IMPROVING SUPPORT FOR GENERIC PROGRAMMING IN C# WITH ASSOCIATED TYPES AND CONSTRAINT PROPAGATION

A Thesis

by

ARAVIND SRINIVASA RAGHAVAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2007

Major Subject: Computer Science

IMPROVING SUPPORT FOR GENERIC PROGRAMMING IN C# WITH ASSOCIATED TYPES AND CONSTRAINT PROPAGATION

A Thesis

by

ARAVIND SRINIVASA RAGHAVAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Jaakko Järvi
Committee Members,	Bjarne Stroustrup
	Marian Eriksson
Head of Department,	Valerie Taylor

May 2007

Major Subject: Computer Science

ABSTRACT

Improving Support for Generic Programming in C# with Associated Types and Constraint Propagation. (May 2007)

Aravind Srinivasa Raghavan, B.E, University of Madras

Chair of Advisory Committee: Dr. Jaakko Järvi

Generics has recently been adopted to many mainstream object oriented languages, such as C# and Java. As a particular design choice, generics in C# and Java use a sub-typing relation to constraint type parameters. Failing to encapsulate type parameters within generic interfaces and inability to encapsulate type constraints as part of an interface definition have been identified as deficiencies in the way this design choice has been implemented in these languages. These deficiencies can lead to verbose and redundant code. In particular, they have been reported to affect the development of highly generic libraries. To address these issues, extending object oriented interfaces and sub-typing with *associated types* and *constraint propagation* has been proposed and studied in an idealized small-scale formal setting. This thesis builds on this previous work and provides a design and implementation of the extensions in full C#. We also present a proof of soundness of the Featherweight Generic Java (FGJ) formalism extended with interfaces. This property was assumed in a proof of type safety of associated types and constraint propagation, but no proof for the property was provided.

To Mom, Dad, Manu, Nithya and Aadhya

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Jaakko Järvi, for his guidance, patience, and encouragement throughout the course of my thesis work. I would also like to thank him for supporting me as a graduate research assistant for pursuing my thesis. It is his vision that shaped this thesis to its final form. I would also like to thank Dr. Bjarne Stroustrup for his useful comments and suggestions.

I take this opportunity to thank the Department of Forest Science and especially Dr. Marian Eriksson for supporting me as a graduate teaching assistant and for agreeing to be on my Master thesis committee.

I would like to express my gratitude to Xiaolong Tang for many useful technical discussions that we shared. My thanks to Sankara Muthukrishnan, Chidambareswaran Raman, Vijay Rajaram, and Shamanth Kuchangi for their support and encouragement.

Lastly, I would like to sincerely thank my family: Mom, Dad, Manu and Nithya for their constant moral support throughout the course of my graduate study and for their enormous trust in me. I thank them for their understanding and patience. If not for them, I would not be writing these final words in my thesis.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Recently many mainstream programming languages have been extended with support for generics. For example, C# included generics in C# 2.0 [1]. C# generics include support for parameterized classes, interfaces and methods. Compared to C# 1.0, these features provide stronger static type checking, require fewer explicit conversions between data types, and reduce the need for boxing operations and run-time type checks.

Generics provides a facility to create types that have *type parameters*, so that concrete types can be constructed by substituting each type parameter by a *type argument* during instantiation. In order to guarantee that the requirements imposed by a generic component on its type arguments are satisfied, C# permits constraints to be supplied for type parameters. These are expressed using *where* clauses in generic classes, interfaces and methods. Constraining type parameters allows generic definitions to be type checked separately from their use.

Although generics provides many benefits, certain shortcomings have been identified in the way in which C# generics constraints its type parameters [2]. These problems can be summarized as follows:

- Type parameters are not encapsulated within interfaces and thus every reference to an interface has to include all the type parameters explicitly.
- Inheriting a generic type does not mean inheriting constraints on its type parameters.

This thesis follows the style of Science of Computer Programming.

These deficiencies may lead to verbose and redundant code. These problems have been reported to affect the use C# for developing highly generic libraries [2].

Earlier work [3] proposes extending object-oriented interfaces and sub-typing to include *associated types* and *constraint propagation* to address the problems of verbose and redundant code. Associated types can be realized as member types in C# interfaces. Member types resemble member **typedefs** of C++ and also share similarities with type members of ML signatures and *virtual types* [4]. Constraint propagation refers to a language mechanism that gathers type parameter constraints that are implied by the use of type parameters of a generic component as arguments to other generic types, and makes those constraints implicitly part of the constraints of the generic component.

The work in [3] studied associated types and constraint propagation formally as an extension to Featherweight Generic Java(FGJ) [5]. In this study, a translation from an object oriented language with associated types and constraint propagation to a standard object oriented language with generics, but no member types or constraint propagation, was described. Also, the type safety of the extensions was established — apart from a gap filled in this thesis — in the idealized setting of FGJ.

This thesis builds on the work described in [3], and describes the design and implementation of associated types and constraint propagation in full C#. The design is based on the translation described in [3]. The formal model developed to prove the soundness of the extension assumes that FGJ extended with interfaces and multiple interface inheritance — we denote this formalism FGJ+I — is type safe. This thesis proves that FGJ+I is type safe and thus fills a gap left out in the formalization for the proposed extensions. The main contributions of this thesis can be summarized as

- Design and implement associated types and constraint propagation in full C#.

- Formally prove the type safety of FGJ+I.

The thesis is structured as follows: Chapter II describes the paradigm of generic programming and discusses Featherweight Generic Java and the .NET Mono framework [6], which we used in our implementation. Chapter III details on how generic programming can be realized in C# and motivates the need for associated types and constraint propagation. Chapter IV describes the C# syntax and semantics we propose for associated types and constraint propagation. Chapter V discusses the framework and algorithms required to implement associated types and constraint propagation in the *gmcs* C# compiler [6]. Chapter VI discusses the FGJ+I formalism and provides a proof of its type safety. Chapter VII surveys the related work and Chapter VIII concludes the thesis and outlines future work.

CHAPTER II

BACKGROUND

This chapter explains relevant background for this thesis. We first discuss about generic programming paradigm and terminologies in generic programming that we use throughout the thesis. We then present a brief overview of Featherweight Generic Java (FGJ) formalism and finally discuss about the Mono .NET framework which was used to implement associated types and constraint propagation in C#.

A. Generic programming

The language features discussed in this thesis are motivated by improving support for generic programming. Generic programming is a programming paradigm aimed at developing efficient, reusable libraries of algorithms. According to Jazayeri et.al., [7]

> Generic programming is a sub-discipline of computer science that deals with finding abstract representation of efficient algorithms, data structures, and other software concepts, and with their systematic organization.

Generic programming works on the principle that software can be decomposed into components which make only minimal assumptions about other components, allowing maximum flexibility in composition. Implementing software components in terms of abstract properties of types, rather than particular types, results in *generic components* whose single generic implementation can cover many concrete implementations. The C++ Standard Template Library (STL) [8] is the first extensive instance of generic programming in wide use. The Boost Graph Library (BGL) [9] and the

Matrix template library (MTL) [10] are other examples and rich sources of generic programming idioms and techniques.

Central notion of generic programming is *concept*. In the terminology of generic programming, a *concept* is the formalization of an abstraction as a set of syntactic and semantic requirements on a type (or on several types) [11]. The *Forward Iterator* or *Container* concepts of STL serve as examples. Concepts can be thought of as a contract between the types instantiating a generic component and types expected by the component. A type that satisfies the requirements of a concept is said to be a *model* of the concept. A generic component specified in terms of concepts can be instantiated with any types that model those concepts. Concepts thus provide a concise means to describe the interfaces of generic components.

In the context of C++, a set of conventions has been developed to describe concepts. Traditionally, a concept consists of four requirements: function signatures, associated types, semantic constraints, and complexity guarantees. *Function signatures* specify the operations that must be implemented for the modeling types. *Associated types* are auxiliary types required by the concept and must be defined by the types that model the concept. As an example, *vertex* and *edge* types could be associated types of a generic *Graph* type. Concepts may also include constraints on associated types. *Complexity guarantees* specify limits for resource consumption (eg., execution time, memory) for function signatures.

A concept may incorporate the requirements of another concept, in which case the first concept is said to *refine* the second. For example, we could define the concept *Bidirectional Iterator* that contains all the requirements of the *Forward Iterator* concept and add the requirement to allow moving backward in a sequence.

B. Comparative study

Languages such as C++, Eiffel, ML have supported various forms of genericity for decades or more. Recently, other mainstream programming languages, Java and C#, have introduced generics as new language extensions. To evaluate the effectiveness of the different approaches to genericity, the current generic programming capabilities of eight different languages are compared in [2]. The study implements a subset of the BGL in eight different languages and identifies eight different language features which must be supported by the language to support generic programming: *support for multi-type concepts, multiple constraints on type parameters, convenient associated type access, constraints on associated types, retroactive modeling, type aliases, separate compilation of algorithms, and implicit argument type deduction for generic algorithms.* According to the study, partial or no support for the above language features may lead to verbose code, poor maintainability, and awkward designs. In this work we focus on particular findings of this study — insufficient support for associated types and constraints on associated types in C# and Java — and demonstrate how to improve the situation. These features are described in Chapter IV.

The study [2] showed how in C#, concepts can be realized as parameterized interfaces. Associated types of a concept are represented in an indirect manner as type parameters of interfaces and constraints concerning associated types as type parameter constraints of interfaces. Several problems, however, with this chosen representation for generic programming constructs were identified. First, type parameters are not encapsulated within interfaces; every reference to an interface must explicitly list all of its type parameters. Due to this effect, the number of type parameters was more than doubled in several generic algorithms of the BGL. Second, in C#, Java and Eiffel, inheriting a generic interface or class does not inherit the constraints. As a re-

sult, constraints on type parameters must be repeated in derived classes even though they could be deduced by the uses of those type parameters in types of base classes or other constraints. The consequence is more verbosity in generic interfaces, classes, and methods. Chapter III explains these problems — the lack of language support for direct representation of associated types and failure to encapsulate the constraints on associated types as a part of an interface (*concept*) — in detail. The identification of the above problems prompted the work described in [3] and subsequently this thesis.

C. Featherweight Generic Java

Formal modeling allows the study of complex artifacts like programming languages, describing some aspect of a design precisely, to state and prove its properties. Commonly formal models cover lightweight versions of programming languages, dropping out complex language features, to enable rigorous arguments about key properties of the language, such as type safety. One of such formalism is the Featherweight Java (FJ) [5]. It is a minimal core calculus for modeling the essential properties of Java with respect to type checking. It bears a similar relation to Java as *lambda-calculus* [12] does to languages such as ML and Haskell. FJ's main application is modeling extensions of Java.

FJ provides classes, methods, fields, inheritance and dynamic typecasts with semantics closely following Java's. In fact, every FJ program is an executable Java program. FJ has five forms of expressions: object constructor, method invocation, field access, casting and a variable reference. A FJ program consists of a set of class definitions and an expression to be evaluated. Figure 1 shows a typical class definition in FJ.

```
class A extends Object {
        A() { super() ; }
        A clone() { return new A() ; }
}
class B extends Object {
        Object b ;
        B(Object b) { super(); this.b = b ; }
        B clone (Object data) { return new B(data) ; }
}
```

Fig. 1. Class definition in Featherweight Java.

To keep the formalism simple, all syntactic sugar is eliminated. For example, the superclass is always included in the class definition, constructors are always written, the receiver *this* for accessing the method or field of the current object must be written explicitly, and so forth. Constructors take one parameter for each field declared in the class and its base classes, and initializes them. The fields of the superclass are initialized by invoking the *super()* constructor. Constructors are the only place where *super* or the assignment operator = appears in a FJ program. Note that the assignment is not destructive, — FJ is a pure functional language. A method body always consists of a single return statement as in the body of *clone* in B's definition in Figure 1.

FJ's semantics are defined via *small-step* operational semantics [13]. There are only three computational rules: one for field access, one for method invocation and one for casts. The normal forms of FJ are object constructor expressions, $new\ N(\overline{w})$, where $\overline{w}$ is a sequence of normal forms. A well-typed FJ program evaluates to a normal form, or an expression containing a valid cast. A proof assuring us of this behavior is detailed in [5]. A computation may get stuck in three different ways in FJ: by attempting to access a field not declared for the class, by attempting to invoke

a method not declared for the class; or by attempting to cast to something other than a superclass of an object's runtime class.

```
class A extends Object {
        A() { super() ; }
        A clone() { return new A() ; }
}
class B<X extends Object> extends Object {
        X fst ;
        B(X fst) {
                super(); this.fst = fst ;
        }
        <Y extends Object> B<Y> clone (Y data) {
                return new B<Y>(data) ;
        }
}
```

Fig. 2. Class definition in Featherweight Generic Java.

Of particular interest to us is the FGJ formalism that extends FJ with generic classes and methods. Figure 2 shows a typical class definition in FGJ, illustrating both parameterized classes and methods: X is a type parameter of the class B, and Y is a type parameter of the generic method *clone*. Each type parameter has a *bound*, the uppermost bound being *Object* — here X and Y are bound by *Object*. Type bounds can be type expressions involving type variables. Plain type variables, however, cannot be used as type bounds. Note that type parameter inference, an important aspect of Java generics, is not modeled in FGJ. Due to this, FGJ is not a proper subset of Java generics; it is assumed to be an *intermediate language* — the form that would result after type parameters are inferred.

This thesis uses the FGJ formalism for two purposes. First, it is the starting point of the formal framework describing the extensions for C# we propose. Even though, FGJ models Java and not C#, FGJ is close enough to C# to be a useful

formalism. Second, we present a proof of type safety of the Featherweight Generic Java (FGJ) formalism extended with Interfaces. This property was assumed in a proof of type safety of associated types and constraint propagation extensions for C# [3] but no proof for it has been provided earlier. Chapter VI details the syntax, typing rules and proof of type safety of the FGJ+I formalism.

D. .NET framework and Mono

Microsoft .NET framework (.NET) [14] is a run-time environment that manages the execution of programs. The .NET introduces a common type system which enables inter-operability across various languages supported by .NET. The .NET framework also provides a substantial *class library* catering for common programming needs such as file reading and writing, database interaction, reflection etc.

The important component of the .NET framework is the *Common Language Infrastructure* (CLI), which provides a language independent platform for application development. The Microsoft implementation of the CLI specification is called *Common Language Runtime* (CLR). Programs written in programming languages supported by the .NET framework are compiled down to *Common Intermediate Language* (CIL) and a *just in time* (JIT) compiler compiles the intermediate language to architecture specific native code.

Mono [6] is an open development initiative to develop an open source implementation of CLI. The component of Mono relevant to this thesis is the free implementation of the C# compiler that supports the C# 2.0 specification which includes generics. We used the Mono C# compiler for implementing the associated types and constraint propagation extensions to C#. The *Shared Source Common Language Infrastructure*

(SSCLI) [14] and DotGNU [15] are other open or shared source implementations of C# compilers.

CHAPTER III

GENERIC PROGRAMMING IN C#

The C++ community has adopted a particular documenting style for describing concepts and their requirements on type parameters of a generic component. The SGI STL [16] and the BGL are well known examples of libraries following this style. We first demonstrate these established notations for describing concepts and the conventions used for describing constraints on type parameters of generic components. For this purpose, we present a small hierarchy of concepts from the domain of graphs taken from BGL [9]. We then show the corresponding definitions of these concept descriptions with language features of C# and describe the problems arising from lack of support for associated types and constraint propagation.

A. A simple hierarchy of concepts from the domain of graphs

GraphEdge concept. Type Edge is a model of *GraphEdge* if the following expressions are valid. Object e is of type Edge.

Expression	**Return Type or Description**
Edge::vertex_type	Associated vertex type
source(e)	Edge::vertex_type
target(e)	Edge::vertex_type

Iterator concept. Type Iter is a model of *Iterator* if the following expressions are valid. Object i is of type Iter.

Expression	**Return Type or Description**
Iter::value_type	Associated value type
next(i)	Iter
at_end(i)	bool
current(i)	Iter::value_type

Fig. 3. C++ concept descriptions for *GraphEdge* and *Iterator* concepts. Only syntactic requirements are shown here.

IncidenceGraph concept. Type Graph is a model of *IncidenceGraph* if the following expressions are valid. Object g is of type Graph and v is of type Graph::vertex_type.

Expression	Return Type or Description
Graph::vertex_type	Associated vertex type
Graph::edge_type	Associated edge type
Graph::out_edge_iterator	Associated iterator type
edge_type models GRAPHEDGE	
out_edge_iterator models ITERATOR	
edge_type::vertex_type == vertex_type	
out_edge_iterator::value_type == edge_type	
out_edges(v,g)	out_edge_iterator
out_degree(v,g)	int

Fig. 4. C++ concept descriptions for *IncidenceGraph* concept. Only syntactic requirements are shown here.

The *GraphEdge* concept in Figure 3 states the requirements for a type to serve as the edge type in the graph data type. The concept requires the existence of an associated type *vertex_type* and the functions *source* and *target*, which return, respectively, the source and target of an edge. The *Iterator* concept in Figure 3 requires an associated type *value_type* and functions for traversing a sequence of values. The *IncidenceGraph* concept in Figure 4 has *vertex_type*, *edge_type*, *out_edge_iterator* as associated types, and requires functions to find the out-degree and out-going edges of a vertex. Additionally, it places the requirement that *edge_type* must model the *GraphEdge* concept, and that *out_edge_iterator* must model the *Iterator* concept. Furthermore, this concept has two same-type constraints to ensure that *out_edge_iterator* iterates over edges of correct type and that *vertex_type* coincides with the *edge_type*'s associated type *vertex_type*.

The *BidirectionalGraph* concept in Figure 5 refines the *IncidenceGraph* concept and adds the ability to find the in-degree and the incoming edges of a vertex. The refinement relation means that all the associated types, requirements on associated

BidirectionalGraph concept refines *IncidenceGraph*. Type Graph is a model of *BidirectionalGraph* if the requirements below are satisfied along with the requirements imposed by *IncidenceGraph*. Object g is of type Graph and v is of type Graph::vertex_type

Expression	**Return Type or Description**
Graph::in_edge_iterator	Associated iterator type
in_edge_iterator models ITERATOR in_edge_iterator::value_type == edge_type	
in_edges(v,g)	in_edge_iterator
in_degree(v,g)	int

Fig. 5. C++ concept descriptions for *BidirectionalGraph* concept. Only syntactic requirements are shown here.

types, and function signatures of the *IncidenceGraph* concept are also part of the *BidirectionalGraph* concept. Additionally, the *BidirectionalGraph* concept requires the existence of the associated type *in_edge_iterator*, requires that *in_edge_iterator* model *Iterator* concept, and establishes a same-type constraint that ensures *in_edge_iterator* iterates over edges of correct type.

It should be noted that, the above concept descriptions are mere documentation and not analyzed by C++ compilers. Current C++ compilers do not support constraints on type parameters.

B. C# concept descriptions

Interfaces in C# can partially capture the requirements of concepts. Figure 6 repeat the *GraphEdge*, *IncidenceGraph*, and *BidirectionalGraph* concepts, now written using C# interfaces. The models relation between types and concepts is represented as classes implementing interfaces. Function signature requirements of concepts are naturally expressed as method signatures. Even though C# does not support the direct representation of associated types, type parameters of interfaces can be used

```
interface GraphEdge<Vertex> {
  Vertex source();
  Vertex target();
}
interface IncidenceGraph<Vertex, Edge, OutEdgeIterator>
    where Edge : GraphEdge<Vertex>
    where OutEdgeIterator : IEnumerable<Edge> {
  OutEdgeIterator out_edges(Vertex v);
  int out_degree(Vertex v);
}
interface BidirectionalGraph <Vertex, Edge, OutEdgeIterator, InEdgeIterator> :
                IncidenceGraph <Vertex, Edge, OutEdgeIterator>
    where Edge : GraphEdge <Vertex>
    where OutEdgeIterator : IEnumerable <Edge>
    where InEdgeIterator : IEnumerable <Edge> {
  InEdgeIterator in_edges (Vertex v) ;
  int in_degree (Vertex v) ;
}
```

Fig. 6. *GraphEdge*, *IncidenceGraph* and *BidirectionalGraph* concepts in C#.

to represent them. The requirement that associated types of concepts model other concepts can be expressed using *where* clauses requiring that a type parameter is a subtype of another interface or class. Concept refinement, for example between the *IncidenceGraph* concept and the *BidirectionalGraph* concept, is represented using inheritance between interfaces as shown in Figure 6. Same type constraints are implicit: for example the constraints *out_edge_iterator::value_type == edge_type* and *in_edge_iterator::value_type == edge_type* are expressed by using the same type parameter *Edge* in the constraints of *OutEdgeIterator* and *InEdgeIterator* in Figure 6. Similarly, the use of *Vertex* as type argument of *GraphEdge* in Figure 6 establishes the other same-type constraint *edge_type::vertex_type == vertex_type* of *IncidenceGraph* concept.

```
bool IsSameInOutDegree <G,GVertex, GEdge, GOutEdgeIter,GInEdgeIter> (G g, GVertex v)
        where G : BidirectionalGraph <GVertex, GEdge, GOutEdgeIter, GInEdgeIter>
        where GEdge : GraphEdge <GVertex>
        where GOutEdgeIter : IEnumerable <GEdge>
        where GInEdgeIter : IEnumerable <GEdge>
{
   return g.out_edges (v) == g.in_edges (v) ;
}
```

Fig. 7. Example generic function in C#.

