

\[
\begin{align*}
E \vdash \text{Sig1} &<: \text{Sig2} & E \vdash f(s1): \text{Rtype1} : \text{Sig1} \rightarrow \text{Rtype1} & E \vdash f(s1 \leftarrow s4: \text{Sig1}) \\
E \vdash f(s2): \text{Rtype2} : \text{Sig2} \rightarrow \text{Rtype2} & E \vdash f(s2 \leftarrow s4: \text{Sig2}) \notin \text{dom}(E) \\
\hline
E \vdash \text{ftype1} <: \text{ftype2}
\end{align*}
\]

Thus ambiguous overloading in the presence of signature subtyping is due to absence of function subtyping as formally shown

\[
\begin{align*}
E \vdash \text{Sig1} &<: \text{Sig2} & E \vdash \text{ftype1} : \text{Sig1} \rightarrow \text{Rtype1} \\
E \vdash f(s1): \text{Rtype1} : \text{Sig1} \rightarrow \text{Rtype1} & E \vdash f(s2): \text{Rtype2} : \text{Sig2} \rightarrow \text{Rtype2} \\
E, f(s2 \leftarrow s4: \text{Sig2}) \vdash f(s1 \leftarrow s4: \text{Sig1}) & E \vdash \text{ftype2} : \text{Sig2} \rightarrow \text{Rtype2} & E \vdash \text{ftype1} \leftarrow \leftrightarrow \text{ftype2} \\
\hline
\end{align*}
\]

\[
\begin{align*}
E \vdash \text{ftype1} \leftarrow \leftrightarrow \text{ftype2} & \sim c2 \\
\text{(Q.E.D.)}
\end{align*}
\]

c. Proof of \((\text{AO} \Rightarrow \sim(\text{c3}))\)

Ambiguity, upon a function call, may rise due to a function call to more than one overloaded function whose signature types do not experience subtyping because the signature of the function call is a common subtype between the invoked functions. This is a violation to condition c3.
With the above three proofs, we prove that the three conditions constitute a necessary condition for non-ambiguous overloading.

7.5. Type Checking for Overloading

To avoid ambiguous overloading, the following type checking process must be followed. It consists of the following stages:

1. **Checking the overloaded function declarations** to avoid ambiguity and to make sure they follow the rules of overloading functions. When a new declared function enters the symbol table, its signature is checked for uniqueness, it is checked against each of the already declared functions (the overloaded ones only) for function specialization. Since the list is always sorted based on the order of function specialization, the first matching function is the most specialized function. Moreover, if two functions turn out to be not subtypes of each other, they undergo the check for a common greatest subtype between them. The generated gcs is searched for in the list of overloaded function. If it is not found, a compilation error occurs indicating that the newly declared function causes ambiguous overloading. If the function is checked against the whole list and turned out to be not related to any of the other overloaded functions, it is inserted at the end of the list. This algorithm is interpreted by the function enter, whose interface is listed below, in appendix B.
function enter(table: symtable{fsymbol}, s: string): ref{fsymbol}
-- enters a new function in the symbol table

2. **Unification:** Unifying a function call against a function signature. As long as the declared function signatures satisfy the above rules, they guarantee no overloading conflicts. Since specialized functions are ordered in their linked list within the symbol table based on their specialization hierarchy, the first matching function is the most specific one. This guarantees that the first matching function is the most specific function appropriate for unification.

3. **Binding function calls to function addresses:**

   If the function call unifies with one of the functions, the hidden parameters are bound to specific values referenced by the binding area. The compiler translates this function call into a code pointer transfer to the code of the unified function. The code address of the function is achieved by generating the code label from the function name concatenated to the index stored in the attribute entry of the function. The actual parameters are stored on stack and the inferred hidden parameters are appended to the actual parameters.
8. CONCLUSION AND FUTURE RESEARCH

This thesis discussed new ideas of subtyping by specialization, where a type instance is a subtype of its base type and the more specific type instance is a subtype of the less specific type instance. As we have seen, the concept of subtyping by specialization opens the door for a powerful function overloading and function specialization based on the covariance relation between function subtyping and signature subtyping. Formal notation representing the SIMPL Type System in formal type rules, and the formal proofs of the proposed rules of overloading are supplied. Function hidden parameters constitute a new concept of parametric polymorphism as supported by SIMPL. This new concept adds power to function declarations and function calls where constraint integrity can be imposed through hidden parameters; accordingly, less number of formal parameters is required i.e. more information with less number of parameters.

Moreover, this research work delves into the idea of inferring hidden parameters and designs an efficient semantic space structure to capture the semantic information of the programs and utilizes it in type checking algorithms.

Yet there are some few parts that need further research. Among which: late binding, constraints on formal parameters, concatenating the signature to the function name, allowing enumeration types to be formal type parameters and accordingly function hidden parameters, the dynamic binding of the union types and adding inheritance as a form of subtyping by extension.
8.1. Late Binding

A more serious problem with function specialization is that we cannot always determine statically at compile time the address of a called function. In other words, it is not always possible to statically bind a function call to a function address. Type information is needed at runtime to achieve dynamic or late binding. The following example clarifies this problem. Function \( g \) calls function \( f \), where there are two overloaded functions share the name \( f \): one accepts \( x \) of any type and the other accepts \( x \) as an array. If \( x \) is of an \texttt{array} \texttt{type}, the second function \( f \) should be called. Otherwise, the first one should be called. The difficulty here is that the specialized (second) function \( f \) can be defined in a different module after function \( g \) has been compiled. Other specialized functions of \( f \) can be defined elsewhere. Therefore, \texttt{SIZEt} is no longer sufficient as a hidden parameter and type information is required when function specialization is permitted in a programming language.

