Associated Types and Constraint Propagation for Mainstream Object-Oriented Generics

Jaakko Järvi
Texas A&M University,
Computer Science
College Station, TX 77843
jarvi@cs.tamu.edu

Jeremiah Willcock
Indiana University,
Open Systems Lab
Bloomington, IN 47405
jewillco@osl.iu.edu

Andrew Lumsdaine
Indiana University,
Open Systems Lab
Bloomington, IN 47405
lums@osl.iu.edu

ABSTRACT

Support for object-oriented programming has become an integral part of mainstream languages, and more recently generic programming has gained widespread acceptance as well. A natural question is how these two paradigms, and their underlying language mechanisms, should interact. One particular design option, that of using subtyping to constrain the type parameters of generic functions, has been chosen in the generics of Java and those planned for a future revision of C#. Certain shortcomings have previously been identified in using subtyping for constraining parametric polymorphism in the context of generic programming. To address these, we propose extending object-oriented interfaces and subtyping to include associated types and constraint propagation. Associated types are type members of interfaces and classes. Constraint propagation allows certain constraints on type parameters to be inferred from other constraints on those parameters and their use in base class type expressions. The paper demonstrates these extensions in the context of C# (with generics), describes a translation of the extended features to C#, and presents a formalism proving their safety. The formalism is applicable to other mainstream object-oriented languages supporting F-bounded polymorphism, such as Java.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects, polymorphism*; D.3.2 [**Programming Languages**]: Language Classifications—*C#, Java*

General Terms

Languages

Keywords

Generics, generic programming, constraint propagation, associated types, C#, Java

1. INTRODUCTION

Generic programming is an emerging paradigm for writing highly reusable libraries of algorithms. The generic programming approach has been used extensively within the C++ community; prime examples of generic libraries include the Standard Template Library (STL) [1,2], STAPL (a parallel STL) [3], Boost Graph Library [4] (BGL), Matrix Template Library [5], and Bioinformatics Template Library [6]. Common to these libraries is their extensive parameterization of algorithms with respect to the types of their arguments, allowing a single implementation of an algorithm to work on a broad class of different argument types. Such a high degree of type parameterization is less common in other mainstream object-oriented languages.

The major difference between generic programming in C++ vs. generics in, say, C# or Java, is that C++ does not require, or allow, any constraints on type parameters, whereas C# and Java do. Note that throughout the paper, with C# we refer to the planned future version 2.0, as specified in [7]. In C++, objects whose types are type parameters can be used in any operation; type checking of generic definitions is delayed until after concrete types have been bound to type parameters. In contrast, in C# and Java, explicit subtype constraints on type parameters allow type checking of generic definitions separately from their uses.

In a previous study [8], we evaluated six mainstream programming languages with respect to their support for generic programming. The evaluation was based on the experiences in implementing a subset of the BGL in each of the languages under study. Mainstream object-oriented languages did not rank highly in this evaluation; practical problems identified included verbose code, redundant code, and difficulties in composing separately defined generic components. Constraints naturally render generic method and class definitions in C# and Java more verbose than their counterparts in C++. Our experience with the BGL implementation showed, however, that the number and size of the required constraint expressions can become excessive in algorithms parameterized over several input types. First, the only means of expressing dependencies between generic types is by type parameterization. This prevents proper encapsulation of such dependencies into interfaces or abstract classes. Second, inheriting a generic class from another generic class does not mean inheriting constraints on type parameters. Rather, a generic class or method definition must often repeat type parameter constraints which are already induced by uses of these parameters in the types of base classes or by uses in other constraints.

This paper advocates *associated types* and *constraint propagation* as solutions to these problems. Associated types resemble member **typedef**s in C++, and also share similarities with type members of ML signatures. Section 6 describes these connections in more detail. Associated types can also be viewed as a restricted form of virtual types [9, 10]; unlike virtual types, associated types are conceptually attached to classes, rather than objects. We believe this solution fits into the generic features of mainstream languages, such as C# or Java, without drastic changes, and yet results in notable improvement in code clarity and expressiveness.

By constraint propagation we refer to a language mechanism that gathers type parameter constraints that are induced by other uses of the same type parameters. Constraints for a type parameter can be propagated from its uses as an argument to other generic types, in type parameter constraints, or in base class or interface expressions.

The contributions of this work are to introduce language extensions for associated types and constraint propagation within the context of languages that support F-bounded polymorphism [11], such as Java or C#. Our approach is motivated by the needs of practical generic programming, and by the techniques commonly used within that community. The proposed extensions are described using a series of examples, and then formalized using a Featherweight Generic Java-like approach [12]; the formalization is used to show that the extensions preserve type-safety. A translation from a simple object-oriented language with these extensions to a standard object-oriented language with F-bounded polymorphism is then presented, to give a possible implementation of the extensions. We provide a formal proof that the translation preserves type-safety.

The paper is structured as follows: Section 2 describes generic programming, and the role of associated types in generic definitions. Section 3 discusses the consequences of representing associated types using type parameters, and the consequences of the lack of constraint propagation in constrained generics. As a remedy to these problems, Section 4 suggests two language extensions for C# and describes informally a source-to-source translation of these extensions to C#. Section 5 provides a formal account of the impact of these features in a setting similar to Featherweight Generic Java (FGJ) [12] and C# minor [13] and proves the soundness of the extensions in this simplified formalism. Section 6 surveys related work, both in object-oriented and in functional programming languages. Section 7 concludes the paper and outlines future work.

2. BACKGROUND

Generic programming is a systematic approach to software reuse that focuses on finding the most general (or abstract) formulations of algorithms and their efficient implementations [14]. Fundamental to realizing generic algorithms is the notion of abstraction: generic algorithms are specified in terms of abstract properties of types, not in terms of particular types. In the terminology of generic programming, a *concept* is the formalization of an abstraction as a set of requirements on a type (or on several types) [15, 16]. These requirements may be semantic as well as syntactic. A concept may incorporate the requirements of another concept, in which case the first concept is said to *refine* the second. Types (or sequences of types) that meet the requirements of a concept are said to *model* the concept. Note that it is not necessarily the case that the requirements of a concept involve just one type: a concept may involve multiple types and specifies their relationships.

A concept consists of four different kinds of requirements: associated types, function signatures, semantic constraints, and complexity guarantees. The *associated types* of a concept specify mappings from the modeling type(s) to other collaborating types (such as the mapping from a container to the type of its elements). The *function signatures* specify the operations that must be defined for the modeling types. Alternatively, the operations may be specified using *valid expressions* that must be syntactically valid for any types that model the concept. This is idiomatic in documenting concepts in C++ libraries, because a single valid expression can be implemented with more than one function signature.