C. Problems in current generic C#

The main problem with representing associated types as type parameters is that type parameters are not properly encapsulated within interfaces. Every reference to an interface, either in a refinement relation or in a type parameter constraint, has to specify all the interface's type parameters explicitly. The *IsSameInOutDegree* in Figure 7 is a generic algorithm, represented as a generic method in C#, that determines whether the "in"-degree and "out"-degree of a vertex in a graph are equal. In this generic method, note the verbose constraint on type parameter *G*. The reference to *BidirectionalGraph* interface must qualify all its type parameters and becomes *BidirectionalGraph<GVertex, GEdge, GOutEdgeIter, GInEdgeIter>*. This is regardless of whether the type parameters are otherwise needed or not. This verbosity occurs within concept refinement, as shown in the declaration of *BidirectionalGraph* concept in Figure 6, where the reference to *IncidenceGraph* interface is *IncidenceGraph<Vertex, Edge, OutEdgeIterator>* instead of *IncidenceGraph*.

The lack of encapsulation of type parameters becomes clear by observing that only two of the type parameters of the generic method *IsSameInOutDegree*, *G* and *GVertex*, are used as types of method parameters. However, all the associated types of the *BidirectionalGraph* concept, (*edge_type, out_edge_iterator,* and *in_edge_iterator*)

must nevertheless be declared as type parameters (*GEdge, GOutEdgeIter, InEdgeIter*) for the generic method to type check. In *IsSameInOutDegree*, we never use *GEdge*, *GInEdgeIter* or *GOutEdgeIter* type parameters, so those type parameters are unnecessary and could be eliminated. This problem can be solved by providing direct support for associated type as *member types* within interfaces. The syntax and semantics of such member types are described in Chapter IV and the implementation framework is detailed in Chapter V.

Another related problem is that interfaces fail to encapsulate the constraints on type parameters. Consider the type parameter constraints in the *where* clause of *IsSameInOutDegree* function in Figure 7. The constraints on *GEdge, GOutEdgeIter,* and *GInEdgeIter* are repetition of the constraint list of the *BidirectionalGraph* interface. This repetition seems unnecessary because no type can be bound to G unless it inherits from *BidirectionalGraph* interface. This in turn requires that the types *GVertex, GEdge, GOutEdgeIter,* and *GInEdgeIter* bound, respectively, to type parameters *Vertex*, *Edge*, *OutEdgeIterator*, and *InEdgeIterator*, satisfy the constraints imposed on these type parameters in the *BidirectionalGraph* interface. Thus based on the constraint on G, the type checker could safely assume that type parameters *GVertex, GEdge, GOutEdgeIter* and *GInEdgeIter* also satisfy the constraints of the *BidirectionalGraph* interface, avoiding thus repeating the constraints in the *IsSameInOutDegree* function. The same problem occurs with concept refinement. For example, in Figure 6, when *BidirectionalGraph* concept refines the *IncidenceGraph* concept, the constraints on type parameters *Edge* and *OutEdgeIterator* from *IncidenceGraph* are repeated in the *BidirectionalGraph* interface.

The problem of repeated constraints can be solved by making the type checker a little more *"intelligent"*, by *propagating* constraints on a type parameter of a generic component that are implied by the use of that type parameter as arguments to other

generic types, and make those constraints implicitly part of the constraints of the generic component. This language mechanism, *constraint propagation* as we name it, is discussed in Chapter IV and the implementation framework for constraint propagation is detailed in Chapter V.

CHAPTER IV

EXTENSIONS TO C#

A. Associated types in C#

Direct language support for associated types would significantly improve support for generic programming in C#. We claim that this support can be implemented without drastic modifications to the language, by introducing *member types* to interfaces and classes [3]. Member types are declared in interfaces and are essentially placeholders for types. One can place constraints (subtype or same-type) on these members. As an example, Figure 8 shows the graph concepts of Figure 6, now taking advantage of member types.

The syntax of member types is as follows: Member types are declared using the keyword **type** followed by the name of the member type. Constraints on member types can be specified by the syntax *A : B*, as in ***type*** *Edge : GraphEdge*, directly at a member type declaration. Alternatively, the syntax ***require*** *A : B* can be used to specify constraints on member types. Same-type constraints are expressed with the syntax **require** A == B, as in ***require*** *Vertex == Edge::Vertex*. The syntax *T::A*, as in *Edge::Vertex*, is used to access the associated type *A* of some type *T*.

Interfaces with member types, *concept interfaces* as we name them, are not traditional object oriented interfaces. In particular, these interfaces cannot be used as a type of a variable, function parameter, or field, or used as type arguments to a generic function or class.

Concept interfaces can directly represent concepts described in Chapter II. In Figure 8, the *GraphEdge* interface declares the member type *Vertex*. The *Incidence Graph* interface declares two member types *Vertex* and *Edge*, and places constraints

on them. Note that these member types correspond directly to the associated types in Figure 6, subtype constraints correspond to requirements that types model concepts, and the same type constraints have direct equivalents as well. As discussed in Chapter III, concept refinement can be expressed as inheritance between interfaces in C#. The *BidirectionalGraph* interface refines *IncidenceGraph* interface and declares a member type *InEdgeIterator* and places constraints on it. Due to the refinement relation, all member type declarations and constraints on member types of *IncidenceGraph* are part of the *BidirectionalGraph* interface as well.

```
interface GraphEdge {
  type Vertex ;
  Vertex source();
  Vertex target();
}
interface IncidenceGraph {
  type Vertex ;
  type Edge : GraphEdge ;
  type OutEdgeIterator : IEnumerable <Edge> ;
  require Vertex == Edge :: Vertex ;

  OutEdgeIterator out_edges(Vertex v);
  int out_degree(Vertex v);
}
interface BidirectionalGraph : IncidenceGraph {
{
  type InEdgeIterator : IEnumerable <Edge> ;
  InEdgeIterator in_edges (Vertex v) ;
  int in_degree (Vertex v) ;
}
```

Fig. 8. *GraphEdge*, *IncidenceGraph* and *BidirectionalGraph* concepts in C# extended with associated types.

Classes that (*model*) implement (concepts) concept interfaces must bind concrete types to the member types declared in the interface. A class that models a concept in-

terface must thus provide a definition for every member type declared in the interface. Figure 9 shows the *AdjacencyList* class modeling the *BidirectionalGraph* concept and *AdjListEdge* class modeling the *GraphEdge* concept. The member type bindings introduced by a class cannot be modified by any of its derived classes. Redefining or leaving a member type unbound in a class is a type error. The semantics for associated types in C# is described in detail in [3]. The syntax for member type declarations in interfaces and definitions in classes are further discussed in Section V.A.

```
class AdjListEdge : GraphEdge {
        type Vertex = int ;
        . . .
}
class AdacencyjList : Graph {
        type Vertex = int ;
        type Edge = AdjListEdge ;
        type OutEdgeIterator = IEnumerable <AdjListEdge> ;
        type InEdgeIterator = IEnumerable <AdjListEdge> ;
        . . .
}
```

Fig. 9. AdjListEdge models *GraphEdge* concept and AdjacencyList models *BidirectionalGraph* concept.

```
bool IsSameInOutDegree <G> (G g, G::Vertex v)
        where G : BidirectionalGraph
        where G::Edge : GraphEdge <G::Vertex>
        where G::OutEdgeIter : IEnumerable <G::Edge>
        where G::InEdgeIter : IEnumerable <G::Edge>
{
   return g.out_edges (v) == g.in_edges (v) ;
}
```

Fig. 10. Example generic function in C# with associated type support but without constraint propagation.

```
bool IsSameInOutDegree <G,GVertex, GEdge, GOutEdgeIter,GInEdgeIter> (G g, GVertex v)
        where G : BidirectionalGraph <GVertex, GEdge, GOutEdgeIter, GInEdgeIter>
        where GEdge : GraphEdge <GVertex>
        where GOutEdgeIter : IEnumerable <GEdge>
        where GInEdgeIter : IEnumerable <GEdge>
{
   return g.out_edges (v) == g.in_edges (v) ;
}
```

Fig. 11. Example generic function in C#.

The rewrite of *IsSameInOutDegree* function, shown in Figure 10, demonstrates the effect of associated types. Compared to the generic function in Figure 11 repeated from Figure 7 for convenience, the number of type parameters are significantly reduced. Also, the reference to *BidirectionalGraph* interface in the constraint on G does not have to include all type parameters explicitly because member types encapsulate the associated types within the *BidirectionalGraph* interface. Note, however, that the constraints on member types are roughly as verbose as the constraints on the type parameters. The remedy is constraint propagation.

B. Constraint propagation in C#

Constraint propagation refers to a language mechanism that gathers constraints on a type parameter that are implied by the use of that type parameter as arguments to other generic types, and makes those constraints implicitly part of the constraints of the generic component being defined. Constraint propagation can apply when a type parameter is constrained by a generic class or interface, or when a generic class or interface inherits from another generic class or interface.

Consider the process of type checking the body of a generic class or method A. Suppose X is a type parameter of A, and X is used as a type argument in some generic

instantiation *B<..., X ,...>* occurring in one of the constraints of *A*, or as a base class or base interface of *A*. Constraint propagation then means that any constraints the definition of *B* places on *X* in the instantiation of *B<..., X ,...>* can be assumed to be true while type-checking *A*. Note that even though fewer constraints must be written, enough information is retained to preserve separate type checking.

As an example of constraint propagation, consider the function in Figure 11. The type parameter *GEdge* is used as a type argument to the *BidirectionalGraph* interface. The *where* clause of the *BidirectionalGraph* requires its second type parameter *Edge* to be a subtype of *GraphEdge <Vertex>*. Substituting *GEdge* to *Edge* and *GVertex* to *Vertex*, as the instantiation of *BidirectionalGraph* implies, the constraint *GEdge : GraphEdge<GVertex>* can be assumed to hold while type checking the generic method *IsSameInOutDegree*. Translation back to C# can be implemented by propagating the constraints with appropriate substitutions of type arguments to type parameters. In the example shown in Figure 11, constraints for *GOutEdgeIter* and *GInEdgeIter* can be propagated as well.

```
bool IsSameInOutDegree <G,GVertex, GEdge, GOutEdgeIter,GInEdgeIter> (G g, GVertex v)
        where G : BidirectionalGraph <GVertex, GEdge, GOutEdgeIter, GInEdgeIter>
{
   return g.out_edges (v) == g.in_edges (v) ;
}
```

Fig. 12. Example generic function in C# with constraint propagation.

Extending C# with constraint propagation will enable writing the *IsSameInOutDegree* function's constraint set very concisely, as shown in Figure 12. Moreover using member types for associated types avoids declaring the unwanted extra type parameters. With the combination of these two language extensions, we can improve even

further: the *IsSameInOutDegree* function in Figure 13 uses the desired constraints without redundancy.

```
bool IsSameInOutDegree <G> (G g, G::Vertex v)
        where G : BidirectionalGraph
{
   return g.out_edges (v) == g.in_edges (v) ;
}
```

Fig. 13. Generic function in C# with constraint propagation and associated types.

Constraint propagation can also be applied when member types are constrained by a generic class or interface, or when member types are used as type arguments of generic type instantiation. Figures 14 to 17 show various scenarios illustrating constraint propagation. The left columns list programs relying on constraint propagation and the right columns list the corresponding programs after propagating the constraints.

```
interface C { }                                  interface C { }
interface B<X,Y>                    ⟹           interface B<X,Y>
  where X : C                    propagate         where X : C
  where Y : new() { ... }                          where Y : new() { ... }
interface A<X,Y> : B<X,Y> {...}                  interface A<X,Y> : B<X,Y> {...}
                                                   where X : C
                                                   where Y : new() { ... }
```

Fig. 14. Constraint propagation when an interface has generic base types.

Figure 14 is an example of constraint propagation when a generic interface inherits from another generic interface. The constraints on type parameters X and Y in the class A<X, Y> are implicitly propagated from the constraint of B<X, Y>. We can safely omit the constraints in A as A cannot be instantiated if B's instantiation does not succeed. Figure 15 shows example requiring substitution. This involves substituting type parameters with type arguments, and substitution of member types.

```
interface C { }                                  interface C { }
interface B<S,T>                    ⟹           interface B<S,T>
  where S : C                    propagate         where S : C
  where T : new() { ... }                          where T : new() { ... }
interface A<X,Y> : B<X,Y> {...}                  interface A<X,Y> : B<X,Y> {...}
                                                   where X : C
                                                   where Y : new() { ... }
```

Fig. 15. Constraint propagation when an interface has generic base types, and when substitution is required.

Type substitution may be necessary before bringing constraints to a new context. For example, while propagating the constraints of *B<S, T>* to the interface *A<X, Y>*, we have to substitute S with X and T with Y — type parameters S and T are obviously invalid in the context of the interface *A<X, Y>*.

Figure 16 is an example of constraint propagation from a base type, as well as from a constraint on a type parameter and a member type. The propagated constraints for *interface A<X,Y>* include both the constraints from *interface B* and *interface D*. We propagate the constraints from B, because it is a base interface of A, and from D because it appears as a constraint of *A's* type parameter as well as member type. The propagated constraints on member type S and the same type constraint $S == T$ come from B. The constraints *Z : new()* and $Z == Y$ originate from interface D. Note the proper qualification *W::Z, W::Y* for the member types Z and Y, illustrating member type substitution. The propagation of constraints for generic classes and generic methods are similar to generic interfaces.

Note that C# does not allow same-type constraints (or member types), thus the code in the right column is not valid C#. The constraint *X::Z==X::Y* in the method *foo* in Figure 17 is shown here to portray the internal representation of the constraint in the compiler.

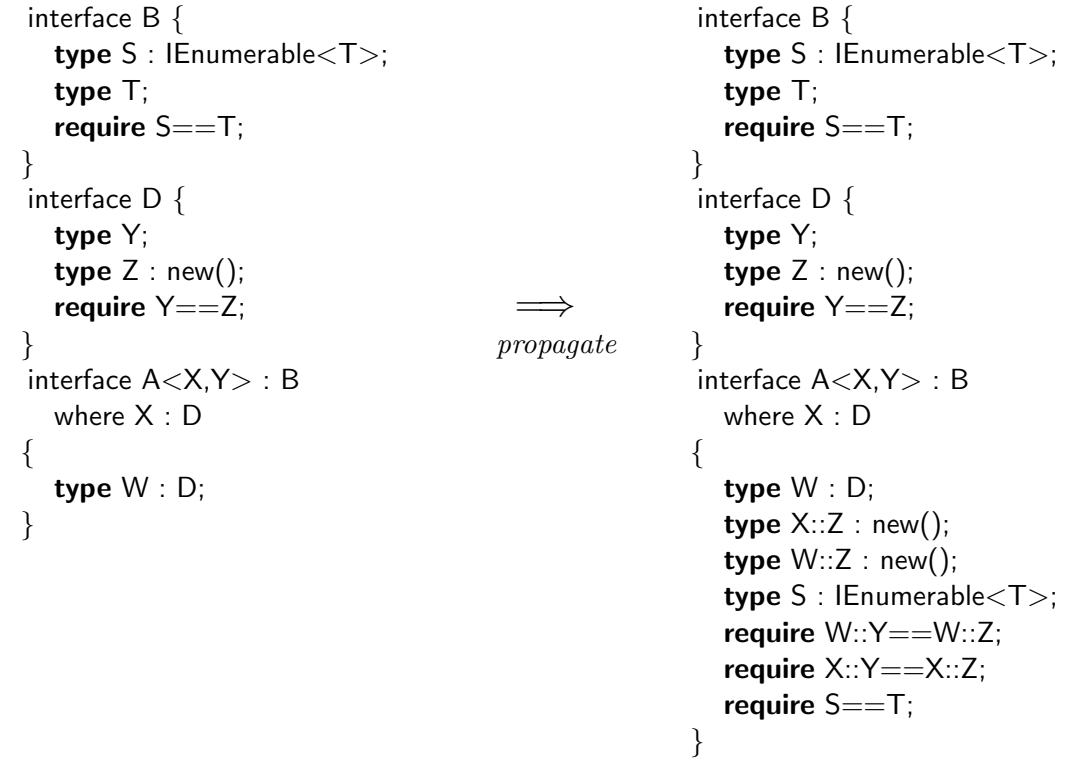

Fig. 16. Constraint propagation when type parameters and member types are constrained by a generic type.

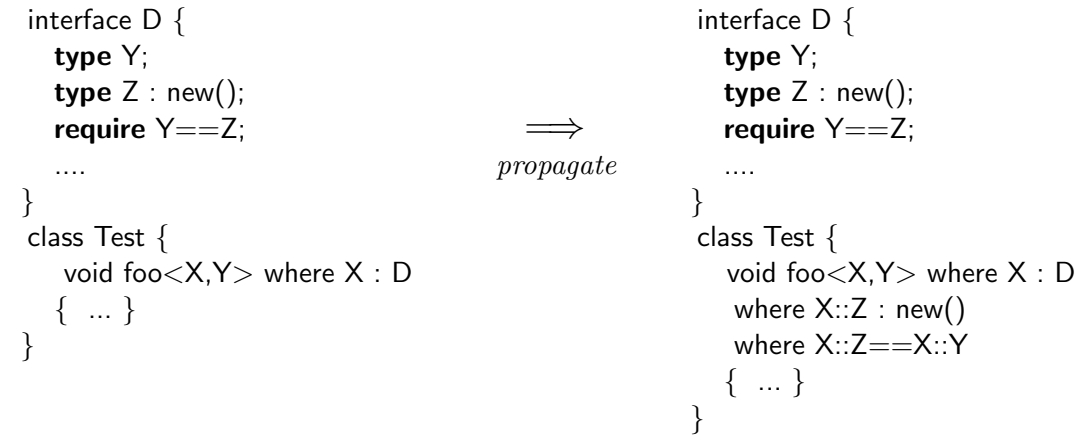

Fig. 17. Constraint propagation for generic methods.

To compile this internal representation to C# without member types or same type constraints, further translation is carried out, explained in detail in Section V.C. Figure 18 gives a glimpse of this with an example of compiling away member types and same type constraints to generate valid C# code. The translation process involves translating the headers and bodies of all interfaces, classes and methods. During the translation process, all member types are converted to type parameters and the constraints on member types are converted to constraints on the corresponding type parameter. For example, in Figure 18, the member types S and U of interface B are converted respectively to type parameters *X0* and *X1*, after the translation process.

```
interface B {                                    interface B<X0,X1>
  type S : IEnumerable<U>;                         where X0 : IEnumerable<X1>
  type T;                          ⟹             {
  type U;                          translate     }
  require S==T;
}
```

Fig. 18. A simple example of translation.

The same-type constraints specified in an interface are compiled away to valid C# using a process of *type canonicalization* which generates the same type parameter for those member types that are involved in a same type constraint. In the above example, member types S and T are mapped to the same type parameter *X0*. The process of canonicalization is explained further in Section V.D. The framework for implementing associated types and constraint propagation in current C# is described in detail in Chapter V.

CHAPTER V

IMPLEMENTATION FRAMEWORK

The following subsections describe how the *mono-gmcs* compiler must be modified to support constraint propagation and associated types.

A. Syntax

Providing support for associated types in the form of *member types* requires parser level modifications to the C# compiler. First, we need to allow declaration of member types and specification of constraints on member types in interfaces. Second, we must allow defining the bindings of member types in the classes that model concept interfaces. Figure 19 shows the modifications in the *interface_member_declaration* and *class_member_declaration* grammar clauses of C# [1] for supporting these additions. The *interface_member_declaration* is modified to support *associated_type_declaration* and *require_clause* grammar rules. The *associated_type_declaration* introduces member types in interfaces in two forms:*type* A, which introduces a member type A without any constraints on it and *type* $A : B$, which introduces A as a member type and places the constraint that A must be a sub-type of B. The *require_clause* grammar rule is used to introduce *sub-type* or *same-type* constraints on type parameters or member types of the form *require* $A{==}B$ or *require* $A : B$ in an interface.

The *class_member_declaration* is modified to support member type definition of the form *type* $A = int$ in classes. The *namespace_or_type_name ASSIGN type* clause, a derivation of *associated_type_declaration* rule introduces member type definitions in a class allowing the syntactic structure *type* $A = ...$ to appear as class members. The grammar clauses for type declaration are also modified to allow usage of associated types as regular types. Currently, C# supports specification of sub-type constraints

```
interface_member_declaration
        :associated_type_declaration
        |require_clause ;
class_member_declaration
        : associated_type_declaration

associated_type_declaration
        : TYPE  associated_type_list SEMICOLON
associated_type_list
        : associated_type
        | associated_type_list COMMA associated_type
associated_type
        : IDENTIFIER
        | IDENTIFIER COLON type_parameter_constraints
        | namespace_or_type_name ASSIGN type

require_clause
        : REQUIRE  associated_type_constraints_list  SEMICOLON
associated_type_constraints_list
        : associated_type_constraint
        | associated_type_constraints_list COMMA associated_type_constraint
associated_type_constraint:
        : namespace_or_type_name OP_EQ type_parameter_constraints
        | namespace_or_type_name COLON type_parameter_constraints
```

Fig. 19. Grammar modifications to support member types.

in *where* clauses: but we allow same-type constraints to be specified in the *where* clause of concept interfaces. So the *where* clause syntax is modified to allow such same-type constraints.

In Figure 19, note that the *associated_type_declartion* rule is used to parse both the declaration of member types in interfaces and their definition in classes. The rationale behind having the same grammar rules for member type declaration and definition is to favor simplicity and to avoid repetition. This, however, allows member type bindings to be part of interfaces, which fails to maintain the semantics of member types introduced in [3]. To avoid this, member type definitions in interfaces will be rejected during the construction of the *Abstract Syntax Tree (AST)*.

B. Framework for constraint propagation

Figure 20 illustrates the design of the constraint propagation framework. The constraint propagation module is invoked after the parser generates the abstract syntax tree (AST) but before the type checker is invoked to check constraints on type parameters.