\[
\begin{align*}
\text{function} & \ f \ \{t: \texttt{type}\}(\texttt{obj} \ x: \ t) \\
\text{function} & \ f \ \{n: \texttt{integer}, \ t: \texttt{type}\}(\texttt{obj} \ x: \texttt{array\{n, \ t\}}) \\
\text{function} & \ g \ \{t: \texttt{type}\}(\texttt{obj} \ x: \ t) \ \textbf{is} \\
& \quad \ldots \ \\
& \quad f(x) \\
& \quad \ldots \\
\end{align*}
\]

8.2. Constraints on Formal Parameters

The work done in the previous chapters considered constraints enforced on hidden parameters as they are referenced by some formal parameters. A further enhancement to this work would be allowing formal parameters to reference other formal parameters. However, the restriction on formal type parameters of parameterized types still apply, i.e. constraints can be applied on formal parameters of the type \texttt{integer} and \texttt{type} variables. For instance, the following function declaration will be allowed in the type system

\[
\begin{align*}
\text{function} & \ g \ \{n: \texttt{integer}, \ t: \texttt{type}\} \ (m: \texttt{integer},
\end{align*}
\]
To support this feature, there must be a `constraints` entry added in the attribute space to keep the number of constrained parameters in the function header. Furthermore, the `intpar` entry might be replaced with an `intval` entry.

8.3. Concatenating the Function Signature to the Function Name

An efficient way to ensure unambiguous overloading is to enforce and store a unique signature per function. This can be achieved by concatenating the signature to the function name in the name field of the symbol table entries. However, the lookup function hashes the function name and the number of formal parameters but not the signature. The next field links all functions that share the same name and same number of parameters. The structure of the signature and what type of information stored in it determine to what degree is overloading allowed. A suggested structure to the signature is listing the type of the formal parameters separated by `'_ '`. The types of formal parameters are represented as a lowercase type signature followed by the list of the type parameters enclosed in `{ }` and separated by commas. 'I' represents any Integer type parameter while 'T' represents any type variable. References to the hidden parameters can be expressed as '#' followed by a serial number. For instance, the function name of the function declaration

```plaintext
function f {t: type, n: integer} (a: array{n, real},
array{n, array{3, t}}) : t
```

as stored in the symbol table entry will be

```plaintext
f_2_arrayIT(#0, real)_arrayIT(#0, arrayIT{3, #1})
```

this signature structure speeds up the overloading and specialization checks but takes big storage space and has its own overhead of generation.
8.4. Enumeration Types as Formal Type Parameters

Allowing parameterized types to use enumeration types as their formal parameters can be of use in some cases such as indexing arrays with enumeration literals. In such a case, parameterized types can have integer variables, enumeration types and type variables as their formal parameters. Accordingly, the type signature stored in the symbol table has to include 'E' to represent an enumeration type. In such a case, all enumeration types are represented by the same letter which means that the system does not allow overloading of types based on different enumeration types; the types \( \forall t: \text{TYPE, } b: \text{boolean} \) array\( \{b, t\} \) and the type \( \forall t: \text{TYPE, } c: \text{char} \) array\( \{c, t\} \) cannot coexist because they own the same type signature \( \text{arrayET} \). There must be a unique representation for each enumeration type.

8.5. Dynamic Binding for Union Types

As mentioned in the SIMPL chapter, union types represent the dynamic typing in SIMPL. This indicates that dynamic type checking must be encoded in the translation of the programs to dynamically check for the current type of a union type – by checking the current value of its tag – and compares it against the corresponding identifier that is passed to or the unique type it is compared against.

8.6. Inheritance as Subtyping by Extension

Another strong feature that can be added to the type system of SIMPL is the support for inheritance. This can be accomplished through the concept of subtyping by extension; where a new type is declared to be of the same structure of an existing type with extra fields or extra functions. Such declaration would consist of the keyword \textbf{type} followed by the new type name and its parameters – if it is parameterized – then the keywords \textbf{is type} and the existing type name with its actual parameters. This is succeeded by the keyword \textbf{with} and the declaration of extra fields or
functions. For instance, the type two, declared below, inherits from type one and extends its structure with the field length. Moreover, it adds the needed functions for this extra field.

```pascal
type two{n: integer, t: type} is type one{n, t} with
    length : integer
    function getlength( ): integer
    function setlength(l: integer)
end
```

-- type two

Accordingly, multiple inheritance can be achieved through listing several types separated by the keyword `and`. For instance, the declared type three inherits from both types one and four.

```pascal
type three{n: integer, t: type} is type one{n, t} and four{n}
    with
    storage : array{n, t}
end
```

-- type three

This feature supplements the object-oriented interface and the data abstraction - achieved through type interface and implementation – in providing the object-oriented programming approach.
Bibliographies


[Bourdoncle97] Bourdoncle, François. “Type-Checking Higher-order Polymorphic


[Kaes88] Kaes, S. “Parametric Polymorphism” In European Symposium on


References


(http://www.research.digital.com/SRC/personal/Luca_Cardelli/Papers.html)

(http://blackcat.brynmawr.edu/~kimberly/Miranda.html)


APPENDIX A:

Subtyping in System $F_{<}$ vs. Subtyping in SIMPL Type System

System $F_{<}$ is an extension to system $F$ (a well known typed $\lambda$-calculus with polymorphic types) that combines parametric polymorphism with subtyping. Although system $F_{<}$ is designed for imperative languages in general, its subtyping rules are based on the object-oriented concepts. The subtype relation is intended, intuitively, as set inclusion between types, where types are seen as sets of values [Cardelli96]. Subtyping – as defined by [Cardelli96] – is a reflexive and transitive binary relation over types, which satisfies subsumption; it asserts the inclusion of collections of values. However, a subtype is meant to be an inheriting subclass from its supertype (the superclass) [Abadi96]. The system declares a type Top that is a supertype of all types.