The above form of describing requirements on type parameters is wide-spread in the C++ community; the SGI STL documentation [17] is the archetypical example. The specification of the C++ standard library in [18] follows the same form as well. One reason for this is arguably the lack of language mechanisms for specifying constraints; structured concept descriptions provide some rigor in requirement descriptions. Another view is that concepts arise by examining concrete algorithms, grouping frequently occurring requirements into reusable entities. Concept descriptions are not bound by language constructs, and have had the freedom to evolve into a form that effectively captures the essential requirements that generic algorithms place on type parameters.

The following example shows a simplistic generic function first_neighbor for finding an adjacent vertex of another vertex in a graph:

```
template <class Graph>
typename Graph::vertex_type
first_neighbor(Graph g, typename Graph::vertex_type v) {
  return target(current(out_edges(v, g)));
}
// Constraints:   Graph models INCIDENCE GRAPH
```

The function constrains, though merely in documentation, its Graph type parameter using the concept INCIDENCE GRAPH documented in Figure 1. This concept requires the existence of the associated types vertex_type, edge_type, and out_edge_iterator. In addition, it places requirements on these types: edge_type must be a model of the GRAPH EDGE concept, and out_edge_iterator is required to model the ITERATOR concept. These two concepts are also shown in Figure 1. Furthermore, the INCIDENCE GRAPH concept has two *same-type constraints* (cf. ML sharing constraints) to ensure that out_edge_iterator iterates over edges with the correct type, and that vertex_type coincides with the edge_type's associated type vertex_type. Concepts can be built from other concepts using refinement. For example, we could define the concept BIDIRECTIONAL ITERATOR to contain all requirements of the ITERATOR concept, and add the ability to move backward in a sequence. In C++, concepts are merely a documentation artifact, and thus structural conformance (presence of the correct associated types and function definitions) to the requirements suffices for a type to model a concept.

In the generic programming approach, taxonomies of concepts of the modeled domain guide the systematic organization of reusable libraries. Concept taxonomies can in principle be developed independently of a particular programming language; the implementation language of a generic library

INCIDENCE GRAPH concept. Type Graph is a model of INCIDENCE GRAPH if the requirements below are satisfied. 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 GRAPH EDGE	
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**

GRAPH EDGE concept. Type Edge is a model of GRAPH EDGE if the following requirements are satisfied. 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 requirements are satisfied. 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

Figure 1: C++ concept descriptions for GRAPH, GRAPH EDGE, and ITERATOR concepts.

must, however, allow their expression. The directness of this varies among languages. For example, ML signatures and Haskell type classes have a relatively close correspondence to concepts. Classes and interfaces, the type parameter bounds of C# and Java, seem to be a less direct fit. In particular, representing associated types is accomplished in a roundabout manner. The next section discusses representing concepts and generic algorithms with object-oriented interfaces, such as those found in C# or Java generics.

3. INTERFACES AS CONCEPTS

In C# or Java, interfaces can capture the valid expression requirements of concepts as method signatures. We use C# syntax in our examples. The modeling relation between types and concepts can be represented as classes implementing interfaces and concept refinement can be represented as inheritance between interfaces. Although associated types do not have a direct counterpart in interfaces, a type parameter can represent an associated type. In that case, constraints on associated types are simply constraints on type parameters. Representing associated types with type parameters is common practice. The C# IEnumerable<T> interface for iterating through containers (from the C# standard library) serves as an example. When a type implements IEnumerable<T>, it must bind a concrete type, the value type of the container, to the type parameter T.

Compare the two concepts in Figure 1, and their representations as C# interfaces (Figure 2). The three type parameters Vertex, Edge, and OutEdgeIterator in the generic IncidenceGraph interface correspond to the three associated types of the INCIDENCE GRAPH concept. The constraints on these types are visible in the **where** clauses. Note that we use the standard interface IEnumerable as the interface for the iterator, rather than defining one based on the ITERATOR concept in Figure 1. Same-type constraints are implicit: out_edge_iterator::value_type == edge_type is expressed by using the type parameter Edge in the constraint of OutEdgeIterator, and the use of Vertex as the type argument to GraphEdge establishes the constraint edge_type::vertex_type == vertex_type. Figure 3 contains the C# version of first_neighbor, which uses IncidenceGraph to constrain one of its type parameters.

```
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);
}
```

Figure 2: GRAPH EDGE and INCIDENCE GRAPH as C# interfaces.

The main problem with representing associated types as type parameters is that type parameters are not properly encapsulated in the interface. Every reference to an interface, whether the interface is being extended (concept refinement) or used as a type parameter constraint, must list all of the type parameters explicitly. In a concept with several associated types, this becomes burdensome. In the study described in [8], the number of type parameters in generic algorithms was often more than doubled due to this effect. Figure 3 demonstrates the problem. Even though the first_neighbor function only takes two parameters, a graph and a vertex, it has four type parameters. Note that only two of these type parameters are referred to in the parameter list.

```
G_Vertex
first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>
  (G g, G_Vertex v)
    where G : IncidenceGraph<G_Vertex,
                             G_Edge, G_OutEdgeIterator>
    where G_Edge : GraphEdge<G_Vertex>
    where G_OutEdgeIterator : IEnumerable<G_Edge> {
  return g.out_edges(v).Current.target();
}
```

Figure 3: Example generic function in C#.

Another related problem is that interfaces fail to encapsulate constraints on associated types. Consider the type parameter constraints in the **where** clause of first_neighbor. The last two lines are a repetition of what is already specified in the **where** clause of the IncidenceGraph interface (see Figure 2). This repetition seems unnecessary: no type can be bound to G unless it inherits from the IncidenceGraph interface. This in turn requires that the types bound to the type parameters Vertex, Edge, and OutEdgeIterator must satisfy the constraints of the IncidenceGraph interface. Thus, based on the constraint on G above, the type-checker could safely assume that the type parameters G_Vertex, G_Edge, and G_OutEdgeIterator in the generic first_neighbor function also satisfy the constraints in IncidenceGraph. Such *constraint propagation*, as we refer to it, is not performed in C#, and thus the last two subtype constraints of first_neighbor are necessary as direct evidence of all type arguments meeting their bounds. In our experience [8], lack of constraint propagation greatly increases the verbosity of generic functions and generic interfaces. It also adds extra dependencies on the exact forms of generic interfaces; a slight change in a constraint on an associated type of an interface may require a change in every use of that interface.