The constraint propagation framework consists of the following functions, *env-for-body*, *propag-c*, *ConstraintPropagate*, *subConstr*, *cConstr*, *GetConstraints* and *Substitute* functions. The AST is traversed once, and for every interface, class and generic method, *env-for-body* is invoked to collect all the constraints that can be propagated from either the constraints list or from the base classes or interfaces. The *ConstraintPropagate* function, which is invoked by *env-for-body* is the main driver of the constraint propagation process. It uses the *GetConstraints* function to retrieve all directly accessible constraints. The *ConstraintPropagate* function then iterates over this constraint set, and propagates the constraints using *subConstr* and *cConstr*

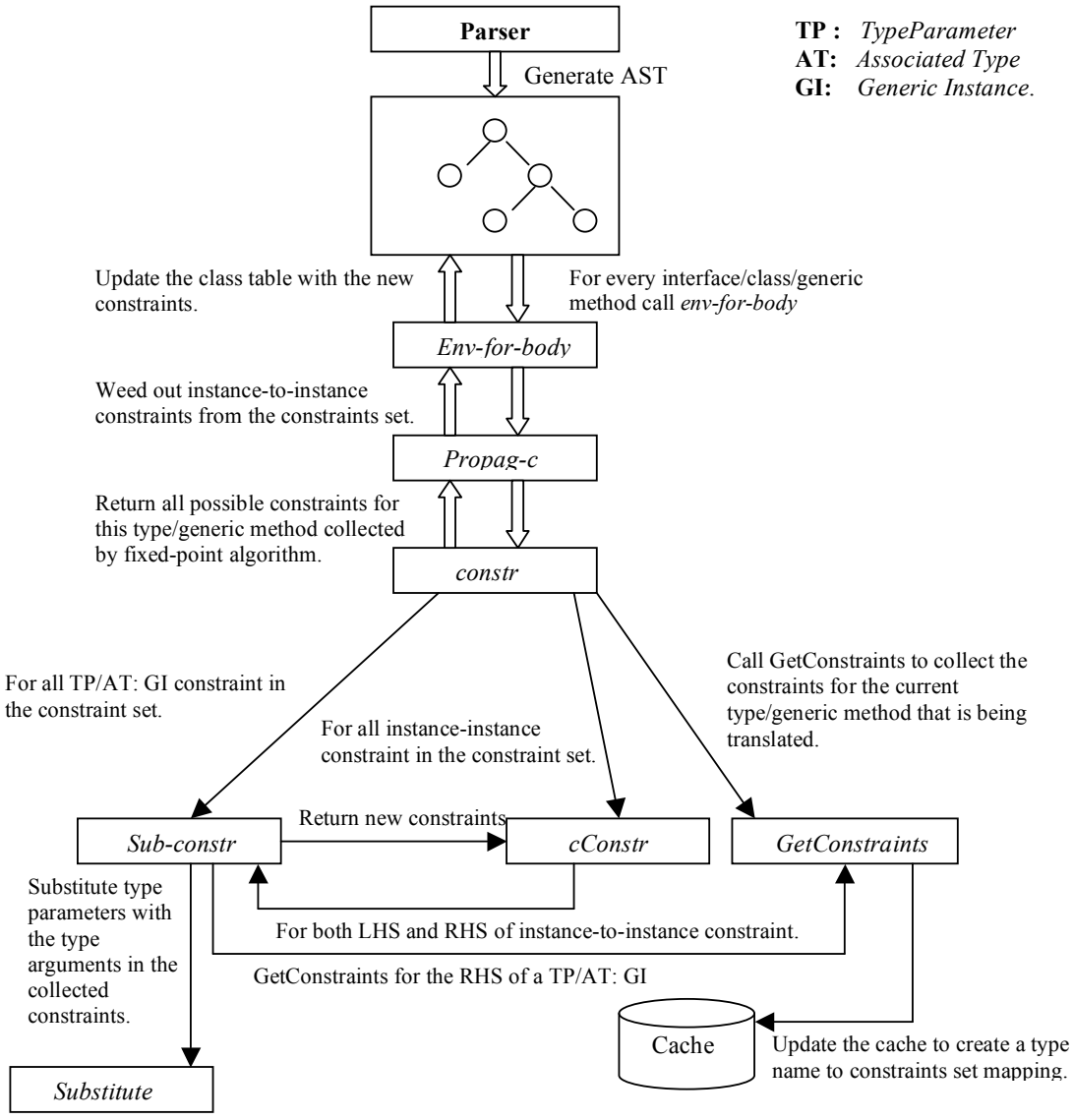

Fig. 20. Constraint propagation framework.

functions. These functions retrieve all directly accessible constraints using *GetConstraints*, and perform the appropriate substitutions for type parameters and member types. These propagated constraints will later be added to the constraints list in *ConstraintPropagate* function, avoiding, however, any duplicate constraints in the set. The constraint propagation is a fixed point computation, in which the algorithm will eventually stop when no more constraints can be added to the environment from the existing constraints. Once the process completes, the class table is updated with the new propagated constraints returned by *env-for-body*

Note that C# allows type parameters to be constrained by *special constraints* such as *class*, *struct* and *new*. These constraints will not be substituted and they remain as such in the new propagated list of constraints.

In the following, we describe the components of the constraint propagation framework. The *env-for-body* given in Algorithm 1 is the entry point to the constraint propagation module. This function builds the environment for type checking classes, interfaces, or generic methods.

Algorithm 1 ***env-for-body***

```
1: procedure ENV-FOR-BODY(sname)
2:     ArrayList ctr ;
3:     ctr = propag-c (sname) ;
4:     return ctr ;
5: end procedure
```

The *propag-c* function given in Algorithm 2 uses *ConstraintPropagate* to collect all the constraints that can be propagated for a generic class or interface. The collected constraint set can also contain *instance-instance* (see below) constraints which are deleted by this function.

Algorithm 2 ***propag-c***

```
1: procedure PROPAG-C(sname)
2:     ArrayList prop_constraints ;
3:     prop_constraints = ConstraintPropagate (sname) ;
4:     for i = 0 to prop_constraints.size() do
5:         Delete all instance-to-instance constraints.
6:     end for
7:     return prop_constraints ;
8: end procedure
```

The *GetConstraints* function given in Algorithm 3 retrieves the set of constraints directly specified on type parameters and member types of a generic class, generic method or generic interface. The process of constraint propagation can lead to constraints of the form *A<X,T> : B<T>*, *instance-instance* — as we name it, when a generic class or interface inherits from another generic class or interface. For example, if a class *P<X,Y>* inherits from classes *B<X>* and *C<Y,X>*, then constraints *P<X,Y> : B<X>* and *P<X,Y> : C<Y,X>* would be constructed and added to the constraint set of *P<X,Y>*. C# does not allow such *instance-instance* constraints, and these extra constraints will be deleted from the constraint set after the constraint propagation is completed. Lines 3–6 construct the instance-instance constraints for every base type and add them to the constraint set. Lines 7–9 collect the constraint on type parameters and lines 10–14 collect the constraints on member types. The constraints on member types are collected only if the type for which *GetConstraints* is invoked for is an interface or a class and not a method.

The process of retrieving constraints for a particular type or method using *GetConstraints* involves a class table lookup to access the constraints and base types. However, this collected set of constraints for a particular type will always be the same, as these constraints are those given in the program text and do not include the propagated constraints. To avoid repeated class table lookups and analyses for the

Algorithm 3 ***Collect constraints for a particular type/generic method***

```
1: procedure GetConstraints(sname)
2:     ArrayList prop_constraints ;
3:     for i = 0 to sname.Bases.size() do
4:         Construct the sname : sname.Bases[i] constraint;
5:         Add the new constraint to prop_constraints;
6:     end for
7:     for i = 0 to sname.TypeParameters.size() do
8:         Add sname.TypeParameters[i].Constraint to prop_constraints ;
9:     end for
10:    if sname is Interface or Class then
11:        for i = 0 to sname.AssocType.size() do
12:            Add sname.AssocType[i].Constraint to prop_constraints ;
13:        end for
14:    end if
15:    Maintain a global cache for prop_constraints ←sname mapping ;
16:    return prop_constraints;
17: end procedure
```

same classes and interfaces: we maintain a cache of the mappings from type names to the constraint sets. The *subConstr* and *ConstraintPropagate* functions use this *Cache* to improve run-time efficiency. Line 15 in Algorithm 3 updates this global cache.

The *ConstraintPropagate* function given in Algorithm 4 relies on *GetConstraints*: line 3 accesses the global cache created by *GetConstraints*. Lines 4–5 invoke *GetConstraints* if the cache does not contain the constraint set for a given class, interface or method. This collected constraint set will contain instance-instance constraints, type parameter constraints and member type constraints; depending on the kind of constraints, we either use *subConstr* or *cConstr* to propagate constraints. Lines 8–9 in Algorithm 4 propagate constraints for both left-hand and right-hand sides of instance-instance constraint. Lines 11–13 propagate constraints if the right side of a type parameter or member type sub-type constraint is a generic class or interface. The propagated constraint from *subConstr* or *cConstr* functions are added to the existing constraint set, if not already in the set.

Algorithm 4 ***Constraint Propagate function for a type or generic method***

```
1: procedure ConstraintPropagate(sname)
2:     ArrayList prop_constraints ;
3:     prop_constraints = cache [sname] ;
4:     if prop_constraints == null then
5:         prop_constraints = GetConstraints (sname) ;
6:     end if
7:     for i = 0 to prop_constraints.size() do
8:         if prop_constraints[i] is Instance-to-Instance constraint then
9:             env = cConstr (prop_constraints[i], prop_constraints[i].Append) ;
10:        else
11:            if prop_constraints[i].Rhs.isGeneric then
12:                env = subConstr (prop_constraints[i].Rhs, prop_constraints[i].Lhs) ;
13:            end if
14:        end if
15:        for j = 0 to env.size() do
16:            if env[j] is not present in prop_constraints then
17:                Add the constraint to prop_constraints ;
18:            end if
19:        end for
20:    end for
21:    return prop_constraints ;
22: end procedure
```

Algorithm 5 ***Propagate for Instance to Instance Constraints.***

```
1: procedure cConstr(ctr, append)
2:     if ctr.Lhs.isGeneric then
3:         ArrayList tempenv = subConstr (ctr.Lhs, append) ;
4:     end if
5:     if ctr.Rhs.isGeneric then
6:         ArrayList env = subConstr (ctr.Rhs, append) ;
7:     end if
8:     for i = 0 to env.size() do
9:         if (!tempenv.present (env[i])) then
10:            Add env[i] to tempenv ;
11:        end if
12:    end for
13:    return tempenv ;
14: end procedure
```

The *cConstr* function given in Algorithm 5 propagates constraints for instance-instance constraints. This function relies on *subConstr* to propagate constraints for left-hand and right-hand side of the constraints. Lines 2–3 and 5–6 propagate constraints for the left-hand side and right-hand side of the constraints, provided they are generic types. The propagated constraints are merged after checking for duplicated constraints and returned to *constr* function.

Algorithm 6 ***Collect constraint with substitution***

```
1: procedure subConstr(sname, append)
2:     ArrayList prop_constraints ;
3:     prop_constraints = cache[sname] ;
4:     if prop_constraints == null then
5:         prop_constraints = GetConstraints (sname) ;
6:     end if
7:     for i = 0 to prop_constraints.size() do
8:         if prop_constraints[i].Lhs is Instance then
9:             substitute (prop_constraints[i].Lhs, append)
10:            prop_constraints[i].Append = append;
11:        else if isTypeParameter (sname, prop_constraints[i].Lhs) then
12:            Substitute type parameter with the type argument from sname.
13:        else
14:            Append append to prop_constraints[i].Lhs
15:        end if
16:        if prop_constraints[i].Rhs is Instance then
17:            substitute (prop_constraints[i].Rhs, append)
18:        else if  isTypeParameter (sname, prop_constraints[i].Rhs)  then
19:            Substitute type parameter with the type argument from sname.
20:        else
21:            Append append to prop_constraints[i].Rhs
22:        end if
23:    end for
24:    Delete all the constraints that has built-in types as Left hand side of the constraints.
25:    return prop_constraints;
26: end procedure
```

The *subConstr* function given in Algorithm 6 retrieves the set of constraints for a generic class, interface or method with proper substitution for the type parameters

and member types. Line 3 accesses the global cache created by *GetConstraints*. Lines 4–5 invoke *GetConstraints* if the cache does not contain the constraint set for a given class or interface. Lines 7–24 traverse this collected constraint set and performs the appropriate type substitution, type parameters are substituted for the corresponding type arguments and the member types are prepended with the method parameter, *append*, of the *subConstr* function. The *append* parameter denotes the left-hand side of a type parameter or member type subtype constraint. This type substitution is valid for any occurrences of type parameters or member types as left-hand side of a constraint or as type arguments in a generic type instantiation. A *substitute* function is used for ensuring proper type substitution for type parameters and member types that occur as type argument in a generic type instantiation. Lines 8–15 and lines 16–24 perform this type substitution for the left-hand and right-hand side of the constraint.

C. Translation of associated types

Our approach to support associated types is to translate associated types to type parameters. The process is a bit more detailed, but as the first approximation, each associated type declaration in an interface is translated into a new type parameter of that interface. The subtype constraints on associated types are translated and added to the constraint set of the corresponding type parameter. The same type constraints in the interface are deleted, but the translation ensures that only one type parameter is created for any two associated types equated with same type constraint.

Concept interfaces can occur as base interfaces of classes or interfaces and in type parameter constraints. Any instance of a concept interface requires an extra type argument for each associated type, so that the instance matches the translated

definition of the interface. Also, references to associated types in class, interface, or method bodies must be translated to references to the corresponding new type parameter. The definition of associated types in classes that implement interfaces will be translated to type arguments to those concept interfaces.

The translation occurs after the parser generates the *abstract syntax tree (AST)*, after constraint propagation, but before the type checker is executed. The associated type translation involves two steps:

- Translating the interface, class and method headers
- Translating the interface, class and method bodies

The class, interface, and method headers include the name of the class, interface, or method, the type parameter list and the constraints list specified in **where** clauses. Translation of a header, for example for interfaces, thus involves collecting constraints using constraint propagation, translating those constraints and generating new type parameters for all the type parameters and associated types declared directly in the interface or accessible through the constraints and base interfaces. The header translation for interfaces as explained here is also applicable to classes and generic methods.

The translation of interface, class or method bodies involves translating all *type-expressions* in these bodies: type parameters, associated types and class or interface instances are translated to reflect the translation done to signatures.

Driver for the translation process: The *translate* function given in Algorithm 7 is the main driver for the associated type translation module. This function is invoked for every generic interface, class, or method given in the program. The main functionalities of the *translate* algorithm are

Algorithm 7 ***Interface, class and method translate.***

```
1: procedure TRANSLATE(sname)
2:     ArrayList env, Eqenv ;
3:     env = env-for-body (sname);
4:     for i = 0 to env.Size() do
5:         if (!env[i].IsSubType()) then
6:             Eqenv.Add (env[i]) ;
7:         end if
8:     end for
9:     Eqenv = getTransitiveClosure (env, Eqenv) ;
10:    ArrayList full_tparams = full-tparams (sname) ;
11:    full_tparams.Sort() ;
12:    newTypeParameter = translatedConstrainableType (sname, full_tparams) ;
13:    if sname is class or interface then
14:        translatedParents = translateParents (sname, env, Eqenv) ;
15:    end if
16:    translatedConstraints = translateEnv (sname, env, Eqenv) ;
17:    Translate body of interface, class and method translating every occurrences of types.
18: end procedure
```

1. *Propagating constraints* from generic instances that occur as base types or as a constraint for a type parameter or associated type. The *env-for-body* function explained in Section B is used to propagate constraints.

2. *Generating new type parameters* for the translated interface, class or method. The *full-tparams* function explained in Section D collects the set of all associated types and type parameters used in the definition of an interface, class or generic method, recursing to its constraints and ancestors. The set of types returned by *full-tparams* function are then ordered by the *Sort* function, to maintain the same ordering of type parameters and associated types in translation of the definition of a type as well as during the translation of any instance of the same type. The translated type or method then needs a type parameter for every element in this set. The *translatedConstrainableType* function given in

Algorithm 13 takes the ordered set of types returned by *full-tparams* function and generates a valid list of new type parameters.

3. *Translation* of *base types*. The *translateParents* function given in Algorithm 15 translates the base types of interfaces and classes.

4. *Translation* of the *propagated constraints*. The *translateEnv* function given in Algorithm 16 translates the set of propagated constraints obtained through *env-for-body*.

5. *Translating* the *bodies of the interface, class or generic method*. The translation of interface, class or method bodies involve translating every occurrence of *type expressions* using the *translateType* function which is explained further in Section D.

6. Finally after translating the base types, constraints and the body of interface, class or method, the class table is *updated* with the new translated construct.

7. The *translate* algorithm also relies on some auxiliary functions listed below.

   (a) Helper functions to collect the declarations of member types in interfaces and definitions of member types in classes.

   - *assoc-decl*, *assoc-def*.

   (b) Algorithm to generate the transitive closure of the same-type constraints.

   - *getTransitiveClosure*.

   (c) Type Canonicalization

   - *canonicalize_weak*, *canonicalize*.

D. Framework for associated type translation

This section provides detailed algorithms for the main components of the associated type translation framework.

Helper functions to collect member types: The *assoc-decl* and *assoc-def* are helper functions used by other algorithms in the framework to collect the names of member types. The *assoc-decl* function collects the names of all member types declared in a given interface or in its ancestors. The *assoc-def* function collects the names of member types defined and bound to types in a given class or in its base class.

The translation process relies on *generating transitive closure* and *type canonicalization* process to reduce the number of type parameters to be generated. We generate the transitive closure of same-type constraints to explore all possible same-type relations from those specified by the user in the program text.

Generating the transitive closure: The *getTransitiveClosure* function generates the symmetric, transitive, congruence closure of the set of same-type constraints. This function takes a set of same-type constraints as the method parameter *Eqenv*. The *getSymmetry* helper function returns the symmetry of a same-type constraint. For example, it would return $B == A$ for the same type constraint $A == B$. First, for the constraints in *Eqenv*, the symmetric relations are determined and added to *Eqenv*. The *getAssocCongruence* function is used to collect the same type relations between member types that can be accessed through the types equated in a same type constraint. For example, if the environment contains the same type constraint $A==B$, then *getAssocCongruence* would construct same-type relations between all member types that can be accessed from A and B. The set of same-type constraints returned

Algorithm 8 ***getTransitiveClosure***

```
1: procedure getTransitiveClosure(env, Eqenv)
2:     for i = 0 to Eqenv.Size() do
3:         Eqenv.Add(getSymmetry(Eqenv[i]) ;
4:     end for
5:     for i = 0 to Eqenv.Size() do
6:         ArrayList assocCongruence = getAssocCongruence (Eqenv[i], env) ;
7:         for j = 0 to assocCongruence.Size() do
8:             Eqenv.Add (assocCongruence[j]) ;
9:             Eqenv.Add(getSymmetry(assocCongruence[j])
10:        end for
11:    end for
12:    for i = 0 to Eqenv.Size() do
13:        ArrayList transConstraints = getTransitivity(Eqenv[i], Eqenv) ;
14:        for j = 0 to transConstraints.Size() do
15:            Eqenv.Add (transConstraints[j]) ;
16:            ArrayList assocCong = getAssocCongruence(transConstraints[j] ,env) ;
17:            for k = 0 to assocCong.Size() do
18:                transConstraints.Add(assocCong[k]) ;
19:            end for
20:            Eqenv.Add(getSymmetry(transConstraints[j]) ;
21:        end for
22:    end for
23:    return Eqenv ;
24: end procedure
```

by *getAssocCongruence* are then populated into *Eqenv*. The symmetric relations are also determined for every new constraint obtained from *getAssocCongruence* function and are added to *Eqenv*.

The *getTransitivity* helper function is used to collect the set of constraints that can be obtained through the transitivity of equality relations. For example, if the environment contains the same type constraints $A == B$ and $B == C$, then *getTransitivity* returns the new same type relation $A == C$. The set of constraints returned by *getTransitivity* will be traversed, and for every constraint the symmetric and associated type congruence relations will be determined and added to *Eqenv*. Note that for every new equality relation, the symmetric, associated type congruence

and transitivity are determined. This algorithm is a fixed point algorithm and it eventually stops when no more new relations can be added to the closure based on the existing equality relations. The algorithm for *getTransitiveClosure* is provided in Algorithm 8.

Type canonicalization: The process of type canonicalization normalizes each type belonging to a particular equivalence class of types into a canonical representation of this equivalence class. This process relies on two main algorithms: *canonicalize-weak* and *canonicalize*. The type canonicalization process is invoked during the translation of a type, or while determining the set of new type parameters for the translated interface, class, or method.

The *canonicalize-weak* function given in Algorithm 9 is used to determine the set of equivalent types for a given type in the transitive closure of same-type constraints. The *expType* parameter is the type and *Eqenv* is the set of same-type constraints. The *canonicalize-weak* function works by ordering equivalent types into a list, with all instances first, followed by type parameters, followed by member types. In Algorithm 9, lines 12–22 determine the set of equivalent types for *expType* by traversing the transitive closure of the same type constraints. The algorithm maintains separate lists for equivalent types of instances, type parameters, and member types. A *Sort* function is used to transform the set of types in each list into an ordered set of types. A particular ordering scheme is not required, we chose to use lexicographic ordering based on the alphabetical ordering of the characters in the type expressions. The three sorted lists are then concatenated and the first element, what we call the *weak* canonical form, is returned to the *canonicalize* function for further processing. The returned type represents the *weak* canonical representation for the set of equivalent types for *expType* in the transitive closure *Eqenv*.