SIMPL provides parametric polymorphism and subtyping via instantiating parameterized types. As long as we consider types to be sets of values, subtypes can be regarded as subsets. All types are members of TYPE where TYPE is the set of all types. $\forall s$ that is a specific type, $s \in$ TYPE. That is, TYPE is a set of sets. Any type variable $t$ is a member of $\text{TYPE}$, $t \in \text{TYPE}$. Actually, type variables can be any type. A type variable is a family of types i.e. it is the intersection of all types. Since any type is a set of values, it can be a subset of the type variable set. That is, a specific type $s$ is a subtype of the type variable $t$. $s <: t$.

System $F_{<}$ considers parameterized types (they are called universally quantified types) $\forall X.A$. Upon applying a subtype constraint on their type parameter, they are called bounded quantifications $\forall(X <: B) A$. Parameterized types (some authors call it polytype) are actually a family of some types. A parameterized type is the intersection of all the types that can be derived from it. For instance, $\forall \{n: \text{integer}, t: \text{TYPE}\} \text{array}\{n, t\}$ is a set that includes all the elements of the set $\forall \{t:\text{TYPE}\} \text{array}\{3, t\}$ and those of the set $\forall \{n: \text{integer}\} \text{array}\{n, \text{real}\}$ and much more. However, all the arrays of real elements belong to the set $\forall \{n: \text{integer}\} \text{array}\{n, \text{real}\}$, and all
arrays that hold 3 elements belong to the set \( \forall \{ t: \text{TYPE}\} \text{array} \{3, t\} \). That is, \( \forall \{ n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, n, t\} \). Accordingly, \( \forall \{ t: \text{TYPE}\} \text{array} \{3, t\} <:\ \forall \{ n: \text{integer}, t: \text{TYPE}\} \text{array} \{n, n, t\} \). Hence, any type instance is a subtype of its parameterized type. The subtyping hierarchy is constructed according to sorting the types based on their parameters and the hierarchy of instantiation of these parameters. That is, a parameterized type \( s \{3, \text{real}\} \) is subtype of \( \forall \{ t: \text{TYPE}\} s \{3, t\} \) which is a subtype of \( \forall \{ n: \text{integer}, t: \text{TYPE}\} s \{n, t\} \). At the top of this hierarchy resides any type variable \( t \). The subtyping rule for parameterized types is

\[
\begin{align*}
\text{(Sub } \forall \text{)} & \\
E \vdash s \{s_1, m\} : \forall(n: \text{integer}, t: \text{TYPE}) S \{t \leftarrow s_1, n \leftarrow m\} & \\
E \vdash s \{s_1, m\} \subseteq \forall(n: \text{integer}, t: \text{TYPE}) S \{t, n\}
\end{align*}
\]

System \( F_\omega \) has a different view of subtyping; it is based on inheritance. Accordingly, a subtype (a subclass) includes more data than its supertype. This leads to the fact that subtyping among universally quantified types is contravariant to the subtyping among their parameters. [Cardelli91] states that a bounded quantifier is antimonotonic in its bound and monotonic in its body under an assumption about the free variable.

\[
\begin{align*}
\text{(Sub } \forall \text{)} & \\
E \vdash A' <: A & E, X <: A' \vdash B <: B' & \\
E \vdash \forall(X <: A) B <: \forall(X <: A') B'
\end{align*}
\]
The subtyping rule for function space (Sub →) declares the subtyping of function types according to the following conditions. For the function type \( f_1 \) to be a subtype of another function type \( f_2 \), the signature type of the \( f_2 \) must be a subtype of \( f_1 \)'s signature and the return type of \( f_1 \) must be a subtype of the return type of \( f_2 \). For instance, if \( \text{integer} \ll 	ext{TYPE} \) and \( \text{real} \ll 	ext{TYPE} \) then \( \text{TYPE} \rightarrow \text{real} \ll \text{integer} \rightarrow \text{TYPE} \)

The subtype relation for function types undergoes inverted inclusion (contravariance\(^4\)) for function arguments, while it goes in the same direction (covariant\(^5\)) for function results. [Cardelli96] explains this as follows. “A function \( M \) of type \( A \rightarrow B \) accepts elements of type \( A \); obviously it also accepts elements of any subtype \( A' \) of \( A \). The same function \( M \) returns elements of type \( B \); obviously it returns elements that belong to any supertype \( B' \) of \( B \). Therefore, any function \( M \) of type \( A \rightarrow B \), by virtue of accepting arguments of type \( A' \) and returning results of type \( B' \), has also type \( A' \rightarrow B' \). The latter is compatible with saying that \( A \rightarrow B \) is a subtype of \( A' \rightarrow B' \).

\[
(\text{Sub } \rightarrow)
E \vdash A' <: A \quad E \vdash B <: B'
\]

\[
E \vdash A \rightarrow B <: A' \rightarrow B'
\]

[Abadi96] states that much controversy in the object-oriented community on whether method argument types should vary covariantly or contravariantly from classes to subclasses. The authors assert they cannot let method argument types to vary covariantly, unless they change the meaning of covariance, subtyping or subsumption. This explains the difference between the two systems. It

\(^4\) Contravariant is a type that varies in the inverse direction from one of its parts with respect to subtyping. The main example is the contravariance of function types in their domain; e.g. if \( A <: B \) and \( X \) varies from \( A \) to \( B \) in \( X \rightarrow C \); we obtain \( B \rightarrow C <: A \rightarrow C \). Thus, \( X \rightarrow C \) varies in the inverse direction of \( X \) [Cardelli96].