4. EXTENDING C# GENERICS

We argue that direct support for associated types and support for constraint propagation would significantly improve generic programming in mainstream object-oriented languages, such as C#, without requiring drastic modifications to the language or its implementation. In this section we describe associated types, in the form of member types in interfaces, and constraint propagation, as extensions to C#. Furthermore, we describe how these features can be translated to standard C#. The presentation is informal; Section 5 gives a detailed formal description of the features and their translations in an idealized model of the language.

4.1 Associated types

Our approach to associated types for C# is to introduce *member types* in interfaces and classes. We allow interfaces to declare members which are placeholders for types, and place constraints, subtype or same-type, on these members. For ease of discussion, we use the term *concept interface* to refer to an interface that defines member types, or inherits from an interface defining member types. We adopt the syntax **type** A : B for declaring a member type A and requiring that any type bound to A must be a subtype of B, where B is an instance of a class or an interface. The constraint can be omitted. Same-type constraints are expressed with the syntax **require** A == B. We also allow subtype constraints of the form **require** A : B in isolation of associated type declarations.

The syntax T::A is used for accessing an associated type A of another type T. In particular, to access an associated type A in the class being defined or in one of its ancestors, one writes **this**::A, and we allow **this**:: to be omitted. Note that **this** in the context of types refers to the class being defined, rather than the current object.

The left-hand side of a subtype constraint, and one side of a same-type constraint, must be a *constrainable* type. We define a constrainable type as either a type parameter; **this**::A, where A is an associated type declared in the class or interface being defined, or in its ancestors; or an associated type of a constrainable type.

A concept interface, that is, an interface with member types that have not been bound to concrete types, can only be used as type parameters' subtype constraints, or as a base interface of another interface or class. In particular, one cannot declare a variable, function parameter, or a field of such a type, nor use it as a type argument to a generic method or type. We discuss this restriction in more detail later in this section. Match-bounded polymorphism [19] includes a similar notion of interfaces that cannot be used as types. A derived interface can tighten constraints on associated types defined in its base interfaces. Classes that implement interfaces must define the member types. These definitions cannot be changed in derived classes, or otherwise substitutability would be compromised. This is in line with the restrictions of type-safe variations of virtual types [10, 20]. We could weaken this restriction without too much complication, and allow *abstract classes* to leave some member types unbound. Any non-abstract class inheriting from an abstract class would then only need to bind the member types left unbound in its ancestors. The stronger rule arose from directly mapping concepts to interfaces, and from the desire to keep the formalization discussed in Section 5 simple.

As an example, Figure 4 shows how the graph concepts from Figure 1 can be expressed using this extension. The GraphEdge interface declares the member type Vertex. The IncidenceGraph interface declares two associated types, Vertex and Edge, and places constraints on them: Edge must be a subtype of GraphEdge and Vertex must be the same type as the associated type, also named Vertex, of Edge. We can observe that the member types correspond directly to the associated types in Figure 1, subtype constraints correspond to requirements that types model concepts, and the same-type constraint has a direct equivalent as well. The constraint on OutEdgeIterator, however, is different: the ITERATOR concept is represented using the standard IEnumerable interface, where the associated type Edge is an extra type parameter rather than a member type. This demonstrates that the two styles of representation for associated types can coexist.

```
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);
}
```

Figure 4: Graph concepts represented as interfaces with associated types.

```
G::Vertex first_neighbor<G>(G g, G::Vertex v)
    where G : IncidenceGraph
    where G::Edge : GraphEdge
    where G::OutEdgeIterator : IEnumerable<G::Edge>
    where G::Vertex == G::Edge::Vertex {
  return g.out_edges(v).Current.target();
}
```

Figure 5: The first_neighbor function that relies on support for associated types, but does not rely on constraint propagation.

The rewrite of the first_neighbor function in Figure 5 demonstrates the effect of associated types: one type parameter, instead of four, suffices. The constraint on G itself is very concise, but the constraints on associated types of G are even more verbose. This is due to the need to provide explicit evidence that the associated types of G satisfy all requirements placed on them by IncidenceGraph. The remedy is constraint propagation, explained in Section 4.2. First, however, we

```
class AdjListEdge : GraphEdge {
  type Vertex = int;
  ...
}
class AdjacencyList : IncidenceGraph {
  type Vertex = int;
  type Edge = AdjListEdge;
  type OutEdgeIterator =
    IEnumerable<AdjListEdge>;
  OutEdgeIterator out_edges(int v) {...}
  int out_degree(int v) {...}
}
```

(a)

```
class AdjListEdge : GraphEdge<int> {
  ...
}
class AdjacencyList
  : IncidenceGraph<int, AdjListEdge,
      IEnumerable<AdjListEdge> > {
  IEnumerable<AdjListEdge>
    out_edges(int v) {...}
  int out_degree(int v) {...}
}
```

(b)

Figure 6: Concrete graph and edge types that model the Incidence Graph and Graph Edge concepts. Associated types are represented using (a) member types, and (b) using type parameters.

explain how associated types are translated into type parameters:

- Each associated type declaration in an interface is translated into a new type parameter of that interface. Subtype constraints on associated types are translated and moved to **where** clauses as constraints on the corresponding type parameters. The translation of the interfaces in Figure 4 results in the interfaces we showed in Figure 2. All three associated types of IncidenceGraph end up as type parameters.

  Same-type constraints between two types are handled by unifying, in the logic programming sense, the translations of the types required to be equal. For example, in Figure 2 the type Vertex is used as the Vertex associated type for both GraphEdge and IncidenceGraph.

- The definitions of associated types in classes that implement interfaces are converted to type arguments of the interfaces. Figure 6 shows two such classes, implementing the GraphEdge and IncidenceGraph interfaces. The code in Figure 6(a) is written using the extension; Figure 6(b) contains the code translated to plain C#. Obviously, this part of the translation must be coordinated with the translations of the corresponding interface definitions. Associated types that are referred to by name in the extended C# are identified based on their positions in the type parameter list in the translated code. Thus, the translation must ensure that the same names are always mapped to the same positions.

- Concept interfaces can occur in constraints of generic functions, or those of generic classes or interfaces. Any use of a such an interface requires an extra type argument for each associated type, so that the instantiation matches the translated definition of the use. Also, references to associated types in the body and constraints of a generic function, class, or interface are converted to references to the corresponding type parameters. The IncidenceGraph and GraphEdge interfaces get three and one, respectively, extra type arguments in the translation of the first_neighbor function from the version in Figure 5 to the version in Figure 3. Again, same-type constraints are handled by using the same type parameter as a type argument to more than one generic interface, or in more than one argument position of one generic interface.