Algorithm 9 ***canonicalize-weak***

```
 1: procedure CANONICALIZE-WEAK(expType, Eqenv, sname)
 2:     ArrayList instances, typeParameters, aTypes ;
 3:     if expType is Instance then
 4:         instances.Add (expType) ;
 5:     else if sname.isTypeParameter (expType) then
 6:         typeParameters.Add (expType) ;
 7:     else
 8:         if sname.isAssocType (expType) then
 9:             aTypes.Add (expType) ;
10:         end if
11:     end if
12:     for i = 0 to Eqenv.Size() do
13:         if Eqenv[i].Lhs == expType then
14:             if Eqenv[i].Rhs is Instance then
15:                 instances.Add (Eqenv[i].Rhs)
16:             else if Eqenv[i].Rhs is TypeParameter then
17:                 typeParameters.Add (Eqenv[i].Rhs) ;
18:             else
19:                 aTypes.Add (Eqenv[i].Rhs) ;
20:             end if
21:         end if
22:     end for
23:     instances.Sort() ;
24:     instances.Add(typeParameters.Sort()) ;
25:     instances.Add(aTypes.Sort()) ;
26:     return instances[0];
27: end procedure
```

The *canonicalize* function determines the canonical representation of a type. The *canonicalize* algorithm, given in Algorithm 10 starts by finding the *"weak"* canonical representation of *expType*, using the *canonicalize-weak* function. Based on this representation, we have three subcases.

1. If *canonicalize-weak* returns a type parameter, then it is the canonical form of the input type *expType*.

Algorithm 10 ***canonicalize***

```
1: procedure CANONICALIZE(expType, Eqenv, env, sname)
2:     ArrayList newTargs ;
3:     canon_weak = canonicalize-weak (expType,Eqenv, sname) ;
4:     if sname.isTypeParameter (canon_weak) then
5:         return canon_weak ;
6:     else if canon_weak is an Instance then
7:         ArrayList targs = canon_weak.TypeArguments ;
8:         for i = 0 to targs.Size() do
9:             newTargs[i] = canonicalize (targs[i], Eqenv, env,sname);
10:        end for
11:        canon_weak.TypeArguments = newTargs ;
12:        return canon_weak ;
13:    else
14:        if !canon_weak.Contains (::) then
15:            if sname is Interface then
16:                return canon_weak ;
17:            else if sname is Class then
18:                Get the member type binding for canon_weak and populate atype_bind.
19:                return canonicalize(atype_bind, Eqenv, env, sname) ;
20:            else
21:                return canon_weak ;
22:            end if
23:        else
24:            canon_left = canonicalize(canon_weak.Left, Eqenv, env, sname) ;
25:            if canon_left is a sub-type of a class then
26:                Get member type binding for canon_weak.Right and populate atype_bind.
27:                return canonicalize(atype_bind, Eqenv, env, sname) ;
28:            else
29:                return canon_left :: canon_weak.Right ;
30:            end if
31:        end if
32:    end if
33: end procedure
```

2. If *canonicalize-weak* returns an instance, then the canonical form is determined by canonicalizing every type argument of the instance. Lines 6–12 find the canonical form for an instance.

3. If *canonicalize-weak* returns a member type, then the canonical form can either be a member type binding, or a member type itself. If the context in which

the canonicalize is invoked is a class or if the member type has a class subtype constraint, then the corresponding member type binding is canonicalized and returned as the canonical form for the member type.

The process of canonicalization usually changes the number of type parameters to be generated during associated type translation. In some cases, the number of type parameters can even be reduced. For example, in Figure 21, X and Y are two associated types declared in *interface A*. With direct translation, the translated interface must have two type parameters. However, since X and Y are declared to be of the same type using the **require** clause, the *canonicalize* function ensures that the translation generates only one type parameter.

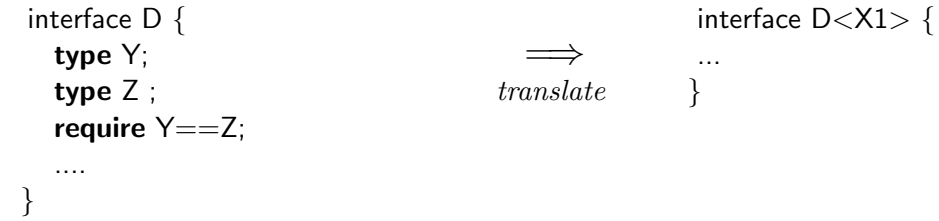

Fig. 21. Translation of a simple interface declaration.

Generation of new type parameters: The generation of new type parameters for the translated generic interface, class, or method relies on three algorithms: *fullassoctparams*, *full-tparams*, *translateConstrainableType* and a mapping function, *generateTypeParameter*.

The *fullassoctparam* function, given in Algorithm 11, collects the set of all type parameters and member types used in the definition of a interface, class, or method recursing to its ancestors and constraints. The type parameters declared in a interface, class, or method are first collected and added to the list *tparams*. The set of member types declared in the current type or in its ancestors are then collected and

Algorithm 11 ***fullassoctparam***

```
1: procedure FULLASSOCTPARAM(sname)
2:     ArrayList tparams, atypes ;
3:     tparams.Add (sname.TypeParameters) ;
4:     if (sname is Interface) then
5:         tparams.Add (assoc-decl (sname)) ;
6:     end if
7:     if (sname is Class) then
8:         tparams.Add (assoc-def (sname)) ;
9:     end if
10:    ArrayList env = env-for-body (sname) ;
11:    for i = 0 to tparams.Size() do
12:        for j = 0 to env.Size() do
13:            Constraint ctr = env[j] ;
14:            if (ctr.Lhs == tparams[i] && ctr.IsSubType()) then
15:                if ctr.Rhs is Interface then
16:                    atypes = assoc-decl (ctr.Rhs) ;
17:                end if
18:                if ctr.Rhs is Class then
19:                    atypes = assoc-def (ctr.Rhs) ;
20:                end if
21:                for k = 0 to atypes.Size() do
22:                    AssocType t = atypes[k] ;
23:                    tparams.Add (tparams[i]::t)
24:                end for
25:            end if
26:        end for
27:    end for
28:    return tparams ;
29: end procedure
```

added to *tparams* by using *assoc-decl* or *assoc-def* algorithms. The constraint set collected using *env-for-body* is then traversed to collect the set of member types that can be accessed through a type parameter or member type based on the sub-type constraints on those type parameters or member types. In Algorithm 11, lines 21–24, populate the list *tparams* with the collected set of member types qualifying every member type in the collected set with the corresponding type parameter or member type through which it can be accessed. The collected set of type parameters and

member types are then returned to the *full-tparams* function.

Algorithm 12 ***full-tparams***

```
1: procedure FULL-TPARAMS(sname)
2:     ArrayList full_assoc_tparams, full_tparams, Eqenv;
3:     ArrayList env = env-for-body (sname) ;
4:     for i = 0 to env.Size() do
5:         if (!env[i].IsSubType()) then
6:             Eqenv.Add (env[i]) ;
7:         end if
8:     end for
9:     full_assoc_tparams = fullassoctparams (sname) ;
10:     ArrayList Eqenv = getTransitiveClosure(Eqenv, env) ;
11:     for i = 0 to full_assoc_tparams.Size() do
12:         canon_ctr = canonicalize (tparams[i], env, Eqenv, sname) ;
13:         if canon_ctr is not an Instance then
14:             full_tparams.Add (canon_ctr ) ;
15:         end if
16:     end for
17:     return full_tparams ;
18: end procedure
```

The *full-tparams* function, given in Algorithm 12, collects all type parameters and member types required for the translated interface, class, or method. This function maintains the semantics of the same-type constraints by *canonicalizing* every type in the collected set of types returned by *fullassoctparam*. During the canonicalization process, if a type parameter or member type canonicalizes to an instance, then such types will not be added to the set of new type parameter for the translated type or method.

The *translateConstrainableType* function, invoked from *translate* function, generates new type parameters for the translated interface, class, or method. The algorithm relies on a mapping function *generateTypeParameter*, that maps member types and type parameters to new type parameter name. The naming scheme for the new type parameter can be arbitrary, however the mapping function ensures that each distinct

type is mapped to a distinct name and to the same name every time. For the naming scheme, we chose to use *X1, X2, X3 .. XN*, where N is the total number of type parameters to be generated with the *full-tparams* function. The *translateConstrainableType* algorithm is given in Algorithm 13.

Algorithm 13 ***translateConstrainableType***

```
1: procedure TRANSLATECONSTRAINABLETYPE(sname, typeParameters)
2:     ArrayList newtypeParameter ;
3:     for i = 0 to typeParameters.Size() do
4:         newtypeParameter.Add(sname.generateTypeParameter(typeParameters[i]))
5:     end for
6:     return newtypeParameters ;
7: end procedure
```

Translation of types: The *translateType* algorithm given in Algorithm 14 is the main function that translates any occurrences of types either as base types, constraints or in the body of an interface, class, or method. The parameter *expType* is the type to be translated. The translation process starts by finding a canonical representation of the type using the *canonicalize* function. Based on the canonical representation *canon_exp*, of *expType*, we distinguish between two cases:

- If the canonical representation *canon_exp* is a type parameter or member type in the context in which the *translateType* is invoked, then *generateTypeParameter* is used to get the mapping of *canon_exp* to the corresponding new type parameter which is returned as the translation of *expType*.

- If the canonical representation *canon_exp* is an instance, then the translation is more involved. The *getTypeParameter* function looks up set of type parameters from the class table for *canon_exp*. The *full-tparams* function gets the set of

Algorithm 14 ***translateType***

```
1: procedure TRANSLATETYPE(expType, env, Eqenv, sname)
2:     ArrayList full_tparams ;
3:     canon_exp = canonicalize (expType, env, Eqenv, sname) ;
4:     if sname.isTypeParameter(canon_exp) OR sname.isAssocType (canon_exp) then
5:         return sname.generateTypeParameter (canon_exp) ;
6:     else
7:         ArrayList targs = canon_exp.TypeArguments ;
8:         ArrayList tParams = getTypeParameters (canon_exp) ;
9:         ArrayList params= full_tparams (canon_exp) ;
10:         params.Sort() ;
11:         for i = 0 to params.Size()  do
12:             if (j = tParams.Contains(params[i])) != null  then
13:                 full_tparams.Add(translateType (targs[j], env, Eqenv, sname)) ;
14:             else
15:                 full_tparams.Add(translateType (params[i], env, Eqenv, sname)) ;
16:             end if
17:         end for
18:     end if
19:     canon_exp.TypeArguments = full_tparams;
20:     return canon_exp ;
21: end procedure
```

all possible type parameters or member types that can be used in *canon_exp*, recursing to its constraints and ancestors. The set of types returned by *full-tparams* is then sorted by using *Sort*, to maintain the same ordering of type parameters and member types in the definition of a interface or class and in its use. The ordered set of types is then traversed and type parameters are substituted with the corresponding type arguments and then translated by invoking *translateType* function, while the member types are mapped to the corresponding new type parameters or the binding type by invoking *translateType*. The type arguments of *canon_exp* are updated and returned.

Note that, the *translateType* guarantees that the type argument list of the translation of any instance will match the type parameter list of the translation of the definition of the class or interface.

Translation of base types: The *translateParents* function given in Algorithm 15 uses *translateType* to translate all the base types of an interface or class.

Algorithm 15 ***translateParents***

```
1: procedure TRANSLATEPARENTS(sname, env, Eqenv)
2:     ArrayList translatedBases ;
3:     for i = 0 to sname.Bases.Size() do
4:         translatedBases.Add (translateType(sname.Bases[i]), env, Eqenv, sname)) ;
5:     end for
6:     return translatedBases ;
7: end procedure
```

Translation of constraints: The *translateEnv* function drives the translation of the propagated constraints. The set of propagated constraints are first traversed and only the sub-type constraints are translated. The same type constraints in the propagated set of constraints are not added to the translated set of constraints. Constraint translation involves translating both the left and right side of a sub-type constraint. The algorithm first canonicalizes the left side of a sub-type constraint. C# do not allow instances to occur in the left side of a sub-type constraint and so, if the canonicalized form is an instance, those constraints are not added to the translated list of constraints. On the other hand, if the canonicalized form is a type parameter or member type, then *generateTypeParameter* obtains the corresponding new type parameter in the translated interface, class, or method.

C# allows type parameters to be constrained by *SpecialConstraints* such as *class*, *struct*, *new()*. These were not considered in the idealized setting in [3]. The right side of such constraints need not be translated. On the other hand, if the right side of a sub-type constraint is an instance, *translateRhs* is used to translate this instance. Finally, *translateEnv* function uses the helper function *constructConstraint* to create

Algorithm 16 ***translateEnv***

```
1: procedure TRANSLATEENV(sname, env, Eqenv)
2:     ArrayList translateEnv ;
3:     for i = 0 to env.Size() do
4:         if env[i].isSubType() then
5:             Lhs = canonicalize (env[i].Lhs, env, Eqenv, sname) ;
6:             if Lhs is not a Instance then
7:                 newLhs = sname.generateTypeParameter(Lhs) ;
8:             else
9:                 continue ;
10:            end if
11:            if env[i].Rhs is SpecialConstraint then
12:                translateEnv.Add (constructConstraint(newLhs, env[i].Rhs));
13:            else
14:                newRhs = translateRhs(env[i].Rhs, env, Eqenv, sname, env[i].Lhs);
15:                translateEnv.Add(constructConstraint(newLhs, newRhs)) ;
16:            end if
17:        end if
18:    end for
19:    return translateEnv ;
20: end procedure
```

a new translated constraint. After translating all constraints, they are returned to *translate* function. The algorithm for *translateEnv* is given in Algorithm 16.

The *translateRhs* algorithm, given in Algorithm 17, is used to translate the instance that appears in the right side of a sub-type constraint of a type parameter or member type. The algorithm for *translateRhs* is similar to *translateType* function except that, during the translation of type arguments of a type that are collected by *full-tparams* function, all the member types are qualified with *lhs*, the left side of the sub-type constraint for which *translateRhs* was invoked. This ensures that all the member types collected using *full-tparams* function are mapped to the already generated new type parameters and that no new type parameters are generated.

Method invocation translation: The translation of generic method header introduces new type parameters for all the member types that can be accessed from

Algorithm 17 ***translateRhs***

```
1: procedure TRANSLATERHS(expType, env, Eqenv, sname, lhs)
2:     ArrayList full_tparams ;
3:     canon_exp = canonicalize (expType, env, Eqenv, sname) ;
4:     ArrayList targs = canon_exp.TypeArguments ;
5:     ArrayList tParams = getTypeParameters (canon_exp) ;
6:     ArrayList params= full_tparams (canon_exp) ;
7:     params.Sort() ;
8:     for i = 0 to params.Size() do
9:         if (j = tParams.Contains(params[i])) != null then
10:            full_tparams.Add(translateType (targs[j], env, Eqenv, sname) ;
11:        else
12:            if canon_exp.isAssocType(params[i]) then
13:                full_tparams.Add(translateType (lhs::params[i], env, Eqenv, sname))
14:            end if
15:        end if
16:    end for
17:    canon_exp.TypeArguments = full_tparams;
18:    return canon_exp ;
19: end procedure
```

the existing type parameter list. For example, Figure 22 shows a method declaration *func1<X, Y>* and a corresponding method invocation statement for this generic method before the translation process and Figure 23 shows the same method declaration and invocation statement after the translation process. Note that, the method *func1<X, Y>* which has two type parameters gets translated to *func1<X1, X2, X3, X4>* with four type parameters. This increase in type parameters makes it necessary to determine the extra type arguments that have to be specified for the new type parameters in the translated method invocation statement, *first.func<Y1, int>*. Moreover during the translation process, the ordering of type parameters in a generic method declaration will be modified and hence the type arguments in the method invocation statement may not match the type parameter list in the method declaration. Due to this, for example in Figure 23, the translated method invocation statement

first.func1<Y1, int> could be invalid. This example necessitates the need to solve two problems listed below in the translation of method invocation statements.

- Determine the type arguments to be specified in the translated method invocation statement for the new extra type parameters.
- Ensure that the method invocation matches the translated method declaration (i.e)., type argument match the corresponding type parameters.

This problem of method invocation translation, — as we name it, was overlooked in [3] and this thesis proposes a solution to this problem and provides a implementation.

```
class first {
   static bool func1 <X, Y> ()
        where . . . . .
   {
        . . . . .
   }
}
class second {
   static bool func2 <Y> () {
        first.func1 <Y, int>();
        . . . . .
   }
}
```

Fig. 22. A method invocation before translation.

The *full-tparams*, *translateType*, and a modified type canonicalization process can be used to translate the method invocation statements. The type arguments for the extra type parameters can be determined from the type arguments specified during the invocation of a method in the program text. During the translation of method invocation statement, a mapping between type parameters of the generic method

```
class first {
   static bool func1 <X1, X2, X3, X4> ()
        where . . . . .
   {
        . . . . .
   }
}
class second {
   static bool func2 <Y1> () {
        first.func1 <Y1, int>(); // Invocation does not match the translated definition
   }
}
```

Fig. 23. A method invocation after translation.

m and the type arguments specified in the method invocation statement has to be created.

The *full-tparams* function can be used to determine the set of type parameters that were generated during the translation of the declaration of the generic method m. This collected set of types has to be ordered to ensure the same ordering of type parameters in the method declaration and in the method invocation. The ordered set of types can then be traversed to determine the canonicalized representation of each type. Based on the canonical representation returned,

- If the canonical form is a type parameter, we retrieve the corresponding type argument for this type parameter using the mapping created between type parameters of method declaration and type arguments specified during method invocation.

- If the canonical form is a member type or instance, the canonicalization process is the same as explained before in *canonicalize* algorithm.

The above process of finding the canonical representation for a type parameter is a slight variation of the type canonicalization process explained in Section D. Here in method invocation statements, the type arguments can even be type parameters from the surrounding context. For example, in Figure 23, in the method invocation *first.func1<Y1, int>*, type parameter *Y1* declared in *func2* is used as a type argument. The above variation of type canonicalization process gives preference to these type arguments rather than to type parameters declared in the generic method m. Now every type in this canonicalized list has to be translated using *translateType* algorithm to ensure that the type parameters and member types are mapped to the corresponding new type parameter or a binding type for the member type in the context of method invocation. This translated list of types is the set of new type arguments to be specified during the invocation of the generic method m.

E. Evaluation of the impact of the language extensions

Using our framework for associated types and constraint propagation, we performed a preliminary evaluation of the impact of these extensions to developing generic libraries. Our experiments consisted of implementing a subset of a state-of-the-art generic library, the Boost Graph Library. The development of this subset also served as a source of testbed for the implementation framework. The subset includes concepts *GraphEdge*, *IncidenceGraph*, *BidirectionalGraph*, *VertexListGraph* and *EdgeListGraph*, and the generic algorithms *Graph Search*, *Breadth first search*, *Bellman Ford*, *Relax*, *first_ neighbor*, *isSameInOutDegree*, and data structures *queue*, and *adjacency list*.

The three factors we used to compare the code verbosity for the subset of BGL in current C# and C# with the proposed extensions are the *number of type parameters*

or member types in concept descriptions or in generic methods, *number of constraint expressions*, and *character count for the constraint expressions*. Tables I to VI list the results.

Table I. Comparison of number of type parameter or member types in concept descriptions in current C# and C# with proposed extensions.

Concepts	**current C#**	**C# with proposed extensions**
GraphEdge	1	1
IncidenceGraph	3	2
BidirectionalGraph	4	1
VertexListGraph	1	1
EdgeListGraph	3	2

Table II. Comparison of number of constraints in concept descriptions in current C# and C# with proposed extensions.

Concepts	**current C#**	**C# with proposed extensions**
GraphEdge	0	0
IncidenceGraph	2	2
BidirectionalGraph	3	1
VertexListGraph	1	1
EdgeListGraph	2	2

The *queue* data structure and few other general concepts such as *StrictWeakOrdering*, *Buffer*, *MutableBuffer* were implemented using type parameters instead of member types demonstrating that both these language features fit together seamlessly and can co-exist without one replacing the other. The verbosity of code while defining concepts in current C# and with the proposed extensions were almost the

Table III. Comparison of character count of constraints in concept descriptions in current C# and C# with proposed extensions.

Concepts	**current C#**	**C# with proposed extensions**
GraphEdge	0	0
IncidenceGraph	50	44
BidirectionalGraph	80	30
VertexListGraph	31	31
EdgeListGraph	50	42

Table IV. Comparison of number of type parameters or member types in generic methods in current C# and C# with proposed extensions.

Generic Methods	**current C#**	**C# with proposed extensions**
GraphSearch	8	4
Relax	8	7
BreadthFirstSearch	7	3
BellmanFord	10	7
first_neighbor	4	1
isSameInOutDegree	5	1

Table V. Comparison of number of constraints in generic methods in current C# and C# with proposed extensions.

Generic Methods	**current C#**	**C# with proposed extensions**
GraphSearch	7	4
Relax	6	6
BreadthFirstSearch	6	3
BellmanFord	8	6
first_neighbor	3	1
isSameInOutDegree	4	1

Table VI. Comparison of character count of constraints in generic methods in current C# and C# with proposed extensions.

Generic Methods	**current C#**	**C# with proposed extensions**
GraphSearch	212	96
Relax	206	212
BreadthFirstSearch	191	74
BellmanFord	295	239
first_neighbor	80	6
isSameInOutDegree	101	6

same, though in the *BidirectionalGraph* concept descriptions, the number of type parameters, constraints and character count of constraints was reduced considerably. On the other hand, verbosity of code in generic methods reduced significantly as is evident from Table IV to VI.