\(^5\) Covariant is a type that varies in the same direction as one of its parts with respect to subtyping; e.g. if \( A <: B \) and \( X \) varies from \( A \) to \( B \) in \( D \rightarrow X \); we obtain \( D \rightarrow A <: D \rightarrow B \). Thus, \( D \rightarrow X \) varies in the same direction of \( X \) [Cardelli96].
is worth to mention that Eiffel, the object-oriented language, supports covariance of method arguments [Meyer92].

SIMPL, on the other hand, states that function subtyping is covariant with both the arguments’ subtyping and result subtyping. It is due to subtyping by instantiation that gives arguments their subtyping hierarchy. For instance, if we have the following declared function type in SIMPL:

```
Type ftype {n: integer, t: type} is function (array {n, t}): t
```

And the following function declarations:

```
Function f {n: integer, t: type} (array {n, t}): t
Function g (array {5, real}): real
Function h {n: integer, t: type} (array{n, array{n, t}}): array{n, t}
```

They are represented in SIMPL Type system as follows:

```
ftype {n: integer, t: type} : (array {n, t}) \rightarrow t
f : ftype { n: integer, t: type }
g : ftype { 5, real }
h: ftype { n: integer, a: array{n, t} }
```

These functions are instances of the function type `ftype`. Yet, they are sorted in a subtyping hierarchy based on the subtyping relation among their arguments and result type. That is, `h <: f` and `g <: f`.

![Figure A.1. Subtyping among Function Types](image-url)
Accordingly, any higher order function

\[
\text{Function } f_2 \{n: \text{integer}, t: \text{type}\}(\text{fun: ftype } \{n, t\}, \ldots)
\]

that accepts, as one of its arguments, a function of type ftype, can be called with any of the three functions – as they are all instances of function subtypes of ftype.

The subtyping rule for function types is

\[
\text{(Sub } \rightarrow) \\
E \vdash S_1 \{\ldots\} \leq S_2 \{\ldots\} \quad E \vdash S_3 \{\ldots\} \leq S_4 \{\ldots\} \\
\hline \\
E \vdash S_1 \{\ldots\} \rightarrow S_3 \{\ldots\} \leq S_2 \{\ldots\} \rightarrow S_4 \{\ldots\}
\]

System F\(_c\) has its own view of method specialization; upon relaxation of the method overriding into method specialization, we allow an overriding method to adopt different argument and result types, specialised for the subclass. [Abadi96] states the rule as such:

“Suppose we use different argument and result types, \(A'\) and \(B'\), for an overriding method \(m\):

\[
\text{class } c \text{ is} \\
\text{method } m(x: A): B \text{ is } \ldots \text{ end;} \\
\text{method } m_1(x: A_1): B_1 \text{ is } \ldots \text{ end;} \\
\text{end;} \\
\text{subclass } c' \text{ of } c \text{ is} \\
\text{override } m(x: A'): B' \text{ is } \ldots \text{ end;} \\
\text{end;}
\]

In determining the admissible \(A'\) and \(B'\) we are constrained by the possibility of subsumption between \(\text{InstanceTypeOf}(c')\) and \(\text{InstanceTypeOf}(c)\). When \(o'\) of type \(\text{InstanceTypeOf}(c')\) is subsumed into \(\text{InstanceTypeOf}(c)\), \(o'.m(a)\) is invoked, it is acceptable for the argument to have static type \(A\) and for the result to have static type \(B\). Therefore it is sufficient to require that \(B' \prec: B\) (covariantly) and that \(A \prec: A'\) (contravariantly). This is called method specialization on override: the result type \(B\) is specialized to \(B'\) and the parameter type \(A\) is generalized to \(A'\), with the net effect that \(A \rightarrow B\) is specialized to \(A' \rightarrow B'\). “

SIMPL Type system has a different definition of function specialization; when the overloaded functions share the same function name, same number of arguments, and the type of
each argument is a subtype of the corresponding argument’s type in the other signature, it is called specialization. Moreover, the result types experience a subtyping relation. Actually, specialization is a special case of overloading.

For instance, the function \( f (x: \text{integer}): \text{integer} \) is a specialization to the function \( \forall t: \text{TYPE} \ f (x: t) : t \) It can be formalized as follows:

\[
\begin{align*}
E \vdash \text{ftype1} \prec \text{ftype2} \\
E \vdash f(\text{Sig1}):\text{Rtype1} : \text{ftype1} \\
E \vdash f(\text{Sig2}):\text{Rtype2} : \text{ftype2} \\
\hline
E \vdash f(\text{Sig1}):\text{Rtype1} < f(\text{Sig2}):\text{Rtype2}
\end{align*}
\]
Appendix B:

Unification Algorithm

This appendix describes the algorithms needed for unification, subtyping checks, non-ambiguous function overloading and function specialization. These algorithms rely mainly on the information stored in the symbol table and the attribute space. The first section lists the peripheral functions needed along the checking.