The translation outlined above is automating a procedure which is performed manually while implementing generic libraries. For example, the manual approach was used to express associated types in implementations of a graph algorithm library in C#, Java, and Eiffel, described in [8]. One can also view certain definitions in the standard libraries of these three languages, such as the types of generic iterators (e.g., the IEnumerable interface in C#), as uses of this technique.

A concept interface is not a traditional object-oriented interface; in particular, such an interface is not a type. As the translation suggests, these interfaces cannot be used without providing, either implicitly or explicitly, bindings for their associated types. As a consequence, concept interfaces can be used as constraints on type parameters, but not as types for variables or function parameters — uses that traditional interfaces allow. For example, the first_neighbor function in Figure 5 cannot be written as:

```
IncidenceGraph::Vertex
first_neighbor(IncidenceGraph g, IncidenceGraph::Vertex v);
```

In this definition, the references to IncidenceGraph::Vertex are undefined; the abstract IncidenceGraph interface does not define a binding for the Vertex associated type. This is a major difference between our translation and systems based on virtual types. In our translation, all associated types are looked up statically, and so the type of g is the interface IncidenceGraph, not a concrete class which implements IncidenceGraph. Compared to virtual types there are situations where extra type parameters are needed. The added verbosity is at most one type parameter for each function parameter.

For the translation described in this section to work with implicit instantiation (inferring type arguments from the types of the function arguments), it is important to be able to infer the definitions of associated types from the types that are bound to the main type parameters. After the translation, this amounts to being able to do type inference based on type parameter constraints, a feature supported, for example, by Java 5.

4.2 Constraint propagation

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 function a. Suppose T is a type parameter of a, and T is used as a type argument in some instantiation Y<..., T, ...> 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 Y places on T in the instantiation Y<..., T, ...> can be assumed to be true while type-checking a. Note that in all uses of a, type arguments will be checked against both explicitly declared and propagated constraints.

As an example of constraint propagation, consider the function in Figure 3. The type parameter G_Edge is used as a type argument to the IncidenceGraph interface. The **where** clause of IncidenceGraph requires its second type parameter Edge to be a subtype of GraphEdge<Vertex>. Substituting G_Edge to Edge and G_Vertex to Vertex, as the instantiation IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator> implies, the constraint G_Edge : GraphEdge<G_Vertex> can be assumed to hold while type-checking first_neighbor. The translation can thus be implemented by copying the constraints, with appropriate substitutions, according to the above description.

Combining the two extensions, associated types and constraint propagation, the constraints of the first_neighbor function can be written very concisely:

```
G::vertex_type first_neighbor<G>(G g, G::Vertex v)
    where G : IncidenceGraph {
  return g.out_edges(v).Current.target();
}
```

Applying first the translation for associated types leads to the definition below, which, after constraint propagation, becomes the definition in Figure 3:

```
G_Vertex
first_neighbor<G, G_Vertex, G_Edge, G_OutEdgeIterator>
  (G g, G_Vertex v)
    where G :
      IncidenceGraph<G_Vertex, G_Edge, G_OutEdgeIterator> {
  return g.out_edges(v).Current.target();
}
```

Our definition of constraint propagation infers additional constraints on type parameters from their uses within constraints, and base classes and interfaces. It would be possible to infer constraints from other uses of type parameters as well, such as their uses in method signatures, or the types of fields. We suspect that this would be surprising, as internal features of a class could leak into the interface. For example, a private field could impose constraints on users of a class. Constraints should not be propagated from entities that are not already part of the exposed interface of the class.

4.3 Associated types vs. type parameters

Associated types complement, rather than replace, explicit type parameters as the language mechanism for supporting generic programming. There are cases, however, where both mechanisms are feasible alternatives for implementing a particular design. This section discusses factors that can affect the decision between these alternatives.

If an associated type of an interface is frequently referred to in generic functions that use the interface as a constraint, it may be more natural to use type parameters. An example of such a situation would be interfaces describing different kinds of container concepts. One can expect that the type of the elements stored in a container class will almost invariably be mentioned in generic functions operating on containers, and thus it likely is more economical to express such an often used associated type as a type parameter. Less frequently needed associated types, on the other hand, are more natural to express as member types.

As another factor in the choice between type parameters and member types, one must consider whether implicit same type constraints with type parameters are more economical than their explicit expressions, as must be done with member types. Also, the C# 2.0 specification allows [7] a class to implement the same interface in two different ways, with different type arguments. With concept interfaces, if the only variation is in the bindings of member types and all type arguments remain the same, this is not possible.

Finally, member types may give better protection against changes in interfaces. For example, adding a new type parameter to an interface with explicit type parameters requires changing all uses of that interface. Adding a new member type to a concept interface, on the other hand, has no effect on the existing uses of the interface, except of course for class definitions that implement the interface.

5. FORMALIZATION

To gain assurance of the soundness of the extensions of associated types and constraint propagation, we developed a formal model for an idealized language, based on Featherweight Generic Java (FGJ) [12], which captures the essential properties of the extensions in a language similar to C# and Java. We refer to this language as FGJ+IAPR. We then define a translation of programs in FGJ+IAPR into a version of FGJ extended with interfaces and multiple interface inheritance, denoted FGJ+I below; we assume this extension can be done while preserving type safety. We show that programs in FGJ+IAPR are translated into programs in FGJ+I with the same type behavior, i.e., well-formedness of classes and interfaces is preserved, and translated expressions have their translated types. Because our language has exactly the same expressions as FGJ, and the translation does not alter the non-type content of expressions, we define our semantics through translation to FGJ+I. FGJ+IAPR is thus type-safe as long as FGJ+I is. FGJ allows type parameters both on methods and classes; we omit generic methods to reduce complexity (hence FGJ+IAPR is not strictly an extension of FGJ). As we do not consider variance or implicit instantiation of type parameters, parameterized methods provide no new insights. For the relevant cases, generic methods can be simulated with generic classes where the type parameters of the method are added as type parameters of the class containing the method.