Generally, access to type parameters are shorter than member type access syntax. For example, in one case the character count of constraints shown in Table VI, for the generic method *relax* increased slightly. This effect is due to the repeated occurrence of member type access through type parameters in the constraint clause of *relax* method, whereas these member types are represented as direct type parameters in current C#. Also note from Table V, the number of constraints in *relax* method remains the same because there are no constraints that can be possibly propagated and hence constraint propagation does not compensate for the increase in character count. In this case, the slight increase in character count is actually compensated by the decrease in the number of type parameters required for the *relax* method written with the proposed extensions. This slight increase in character count of constraints was not found in other generic method implementations due to the significant decrease in the number of constraints that has to be repeated because of the constraint propagation language extension.

In summary, the empirical results suggest that adding support for associated types and constraint propagation does improve the generic programming capabilities in C# by reducing the verbosity of the code.

CHAPTER VI

FEATHERWEIGHT GENERIC JAVA EXTENDED WITH INTERFACES

The formal study of properties of associated types and constraint propagation in [3] was based on the Featherweight Generic Java formalism extended with Interfaces (FGJ+I). Unlike FGJ, FGJ+I formalism has no established proof of type safety, and we prove the type soundness of FGJ+I in this chapter.

FGJ+I extends FGJ with generic interfaces. Interfaces in FGJ+I follow the conventional object oriented behavior found in Java or C#. Classes in FGJ+I can implement multiple interfaces but only inherit from a single class. Interfaces can inherit from multiple interfaces. All the methods declared in an interface must be defined by the class implementing that interface. Further, fields cannot be declared inside interfaces. Interfaces in FGJ+I can be generic and bounds can be specified on type parameters of generic interfaces. Similar to FGJ, a bound of a type variable may not be another type variable. Figure 24 shows a simple program in FGJ+I.

Generics can in principle be realized by two implementation styles: *type-passing*, augmenting the runtime system to carry information about type parameters, or *erasure*, removing all information about type parameters at runtime [17]. Generics implementation of C# maintains the information about type parameters at run-time. To stay close to the semantics of C#, this thesis explores only the type-passing implementation style, giving a direct semantics for FGJ+I and proving a type soundness theorem.

A. Syntax

The abstract syntax of FGJ+I class declaration, interface declaration, constructor declaration, method declaration, method definition and expressions are given in Fig-

ure 25. Many of the rules are either directly from, or based on, FGJ [5]. For the sake of conciseness of notation, we abbreviate the keyword *extends* to the symbol $\triangleleft$. The metavariables *C, D* range over class names; *I, J* range over interface names and *E, F* range over both class and interface names; *X, Y, and Z* ranges over type variables; *S, T, U, and V* range over types; *N, P, and Q* range over non-variable types (types other than type variables); *M* ranges over interface instances; *K* ranges over class instances; *f* and *g* range over field names; *m* ranges over method names; *x* ranges over variables; *d* and *e* range over expressions; *L* ranges over class declarations; *Id* ranges over interface declarations; *kd* ranges over constructor declarations; *md* ranges over method definition; and *ms* ranges over method declarations.

```
interface convertibleTo<A extends Object> {
        A convert() ;
}
class A extends Object {
        A() { super() ; }
}
class B extends Object {
        B() { super() ; }
}
class Pair <X extends Object, Y extends Object> implements convertibleTo<X> {
        X fst ;
        Y snd ;
        Pair(X fst, Y snd) {
                super() ; this.fst = fst ; this.snd = snd ;
        }
        <Z extends Object> Pair<Z,Y> setfst (Z newfst) {
                return new Pair<Z,Y> (newfst, this.snd) ;
        }
        X convert() { return this.fst; }
}
```

Fig. 24. A simple FGJ+I program.

Types in FGJ+I are either class or interface instances, or type variables. Borrowing from FGJ, we use $\overline{f}$ as shorthand for a possibly empty sequence $f_1, \ldots, f_n$ (and similarly for $\overline{N}, \overline{T}, \overline{x}, \overline{e}$, etc.). We also use $\overline{X}$ as shorthand for $X_1, \ldots, X_n$ and assume sequences of type variables contain no duplicate names. We abbreviate the list of type parameters and their bounds, $X_1 \lhd \overline{N_1}, \ldots, X_n \lhd \overline{N_n}$ in a obvious way as $\overline{X \lhd \overline{N}}$. We write the empty sequence as $\bullet$ and denote the concatenation of sequences using a comma. The length of a sequence $\overline{x}$ is written as $\#(\overline{x})$. The operations on pairs of sequences are abbreviated in an obvious way, writing "$\overline{T}\ \overline{f}$" for "$T_1\ f_1, \ldots, T_n\ f_n$", where n is the length of $\overline{T}$ and $\overline{f}$. Sequences of field declarations, parameter names, method definitions, method declarations are assumed to contain no duplicate names. Further we assume that inheritance does not cause method or field duplication in the derived class or interface. Also, sometimes a class definition is written as $class\ C\texttt{<}\overline{X \lhd \overline{P}}\texttt{>} : \overline{N} \ldots$, in which case we assume that at most one N_i is a class, and the other elements in $\overline{N}$ are interfaces. Further, we sometimes use the syntax $class/interface\ E\texttt{<}\overline{X \lhd \overline{P}}\texttt{>} : \overline{N} \ldots$ when describing behavior common to classes and interfaces; it is assumed that if E is an interface, all elements of $\overline{N}$ are interfaces.

As in FGJ, we assume a class table CT as a mapping from a class or interface names E to their corresponding declarations. A program is a pair *(CT, e)* of a class table and an expression. The given class table is assumed to satisfy some sanity conditions: (1) $CT(E) = class/interface$ E... for every $E \in dom(CT)$; (2) $Object \notin dom(CT)$; (3) for every class/interface name E (except $Object$) appearing anywhere in CT, we have $E \in dom(CT)$; and (4) there are no cycles in the subtype relation (<:) induced by CT, i.e., the relation <: is antisymmetric.

For the typing and reduction rules we require a few auxiliary functions, given in Figure 26. We write $m \notin \overline{md}$ to mean that the method definition of the name m is

(interface def) Id	$::= \texttt{interface}\ I\texttt{<}\overline{X \lhd \overline{N}}\texttt{>} \lhd \overline{M}\ \{\overline{ms}\}$
(class def) L	$::= \texttt{class}\ C\texttt{<}\overline{X \lhd \overline{N}}\texttt{>} \lhd \overline{M}, K\ \{\overline{T}\ \overline{f};\ kd\ \overline{md}\}$
(constructor def) kd	$::= C(\overline{T}\ \overline{f})\ \{super(\overline{f}); \texttt{this}.\overline{f} = \overline{f};\}$
(method signature) ms	$::=\texttt{<}\overline{X \lhd \overline{N}}\texttt{>}\ T\ m(\overline{U}\ \overline{x});$
(method def) md	$::=\texttt{<}\overline{X \lhd \overline{N}}\texttt{>}\ T\ m(\overline{U}\ \overline{x})\ \{\texttt{return}\ e;\}$
(expression) e	$::= x \mid e.f \mid e.m\texttt{<}\overline{T}\texttt{>}(\overline{e}) \mid \texttt{new}\ K(\overline{e}) \mid (N)e$
(instantiated interface) M, N	$::= I\texttt{<}\overline{T}\texttt{>}$
(instantiated class) K, L	$::= C\texttt{<}\overline{T}\texttt{>}$
(instantiated class or interface) N, P, Q	$::= M \mid K$
(Type Variables and non-variable types) S, T, U, V	$::=X \mid N$
Class names	$::= C, D$
Interface names	$::= I, J$
(class or interface name) E, F	$::= C \mid I$
Type Variable names	$::= X, Y, Z$
Field names	$::= f, g$
Method name	$::= m$
$fields(K)$	$::=\overline{T}\ \overline{f}$
$mtype(m, N)$	$::=\texttt{<}\overline{X \lhd \overline{N}}\texttt{>}\overline{U} \to U$
$mbody(m\texttt{<}\overline{V}\texttt{>}, K)$	$::=\overline{x}.e$

Fig. 25. Syntax of FGJ+I.

$$\text{SUBCLASSING}\quad E \trianglelefteq E \qquad \frac{E \trianglelefteq F \qquad F \trianglelefteq G}{E \trianglelefteq G} \qquad \frac{\texttt{class}\ E\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{F}\{...\}}{E \trianglelefteq F_i} \qquad \text{F-OBJECT}\quad \mathit{fields}(\mathit{object}) = \bullet$$

$$\text{F-CLASS}\quad \frac{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}, K\ \{\overline{S}\ \overline{f};\ kd\ \overline{md}\} \qquad \mathit{fields}([\overline{T}/\overline{X}]K) = \overline{U}\ \overline{g}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{U}\ \overline{g}, [\overline{T}/\overline{X}]\overline{S}\ \overline{f}}$$

$$\text{MT-CLASS}\quad \frac{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}, K\ \{\overline{S}\ \overline{f};\ kd\ \overline{md}\} \qquad \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}U\ m(\overline{U}\ \overline{x})\{return\ e;\} \in \overline{md}}{\mathit{mtype}(m, C\texttt{<}\overline{T}\texttt{>}) = [\overline{T}/\overline{X}](\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{U} \rightarrow\ U)}$$

$$\text{MT-SUPER}\quad \frac{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}, K\ \{\overline{S}\ \overline{f};\ kd\ \overline{md}\} \qquad m \notin \overline{md}}{\mathit{mtype}(m, C\texttt{<}\overline{T}\texttt{>}) = \mathit{mtype}(m, [\overline{T}/\overline{X}]K)}$$

$$\text{MT-INTERFACE}\quad \frac{\texttt{interface}\ I\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}\ \{\overline{ms}\} \qquad \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}U\ m(\overline{U}\ \overline{x}) \in \overline{ms}}{\mathit{mtype}(m, I\texttt{<}\overline{T}\texttt{>}) = [\overline{T}/\overline{X}](\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{U} \rightarrow\ U)}$$

$$\text{MT-SUPER-INTERFACE}\quad \frac{\texttt{interface}\ I\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}\ \{\overline{ms}\} \quad \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}U\ m(\overline{U}\ \overline{x}) \notin \overline{ms} \quad \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{U} \rightarrow\ U = \mathit{mtype}(m, \overline{M})}{\mathit{mtype}(m, I\texttt{<}\overline{T}\texttt{>}) = [\overline{T}/\overline{X}](\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{U} \rightarrow\ U)}$$

$$\text{MB-CLASS}\quad \frac{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}, K\ \{\overline{S}\ \overline{f};\ kd\ \overline{md}\} \quad \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}U\ m(\overline{U}\ \overline{x})\{return\ e_0;\} \in \overline{md}}{\mathit{mbody}(m\texttt{<}\overline{V}\texttt{>}, C\texttt{<}\overline{T}\texttt{>}) = \overline{x}.[\overline{T}/\overline{X}, \overline{V}/\overline{Y}]e_0}$$

$$\text{MB-SUPER}\quad \frac{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}, K\ \{\overline{S}\ \overline{f};\ kd\ \overline{md}\} \quad m \notin \overline{md}}{\mathit{mbody}(m\texttt{<}\overline{V}\texttt{>}, C\texttt{<}\overline{T}\texttt{>}) = \mathit{mbody}(m\texttt{<}\overline{V}\texttt{>}, [\overline{T}/\overline{X}]K)} \qquad \text{MT-MULTIPLE}\quad \frac{\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{U} \rightarrow\ U = \mathit{mtype}(m, N_i)}{\mathit{mtype}(m, \overline{N}) = \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{U} \rightarrow\ U}$$

Fig. 26. FGJ+I auxillary functions.

not included in $\overline{md}$ and the notation $m \notin \overline{ms}$ to mean that method declaration m is not included in $\overline{ms}$. The fields of a non-variable type (class) K, written $fields(K)$, is a sequence of corresponding types and field names $\overline{T}\ \overline{f}$. The type of the method invocation on a non variable type obtained using the $mtype(m, N)$ function is a type of the form $\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{U} \rightarrow U$. The body of the method invocation m at non-variable type K with type parameters $\overline{V}$, written $mbody(m\texttt{<}\overline{V}\texttt{>}, K)$, is a pair, written $\overline{x}.e$, of a sequence of parameters $\overline{x}$ and an expression e. Note that the *mtype* function is defined for interfaces and classes whereas *mbody* is defined only for classes. Additionally the typing and reduction rules use the following helper functions, whose definitions we omit.

- *name* function takes a non-variable type N and returns the name of the class or interface of which the type N is an instance.
- *getclass* function takes a sequence of non-variable types $\overline{N}$ and returns a class K, if $K \in \overline{N}$, otherwise returns an empty sequence
- *msig_decl_names* function collects the names of all methods declared in a given interface or in its ancestors
- *msig_def_names* function collects the names of all methods defined in a class or in its base class.

B. Typing

In our formalism, we use two typing environments Γ and Δ. The environment Γ maps variables to their types, written as $\overline{x} : \overline{T}$. The type environment Δ maps type variables to non-variable types, written as $\overline{X <: \overline{N}}$. Note that, a type variable can have multiple bounds in FGJ+I due to the inclusion of interfaces to the language.

The forms of judgments in the FGJ+I type system consists of one for subtyping, $\Delta \vdash S <: T$, one for type well-formedness, $\Delta \vdash T$ ok and one for typing, $\Delta; \Gamma \vdash e : T$. Sequences of judgments are abbreviated in the obvious way: $X <: \overline{N}$ for $X <: N_1$ and $X <: N_2 \ldots X <: N_n$; $\Delta \vdash S_1 <: T_1, T_2, \ldots, T_m, \ldots\ldots, \Delta \vdash S_n <: T_1, T_2, \ldots, T_m$ is abbreviated to $\Delta \vdash \overline{S <: \overline{T}}$; $\Delta \vdash T_1$ *ok*, $\ldots$, $\Delta \vdash T_n$ *ok* to $\Delta \vdash \overline{T}$ ok; $\Delta \vdash S_1 <: T_1, \Delta \vdash S_2 <: T_2, \ldots, \Delta \vdash S_n <: T_n$ to $\Delta \vdash \overline{S} <: \overline{T}$ and $\Delta; \Gamma \vdash e_1 : T_1, \ldots,$ $\Delta; \Gamma \vdash e_n : T_n$ to $\Delta; \Gamma \vdash \overline{e} : \overline{T}$.

The sub-typing and type well-formedness rules are given in Figure 27. The function $bound_\Delta(T)$, returns the upper bound of a type T. If T is a type variable then *bound* returns a sequence of non-variable types given as bound for T. For non-variable type N, $bound_\Delta(N)$ returns the non-variable type N itself or a singleton sequence. Since FGJ+I does not support type variables to be constrained by another type variable, the *bound* function always returns either a sequence of non-variable types $\overline{N}$ or a single non-variable type N.

The subtyping relation $\Delta \vdash S <: T$, read as "S is a subtype of T in Δ", is the reflexive and transitive closure of the *extends* relation. The rule S-CLASS introduces subtyping relation between a class and its base types in the environment Δ. Similarly, rule S-INTERFACE introduces subtyping relation between an interface and its base types in the environment Δ. The rule S-VAR introduces the subtyping relation between type variables and its bounds.

A class or interface instance, $C\texttt{<}\overline{T}\texttt{>}$ or $I\texttt{<}\overline{T}\texttt{>}$, is said to be *well-formed* only if substituting $\overline{T}$ for $\overline{X}$ in $C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>}$ respects the bounds $\overline{N}$ on every type parameter, i.e., $\overline{T <: \overline{N}}$. The type *Object* is assumed to be well-formed in the environment Δ and a type variable X is well-formed in Δ if it belongs to the $dom(\Delta)$. We use the notation, $\Delta \vdash T$ *ok* if the type T is well-formed in the environment Δ.

$$bound_\Delta(X) = \Delta(X) \qquad bound_\Delta(N) = N$$

$$\text{S-REFL}\ \ \Delta \vdash T <: T$$

$$\text{S-TRANS}\ \ \frac{\Delta \vdash S <: T \qquad \Delta \vdash T <: U}{\Delta \vdash S <: U}$$

$$\text{S-VAR}\ \ \Delta \vdash X <: \Delta(X)$$

$$\text{S-CLASS}\ \ \frac{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{P}}\texttt{>} \triangleleft \overline{N}\ \{....\}}{\Delta \vdash C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]N_i,\ \text{for all } i}$$

$$\text{S-INTERFACE}\ \ \frac{\texttt{interface}\ I\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}\ \{....\}}{\Delta \vdash I\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]M_i,\ \text{for all } i}$$

$$\text{Wf-object}\ \ \Delta \vdash Object\ ok$$

$$\text{WF-VAR}\ \ \frac{X \in dom(\Delta)}{\Delta \vdash X\, ok}$$

$$\text{WF-Interface}\ \ \frac{\texttt{interface}\ I\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}\{....\} \qquad \Delta \vdash \overline{T}\ ok \qquad \Delta \vdash \overline{T <: [\overline{T}/\overline{X}]\overline{N}}}{\Delta \vdash I\texttt{<}\overline{T}\texttt{>}\ ok}$$

$$\text{WF-CLASS}\ \ \frac{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}, K\ \{...\} \qquad \Delta \vdash \overline{T}\ ok \qquad \Delta \vdash \overline{T <: [\overline{T}/\overline{X}]\overline{N}}}{\Delta \vdash C\texttt{<}\overline{T}\texttt{>}\ ok}$$

$$\text{VALID-DOWNCAST}\ \ \frac{\texttt{class/interface}\ E\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{F\texttt{<}\overline{T}\texttt{>}}\ \{...\} \quad \overline{X} = FV(\overline{T})}{dcast(E, F_i), for\ all\ i}$$

$$\text{TRANS-DOWNCAST}\ \ \frac{dcast(E, F) \qquad dcast(F, G)}{dcast(E, G)}$$

$$\text{VALID-METHOD-OVERRIDE}\ \ \frac{mtype(m, N)\ = \texttt{<}Z \triangleleft \overline{Q}\texttt{>}\overline{U} \to U_0\ implies\ \overline{P}, \overline{T} = [\overline{Y}/\overline{Z}](\overline{Q}, \overline{U})\ and\ \overline{Y <: \overline{P}} \vdash T_0 <: [\overline{Y}/\overline{Z}]U_0}{override(m, N, \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{T} \to T_0)}$$

$$\text{OVERRIDE-MULTIPLE}\ \ \frac{override(m, N_i, \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{T} \to T_0)}{override(m, \overline{N}, \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{T} \to T_0)}$$

Fig. 27. FGJ+I bound of types and subtyping rules.

GT-CLASS

$$\frac{\overline{X <: \overline{N}} \vdash \overline{N}, \overline{M}, \overline{T}, K \ ok \qquad \mathit{fields}(K) = \overline{U}\ \overline{g} \qquad kd = C(\overline{U}\ \overline{g}, \overline{T}\ \overline{f})\{\mathit{super}(\overline{g}); \texttt{this}.\overline{f} = \overline{f};\} \qquad \overline{md}\ ok\ in\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \qquad A = \mathit{msig\_decl\_names}(C) \qquad B = \mathit{msig\_def\_names}(C) \qquad A - B = \emptyset}{\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}, K\ \{\overline{T}\ \overline{f};\ kd\ \overline{md}\}\ ok}$$

GT-INTERFACE

$$\frac{\overline{ms}\ ok\ in\ I\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \qquad \overline{X <: \overline{N}} \vdash \overline{M}\ ok \qquad \overline{X <: \overline{N}} \vdash \overline{N}\ ok}{\texttt{interface}\ I\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>} \triangleleft \overline{M}\ \{\overline{ms}\}\ ok}$$

GT-METHODDECL

$$\frac{\Delta = \overline{X <: \overline{N}}, \overline{Y <: \overline{P}} \qquad \Delta \vdash \overline{T}, T, \overline{P}\ ok}{\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\ T\ m(\overline{T}\ \overline{x})\ ok\ in\ I\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>}}$$

GT-METHOD

$$\frac{\Delta = \overline{X <: \overline{N}}, \overline{Y <: \overline{P}} \qquad \Delta \vdash T, \overline{T}, \overline{P}\ ok \qquad \Delta; \overline{x} : \overline{T}; \mathit{this} : C\texttt{<}\overline{X}\texttt{>} \vdash e_0 : S \qquad \Delta \vdash S <: T \qquad \texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{Q}}\texttt{>} \triangleleft \overline{N}\{...\} \qquad \mathit{override}(m, \overline{N}, \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{T} \to T)}{\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\ T\ m(\overline{T}\ \overline{x})\ \ \{\mathit{return}\ e_0;\}\ ok\ in\ C\texttt{<}\overline{X \triangleleft \overline{N}}\texttt{>}}$$

Fig. 28. FGJ+I class, interface and method typing.

$$\text{GT-VAR}\quad \Delta;\Gamma \vdash x : \Gamma(x)$$

$$\text{GT-FIELD}\quad \frac{\Delta;\Gamma \vdash e_0 : T_0 \quad T_0 \textit{ not } \texttt{interface} \quad \textit{fields}(\textit{getclass}(\textit{bound}_\Delta(T_0))) = \overline{T}\ \overline{f}}{\Delta;\Gamma \vdash e_0.f_i : T_i}$$

$$\text{GT-INVK}\quad \frac{\Delta;\Gamma \vdash e_0 : T_0 \qquad \textit{mtype}(m, \textit{bound}_\Delta(T_0)) = \texttt{<}\overline{Y \lhd \overline{P}}\texttt{>}\overline{U} \to U \qquad \Delta \vdash \overline{V}\ \textit{ok} \qquad \Delta \vdash \overline{V} <: [\overline{V}/\overline{Y}]\overline{P} \qquad \Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}}{\Delta;\Gamma \vdash e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V}/\overline{Y}]U}$$

$$\text{GT-NEW}\quad \frac{\Delta \vdash K\ \textit{ok} \qquad \textit{fields}(K) = \overline{T}\ \overline{f} \qquad \Delta;\Gamma \vdash \overline{e} : \overline{S} \qquad \Delta \vdash \overline{S} <: \overline{T}}{\Delta;\Gamma \vdash \texttt{new}\ K(\overline{e}) : K}$$

$$\text{GT-UCAST}\quad \frac{\Delta;\Gamma \vdash e_0 : T_0 \qquad N_0 \textit{ in } \textit{bound}_\Delta(T_0) \qquad \Delta \vdash N_0 <: N}{\Delta;\Gamma \vdash (N)e_0 : N}$$

$$\text{GT-DCAST}\quad \frac{\Delta;\Gamma \vdash e_0 : T_0 \qquad \Delta \vdash N\ \textit{ok} \qquad N_0 \textit{ in } \textit{bound}_\Delta(T_0) \qquad \Delta \vdash N <: N_0 \qquad N = E\texttt{<}\overline{T}\texttt{>} \qquad N_0 = F\texttt{<}\overline{U}\texttt{>} \qquad \textit{dcast}(E, F)}{\Delta;\Gamma \vdash (N)e_0 : N}$$

$$\text{GT-SCAST}\quad \frac{\Delta;\Gamma \vdash e_0 : T_0 \qquad \Delta \vdash N\ \textit{ok} \qquad \overline{N} = \textit{bound}_\Delta(T_0) \qquad \textit{name}(N) \not\triangleleft \textit{name}(N_i) \qquad \textit{name}(N_i) \not\triangleleft \textit{name}(N) \qquad \textit{stupid warning}}{\Delta;\Gamma \vdash (N)e_0 : N}$$

Fig. 29. FGJ+I typing rules for expression.