B.1. Implementation of Some Auxiliary Functions

Function num_fparams(a: attref): integer is
-- returns the number of formal parameters in a tuple
    result := a^.value
end

function tag(a: attref): tagtype is
-- returns the tag of an attribute pointed to by a
    result := a^.tag
end

function value(a: attref): valtype is
-- returns the value of an attribute pointed to by a
    result := a^.value
end

function name(s: symbol): string is
-- returns the name of the symbol pointed to by s
    result := s.name
end

function attribute(s: symbol): attref is
-- returns a pointer to the attribute entry of the symbol pointed to by s
    result := s.attribute
end
function setvalue(a: attref, v: valtype) is
-- sets the value of an attribute pointed to by a to the value v
    a^.value := v
end

function settag(a: attref, v: tagtype) is
-- sets the tag of an attribute pointed to by a with the value v
    a^.tag := v
end

function hidden(a: attref): integer is
-- returns the number of hidden parameters of the function
-- attribute pointed to by a
    if tag(a) = 'function'
        then result := value(a+1)
        else result := value(a)
    end

function tuple(a: attref): attref is
-- returns a pointer to the tuple attribute of the function
-- pointed to by a
    if tag(a) = 'function'
        then result := a+ hidden(a) + 2
        else result := a+ hidden(a) + 1
    end

function result_type(a: attref): attref is
-- returns a pointer to the result type of the function
-- attribute pointed to by a
    if tag(a) = 'function'
        then
            result := value(a + hidden(a) + num_fparams(tuple(a))+ 3)
        else
            result := value(a + hidden(a) + num_fparams(tuple(a))+ 2)
        end

function parameter(a: attref, I: integer): attref is
-- needs low-level programming for pointer arithmetic
    result := a + I
-- overloaded '+'
end
function initialize (c: attref, n: integer) is
-- initializes all the binding entries in the constraint area
    for I:= 0 to n-1 do
        value(c + I) := 'null'
    end
-- for loop
end
-- initialize

function next(s: ref{fsymbol}): ref{fsymbol} is
-- returns a pointer to the symbol entry linked next to the
-- symbol pointed to by s
    result := s^.next
end
-- next

function inttype (a: attref): boolean is
-- checks if the type of a is integer
    result := value(a) = value(attribute(lookup(symbol_table, "integer")))
end

B.2. The Unify Function

This function unifies a function call against a declared function. If they unify the function
returns the binding area with each entry that corresponds to a hidden parameter bound to the
appropriate binding. This is a recursive function that unifies the individual parameters of the tuple
and the result type in sequence.

function unify (formal, actual, bind: attref):boolean is
-- unifies the formal against the actual and stores the inferred
-- values of the hidden parameters
    obj j : integer := 1
    if tag(formal) = 'ref'
        then
            result := unify(value(formal),actual, bind)
    elseif tag(actual) = 'ref'
        then
            result := unify(formal, value(actual) , bind)
    elsif tag(formal) = 'tuple' and tag(formal) = 'tuple' and
        value(formal)= value(actual) then
            result := 'true'
            while result and j <= value(formal) do
                result := unify(parameter(formal, j),
parameter(actual, j), bind)
    j := j + 1  -- while
elsif tag(formal) = 'obj' and tag(actual) = 'obj' then
    result := unify(value(formal), value(actual), bind)
elsif tag(formal) = 'val' and
    (tag(actual) = 'val' or tag(actual) = 'obj')
then
    result := unify(value(formal), value(actual), bind)
elsif tag(formal) = 'typevar' and
    value(parameter(bind, value(formal))) = 'null' -- unbound
    and (tag(actual) = 'type' or tag(actual) = 'typeinst'
    or tag(actual) = 'typevar' or tag(actual) = 'uniontype'
    or tag(actual) = 'syntype' or tag(actual) = 'ftype')
then
    setvalue(parameter(bind, value(formal)), actual)
    result := 'true'
elsif tag(formal) = 'typevar' and
    value(parameter(bind, value(formal))) <> 'null' -- bound
then
    result := unify(parameter(bind, value(formal)),
    actual, bind)
elsif tag(formal) = 'intpar' and
    value(parameter(bind, value(formal))) = 'null' -- unbound
    and (tag(actual) = 'iliteral' or tag(actual) = 'intpar'
    or (tag(actual) = 'val' and inttype(actual) ) )
then
    setvalue(parameter(bind, value(formal)), actual)
    result := 'true'
elsif tag(formal) = 'intpar' and
    value(parameter(bind, value(formal))) <> 'null' -- bound
then
    result := unify(parameter(bind, value(formal)),
    actual, bind)
elsif tag(formal) = 'typeinst' and tag(actual) = 'typeinst'
    and value(formal) = value(actual) -- same type
then
    result := 'true'  -- unify their actual parameters
    while result and j <= value(value(formal)) do
        result := unify(parameter(formal, j),
        parameter(actual, j), bind)
        j := j + 1  -- while
    end
elsif tag(formal) = 'typeinst' and tag(actual) = 'typeinst'
then
    if tag(value(formal)) = 'syntype' and
    tag(value(actual)) <> 'syntype' and
    then
        result := unify(value(formal), actual, bind)
    elseif tag(value(formal)) <> 'syntype' and
    tag(value(actual)) = 'syntype' and
    then
        result := unify(formal, value(actual), bind)
elsif tag(value(formal)) = 'syntype' and
tag(value(actual)) = 'syntype' and
unify(value(formal), value(actual), bind)
then
result := 'true'  -- unify their actual parameters
while result and  j <= value(value(formal)) do
  result := unify(parameter(formal, j),
               parameter(actual, j), bind)
  j := j + 1  -- while
end  -- if
elsif tag(formal) = 'syntype'
then
  result := unify(formal + hidden(formal)+1, actual, bind)
elsif tag(actual) = 'syntype'
then
  result := unify(formal, actual + hidden(actual)+1, bind)
elsif tag(formal) = 'uniontype'
then
  result := 'true'
  for i := 1 to value(formal + value(formal)+ 2) do
    result := result and unify(parameter(formal +
                                 value(formal)+ 2), i), actual, bind)
  end  -- for loop
elsif tag(actual) = 'uniontype'
then
  result := 'true'
  for i := 1 to value(actual + value(actual)+ 2) do
    result := result and unify(formal, parameter(actual +
                                 value(actual)+ 2), i), bind)
  end  -- for loop
elsif tag(formal) = 'ftype' and tag(actual) = 'ftype'
then
  -- unify formal parameters
  initialize(bind, value(formal))
  result := unify(tuple(formal) -1, tuple(actual) -1, bind)
if result
  result := unify(value(tuple(formal) +
                        value(tuple(formal) - 1)),
               value(tuple(actual) +
                        value(tuple(actual) - 1)), bind)
  ;; if
elsif tag(formal) = 'ftype' and tag(actual) = 'function'
then
  -- unify the formal parameters
  initialize(bind, value(formal))
  result := unify(tuple(formal) -1, tuple(actual), bind)
if result
  -- unify result types
  result := unify(value(tuple(formal) +
                        value(tuple(formal) - 1)),
               result_type(actual), bind)
  ;; if
elsif tag(formal) = 'function' and tag(actual) = 'function' then
    initialize(bind, value(formal+1))
    result := unify(tuple(formal), tuple(actual), bind)
    if result
        -- if formal parameters unify
        result := unify(result_type(formal),
                        result_type(actual), bind)
    end
    -- if
elsif formal = actual
    -- same attribute
then
    result := 'true'
else
    result := 'false'
end
-- if
-- unify