5.1 Syntax

Figure 7 shows the syntax of FGJ+IAPR; many of the rules are either directly from, or based on, FGJ [12]. We summarize the language and notation. The metavariables I and J range over interface names; C and D over class names; E and F over either interface or class names; A and B over associated type names; X, Y, and Z over type variables; S, T, U, V, and W over arbitrary types; M, N over instantiated interfaces; K, L over instantiated classes; O, P

(interface def) id	$::=$ `interface` I<$\overline{X}$> : $\overline{M}$ `where` $\overline{rd}$
	{`type` $\overline{A}$; `require` $\overline{rd}$; $\overline{ms}$}
(class def) cd	$::=$ `class` C<$\overline{X}$> : $\overline{M}, K$ `where` $\overline{rd}$
	{`type` $\overline{A} = \overline{U}$; $\overline{T}\ \overline{f}$; $kd\ \overline{md}$}
(constraint def) rd, Q, R	$::= T : P \mid T == U$
(constructor def) kd	$::= C(\overline{T}\ \overline{f})$: `base`$(\overline{f})$ {`this`.$\overline{f} = \overline{f}$; }
(method signature) ms	$::= T\ m(\overline{U}\ \overline{x})$;
(method def) md	$::= T\ m(\overline{U}\ \overline{x})$ {`return` e; }
(expression) e	$::= x \mid e.f \mid e.m(\overline{e}) \mid$ `new` $K(\overline{e}) \mid (T)e$
(constrainable type) G, H	$::= X \mid$ `this` $:: A \mid G :: A$
(instantiated interface) M, N	$::= I$<$\overline{T}$>
(instantiated class) K, L	$::= C$<$\overline{T}$>
(class or interface name) E, F	$::= C \mid I$
(instantiated class or interface) O, P	$::= M \mid K$
(type) S, T, U, V, W	$::= G \mid P \mid T :: A$

Figure 7: Syntax of FGJ+IAPR, and the metavariables used for each syntactic construct.

over instantiated interfaces or classes (we refer to these as *instances*); G, H over *constrainable* types (these are either type parameters, or possibly nested associated types of type parameters or `this`); d and e over expressions; f and g over field names; and x over method parameters.

Types in FGJ+IAPR are either class or interface instances, type variables, or associated types. We always require associated types to be qualified, either with a class instance, type parameter name, another explicitly qualified associated type name, or `this`. In the examples of Section 4, such as in the function interface Vertex source(); of Figure 4, we wrote associated types without qualification. This is considered to be a shorthand notation for explicit qualification with `this`, and not allowed in the formalization. Within the formalization, the interface would have to be written as **this**::Vertex source();. Hence, as a qualifier preceding ::, `this` is a type that refers to the current class name, rather than an object.

Borrowing from the FGJ notation, a variable with a horizontal bar above it stands for a possibly empty sequence of elements from the variable's domain. The separator parameter is determined from the context. For example $\overline{A}$ is a shorthand for the comma separated sequence of associated types $A_1, A_2, \ldots, A_n$; and $\overline{ms}$; a sequence of method signatures delimited by semicolons. Further, $\overline{rd}$ represents a comma separated sequence of constraints. Such constraints can be of two forms: $T : P$ or $T == U$. We refer to the former kind as subtype constraints and the latter as same-type constraints. Note that in program code, the left-hand side of these constraints must be a constrainable type, but this restriction does not hold in the formalism in general. The horizontal bar is used with constraints as well: $\overline{T} : \overline{P}$ represents the sequence $T_1 : P_1, T_2 : P_2, \ldots, T_n : P_n$ and $\overline{T} == \overline{U}$ the sequence $T_1 == U_1, T_2 == U_2, \ldots, T_n == U_n$. The horizontal bar is also used with helper functions. For example, we define $constr(T)$ as the set of constraints induced by T, and take $constr(\overline{T})$ to mean $constr(T_1), constr(T_2), \ldots, constr(T_n)$. We write the type of a method $V\ m(\overline{U}\ \overline{x})$ { `return` e; } in class C<$\overline{T}$> as $\overline{U} \to V$; its body is the pair $\langle \overline{x}, e\rangle$, accessed by $mbody(C$<$\overline{T}$>$.m)$.

We require that sequences of names of type parameters, methods, associated types, and method parameters do not contain duplicates; and that inheritance does not cause a class or interface to have multiple members (associated types, fields, and methods) with the same name. Also, we sometimes write a class definition as `class` C<$\overline{X}$> : $\overline{P}\ldots$, in which case we assume that at most one P_i is a class, and the other elements in $\overline{P}$ are interfaces. Further, we sometimes use the syntax `class/interface` E<$\overline{X}$> : $\overline{P}\ldots$ when describing behavior common to classes and interfaces; it is assumed that if E is an interface, all elements of $\overline{P}$ are interfaces.

We write $[\overline{T}/\overline{X}]U$ for the simultaneous substitution of types $\overline{T}$ for $\overline{X}$ in U. A substitution σ is well-formed in some environment Δ (written $\Delta \vdash \sigma\ ok$) if $\overline{T}$ are well-formed in Δ. Well-formedness is defined below in this section.

CT is a class table mapping the name of a class or interface to its definition. Then an FGJ+IAPR program is a fixed class table and a single expression e, whose evaluation is the program execution. We assume CT satisfies the following conditions: (1) $CT(C) =$ `class` $C\ldots$ and $CT(I) =$ `interface` $I\ldots$ for every $C, I \in dom(CT)$; (2) for every class or interface name E, appearing anywhere in CT, $E \in dom(CT)$; (3) there are no cycles in the inheritance relation induced by CT; (4) and that associated type definitions in classes do not form a cycle. By the third condition we mean that a class or interface cannot inherit from another instance of itself. Since inheriting is only allowed from instances of classes or interfaces, not from type parameters or associated types, this condition can be guaranteed easily. The fourth condition can be checked by following the definitions of each associated type in all classes, and ensuring that they eventually are bound to an instance or a constrainable type. For shorter presentation, we define another function $\mathcal{D}$, in Figure 8, that looks up a class or interface from CT and performs simultaneous substitution

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : \overline{P}\ \texttt{where}\ \overline{rd}\ \{\texttt{type}\ \overline{A} = \overline{V};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\} \qquad |\overline{T}| = |\overline{X}| \qquad \sigma = [\overline{T}/\overline{X}, U/\texttt{this}]}{\mathcal{D}(C\texttt{<}\overline{T}\texttt{>}, U) = \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \sigma\overline{P}\ \texttt{where}\ \sigma\overline{rd}\ \{\texttt{type}\ \overline{A} = \sigma\overline{V};\ \sigma\overline{W}\ \overline{f};\ \sigma kd\ \sigma\overline{md}\}}$$

$$\frac{CT(I) = \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\{\texttt{type}\ \overline{A};\ \texttt{require}\ \overline{rd_2};\ \overline{ms};\} \qquad |\overline{T}| = |\overline{X}| \qquad \sigma = [\overline{T}/\overline{X}, U/\texttt{this}]}{\mathcal{D}(I\texttt{<}\overline{T}\texttt{>}, U) = \texttt{interface}\ I\texttt{<}\overline{T}\texttt{>} : \sigma\overline{M}\ \texttt{where}\ \sigma\overline{rd_1}\ \{\texttt{type}\ \overline{A}; \texttt{require}\ \sigma\overline{rd_2};\ \sigma\overline{ms};\}}$$

Figure 8: Lookup function that performs substitution.

of type arguments to type parameters and an instance type to `this` everywhere in the class or interface definition.