Similar to FGJ, FGJ+I allows co-variant overriding of methods. The *override* $(m, N, \overline{<Y \triangleleft \overline{P}>T} \rightarrow T_0)$ helper function given in Figure 27 is used to check for valid method overriding when type checking method definitions in class bodies.

The typing rules for type checking class, interface, method definition and method declarations are given in Figure 28. Note that in the rule for typing class definition, GT-CLASS, the helper functions *msig_decl_names* and *msig_def_names* are used to check if every method declaration has a corresponding definition.

The typing rules for expressions are given in Figure 29. The typing judgment for expression typing is of the form $\Delta; \Gamma \vdash$ e : T, read as "in the typing environment Δ and the environment Γ, the expression e has type T".

C. Reduction

The reduction relation is of the form $e \rightarrow e'$, read as "expression e reduces to expression e' in one step". We use the notation $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. There are three reduction rules, one for field access, one for method invocation, and one for casting. We use the notation $[\overline{d}/\overline{x}, e/y]e_0$ for the result of simultaneously replacing x_1 by $d_1, \ldots, x_n$ by d_n, and y by e in the expression e_0. Introducing interfaces to FGJ does not introduce any new forms of expressions; the reduction rules thus remain the same as in FGJ. The reduction rules for FGJ+I are given in Figure 30.

D. Properties of the formalization

FGJ+I programs enjoy the subject reduction and progress properties, and thus a type soundness property like FGJ programs. Since the evaluation and reduction are the same as FGJ, the structure of the type soundness proof did not change radically

$$\frac{\textsc{GR-field} \quad \mathit{fields}(K) = \overline{T}\ \overline{f}}{(\texttt{new}\ K(\overline{e})).f_i \rightarrow e_i}$$

$$\frac{\textsc{Gr-invk} \quad \mathit{mbody}(m\texttt{<}\overline{V}\texttt{>},\ K)\ =\ \overline{x}.e_0}{(\texttt{new}\ K(\overline{e})).m\texttt{<}\overline{V}\texttt{>}(\overline{d}) \rightarrow [\overline{d}/\overline{x}, \texttt{new}\ K(\overline{e})/\mathit{this}]e_0}$$

$$\frac{\textsc{Gr-cast} \quad \emptyset \vdash K <: P}{(P)(\texttt{new}\ K(\overline{e})) \rightarrow \texttt{new}\ K(\overline{e})}$$

$$\frac{\textsc{GRC-field} \quad e_0 \rightarrow e_0'}{e_0.f \rightarrow e_0'.f}$$

$$\frac{\textsc{GRC-inv-recv} \quad e_0 \rightarrow e_0'}{e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) \rightarrow e_0'.m\texttt{<}\overline{V}\texttt{>}(\overline{e})}$$

$$\frac{\textsc{GRC-inv-arg} \quad e_i \rightarrow e_i'}{e_0.m\texttt{<}\overline{V}\texttt{>}(..., e_i, ...) \rightarrow e_0.m\texttt{<}\overline{V}\texttt{>}(..., e_i', ...)}$$

$$\frac{\textsc{GRC-new-arg} \quad e_i \rightarrow e_i'}{\texttt{new}\ N(...., e_i, ...) \rightarrow \texttt{new}\ N(...., e_i', ...)}$$

$$\frac{\textsc{GRC-cast} \quad e_0 \rightarrow e_0'}{(N)e_0 \rightarrow (N)e_0'}$$

Fig. 30. FGJ+I reduction rule.

from that of FGJ [5]. However due to the addition of interfaces to the formalism, we need to prove the preservation of sub-typing and well-formedness of interfaces under type substitution lemmas, in order to reuse the proof structure from [5]. We proceed by first proving the lemmas that need to change and then outline the entire proof, relying on [5] where no change from the FGJ proof was necessary. The main lemmas required are term substitution lemma, as in FGJ, and a similar lemma to prove the preservation of typing under type substitution. The required lemmas are proved by induction on derivation of $\Delta \vdash S <: T$ or $\Delta; \Gamma \vdash e : T$. The proof of type soundness of FGJ+I follows by establishing the subject reduction and progress properties. We assume that the underlying class table is *ok* in the following proofs.

Lemma 1 (Type substitution preserves Subtyping)**.** *If* $\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash S <: T$ *and* $\Delta_1 \vdash \overline{U <: [\overline{U}/\overline{X}]\overline{N}}$ *with* $\Delta_1 \vdash \overline{U}$ *ok and none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]S <: [\overline{U}/\overline{X}]T$.

Proof. By induction on the derivation of $\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash S <: T$. The subtyping rules for classes and variables and the transitivity and reflexivity remains the same as in FGJ. It is straightforward to see that lemma holds for those cases. Now consider the case of sub-typing rules for interfaces. By rule S-INTERFACE, we have

$$\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash I\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{Y}]\overline{M}$$
$$S = I\texttt{<}\overline{T}\texttt{>}$$
$$T = [\overline{T}/\overline{Y}]\overline{M}$$
$$\texttt{interface I<}\overline{Y \triangleleft \overline{P}}\texttt{>} \triangleleft \overline{M} \texttt{ \{ ..... \}}$$

By rule GT-INTERFACE, we have $\overline{Y <: \overline{P}} \vdash \overline{M}$ *ok* implies that $\overline{M}$ does not include any of $\overline{X}$ as a free variable. Thus, $[\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{M} = [[\overline{U}/\overline{X}]\overline{T}/\overline{Y}]\overline{M}$. Now with substitution, $[\overline{U}/\overline{X}]I\texttt{<}\overline{T}\texttt{>} <: [\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{M}$ implies $I\texttt{<}[\overline{U}/\overline{X}]\overline{T}\texttt{>} <: [[\overline{U}/\overline{X}]\overline{T}/\overline{Y}]\overline{M}$.

Finally by rule S-INTERFACE, we have $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]S <: [\overline{U}/\overline{X}]T$ finishing the proof.

□

Lemma 2 (Type Substitution preserves Type Well-Formedness)**.** *If* $\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash T$ *ok and* $\Delta_1 \vdash \overline{U <: [\overline{U}/\overline{X}]\overline{N}}$ *with* $\Delta_1 \vdash \overline{U}$ *ok and none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]T$ *ok.*

Proof. By induction on the derivation of $\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash T$ *ok*, with a case analysis of the last rule used. The well-formedness rules for Objects and variables and classes remains the same as in FGJ. It is straightforward to see that lemma holds for those cases. Consider the case of well-formedness rules for interfaces. By rule WF-INTERFACE, we have

$$T = I\texttt{<}\overline{T}\texttt{>}$$
$$\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash \overline{T}\ ok$$
$$\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash \overline{T <: [\overline{T}/\overline{Y}]\overline{P}}$$
$$\texttt{interface}\ \ \text{I}\texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>} \triangleleft \overline{M}\ \{\ .....\ \}$$

By induction hypothesis, we have

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{T}\ \ \text{ok}$$

On the other hand by Lemma 1, we have $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \overline{[\overline{U}/\overline{X}]T <: [\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{P}}$. Since $\overline{Y <: \overline{P}} \vdash \overline{P}$ *ok* by the rule GT-INTERFACE, implies that $\overline{P}$ does not include any of $\overline{X}$ as a free variable. Thus, $[\overline{U}/\overline{X}][\overline{T}/\overline{Y}]\overline{P} = [[\overline{U}/\overline{X}]\overline{T}/\overline{Y}]\overline{P}$ and hence we have,

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \overline{[\overline{U}/\overline{X}]T <: [[\overline{U}/\overline{X}]\overline{T}/\overline{Y}]\overline{P}}$$

This implies that all the type arguments with substitution satisfies the bounds on the corresponding type parameters and that they are well formed. Therefore by rule WF-INTERFACE, we have

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash I\texttt{<}[\overline{U}/\overline{X}]\overline{T}\texttt{>}\ \text{ok}$$

□

Lemma 3 (Weakening)**.** *Suppose* $\Delta,\ \overline{X <: \overline{N}} \vdash \overline{N}$ *ok and* $\Delta \vdash U$ *ok.*

(i) If $\Delta \vdash S <: T$*, then* $\Delta, \overline{X <: \overline{N}} \vdash S <: T$*.*

(ii) If $\Delta \vdash S$ *ok, then* $\Delta, \overline{X <: \overline{N}} \vdash S$ *ok.*

(iii) If $\Delta; \Gamma \vdash e : T$*, then* $\Delta; \Gamma, x : U \vdash e : T$ *and* $\Delta, \overline{X <: \overline{N}}; \Gamma \vdash e : T$

Proof. Each of the above cases can be proved by induction on the derivation of $\Delta \vdash S <: T$ and $\Delta \vdash S$ ok and $\Delta;\ \Gamma \vdash$ e : T. The proof given in [5] does not change except for the added cases WF-INTERFACE and S-INTERFACE which are trivial.

□

Lemma 4. *If* $\Delta \vdash T$ *ok and* $mtype(m, bound_\Delta(T)) = \texttt{<}\overline{Y \lhd \overline{P}}\texttt{>}\overline{U} \to U_0$*, then for any* S *such that* $\Delta \vdash S <: T$ *and* $\Delta \vdash S$ *ok, we have* $mtype(m, bound_\Delta(S)) = \texttt{<}\overline{Y \lhd \overline{P}}\texttt{>}\overline{U} \to U_0'$ *and* $\Delta, \overline{Y <: \overline{P}} \vdash U_0' <: U_0$*.*

Proof. By induction on the derivation of $\Delta \vdash S <: T$ with a case analysis of the last rule used.

The subtyping rules for reflexivity and transitivity, S-REFL and S-TRANS are the same as in FGJ. It is straightforward to see that lemma holds for those cases. We now provide the proof for all the other cases, S-VAR, S-INTERFACE, S-CLASS.

Case S-VAR: Given $\Delta \vdash X <: \Delta(X)$ $\quad$ S = X $\quad$ $T \in \Delta(X)$ or $T \in bound_\Delta(S)$.

In our formalism, we assume that methods are not duplicated due to inheritance. This implies that the method type of m returned by $mtype(m, bound_\Delta(T))$ is the same as $mtype(m, bound_\Delta(S))$ since the lookup for m will resolve to the same type T in both cases, where, $T = bound_\Delta(T)$ and $T \in bound_\Delta(S)$. Therefore by reflexive property of sub-typing relation, we have $\Delta, \overline{Y <: \overline{P}} \vdash U_0' <: U_0$.

Case S-INTERFACE: Given

$$S = I\texttt{<}\overline{T}\texttt{>} \quad T \in [\overline{T}/\overline{X}]\overline{M}$$
$$\texttt{interface } I\texttt{<}\overline{X \lhd \overline{N}}\texttt{>} \lhd \overline{M}\{\overline{ms};\}$$

In our formalism, we do not allow overriding of method declarations in interfaces. So if m is found in $\overline{ms}$, then it will not be present in its base type T which contradicts the induction hypothesis that the method is present in the base type T. So we can safely omit this case.

On the other hand, if $m \notin \overline{ms}$, then $mtype(m, bound_\Delta(S))$ would return the type of the method m in one of the base types M_i by MT-SUPER-INTERFACE rule. Moreover by the assumption that methods are not duplicated due to inheritance, we can be sure that the method lookup will resolve to the same type T, where $T = M_i$ while determining the type of the method in $bound_\Delta(T)$ as well as in $bound_\Delta(S)$. Hence by reflexive property of the subtyping relation, we have $\Delta, \overline{Y <: \overline{P}} \vdash U_0' <: U_0$.

Case S-CLASS:

$$S = C\texttt{<}\overline{T}\texttt{>} \quad T \in [\overline{T}/\overline{X}]\overline{N}$$
$$\texttt{class } C\texttt{<}\overline{X \lhd \overline{Q}}\texttt{>} \lhd \overline{N}\{\ldots \overline{md};\}$$

If $m \notin \overline{md}$, there are two sub-cases depending on whether T is a class or interface. If T is a class, it is easy to show the conclusion as $mtype(m, bound_\Delta(S)) = mtype(m, [\overline{T}/\overline{X}]K)$ for some $K \in \overline{N}$ by MT-SUPER rule. So by reflexivity property of subtyping we have $\Delta, \overline{Y <: \overline{P}} \vdash U_0' <: U_0$. If T is an interface and $m \notin \overline{md}$ cannot occur as every method declaration in the base interface must be given a definition in the class that derives the interface. This property is checked while type checking the definition of the class.

If $m \in \overline{md}$, then by $mtype(m, \overline{N})$, we have

$$mtype(m, \overline{N}) = [\overline{T}/\overline{X}](\texttt{<}\overline{Y \lhd \overline{P'}}\texttt{>}\overline{U'} \to U_0'')$$

where $(\texttt{<}\overline{Y \lhd \overline{P'}}\texttt{>}\overline{U'} \to U_0'') = mtype(m, N_i)$ and $N_i = T$. We can safely assume that $\overline{X}$ and $\overline{Y}$ are distinct and in particular, that $[\overline{T}/\overline{X}]U_0'' = U_0$. By GT-METHOD, it must be the case that

$$\texttt{<}\overline{Y \lhd \overline{P'}}\texttt{>}\ W_0'\ m(\overline{U'}\ \overline{x})\{\ldots\} \in \overline{md} \text{ and}$$
$$\overline{X \lhd \overline{Q}}, \overline{Y \lhd \overline{P'}} \vdash W_0' <: U_0''$$

Now by Lemma 1 and Lemma 3, we have

$$\Delta, \overline{Y <: \overline{P}} \vdash [\overline{T}/\overline{X}]W_0' <: U_0$$

Since $mtype(m, bound_\Delta(S)) = mtype(m, S) = [\overline{T}/\overline{X}]\texttt{<}\overline{Y \lhd \overline{P'}}\texttt{>}\overline{U'} \to W_0'$ by MT-CLASS, letting $U_0' = [\overline{T}/\overline{X}]W_0'$ finishes the case. □

Lemma 5. *If $\Delta \vdash S <: T$ and $fields(getclass(bound_\Delta(T))) = \overline{T}\ \overline{f}$, then $fields(getclass(bound_\Delta(S))) = \overline{S}\ \overline{g}$ and $S_i = T_i$ and $g_i = f_i$ for all $i \leq \#(\overline{f})$.*

Proof. By straightforward induction on the derivation of $\Delta \vdash S <: T$.

The subtyping rules for reflexivity and transitivity, S-REFL and S-TRANS are the same as in FGJ. It is straightforward to see that the lemma holds for those cases. We now provide the proof for all the other cases, S-VAR, S-INTERFACE, S-CLASS.

Case S-VAR: Given $S = X \quad T \in \Delta(X)$. If T is a class, the lemma holds since $getclass(bound_\Delta(S)) <: getclass(bound_\Delta(T))$. The case where T is an interface is trivial as $\#(\overline{f}) = 0$.

Case S-CLASS:

$$S = C\texttt{<}\overline{T}\texttt{>} \quad T \in [\overline{T}/\overline{X}]\overline{N}$$
$$\texttt{class}\ C\texttt{<}\overline{X \lhd \overline{P}}\texttt{>} \lhd \overline{N}\{\overline{S}\ \overline{g}; \ldots\}$$

If T is a class, then $T = K$ for some $K \in \overline{N}$. By rule F-CLASS, $fields(C\texttt{<}\overline{T}\texttt{>}) = \overline{U}\ \overline{f}, [\overline{T}/\overline{X}]\overline{S}\ \overline{g}$ where $\overline{U}\ \overline{f} = fields([\overline{T}/\overline{X}]K)$ for some $K \in \overline{N}$. If T is an interface, the lemma trivially holds as $\#(\overline{f}) = 0$.

Case S-INTERFACE: Since fields cannot be declared inside interface, this lemma trivially holds for this case as $fields(getclass(bound_\Delta(T)))$ and $fields(getclass(bound_\Delta(S)))$ returns an empty sequence.

□

Lemma 6 (Type Substitution preserves Typing)**.** *If* $\Delta_1, \overline{X <: \overline{N}}, \Delta_2; \Gamma \vdash e : T$ *and* $\Delta_1 \vdash \overline{U <: [\overline{U}/\overline{X}]\overline{N}}$ *with* $\Delta_1 \vdash \overline{U}$ *ok and none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e : S$ *for some* S *such that* $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S <: [\overline{U}/\overline{X}]T$.

Proof. By induction on the derivation of $\Delta_1, \overline{X <: \overline{N}}, \Delta_2; \Gamma \vdash \mathrm{e} : \mathrm{T}$ with a case analysis on the last rule used.

Case GT-VAR: Trivial

Case GT-FIELD:

$$e = e_0.f_i \qquad \Delta_1, \overline{X <: \overline{N}}, \Delta_2; \Gamma \vdash e_0 : T_0$$
$$T_0\ not\ interface \qquad fields(getclass(bound_{\Delta_1, \overline{X <: \overline{N}}, \Delta_2}(T_0))) = \overline{T}\ \overline{f} \qquad T = T_i$$

By induction hypothesis, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0 : S_0$ and $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 <: [\overline{U}/\overline{X}]T_0$. Assume $\Delta = \Delta_1, [\overline{U}/\overline{X}]\Delta_2$.

If T_0 is an instance, then $bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}(S_0)$ and $[\overline{U}/\overline{X}]bound_{\Delta_1, \overline{X <: \overline{N}}, \Delta_2}(T_0)$ are the same type. If $T_0 <: U_0$ where $T_0 <: U_0 \in \Delta_1, \Delta_2$, then $bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}(S_0)$ and $[\overline{U}/\overline{X}]bound_{\Delta_1, \overline{X <: \overline{N}}, \Delta_2}(T_0)$ will be the same sequence of types. On the other hand, if T_0 is a type parameter such that $T_0 \in \overline{X}$, then $S_0 = U_i$. This implies that every type T_i in $[\overline{U}/\overline{X}]bound_{\Delta_1, \overline{X <: \overline{N}}, \Delta_2}(T_0)$ must have a type S_i in $bound_{\Delta_1, [\overline{U}/\overline{X}]\Delta_2}(S_0)$ that

is either the same type as T_i or $\Delta \vdash S_i <: T_i$ since U_i must satisfy the constraints imposed by $X_i(T_0)$.

From the above, we can conclude that $getclass(bound_{\Delta_1,\overline{X<:\overline{N}},\Delta_2}(T_0))$ and $getclass(bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0))$ must be same type or $\Delta \vdash getclass(bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)) <: getclass(bound_{\Delta_1,\overline{X<:\overline{N}},\Delta_2}(T_0))$ or empty sequence.

Now by Lemma 5, $fields(getclass(bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0))) = \overline{S}\ \overline{g}$ such that $S_j = T_j$ and $f_j = g_j$ for all j $\leq \#(\overline{T})$. Therefore by the rule GT-FIELD, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0.f_i : S_i$. Letting S $= S_i = ([\overline{U}/\overline{X}]T_i)$ finishes the case.

Case GT-INVK:

$$e = e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e})$$
$$\Delta_1, \overline{X <: \overline{N}}, \Delta_2; \Gamma \vdash e_0 : T_0$$
$$mtype(m, bound_{\Delta_1,\overline{X<:\overline{N}},\Delta_2}(T_0)) = \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{W} \to W_0$$
$$\Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash \overline{V}\ ok \quad \Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash \overline{V <: [\overline{V}/\overline{Y}]\overline{P}}$$
$$\Delta_1, \overline{X <: \overline{N}}, \Delta_2; \Gamma \vdash \overline{e} : \overline{S} \quad \Delta_1, \overline{X <: \overline{N}}, \Delta_2 \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{W}$$
$$T = [\overline{V}/\overline{Y}]W_0$$

Let us consider that the method m is found in the type T_k where $T_k \in bound_{\Delta_1,\overline{X<:\overline{N}},\Delta_2}(T_0)$.