B.3. Subtyping Check

This function, in general, checks if the first type is a subtype of the second type. Its main use is in
detecting function subtyping. Yet, being a recursive function, it can check the subtype relation
between any two types. The kind of subtyping checked here means subtyping by specialization.

function subtype(t1, t2, c1, c2: attref): boolean is
    -- checks if t1 is a subtype of t2 and stores inferred values
    -- for both types
i : integer

if tag(t1) = 'function' and tag(t2) = 'function' then
    result := subtype(tuple(t1), tuple(t2), c1, c2)
    if result then
        result := subtype(result_type(t1), result_type(t2),
                        c1, c2)
    end
    -- if
elsif tag(t1) = 'tuple' and tag(t2) = 'tuple' and
value(t1)=value(t2) then
    result := 'true'
i := 1
while i <= value(t1) and result do
    result := subtype(parameter(t1,i), parameter(t2,i),
                        c1, c2)
end
-- while
elsif tag(t1)='obj' and tag(t2)='obj' then
    result := subtype(value(t1), value(t2), c1, c2)
elsif tag(t1)='val' and tag(t2)='val' then
    result := subtype(value(t1), value(t2), c1, c2)
if result then
  if value(t1+1) > 0 and value(t2+1) = 0 then
    result := subtype(value(parameter(c1, value(t1+1), t2, c1, c2)
  elsif value(t1+1) = 0 and value(t2+1) > 0 then
    result := subtype(t1, value(parameter(c2, value(t2+1), c1, c2)
  elsif value(t1+1) > 0 and value(t2+1) > 0 then
    result := subtype(value(parameter(c1, value(t1+1), value(t2+1), c1, c2)
  end
end
elsif (tag(t2) = 'type' and tag(t1) = 'typeinst') then
  result := (t2 = value(t1))
elseif tag(t2) = 'typevar' and
  value(parameter(c2, value(t2))) = 'null' and
  (tag(t1) = 'typevar' or tag(t1) = 'type')
  or tag(t1) = 'typeinst')
  then
  result := 'true'
  setvalue(parameter(c2, value(t2)), t1) -- bind it
end
elsif tag(t2) = 'val' and tag(t1) = 'iliteral'
  then if value(t2+1) = 0 then
    result := 'true'
  elsif value(t2+1) > 0 and
    value(parameter(c2, value(t2+1)) <> 'null' then
    result := subtype(t1, value(parameter(c2, value(t2+1)), c1, c2)
  else
    result := 'true'
  end
elsif tag(t2) = 'val' and tag(t1) = 'eliteral'
  and value(t1) = value(t2) then
  result := 'true'
elsif value(t2+1) > 0 and
  value(parameter(c2, value(t2+1)) <> 'null' then
  result := subtype(t1, value(parameter(c2, value(t2+1)), c1, c2)
else
  result := 'true'
elsif tag(t1) = 'ref'
    then result := subtype(value(t1), t2, c1, c2)
elsif tag(t2) = 'ref'
    then result := subtype(t1, value(t2), c1, c2)
elsif tag(t2) = 'typevar' and value(parameter(c2, value(t2)) <> 'null' -- bound type
    then
        if tag(t1) = 'typeinst' then
            result := subtype(value(t1), value(parameter(c2, value(t2))), c1, c2)
elsif tag(t1) = 'type' then
            result := subtype(t1, value(parameter(c2, value(t2))), c1, c2)
elsif tag(t1) = 'typevar' and value(parameter(c1, value(t1)) <> 'null' -- bound type
    then
        result := subtype(value(parameter(c1, value(t1))),
            value(parameter(c2, value(t2))), c1, c2)
elsif tag(t1) = 'typevar' and value(parameter(c1, value(t1)) = 'null' -- unbound type
    then
        result := 'true'
selse: parameter(c1, value(t1)),
            value(parameter(c2, value(t2)))
else
    result := 'false'
end -- if
elsif (tag(t1) = 'typeinst' and tag(t2) = 'typeinst')
    then -- both of them are type instances
        if (value(t1) = value(t2))
    then
            -- share the same base type
            for I := 1 to value(value(t1)) do
                -- for each parameter
                    result := subtype(parameter(t1, I),
                        parameter(t2, I))
            end -- for
        end -- if
elsif tag(t1) = 'eliteral' and tag(t1) = 'eliteral'
    and value(t1) = value(t2) and value(t1+1) = value(t2+1)
then
    -- same enumeration literals
elsif tag(t1) = 'eliteral' and tag(t1) = 'eliteral'
    and value(t1) = value(t2)
then
    -- same integer literals
elsif t1 = t2 then result := 'true' -- point to the same
    -- place in the attribute space
else result := 'false'
end -- if
end -- subtype
B.4. Check for Unique Function Signature