5.2 Typing

We use two typing environments Γ and Δ. The environment Γ maps variables (method parameters) to their types; its elements have the form $\overline{x} : \overline{T}$. The type environment Δ can contain three kinds of elements: type parameter names; subtype constraints, of the form $T : P$; and same-type constraints, of the form $T == U$. We define the helper function $\cdot_v$ for extracting the type variables from a type environment, and $\cdot_c$ for extracting the constraints. These two functions are used as Δ_v and Δ_c. We use the metavariables Q and R to range over constraints. A typing judgment $\Delta; \Gamma \vdash e : T$ is read as "e has type T in the environments $\Delta; \Gamma$." The typing rules for expressions, shown in Figure 10, directly correspond to those in FGJ, except for TY-DCAST, which omits for brevity a technical condition not significant for our formalism.

In Figure 9, we define type equality $==$ as the symmetric, reflexive, and transitive congruence closure of the same-type constraints in the environment. The subtyping relation $<:$ is defined as the closure, under type equality, reflexivity, and transitivity, of the subtype constraints in the environment. We include well-formedness rules for types, environments, constraints, class and interface definitions, and method signatures and definitions. Except for class and interface definitions, well-formedness is always defined with respect to some environment Δ. The judgment $\Delta \vdash E\texttt{<}\overline{T}\texttt{>}\ ok$ is true if $\overline{T}$ satisfy the constraints the definition of E places on its type parameters. This set of constraints, defined using the *propag-c* function shown in Figure 13, is the result of constraint propagation from E's ancestors and from all constraints explicitly expressed in E. For technical reasons, we define two forms of well-formedness for constraints: *semi-ok* and *ok*. A subtype or same-type constraint is *semi-ok* in some environment if both its left-hand and right-hand sides are *ok* in that environment. In the user syntax of FGJ+IAPR the left-hand side of a constraint must always be a constrainable type. This is the additional requirement for a constraint to be *ok*. Constraints with an instance on the left-hand side can, however, occur internally in the formalism; the definition of *semi-ok* allows this. A type environment is well-formed if all constraints and types in it are well-formed.

The well-formedness rules for class and interface definitions, in Figure 11, use the *env-for-body* function to create the type environment Δ, in which all subparts of the class or interface definition must be well-formed. The *env-for-body* function should result in the same environment that must be proven well-formed in the "instance *ok*" rules, and its definition in terms of *propag-c* does exactly this. The CLASS-DEF-OK rule also requires that all constraints propagated from the base interfaces are satisfied in the environment of the class definition. In this premise, the class being checked is substituted for `this` to ensure that the definitions for associated types given in the class are checked against the constraints placed on them by the class's base class and base interfaces.

In contrast to FGJ, in which only the direct constraints of type parameters comprise the environment for checking well-formedness of the subparts of the class, the environment in our formalism is larger as a result of constraint propagation. This amounts to fewer explicit constraints on type parameters being necessary. Note, however, that generic definitions are still type-checked separately. Type checking may require examining classes or interfaces used as ancestors of the class being checked, or classes or interfaces used in its constraints. This must be done recursively, but it is not significantly different than guaranteeing that, say, references to fields are valid. Regarding well-formed classes, the special class `object` is allowed to have no base class, and is assumed to be defined as an empty class with no bases.

Other helper definitions for these rules include *assoc-decl*, which collects the names of all associated types declared in a given class or interface or in its ancestors; *assoc-def* collects the names of associated types defined and bound to types in a given class or in its base classes. These definitions help to establish that no associated type is left unbound in a class definition. The functions *msigs-decl* and *msigs-def* serve a similar purpose for methods. The function *fields* collects all field definitions from a given class and its superclasses.

The rules for constraint propagation are defined in Figure 13. For some instance $E\texttt{<}\overline{T}\texttt{>}$, *constr*$(E\texttt{<}\overline{T}\texttt{>})$ includes all direct constraints of E, both from **where** clauses and **require** clauses, and constraints for all instances that occur as direct bases or in direct constraints of E (recursively). This set may contain constraints of the form $O : P$, which are between two instances. Such constraints are filtered out using *propag-c*, and only constraints of the form $G : P$ or $G == T$ remain. We call such constraints *propagable.*

5.3 Translation

Figure 15 shows the definition of the translation function $[\![\cdot]\!]_\Delta$ from FGJ+IAPR to FGJ+I. Regarding notation, we apply the translation freely to a set of types, or constraints, and take it to mean that each element of the set is translated, and a new set produced as a result. In particular, type environments contain subtype and same-type constraints, and type parameter names. Translating such an environment means translating each element with the appropriate rule and combining the results into a set. Note that $\emptyset$ as the result of a translation in the constraint translation rules means that the constraint is removed in the translation. The interesting parts of the rules are the translations of constrainable types, instances, and class and interface definitions, particularly the coordination to ensure that the type parameters

TE-REFL
$$\frac{}{\Delta \vdash T == T}$$

TE-FROM-ENV
$$\frac{}{\Delta, T == U \vdash T == U}$$

TE-TRANS
$$\frac{\Delta \vdash T == U \qquad \Delta \vdash U == V}{\Delta \vdash T == V}$$

TE-SYM
$$\frac{\Delta \vdash T == U}{\Delta \vdash U == T}$$

TE-ASSOC-CRG
$$\frac{\Delta \vdash T == U}{\Delta \vdash T :: A == U :: A}$$

TE-TPARAM-CRG
$$\frac{\Delta \vdash \overline{T} == \overline{U}}{\Delta \vdash E\texttt{<}\overline{T}\texttt{>} == E\texttt{<}\overline{U}\texttt{>}}$$

TE-ASSOC-TYPE
$$\frac{\mathcal{D}(C\texttt{<}\overline{T}\texttt{>}, U) = \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A} = \overline{V};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\} \qquad \Delta \vdash U <: C\texttt{<}\overline{T}\texttt{>}}{\Delta \vdash U :: A_i == V_i,\ \text{for all } i}$$