By induction hypothesis,

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0 : S_0$$
$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 <: [\overline{U}/\overline{X}]T_0$$

and

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]\overline{e} : \overline{S'}$$
$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash \overline{S'} <: [\overline{U}/\overline{X}]\overline{S}$$

If T_0 is an instance, then $bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)$ and $[\overline{U}/\overline{X}]bound_{\Delta_1,\overline{X<:\overline{N}},\Delta_2}(T_0)$ are the same type. If $T_0 <: U_0$ where $T_0 <: U_0 \in \Delta_1, \Delta_2$, then $bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)$ and $[\overline{U}/\overline{X}]bound_{\Delta_1,\overline{X<:\overline{N}},\Delta_2}(T_0)$ will be the same sequence of types. On the other hand, if T_0 is a type parameter such that $T_0 \in \overline{X}$, then $S_0 = U_i$. This implies that every type T_i in $[\overline{U}/\overline{X}]bound_{\Delta_1,\overline{X<:\overline{N}},\Delta_2}(T_0)$ must have a type S_i in $bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)$ that is either the same type as T_i or $\Delta \vdash S_i <: T_i$ since U_i must satisfy the constraints imposed by $X_i(T_0)$. By the assumption that methods are not duplicated due to inheritance and by the above argument, the method lookup for m in $bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}(S_0)$ will resolve to a type T_l such that $\Delta \vdash T_l <: T_k$. Now by Lemma 4,

$$mtype(m, T_l) = \texttt{<}\overline{Y \triangleleft [\overline{U}/\overline{X}]\overline{P}}\texttt{>}[\overline{U}/\overline{X}]\overline{W} \to W_0'$$
$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2, \overline{Y <: [\overline{U}/\overline{X}]\overline{P}} \vdash W_0' <: [\overline{U}/\overline{X}]W_0$$

By Lemma 2

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{V} \text{ ok}$$

Without loss of generality, we can assume that $\overline{X}$ and $\overline{Y}$ are distinct and that none of $\overline{Y}$ appear in $\overline{U}$, then $[\overline{U}/\overline{X}][\overline{V}/\overline{Y}] = [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}][\overline{U}/\overline{X}]$. By Lemma 1, we have

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{V <: [\overline{U}/\overline{X}][\overline{V}/\overline{Y}]\overline{P}} \quad (= [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}][\overline{U}/\overline{X}]\overline{P})$$
$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{U}/\overline{X}]\overline{S <: [\overline{U}/\overline{X}][\overline{V}/\overline{Y}]\overline{W}} \quad (= [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}][\overline{U}/\overline{X}]\overline{W})$$

By Lemma 1,

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash [\overline{V}/\overline{Y}]W_0' <: [\overline{U}/\overline{X}][\overline{V}/\overline{Y}]W_0 \quad (= [[\overline{U}/\overline{X}]\overline{V}/\overline{Y}][\overline{U}/\overline{X}]W_0)$$

Finally by the rule GT-INVK,

$$\Delta_1, [\overline{U}/\overline{X}]\Delta_2, [\overline{U}/\overline{X}]\Gamma \vdash ([\overline{U}/\overline{X}]e_0).m\texttt{<}[\overline{U}/\overline{X}]\overline{V}\texttt{>}([\overline{U}/\overline{X}]\overline{d}) : S$$

where $S = [\overline{V}/\overline{Y}]W_0'$ finishes the case.

Case GT-NEW, GT-UCAST: Easy.

Case GT-DCAST:

$$e = (N)e_0 \qquad \Delta = \Delta_1, \overline{X <: \overline{N}}, \Delta_2$$
$$\Delta; \Gamma \vdash e_0 : T_0 \qquad \Delta \vdash N \; ok$$
$$N_0 \; in \; bound_\Delta(T_0) \qquad \Delta \vdash N <: N_0 \qquad N = E\texttt{<}\overline{T}\texttt{>} \qquad N_0 = G\texttt{<}\overline{U}\texttt{>} \qquad dcast(E, G)$$

By induction hypothesis, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0 : S_0$ for some S_0 such that $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 <: [\overline{U}/\overline{X}]T_0$. Let $\Delta' = \Delta_1, [\overline{U}/\overline{X}]\Delta_2$.

Subcase 1: $\Delta' \vdash N_0' <: [\overline{U}/\overline{X}]N$ where $N_0' \in bound_{\Delta'}(S_0)$

By rule GT-UCAST, we have $\Delta'; \Gamma \vdash [\overline{U}/\overline{X}]((N)e_0) : [\overline{U}/\overline{X}]N$

Subcase 2: $\Delta' \vdash [\overline{U}/\overline{X}]N <: N_0'$ where $N_0' \in bound_{\Delta'}(S_0)$ and $[\overline{U}/\overline{X}]N \neq N_0'$.

By rule GT-DCAST and VALID-DOWNCAST, we have $\Delta'; \Gamma \vdash [\overline{U}/\overline{X}]((N)e_0) : [\overline{U}/\overline{X}]N$.

Subcase 3: $\Delta' \vdash [\overline{U}/\overline{X}]N \not<: bound_{\Delta'}(S_0)$ and $\Delta' \vdash bound_{\Delta'}(S_0) \not<: [\overline{U}/\overline{X}]N$.

If none of the type in $bound_{\Delta'}(S_0)$ are related to $[\overline{U}/\overline{X}]N$ implies $name(N) \notin name(N_i)$ and $name(N_i) \notin name(N)$ for all N_i in $bound_{\Delta'}(S_0)$. Now by GT-SCAST rule, we have $\Delta'; \Gamma \vdash [\overline{U}/\overline{X}]((N)e_0) : [\overline{U}/\overline{X}]N$ finishing this case.

Case GT-SCAST:

$$e = (N)e_0 \qquad \Delta = \Delta_1, \overline{X <: \overline{N}}, \Delta_2$$
$$\Delta; \Gamma \vdash e_0 : T_0 \qquad \Delta \vdash N \; ok$$
$$name(N) \notin name(N_i) \qquad name(N_i) \notin name(N)$$
$$\overline{N} = bound_\Delta(T_0) \qquad N = E\texttt{<}\overline{T}\texttt{>} \qquad stupid \; warning$$

By induction hypothesis, $\Delta_1, [\overline{U}/\overline{X}]\Delta_2; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]e_0 : S_0$ for some S_0 such that $\Delta_1, [\overline{U}/\overline{X}]\Delta_2 \vdash S_0 <: [\overline{U}/\overline{X}]T_0$. Let $\Delta' = \Delta_1, [\overline{U}/\overline{X}]\Delta_2$..

If $bound_{\Delta'}(S_0) \not<: [\overline{U}/\overline{X}]N$ and $[\overline{U}/\overline{X}]N \not<: bound_{\Delta'}(S_0)$, i.e., none of the types N_i in $bound_{\Delta'}(S_0)$ is related to $[\overline{U}/\overline{X}]N$, then by rule GT-SCAST we have $\Delta'; [\overline{U}/\overline{X}]\Gamma \vdash [\overline{U}/\overline{X}]((N)e_0) : [\overline{U}/\overline{X}]N$ with a *stupid warning*. On the other hand, if for some type $N_i \in bound_{\Delta'}(S_0)$, $\Delta' \vdash N_i <: [\overline{U}/\overline{X}]N$, then by rule GT-UCAST, we have $\Delta'; [\overline{U}/\overline{X}]\Gamma \vdash$

$[\overline{U}/\overline{X}]((N)e_0) : [\overline{U}/\overline{X}]N$. The case where for some type $N_i \in bound_{\Delta'}(S_0)$, $\Delta' \vdash [\overline{U}/\overline{X}]N <: N_i$ cannot occur as it implies a relationship between $bound_{\Delta}(T_0)$ and N which contradicts the assumption finishing the lemma.

□

Lemma 7 (Term Substitution preserves Typing)**.** *If* $\Delta;\Gamma;\overline{x} : \overline{T} \vdash e : T$ *and* $\Delta;\Gamma \vdash \overline{d} : \overline{S}$ *where* $\Delta \vdash \overline{S} <: \overline{T}$ *then* $\Delta;\Gamma \vdash [\overline{d}/\overline{x}]\ e : S$ *for some* S *such that* $\Delta \vdash S <: T$.

Proof. By induction on the derivation of $\Delta;\Gamma,\overline{x} : \overline{T} \vdash$ e : T with a case analysis on the last rule used.

Case GT-VAR:

$$\Delta,\Gamma \vdash x : \Gamma(x)$$

If $x \in dom(\Gamma)$, then the conclusion is immediate as the substitution do not have effect and hence $[\overline{d}/\overline{x}]x = x$. On the other hand, if $x = x_i$ and $T = T_i$, then x_i must be substituted by $d_i : S_i$ and from given we have $\Delta \vdash S_i <: T_i$ which implies $\Delta \vdash S <: T$ finishing the case.

Case GT-FIELD:

$$e = e_0.f_i \qquad \Delta,\Gamma,\overline{x} : \overline{T} \vdash e_0 : T_0$$
$$T_0\ not\ interface \qquad fields(getclass(bound_{\Delta}(T_0))) = \overline{T}\ \overline{f} \qquad T = T_i$$

By induction hypothesis, $\Delta,\Gamma \vdash [\overline{d}/\overline{x}]e_0 : S_0$ for some S_0 such that $\Delta \vdash S_0 <: T_0$. The case where T_0 is a type of interface is vacuously true. Now by Lemma 5, $fields(getclass(bound(S_0))) = \overline{S}\ \overline{g}$ such that $S_j = T_j$ and $f_j = g_j$ for all j $\leq \#(\overline{T})$. Therefore by the rule GT-FIELD, $\Delta;\Gamma \vdash [\overline{d}/\overline{x}]e_0.f_i : T$.

Case GT-INVK:

$$e = e_0.m\langle\overline{V}\rangle(\overline{e}) \qquad \Delta,\Gamma,\overline{x} : \overline{T} \vdash e_0 : T_0$$
$$mtype(m, bound_{\Delta}(T_0)) = \langle\overline{Y \lhd \overline{P}}\rangle\overline{U} \to U$$

$$\Delta \vdash \overline{V}\ ok \quad \Delta \vdash \overline{V <: [\overline{V}/\overline{Y}]\overline{P}}$$
$$\Delta; \Gamma \vdash \overline{e} : \overline{S} \quad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}$$
$$T = [\overline{V}/\overline{Y}]U$$

By induction hypothesis, $\Delta, \Gamma \vdash [\overline{d}/\overline{x}]e_0 : S_0$ for some S_0 such that $\Delta \vdash S_0 <: T_0$ and $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]\overline{e} : \overline{W}$ for some $\overline{W}$ such that $\Delta \vdash \overline{W} <: \overline{S}$. By Lemma 4, $mtype(m, bound(S_0)) = \texttt{<}\overline{Y \lhd \overline{P}}\texttt{>}\overline{U} \to U'$ and $\Delta, \overline{Y <: \overline{P}} \vdash U' <: U$. By Lemma 1, we have $\Delta \vdash [\overline{V}/\overline{Y}]U' <: [\overline{V}/\overline{Y}]U$. By the rule GT-INVK, $\Delta, \Gamma \vdash [\overline{d}/\overline{x}](e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V}/\overline{Y}]U'$. Letting $S = [\overline{V}/\overline{Y}]U'$. finishes the case.

Case GT-NEW:

$$e = \texttt{new}\ K(\overline{e}) \qquad \Delta; \Gamma, \overline{x} : \overline{T} \vdash \overline{e} : \overline{S}$$
$$\Delta \vdash K\ ok \qquad fields(K) = \overline{T}\ \overline{f} \qquad \Delta \vdash \overline{S} <: \overline{T}$$
$$T = K$$

By induction hypothesis, $\Delta; \Gamma, \overline{x} : \overline{T} \vdash [\overline{d}/\overline{x}]\overline{e} : \overline{W}$ such that $\Delta \vdash \overline{W} <: \overline{S}$. By transitivity of subtype relation, we have $\Delta \vdash \overline{W} <: \overline{T}$. Now by GT-NEW, $\Delta, \Gamma \vdash \texttt{new}\ K([\overline{d}/\overline{x}]\overline{e})$: K, finishing this case.

Case GT-UCAST:

$$e = (N)e_0 \quad \Delta; \Gamma \vdash e_0 : T_0$$
$$N_0\ in\ bound_\Delta(T_0) \quad \Delta \vdash N_0 <: N \quad T = N$$

By induction hypothesis, $\Delta, \Gamma \vdash [\overline{d}/\overline{x}]e_0 : S_0$ for some S_0 such that $\Delta \vdash S_0 <: T_0$. Now from the GT-UCAST rule we have, $\Delta \vdash T_0 <: N_0$. By transitivity of subtype relation, this implies $\Delta \vdash S_0 <: N_0$ and $\Delta \vdash S_0 <: N$. Now by GT-UCAST rule, $e = (N)[\overline{d}/\overline{x}]e_0$: N, finishing the proof.

Case GT-DCAST:

$$e = (N)e_0 \quad \Delta; \Gamma, \overline{x} : \overline{T} \vdash e_0 : T_0 \quad \Delta \vdash N\ ok$$

$$N_0 \; in \; bound_\Delta(T_0) \quad \Delta \vdash N <: N_0 \quad N = E\texttt{<}\overline{T}\texttt{>} \quad N_0 = G\texttt{<}\overline{U}\texttt{>} \quad dcast(E, F)$$

By induction hypothesis, $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]e_0 : S_0$ for some S_0 such that $\Delta \vdash S_0 <: T_0$. Now we have three subcases according to the relation between S_0 and N.

Subcase 1: $\Delta \vdash N_0' <: N$ where $N_0' \in bound_\Delta(S_0)$

By rule GT-UCAST, we have $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]((N)e_0) : N$

Subcase 2: $\Delta \vdash N <: N_0'$ where $N_0' \in bound_\Delta(S_0) \quad N \neq N_0'$.

By rule GT-DCAST and VALID-DOWNCAST, we have $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]((N)e_0) : N$.

Subcase 3: $\Delta \vdash N \not<: bound_\Delta(S_0)$ and $\Delta \vdash bound_\Delta(S_0) \not<: N$.

If none of the type in $bound_\Delta(S_0)$ are related to N implies $name(N) \notin name(N_i)$ and $name(N_i) \notin name(N)$ for all i in $bound_\Delta(S_0)$. Now by GT-SCAST rule, we have $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]((N)e_0) : N$ with a *stupid warning* finishing this case.

Case GT-SCAST:

$$e = (N)e_0 \quad \Delta; \Gamma, \overline{x} : \overline{T} \vdash e_0 : T_0 \quad \Delta \vdash N \; ok$$
$$name(N) \notin name(N_i) \quad name(N_i) \notin name(N)$$
$$\overline{N} = bound_\Delta(T_0) \quad N = E\texttt{<}\overline{T}\texttt{>} \quad stupid \; warning$$

By induction hypothesis, $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]e_0 : S_0$ for some S_0 such that $\Delta \vdash S_0 <: T_0$.

If $bound_\Delta(S_0) \not<: N$ and $N \not<: bound_\Delta(S_0)$, (i.e)., none of the types N_i in $bound_\Delta(S_0)$ is related to N, then by rule GT-SCAST we have $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]((N)e_0) : N$ with a *stupid warning*. On the other hand, if for some type $N_i \in bound_\Delta(S_0)$, $\Delta \vdash N_i <: N$, then by rule GT-UCAST, we have $\Delta; \Gamma \vdash [\overline{d}/\overline{x}]((N)e_0) : N$. The case where for some type $N_i \in bound_\Delta(S_0)$, $\Delta \vdash [\overline{d}/\overline{x}]N <: N_i$ cannot occur as it implies a relationship between $bound_\Delta(T_0)$ and N which contradicts the assumption finishing the lemma.

□

Lemma 8. *If* $mtype(m, C\langle\overline{T}\rangle) = \langle\overline{Y \lhd \overline{P}}\rangle\overline{U} \to U$ *and* $mbody(m\langle\overline{V}\rangle, C\langle\overline{T}\rangle) = \overline{x}.e_0$ *where* $\Delta \vdash C\langle\overline{T}\rangle$ *ok and* $\Delta \vdash \overline{V}$ *ok and* $\Delta \vdash \overline{V <: [\overline{V}/\overline{Y}]\overline{P}}$*, then there exists some* K *and* S *such that* $\Delta \vdash C\langle\overline{T}\rangle <: K$ *and* $\Delta \vdash K$ *ok and* $\Delta \vdash S <: [\overline{V}/\overline{Y}]U$ *and* $\Delta \vdash S$ *ok and* $\Delta; \overline{x} : [\overline{V}/\overline{Y}]\overline{U}, this : K \vdash e_0 : S$.

Proof. By induction on the derivation of $mbody(m\langle\overline{V}\rangle, C\langle\overline{T}\rangle) = \overline{x}.e$ using Lemma 1 and 6. The MB-CLASS and MB-SUPER rules did not change from FGJ and hence the proof of this lemma directly falls out from the proof given in [5]. We now outline the proof paraphrasing the proof given in [5].

Case MB-CLASS:

$$\texttt{class}\ C\langle\overline{X \lhd \overline{P}}\rangle \lhd \overline{N}\{\ldots \overline{md}\}$$
$$\langle\overline{Y \lhd \overline{Q}}\rangle\ T_0\ m(\overline{S}\ \overline{x})\{return\ e;\} \in \overline{md}$$

Let $\Gamma = \overline{x} : \overline{S}, this : C\langle\overline{X}\rangle$ and $\Delta' = \overline{X <: \overline{P}}, \overline{Y <: \overline{Q}}$. By the rules GT-CLASS and GT-METHOD, we have $\Delta'; \Gamma \vdash e : S_0$ and $\Delta'; \Gamma \vdash S_0 <: T_0$ for some S_0. Since $\Delta \vdash C\langle\overline{T}\rangle ok$, we have $\Delta \vdash \overline{T <: [\overline{T}/\overline{X}]\overline{P}}$ by rule WF-CLASS. By Lemmas 3, 1 and 6, we have

$$\Delta, \overline{Y <: [\overline{T}/\overline{X}]\overline{Q}} \vdash [\overline{T}/\overline{X}]S_0 <: [\overline{T}/\overline{X}]T_0$$
$$\Delta, \overline{Y <: [\overline{T}/\overline{X}]\overline{Q}}; \overline{x} : [\overline{T}/\overline{X}]\overline{S}, this : C\langle\overline{T}\rangle \vdash [\overline{T}/\overline{X}]e : S_0'$$
$$\Delta, \overline{Y <: [\overline{T}/\overline{X}]\overline{Q}} \vdash S_0' <: [\overline{T}/\overline{X}]S_0$$

Now we can assume $\overline{X}$ and $\overline{Y}$ are distinct without loss of generality. By the rule MT-CLASS, we have

$$[\overline{T}/\overline{X}]\overline{Q} = \overline{P} \quad [\overline{T}/\overline{X}]\overline{S} = \overline{U} \quad [\overline{T}/\overline{X}]T_0 = U$$

Again, by rule S-TRANS and Lemmas 1 and 6,

$$\Delta \vdash [\overline{V}/\overline{Y}]S_0' <: [\overline{V}/\overline{Y}]U$$

$$\Delta; \overline{x} : [\overline{V}/\overline{Y}]\overline{U}, this : C\texttt{<}\overline{T}\texttt{>} \vdash [\overline{V}/\overline{Y}][\overline{T}/\overline{X}]e : \ S_0''$$
$$\Delta \vdash S_0'' <: [\overline{V}/\overline{Y}]S_0'$$

Since we can assume that any of $\overline{Y}$ does not occur in $\overline{T}$ without loss of generality,

$$e_0 = [\overline{T}/\overline{X}, \overline{V}/\overline{Y}]e = [\overline{V}/\overline{Y}][\overline{T}/\overline{X}]e$$

Letting $K = C\texttt{<}\overline{T}\texttt{>}$ and $S = S_0''$ finishes the case.

Case MB-SUPER:

$$\texttt{class}\ C\texttt{<}\overline{X \triangleleft \overline{P}}\texttt{>} \triangleleft \overline{N}\{\ldots \overline{md}\} \quad m \notin \overline{md}$$

Immediate from the induction hypothesis and the fact that $\Delta \vdash C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]K$ for some $K \in \overline{N}$.

□

Theorem 1 (Subject Reduction)**.** *If* $\Delta; \Gamma \vdash e : T$ *and* $e \rightarrow e'$*, then* $\Delta; \Gamma \vdash e' : T'$*, for some* T' *such that* $\Delta \vdash T' <: T$*.*

Proof. By induction on derivation of $e \rightarrow e'$ with a case analysis of the reduction rule used. The reduction rules did not change from FGJ and hence the proof of this theorem directly falls out from the proof given in [5]. We now outline the proof paraphrasing the proof given in [5].

Case GR-FIELD:

$$e = \texttt{new}\ K(\overline{e}).f_i \qquad fields(K) = \overline{T}\ \overline{f}$$
$$e' = e_i \qquad T = T_i$$

By rule GT-FIELD and GT-NEW, we have

$$\Delta, \Gamma \vdash \texttt{new}\ K(\overline{e}) : K$$
$$\Delta, \Gamma \vdash \overline{e} : \overline{S} \quad \Delta \vdash \overline{S} <: \overline{T}$$

Now in $\Delta, \Gamma \vdash e_i : S_i$ and $\Delta \vdash S_i <: T_i$. Letting $T' = S_i$ and $T = T_i$ finishes the case.