This function performs a check whether the input function has a unique signature compared to the rest of overloaded functions stored in the symbol table. This function calls another function named identical. Identical function checks if two parameters are identical in their mode and type.

```plaintext
function identical(p1, p2: attref): boolean is
  -- checks if p1 and p2 share the same mode and type
  result := tag(p1) = tag(p2) and
             tag(value(p1)) = 'type' and
             tag(value(p1)) = 'type' and
             value(p1) = value(p2) or
             (tag(value(p1)) = 'typeinst' and tag(value(p1)) = 'typeinst'
               and value(value(p1)) = value(value(p2)))
end -- identical

function unique_signature(f: ref{fsymbol},
                           table: symtable{fsymbol}) : boolean is
  -- checks if the function f has a unique signature
  -- in the symbol table
  nextf: ref{fsymbol}
  a1, a2: attref
  k: integer

  result := 'true'
  a1 := tuple(attribute(f))
  nextf := next(f) -- picks the next function
                  -- with the same name
  while (nextf <> 'null') do
    k = 0 -- counts identical parameters
    a2 := tuple(attribute(nextf))
    for I := 1 to num_fparams(a1) do
      if identical(parameter(a1, I), parameter(a2, I))
        k = k + 1 -- another identical parameters
      end -- if
    end -- for
    if k = num_fparams(f) -- identical signatures
      result := 'false'
      exit -- if
    end
    nextf := next(nextf) -- pick the next function
  end -- while
end -- unique_signature
```
B.5. Check for the Greatest Common Subtype

The \texttt{gcs} function generates the greatest common function subtype between two functions. In addition, it reserves binding space equivalent to the sum of hidden parameters in both functions due to the fact that the constructed subtype holds the union of both sets of hidden parameters. The \texttt{gcs} function uses the \texttt{copylink} function after generating a \texttt{gcs} to copy the list of bindings of the hidden parameters into the produced greatest common subtype.

\begin{verbatim}
function copylink(final, x, y, h1, h2: attref): attref is
    -- copies the hidden parameters of x and y in final
val j : integer
result := final
if tag(result) = 'function' then j := 2 else j := 1
for i:= 1 to hidden(x) do
    tag(result + j + i) := tag(x + j + i)
    value(result + j + i) := value(h1 + i - 1)
end -- for loop
if tag(result) = 'function'
then j := 2 + hidden(x)
else j := 1 + hidden(x)
for i:= 1 to hidden(y) do
    tag(result + j + i) := tag(x + j + i)
    value(result + j + i) := value(h1 + i - 1)
end -- for loop
end

function gcs (x, y, h1, h2: attref): attref is
    if subtype(x, y, h1) -- if x is a subtype of y
        then result := x
    elsif subtype(y, x, h2) -- if y is a subtype of x
        then result := y
    elsif tag(y) = 'typeinst' and tag(x) = 'typeinst'
        and value(x) = value(y) -- same type
        and value(value(x)) = value(value(y)) -- same no. of parameters
        then
            settag(result, 'typeinst')
if value(value(x)) > 0 then -- if they are parameterized
    I := 0
    Do
        I := I + 1
end
\end{verbatim}
\[
\text{parameter}(\text{result}, I) := \text{gcs}([\text{parameter}(x, I), \text{parameter}(y, I), h1, h2])
\]

\[\text{while} \quad \text{parameter}(\text{result}, I) \not= 'null'
\quad \text{and} \quad I \leq \text{value}(\text{value}(x))\]
\[\text{end} \quad -- \text{if}\]

\[\text{elsif} \quad \text{tag}(y) = 'type' \quad \text{and} \quad \text{tag}(x) = 'type'
\quad \text{and} \quad \text{value}(x) = \text{value}(y) \quad -- \text{same no. of}
\quad \text{parameters}\]
\[\text{then}\]
\[\text{settag}(\text{result}, 'type')\]
\[\text{for} \quad I := 1 \quad \text{to} \quad \text{value}(x) \quad \text{do}\]
\[\quad \text{parameter}(\text{result}, I) := \text{gcs}([\text{parameter}(x, I), \text{parameter}(y, I), h1, h2])
\quad \text{if} \quad \text{parameter}(\text{result}, I) = 'null' \quad \text{then exit end}\]
\[\text{end} \quad -- \text{for loop}\]