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

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

S-FROM-ENV
$$\frac{}{\Delta, G : P \vdash G <: P}$$

S-VIA-EQ
$$\frac{\Delta \vdash T == U \qquad \Delta \vdash U <: V}{\Delta \vdash T <: V}$$

S-CLASS
$$\frac{\mathcal{D}(E\texttt{<}\overline{T}\texttt{>}, E\texttt{<}\overline{T}\texttt{>}) = \texttt{class/interface}\ E\texttt{<}\overline{T}\texttt{>} : \overline{P} \ldots}{\Delta \vdash E\texttt{<}\overline{T}\texttt{>} <: P_i,\ \text{for all } i}$$

WF-VAR
$$\frac{X \in \Delta}{\Delta \vdash X\, ok}$$

WF-INTERF-INSTANCE
$$\frac{\begin{array}{c} CT(I) = \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} \ldots \qquad \Delta \vdash \overline{T}\ ok \\ \Delta \vdash [\overline{T}/\overline{X}]\mathit{propag\text{-}c}(I\texttt{<}\overline{X}\texttt{>})\ \mathit{satisfied} \qquad \mathit{assoc\text{-}decl}(I\texttt{<}\overline{T}\texttt{>}) = \emptyset \end{array}}{\Delta \vdash I\texttt{<}\overline{T}\texttt{>}\ ok}$$

WF-CONSTRAINT-INTERF-INSTANCE
$$\frac{\begin{array}{c} CT(I) = \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} \ldots \qquad \Delta \vdash \overline{T}\ ok \\ \Delta \vdash [\overline{T}/\overline{X}]\mathit{propag\text{-}c}(I\texttt{<}\overline{X}\texttt{>})\ \mathit{satisfied} \end{array}}{\Delta \vdash I\texttt{<}\overline{T}\texttt{>}\ \mathit{ok\text{-}constraint}}$$

OK-IS-OK-CONSTRAINT
$$\frac{\Delta \vdash T\ ok}{\Delta \vdash T\ \mathit{ok\text{-}constraint}}$$

WF-CLASS-INSTANCE
$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : \overline{M}, K\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A}\ = \overline{V};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\} \\ \Delta \vdash \overline{T}\ ok \qquad \Delta \vdash [\overline{T}/\overline{X}, C\texttt{<}\overline{T}\texttt{>}/\texttt{this}]\mathit{propag\text{-}c}(C\texttt{<}\overline{X}\texttt{>})\ \mathit{satisfied} \end{array}}{\Delta \vdash C\texttt{<}\overline{T}\texttt{>}\ ok}$$

WF-ASSOC-TYPE-INTERF
$$\frac{\mathcal{D}(I\texttt{<}T\texttt{>}, G) = \texttt{interface}\ I\texttt{<}\overline{T}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A};\ \texttt{require}\ \overline{rd_2};\ \overline{ms};\} \qquad \Delta \vdash G <: I\texttt{<}\overline{T}\texttt{>} \qquad \Delta \vdash G\ ok \text{ or } G = \texttt{this}}{\Delta \vdash G :: A_i\ ok,\ \text{for all } i}$$

WF-ASSOC-TYPE-CLASS
$$\frac{\mathcal{D}(C\texttt{<}T\texttt{>}, U) = \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A} = \overline{V};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\} \qquad \Delta \vdash U <: C\texttt{<}\overline{T}\texttt{>} \qquad \Delta \vdash U\ ok \text{ or } U = \texttt{this}}{\Delta \vdash U :: A_i\ ok,\ \text{for all } i}$$

Figure 9: Subtyping, type equality, and well-formedness rules of FGJ+IAPR.

TY-VAR
$$\frac{}{\Delta; \Gamma \vdash x : \Gamma(x)}$$

TY-SUB
$$\frac{\Delta; \Gamma \vdash e : T \qquad \Delta \vdash T <: U}{\Delta; \Gamma \vdash e : U}$$

TY-NEW
$$\frac{\begin{array}{c} \Gamma; \Delta \vdash \overline{e} : \overline{W} \qquad \Delta \vdash K\ ok \\ \mathit{fields}(K) = \overline{W}\ \overline{f} \end{array}}{\Delta \vdash \texttt{new}\ K(\overline{e}) : K}$$

TY-FLD
$$\frac{\Delta; \Gamma \vdash e : K \qquad \mathit{fields}(K) = \overline{W}\ \overline{f}}{\Delta; \Gamma \vdash e.f_i : W_i}$$

TY-METH
$$\frac{\begin{array}{c} \Delta; \Gamma \vdash e : W \qquad \Delta \vdash W <: P \qquad \Delta; \Gamma \vdash e : U \\ \langle m, \overline{U} \rightarrow V \rangle \in \mathit{msigs\text{-}decl}(P, W) \end{array}}{\Delta; \Gamma \vdash e.m(\overline{e}) : V}$$

TY-UCAST
$$\frac{\begin{array}{c} \Delta; \Gamma \vdash e : T \\ \Delta \vdash T <: O \qquad \Delta \vdash O\ ok \end{array}}{\Delta; \Gamma \vdash (O)e : O}$$

TY-DCAST
$$\frac{\begin{array}{c} \Delta; \Gamma \vdash e : T \qquad \Delta \vdash O\ ok \\ \Delta \nvdash O == T \qquad \Delta \vdash O <: T \end{array}}{\Delta; \Gamma \vdash (O)e : O}$$

TY-SCAST
$$\frac{\begin{array}{c} \Delta; \Gamma \vdash e : T \qquad \Delta \vdash O\ ok \qquad \Delta \vdash O \not<: T \qquad \Delta \vdash T \not<: O \\ T \text{ is a class instance} \qquad O \text{ is a class instance} \qquad \mathit{stupid\ warning} \end{array}}{\Delta; \Gamma \vdash (O)e : O}$$

Figure 10: Typing rules for expressions.