Case GR-INVK:

$$mbody(m\texttt{<}\overline{V}\texttt{>},\ K)\ =\ \overline{x}.e_0$$

$$e = \texttt{new}\ K(\overline{e}).m\texttt{<}\overline{V}\texttt{>}(\overline{d}) \quad e' = [\overline{d}/\overline{x}, \texttt{new}\ K(\overline{e})/this]e_0$$

By rules GT-INVK and GT-NEW, we have

$$\Delta, \Gamma \vdash \texttt{new}\ K(\overline{e}) : K$$

$$mtype(m, bound_\Delta(K)) = \texttt{<}\overline{Y \lhd \overline{P}}\texttt{>}\overline{U} \to U$$

$$\Delta \vdash \overline{V}\ ok \quad \Delta \vdash \overline{V <: [\overline{V}/\overline{Y}]\overline{P}}$$

$$\Delta; \Gamma \vdash \overline{d} : \overline{S} \quad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{U}$$

$$T = [\overline{V}/\overline{Y}]U \quad \Delta \vdash K\ ok$$

By Lemma 8, $\Delta; \overline{x}; [\overline{V}/\overline{Y}]\overline{U}, this : P \vdash e_0 :$ S for some P and S such that $\Delta \vdash K <: P$ where $\Delta \vdash P$ ok and $\Delta \vdash S <: [\overline{V}/\overline{Y}]U$ where $\Delta \vdash S$ ok. We know that term substitution preserves typing by Lemma 7 and hence $\Delta; \Gamma \vdash [\overline{d}/\overline{x}, \texttt{new}\ K(\overline{e})/this]e_0 : T_0$ for some T_0 such that $\Delta \vdash T_0 <: S$. Now by transitivity of the subtyping relation, we have $\Delta \vdash T_0 <: [\overline{V}/\overline{Y}]U$. Letting $T' = T_0$ and $T = [\overline{V}/\overline{Y}]U$ finishes the case with $\Delta \vdash T' <: T$.

Case GR-CAST:

$$\emptyset \vdash K <: P$$

$$e = (P)(\texttt{new}\ K(\overline{e}))\ \ e' = \texttt{new}\ K(\overline{e})$$

By rules GT-UCAST and GT-NEW, we have

$$\Delta; \Gamma \vdash \texttt{new}\ K(\overline{e}) : K$$

$$N_0\ in\ bound_\Delta(K) \quad \Delta \vdash N_0 <: P$$

and hence T = P. Now $T' = K$ since the type of `new` $K(\overline{e}) : K$ by GT-NEW. We also know that $\Delta; \Gamma \vdash K <: N_0$. Then by transitivity of $<:$, we have $K <: P$, which implies $\Delta \vdash T' <: T$ finishing the case.

Case GRC-FIELD:

$$e = e_0.f \quad e' = e'_0.f \quad e_0 \to e'_0$$

By rule GT-FIELD, we have

$$\Delta, \Gamma \vdash e_0 : T_0 \quad T_0 \; not \; interface$$
$$fields(getclass(bound_\Delta(T_0))) = \overline{T} \; \overline{f} \quad T = T_i$$

By induction hypothesis, we have $\Delta, \Gamma \vdash e'_0 : T'_0$ for some T'_0 such that $T'_0 <: T_0$. Now by Lemma 5, we have $fields(getclass(bound_\Delta(T'_0))) = \overline{T'} \; \overline{g}$ and for some $j \leq \#(\overline{f})$, we have $g_j = f_j$ and $T'_j = T_j$. Therefore by rule GT-FIELD, $\Delta; \Gamma \vdash e'_0.f : T'_j$ which is the type of the field accessed from e_0. Hence $T' <: T$ by reflexivity of the subtyping relation.

Case GRC-INVK-RECV:

$$e = e_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) \quad e' = e'_0.m\texttt{<}\overline{V}\texttt{>}(\overline{e})$$
$$e_0 \to e'_0$$

By rule GT-INVK, we have

$$\Delta, \Gamma \vdash e_0 : T_0$$
$$mtype(m, bound_\Delta(T_0)) = \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{T} \to U$$
$$\Delta \vdash \overline{V} \; ok \quad \Delta \vdash \overline{V <: [\overline{V}/\overline{Y}]\overline{P}}$$
$$\Delta; \Gamma \vdash \overline{e} : \overline{S} \quad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{T}$$
$$T = [\overline{V}/\overline{Y}]U$$

By induction hypothesis, we have $\Delta; \Gamma \vdash e'_0 : T'_0$ for some T'_0 such that $T'_0 <: T_0$. Now by Lemma 4, we have $mtype(m, bound_\Delta(T'_0)) = \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{T} \to V$ and in

$\Delta, \overline{Y <: \overline{P}} \vdash V <: U$. Since we already proved that type substitution preserves subtyping by Lemma 1, we have $\Delta \vdash [\overline{V}/\overline{Y}]V <: [\overline{V}/\overline{Y}]U$. Therefore by GT-INVK, $\Delta, \Gamma \vdash e_0'.m\texttt{<}\overline{V}\texttt{>}(\overline{e}) : [\overline{V}/\overline{Y}]V$. Letting $T' = [\overline{V}/\overline{Y}]V$ implies that $\Delta \vdash T' <: T$ which finishes the case.

Case GRC-INVK-ARG:

$$e = e_0.m\texttt{<}\overline{V}\texttt{>}(....., e_i, ...)$$
$$e' = e_0.m\texttt{<}\overline{V}\texttt{>}(....., e_i', ...) \qquad e_i \rightarrow e_i'$$

By rule GT-INVK, we have

$$\Delta, \Gamma \vdash e_0 : T_0 \qquad mtype(m, bound_\Delta(T_0)) = \texttt{<}\overline{Y \triangleleft \overline{P}}\texttt{>}\overline{T} \rightarrow U$$
$$\Delta \vdash \overline{V}\ ok \quad \Delta \vdash \overline{V <: [\overline{V}/\overline{Y}]\overline{P}}$$
$$\Delta; \Gamma \vdash \overline{e} : \overline{S} \quad \Delta \vdash \overline{S} <: [\overline{V}/\overline{Y}]\overline{T}$$
$$T = [\overline{V}/\overline{Y}]U \quad \Delta; \Gamma \vdash e_i : T_i$$

By induction hypothesis, we have $\Delta; \Gamma \vdash e_i' : T_i'$ such that $\Delta \vdash T_i' <: T_i$. Since a method takes any argument that is a subtype of $\overline{T}$, we can pass $\overline{T'}$ to the method without affecting the type of the return type (i.e)., the type of the method body. Therefore $T' = [\overline{V}/\overline{Y}]U$ which is a subtype of T by reflexive property of subtyping relation.

Case GRC-CAST:

$$e = (N)e_0 \qquad e_0 \rightarrow e_0' \qquad e' = (N)e_0'$$

There are three subcases according to the last typing rule: GT-UCAST, GT-DCAST, GT-SCAST. These subcases are similar to the subcases in the case for GT-DCAST in the proof of Lemma 7.

Case GRC-NEW-ARG: Similar argument as GRC-INVK-ARG

□

Theorem 2 (Progress)**.** *Suppose e is a well-typed expression.*

1. If e includes new $K_0(\overline{e}).f$ *as sub-expression, then fields(*K_0*)* $= \overline{T}\ \overline{f}$ *and f in* $\overline{f}$ *for some* $\overline{T}$ *and* $\overline{f}$

2. If e includes new $K_0(\overline{e}).m\langle\overline{V}\rangle(\overline{d})$ *as sub-expression, then mbody(*$m\langle\overline{V}\rangle, K_0$*)* $= \overline{x}.e_0$ *and* $\#(\overline{x}) = \#(\overline{d})$ *for some* $\overline{x}$ *and* e_0

Proof. The reduction rules of FGJ did not change because of the inclusion of interface to the formalism and hence the progress property trivially holds for FGJ+I. So we outline the entire proof of this theorem as given in [5]. If e has *new* $K_0(\overline{e}).f$ as sub-expression, then by well-typedness of the subexpression, it is simple to check that $fields(K_0)$ are well defined and f appears in it. On a similar line, if e has *new* $K_0(\overline{e}).\mathrm{m}\langle\overline{V}\rangle(\overline{d})$ as sub-expression, then we have $mbody(m\langle\overline{V}\rangle, K_0) = \overline{x}.e_0$ and it is simple to show that $\#(\overline{x}) = \#(\overline{d})$ from the fact that $mtype(m, K_0) = \overline{U} \rightarrow U$ where $\#(\overline{x}) = \#(\overline{U})$ □

To state the type soundness formally, we give the definition of values, given by the following syntax : $v := \texttt{new}\ K(\overline{w})$.

Theorem 3 (FGJ+I Type Soundness)**.** *If* $\emptyset;\emptyset \vdash e : T$ *and* $e \rightarrow^* e'$ *with e' a normal form, then e' is either (1) a FGJ+I value w with* $\emptyset;\emptyset \vdash w : S$ *and* $\emptyset \vdash S <: T$ *or an expression containing (P) new K(*$\overline{e}$*) where* $\emptyset \vdash K \not<: P$.

Proof. Immediate from the progress and preservation properties. □

CHAPTER VII

RELATED WORK

The earliest approach to member types was *Beta*'s virtual types. Variations of virtual types has appeared in the literature, e.g., virtual types proposed to Java as a language feature [4,18]. Unlike our approach, virtual types associate member types with objects rather than classes. Without adding restrictions, run-time type checking is necessary to ensure full type safety in that approach. By introducing suitable restrictions (see e.g. [19]) total or partial static type safety can be guaranteed. These restrictions are similar to the requirement we impose that associated types are not redefined in subclasses of the class in which they are defined. Indeed, associated types as proposed in this thesis can be viewed as type-safe variation of virtual types.

Often virtual types are viewed as an alternative to parameterized types; e.g., virtual types as described in [4,18] do not include type parameterization. Thorup and Torgersen [20] argue that type parameterization is beneficial even in the presence of virtual types. We agree with this view — this thesis combines type parameterization and associated types, and advocates the importance of constraint propagation in this combination. The translation implemented in this thesis precisely describes a correspondence between parameterized types and associated types in Generic C# and allows both of these language features to coexist rather than one replacing the other.

Various related formalisms and language features for associated types have been developed. *Nested inheritance* [21] is a Java extension that can be translated to standard Java with techniques similar to those implemented in this thesis. Nested inheritance closely follows our approach to associated types by associating nested member types with classes rather than objects. Unlike our approach, nested inheri-

tance does not support parameterized types, and nested types can only be bound to newly defined classes, not to existing types.

Virtual Classes [18] are a language mechanism that allows part of the specification of the class attributes to be deferred to a subclass. Virtual classes were first introduced in *Beta*. Similar to the approach taken in our extensions to C#, virtual classes cannot be redefined in a subclass, but the definition may be extended. Recently, a formal object calculus *Deep*, which implements virtual classes in a type safe manner, was introduced [22]. *Deep*'s type system is based on *prototypes* which blurs the distinction between compile and run time. A *vc* calculus that captures the essence of virtual classes and a proof of soundness of *vc* is presented in [23]. Similar to virtual types, and unlike our approach, virtual classes are bound to objects rather a class.

Support for member types is also found in various languages including C++, ML, Haskell, and Scala. C++ supports member types through **typedefs** inside classes. Such typedefs are used to implement *traits classes* [24], which are an approach to representing associated types. With the recent proposal for concepts as a first class language feature for C++ [25], associated types are directly supported in the language. *ConceptGCC* provides a prototype implementation of associated types, and constraint propagation, for C++ [26].

Haskell's *functional dependencies* among type classes' parameters offer partial support for associated types. Further, a recent extension of associated type synonyms [27,28] to Haskell type classes allows a type class declaration to define member types. An instance declaration gives a definition for these member types in a way similar to how classes provide definition for associated types in our proposed extension. Similar to our approach of implementing associated types, these extensions to Haskell can be implemented in an existing Haskell compiler with changes only to the front-end. ML

supports associated types through *nested type* declarations in ML signatures but not in combination with parameterized types.

Scala [29] supports member types to be declared inside *traits* (these are unrelated to C++ traits classes), which are essentially interfaces that allow methods with default implementations. Scala's type system is modeled after the μObj calculus [30], which formally examines the properties and behavior of nested types. Member types of Scala are associated to objects rather than classes. In Scala, *same-type* constraints can only be specified between a type parameter and a member type and not between two member types. This requires introducing type parameters for all member types that are involved in *same-type* constraints. This round-about manner of specifying *same-type* constraints has been reported to lead to verbose constraints in generic methods [31].

In contrast to systems related to associated types, work on constraint propagation is infrequent in the literature. The *concepts* mechanism for C++ supports constraint propagation [25]. Java's wildcards [32] allow a limited form of constraint propagation. Scala partially supports constraint propagation when traits extend other traits. This is, however, of limited usefulness: type parameters must be introduced for member types that are involved in *same-type* constraints, and the constraints on member types are not implicitly propagated as constraints on type parameters representing these member types. In practice, this requires generic components to repeat constraints on member types [31]. Except the *concepts* proposal for C++ [25], we are not aware of any work combining associated types or type members with constraint propagation in object oriented languages.

There are several works in the literature proving safety properties for subsets of Java. A proof of type soundness for a fairly large subset of sequential Java including interfaces but not generic interfaces is provided in [33]. Like FGJ, this formalism

follows the small step semantics but omits the intricate details of stupid casts. Proof of soundness for a larger subset of Java formulating the formalism in terms of big step semantics was given in [34]. Neither of these formalism consider generic Java with interfaces. A number of extensions to Java with generic classes, generic interfaces and generic methods have been proposed [17, 35, 36]. While all these languages are believed to be typesafe, FGJ was the first formalism, including a type safety proof for a subset of Java with generics. FGJ omits interfaces for the sake of compactness. To our knowledge, a type safety proof for a subset of generic Java including interfaces has not been reported, a gap filled in this thesis.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

Developing generic libraries in C# stretches the practical limits of its generics facilities due to the high degree of parameterization on types, typical of generic libraries. In particular, lack of direct support for associated types and constraint propagation leads to redundant and verbose code. This issue was partially addressed in [3], where associated types and constraint propagation were proposed as extensions to C#, mostly to avoid redundant and verbose code. In this thesis, we extend this work by taking the proposal a step further by developing an initial version of C# compiler supporting associated types and constraint propagation. Our approach to implement these two language extensions is through translation at the abstract syntax tree level of programs written with the extensions to current C#. Major contribution of this thesis is the design and development of a framework of algorithms for implementing constraint propagation and translation of associated types. Besides taking into consideration many features intentionally left out from the treatment in [3], the implementation of these language extensions in a real compiler revealed corner cases that were not properly handled in the translation developed in [3].

With this initial implementation, we are able to evaluate the impact of the proposed extensions to developing generic libraries. To demonstrate, we implemented a subset of a state-of-the art generic library, the Boost Graph Library, which also served as a testbed for the implementation framework. The experimental results confirmed that the verbosity of the code reduces considerably if the proposed extensions can be used. This assures that the language extensions implemented improve the generic programming experience in C#.

This thesis complements the work of [3] in another direction as well. We present a proof of type safety for the Featherweight Generic Java extended with interfaces (FGJ+I). The formal treatment of [3] was based on FGJ+I, but its type safety has not been previously established formally.

There are three directions in which we have plans to extend this work. First, improving the existing implementation closer to a production level compiler and releasing it to the *Mono* open source community. This will enable the evaluation of the true impact of these language extensions. Second, to provide support for type aliasing, a mechanism to provide an alternative name for a type. C# provides type aliasing to some extent with the **using** clause, but it cannot be used within a type or a method. The support for associated types, especially the same-type constraints introduced within type declarations, can be extended to provide language level support for type aliasing. A simple example of type aliasing, with the syntax of the associated types extension proposed in this thesis is given in Figure 31. We believe this feature can be added seamlessly to the existing framework for associated type translation.

```
interface B<T, X, Y, V> { ... }
interface A<T, X, Y, V> {
        type X1 == B<T, X, Y, V> ;
}
```

Fig. 31. Example of type aliasing. Further usage of X1 in the interface will be translated to B<T, X, Y, V> as the associated type translation favors the instances to associated type during the canonicalization process.

Third, we plan to strengthen the support for implicit instantiation in C#, which allows type argument deduction based on the constraints on type parameters of a generic method. This will eliminate, apart from pathological cases, the need to explicitly specify the argument types during generic method invocations, which one

often must do in current C#. Type aliases and implicit instantiation along with the extensions proposed in this thesis will improve the support for generic programming paradigm in C#.

REFERENCES

[1] Microsoft Corporation, C# version 2.0 specification, Available at http://msdn.microsoft.com/vcsharp/programming/language/, March 2005.

[2] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, J. Willcock, A comparative study of language support for generic programming, in: Proceedings of the 18th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 2003, pp. 115–134.

[3] J. Järvi, J. Willcock, A. Lumsdaine, Associated types and constraint propagation for mainstream object-oriented generics, in: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 2005, pp. 1–19.

[4] K. K. Thorup, Genericity in Java with virtual types, in: European Conference on Object-Oriented Programming, Vol. 1241, Springer Verlag, 1997, pp. 444–471.

[5] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.

[6] Mono .NET framework, Available from http://www.mono-project.com/, 2004.

[7] M. Jazayeri, R. Loos, D. R. Musser (Eds.), Generic programming, in: Selected papers from International Seminar on Generic Programming, Vol. 1766 of Lecture Notes in Computer Science, pp. 1-11, Springer Verlag, 1998.

[8] A. A. Stepanov, M. Lee, The Standard Template Library, Tech Rep., Hewlett Packard Laboratories, Available at http://www.hpl.hp.com/techreports (1994).

[9] J. G. Siek, L. Q. Lee, A. Lumsdaine, The Boost Graph Library, User Guide and Reference Manual, Addison-Wesley, 2002.

[10] J. G. Siek, A. Lumsdaine, The matrix template library: A generic programming approach to high performance numerical linear algebra, in: International Symposium on Computing in Object-Oriented Parallel Environments, Vol. 1505, Springer Verlag, 1998, pp. 59–70.

[11] M. H. Austern, Generic Programming and STL, Professional Computing Series, Addison Wesley, 1999.

[12] H.P.Barendredgt, The Lambda Calculus, revised ed., North Holland, Amsterdam, The Netherlands, 1984.

[13] B. C. Pierce, Types and Programming Languages, MIT Press, 2002.

[14] Microsoft .NET framework, Available at http://msdn.microsoft.com/netframework/, 2004.

[15] DotGNU Project, Available from http://www.dotgnu.org/, 2004.

[16] SGI Standard Template Library, Available at http://www.sgi.com/tech/stl/, 2004.

[17] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler, Making the future safe for the past: Adding genericity to the Java programming language, in: Proceedings of the 13th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 1998, pp. 183–200.

[18] O. L. Madsen, B. Moller-Pedersen, Virtual classes: A powerful mechanism in object-oriented programming, in: Proceedings of the ACM SIGPLAN Conference

on Object Oriented Programming, Systems, Languages, and Applications, 1989, pp. 397–406.

[19] M. Torgersen, Virtual types are statically safe, in: FOOL 5: Workshop on Foundations of Object-Oriented Languages, 1998, pp. 1–9.

[20] K. K. Thorup, M. Torgersen, Unifying genericity - combining the benefits of virtual types and parameterized classes, in: Proceedings of the 13th European Conference on Object-Oriented Programming, 1999, pp. 186–204.

[21] N. Nystrom, S. Chong, A. C. Myers, Scalable extensibility via nested inheritance, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 2004, pp. 99–115.

[22] D. Hutchins, Eliminating distinctions of class: Using prototypes to model virtual classes, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 2006, pp. 1–20.

[23] E. Ernst, K. Ostermann, W. R. Cook, A virtual class calculus, in: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2006, pp. 270–282.

[24] N. Myers, A new and useful template technique: Traits, in: C++ Gems, Sigs Reference Library Series, 1996, pp. 451–457.

[25] D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, A. Lumsdaine, Concepts: Linguistic support for generic programming in C++, in: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 2006, pp. 291–310.

[26] Douglas Gregor, ConceptGCC, Accessed on http://www.generic-programming.org/software/conceptgcc/, December 2006.

[27] M. M. T. Chakravarty, G. Keller, S. P. Jones, S. Marlow, Associated types with class, in: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2005, pp. 1–13.

[28] M. M. T. Chakravarty, G. Keller, S. P. Jones, Associated type synonyms, in: Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, 2005, pp. 241–253.

[29] M. Odersky, P. Altherr, V. Cremet, B. Emir, G. Dubochet, et al., The Scala Language Specification, Programming Methods Laboratory, EPFL, Available at http://scala.epfl.ch/docu/files/ScalaReference.pdf, 2005.

[30] M. Odersky, V. Cremet, C. Röckl, M. Zenger, A nominal theory of objects with dependent types, in: Proceedings of the European Conference on Object Oriented Programming, Vol. 2743, 2003, pp. 201–224.

[31] S. O. N'Guessan, Generic programming in Scala, Master's thesis, Texas A&M Univeristy (2006).

[32] M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahe, G. Bracha, N. Gafter, Adding wildcards to the Java programming language, in: Proceedings of the 2004 ACM Symposium on Applied Computing, 2004, pp. 1289–1296.

[33] S. Drossopoulou, S. Eisenbach, S. Khurshid, Is the Java type system sound?, Theory and Practice of Object Systems 5 (1) (1999) 3–24.

[34] T. Nipkow, D. V. Oheimb, Javalight is type-safe definitely, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1998, pp. 161–170.

[35] O. Agesen, S. N. Freund, J. C. Mitchell, Adding type parameterization to the Java language, in: Proceedings of the 12th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, 1997, pp. 49–65.

[36] M. Odersky, P. Wadler, Pizza into Java: Translating theory into practice, in: Proceedings of the 24th ACM Symposium on Principles of Programming Languages, 1997, pp. 146–159.

VITA

Aravind Srinivasa Raghavan was born in Chingleput, Tamil Nadu, India. He received a Bachelor of Engineering degree from University of Madras specializing in computer science in April 2004. He began pursuing his Master of Science degree in computer science at Texas A&M University in the fall 2004. Initially he worked as a graduate teaching assistant for Dr. Marian Eriksson and then joined as a graduate research assistant under Dr. Jaakko Järvi. Starting March 2007, he will be working with Cisco Systems, San Jose, CA. He can be reached at aravindraghavan@gmail.com. His permanent address is 80, Kumaran Nagar, Viswas Sai Homes, Second Main Road, Virugambakkam, Chennai, Tamil Nadu - 600092, India.