\[\text{elsif} \quad (\text{tag}(y) = 'functiontype' \quad \text{and} \quad \text{tag}(x) = 'functiontype')
\quad \text{or} \quad (\text{tag}(y) = 'function' \quad \text{and} \quad \text{tag}(x) = 'function') \quad \text{and}
\quad (\text{num\_fparams}(\text{tuple}(x)) = \text{num\_fparams}(\text{tuple}(y)))\]
\[\text{then}\]
\[\text{settag}(\text{result}, \text{tag}(x))\]
\[\text{if} \quad \text{tag}(x) = 'functiontype'
\quad \text{then} \quad \text{setvalue}(\text{result}, \text{value}(x) + \text{value}(y))\]
\[\text{else} \quad \text{setvalue}(\text{result}+1, \text{value}(x+1) + \text{value}(y+1))\]
\[\text{end} \quad -- \text{if}\]
\[\text{for} \quad I := 1 \quad \text{to} \quad \text{num\_fparams}(\text{tuple}(x)) \quad \text{do}\]
\[\quad \text{parameter}(\text{tuple}(\text{result}), I) :=
\quad \quad \text{gcs}([\text{parameter}(\text{tuple}(x), I), \text{parameter}(\text{tuple}(y), I),
\quad \quad h1, h2])\]
\[\quad \text{if} \quad \text{parameter}(\text{tuple}(\text{result}), I) = 'null' \quad \text{then exit end}\]
\[\text{end} \quad -- \text{for loop}\]
\[\text{if} \quad \text{parameter}(\text{tuple}(\text{result}), \text{num\_fparams}(\text{tuple}(x))) \not= 'null'\]
\[\text{then}\]
\[\quad \text{settag}(\text{result} + \text{hidden}(\text{result}) + 2 +
\quad \text{num\_fparams}(\text{tuple}(x)), 'result')\]
\[\quad \text{setvalue}(\text{result} + \text{hidden}(\text{result}) + 2 +
\quad \text{num\_fparams}(\text{tuple}(x)),
\quad \text{gcs}(\text{result\_type}(x), \text{result\_type}(y), h1, h2))\]
\[\quad \text{result} := \text{copylink}(\text{result}, x, y, h1, h2)\]
\[\text{else} \quad \text{result} := 'null' \quad -- \text{if}\]
\[\text{end} \quad -- \text{function gcs}\]

\[\text{B.6. Function Specialization Check}\]

This check is needed to detect if function f1 is a correct specialization of f2. It is used in sorting overloaded functions in the symbol table. It utilizes the unify function in its check.
function specialization(f1, f2: fsymbol): boolean is
val a1, a2 : attref

    a1 := attribute(f1)
a2 := attribute(f2)
if name(f1) = name(f2) and tag(a1) = 'function'
    and tag(a2) = 'function'
then
    result := 'true'
for i:= 1 to num_fparams(tuple(a1))do
    -- check exact modes
    result := result and (tag(parameter(tuple(a1),i)) =
    tag(parameter(tuple(a2),i)))
end
if result
then
    -- reserve space for the hidden
    -- parameters of function f2
    initialize(bind, hidden(f2))
if unify(tuple(a1), tuple(a2), bind)
then
    result := unify(result_type(a1), result_type(a2),
    bind)
end
end
end
end
-- specialization

B.7. Sorting Overloaded Functions in the Symbol Table

The sorting of overloaded functions is achieved as they enter the symbol table; each newly entered
function is checked against the already stored functions for the uniqueness of its signature, whether
it is a specialization of any of the other functions. Once the specialization function confirms
that it is a specialization of a certain function, it is inserted before this function to preserve the order
of specialization among the overloaded functions. Even if the two functions turned out to be not
subtypes of each other, a check for the existence of a common subtype between them. In case they
have a common subtype, the algorithm searches for it among the overloaded functions; if it is found
then the overloading is acceptable and the function can proceed with the tests. Otherwise, this
overloading may create ambiguity later on; accordingly, the entrance of this function is rejected
and a warning of a missing function with the header of the greatest common subtype is required. If
the overloaded function is not a specialization of any of the other functions, it is stored at the end of the list. At the beginning, enter function extracts the semantic information from the input string and stores it in a temp area in the symbol table. This functionality is achieved by the construct_symbol function.

```plaintext
function construct_symbol(s: string): ref{fsymbol} is
  -- constructs a symbol with the information extracted from
  -- the input string
end

function insert_before(obj list, one, two: ref{fsymbol}) is
  -- inserts the element one before the element two in the list
end

function found(element, list: ref{fsymbol}): boolean is
  -- returns true if the element is found in the list
end

function insert_last(list, element: ref{fsymbol}) is
  -- inserts the element at the end of the list
end

function enter(table: symtable{fsymbol}, s: string): ref{fsymbol} is
  -- enters a new function in the symbol table

obj first : ref{fsymbol}
temp, current : ref{fsymbol}
inserted, nogcs: boolean
gcsref, h1 , h2: attref

temp := construct_symbol(s)
first := lookup(table, s)
result := first
if unique_signature(temp, table) then
  inserted := 'false'
  while result <> 'null' and not(inserted) do
    if specialization(temp, result) then
      insert_before(first, temp, result)
      inserted := 'true'
    else
      gcsref := gcs (temp , current, h1 , h2)
      if gcsref <> 'null' then -- there is a gcs
        if found(gcsref, first) then -- found in the list
          result := next(result) -- check next entry
          nogcs := 'false'
        else
          error('you need to define a function with the
```
signature', name(gcsref),'otherwise ambiguous overloading takes place')
nogcs := 'true'
result := 'null' -- end the checks
exit
end -- if
else
result := next(result) -- check next entry
end -- if
end -- while
if not(nogcs) and result = 'null' and not(inserted) then
result := insert_last(first, temp)
end -- if
end -- if
end -- function enter