WF-SUBT-CONSTR

$$\frac{\Delta \vdash G\ \mathit{ok} \qquad \Delta \vdash L\ \mathit{ok\text{-}constraint}}{\Delta \vdash G : L\ \mathit{ok}}$$

WF-SAMET-CONSTR

$$\frac{\Delta \vdash G\ \mathit{ok} \qquad \Delta \vdash U\ \mathit{ok}}{\Delta \vdash G == U\ \mathit{ok}}$$

SEMI-WF-SUBT-CONSTR

$$\frac{\Delta \vdash O\ \mathit{ok} \qquad \Delta \vdash P\ \mathit{ok\text{-}constraint}}{\Delta \vdash O : P\ \mathit{semi\text{-}ok}}$$

SEMI-WF-SAMET-CONSTR

$$\frac{\Delta \vdash O\ \mathit{ok} \qquad \Delta \vdash P\ \mathit{ok}}{\Delta \vdash O == P\ \mathit{semi\text{-}ok}}$$

SEMI-WF-THIS-CONSTR

$$\frac{\Delta \vdash P\ \mathit{ok\text{-}constraint}}{\Delta \vdash \mathtt{this} : P\ \mathit{semi\text{-}ok}}$$

$$\frac{\Delta \vdash U <: P \qquad \Delta \vdash U : P\ \mathit{semi\text{-}ok}}{\Delta \vdash U : P\ \mathit{satisfied}} \qquad \frac{\Delta \vdash U == T \qquad \Delta \vdash U == T\ \mathit{semi\text{-}ok}}{\Delta \vdash U == T\ \mathit{satisfied}}$$

CLASS-DEF-OK

$$\frac{\begin{array}{c}\Delta = \mathit{env\text{-}for\text{-}body}(C) \qquad \sigma = [C\mathtt{<}\overline{X}\mathtt{>}/\mathtt{this}] \\ \Delta \vdash \overline{M}\ \mathit{ok\text{-}constraint} \qquad \Delta \vdash K\ \mathit{ok} \qquad \Delta \vdash \overline{rd_1}\ \mathit{ok} \qquad \mathit{assoc\text{-}decl}(\overline{M}) \subseteq \overline{A}, \mathit{assoc\text{-}def}(K) \qquad \overline{A} \text{ and } \mathit{assoc\text{-}def}(K) \text{ disjoint} \\ \Delta \vdash \sigma\overline{V}\ \mathit{ok} \qquad \Delta \vdash \sigma\overline{W}\ \mathit{ok} \qquad \mathit{fields}(K) = \overline{U}\ \overline{g} \qquad \overline{f} \text{ and } \overline{g} \text{ disjoint} \qquad \mathit{kd} = C(\overline{U}\ \overline{g}, \overline{W}\ \overline{f})\{\mathtt{base}(\overline{g}); \mathtt{this}.\overline{f} = \overline{f};\} \\ \Delta \vdash \sigma\overline{md}\ \mathit{ok} \qquad \langle m, T\rangle \in \mathit{msigs\text{-}decl}(C\mathtt{<}\overline{X}\mathtt{>}, \mathtt{this}) \text{ and } \langle m, U\rangle \in \mathit{msigs\text{-}decl}(C\mathtt{<}\overline{X}\mathtt{>}, \mathtt{this}) \text{ implies } \Delta \vdash T == U \\ \Delta \vdash \sigma(\mathit{propag\text{-}c}(C\mathtt{<}\overline{X}\mathtt{>}))\mathit{satisfied}\end{array}}{\mathtt{class}\ C\mathtt{<}\overline{X}\mathtt{>} : \overline{M}, K\ \mathtt{where}\ \overline{rd_1}\ \{\mathtt{type}\ \overline{A}\ = \overline{V};\ \overline{W}\ \overline{f};\ \mathit{kd}\ \overline{md}\}\ \mathit{ok}}$$

INTERFACE-DEF-OK

$$\frac{\begin{array}{c}\Delta = \mathit{env\text{-}for\text{-}body}(I) \qquad \Delta \vdash \overline{M}\ \mathit{ok\text{-}constraint} \qquad \Delta \vdash \overline{rd_1}\ \mathit{ok} \\ \Delta \vdash \overline{rd_2}\ \mathit{ok} \qquad \Delta \vdash \overline{ms}\ \mathit{ok} \qquad \langle m, T\rangle \in \mathit{msigs\text{-}decl}(I\mathtt{<}\overline{X}\mathtt{>}, \mathtt{this}) \text{ and } \langle m, U\rangle \in \mathit{msigs\text{-}decl}(I\mathtt{<}\overline{X}\mathtt{>}, \mathtt{this}) \text{ implies } \Delta \vdash T == U\end{array}}{\mathtt{interface}\ I\mathtt{<}\overline{X}\mathtt{>} : \overline{M}\ \mathtt{where}\ \overline{rd_1}\ \{\mathtt{type}\ \overline{A};\ \mathtt{require}\ \overline{rd_2};\ \overline{ms};\}\ \mathit{ok}}$$

$$\frac{\Delta \vdash T\ \mathit{ok} \qquad \vdash \overline{V}\ \mathit{ok}}{\Delta \vdash T\ m(\overline{V}\ \overline{x})\ \mathit{ok}} \qquad \frac{\Delta \vdash T\ m(\overline{V}\ \overline{x})\ \mathit{ok} \qquad \Delta; \overline{x} : \overline{V}, \mathtt{this} : \mathtt{this} \vdash e : T}{\Delta \vdash T\ m(\overline{V}\ \overline{x})\ \{\mathtt{return}\ e;\}\ \mathit{ok}}$$

$$\frac{CT(C) = \mathtt{class}\ C\mathtt{<}\overline{X}\mathtt{>} : \overline{M}, K\ \mathtt{where}\ \overline{rd_1}\ \{\mathtt{type}\ \overline{A}\ = \overline{V};\ \overline{W}\ \overline{f};\ \mathit{kd}\ \overline{md}\}}{\mathit{env\text{-}for\text{-}body}(C) = \overline{X}, \mathit{propag\text{-}c}(C\mathtt{<}\overline{X}\mathtt{>}), \mathtt{this} : C\mathtt{<}\overline{X}\mathtt{>}}$$

$$\frac{CT(I) = \mathtt{interface}\ I\mathtt{<}\overline{X}\mathtt{>} : \overline{M}\ \mathtt{where}\ \overline{rd_1}\ \{\mathtt{type}\ \overline{A};\ \mathtt{require}\ \overline{rd_2};\ \overline{ms};\}}{\mathit{env\text{-}for\text{-}body}(I) = \overline{X}, \mathit{propag\text{-}c}(I\mathtt{<}\overline{X}\mathtt{>}), \mathtt{this} : I\mathtt{<}\overline{X}\mathtt{>}}$$

Figure 11: Well-formed constraints, classes and interfaces, and necessary helper definitions.

$$\frac{CT(I) = \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A};\ \texttt{require}\ \overline{rd_2};\ \overline{ms};\}}{A_i \in \mathit{assoc\text{-}decl}(I\texttt{<}\overline{T}\texttt{>}),\ \text{for all } \overline{T} \text{ and } i}$$

$$\frac{CT(I) = \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} : \overline{M}\ \ldots \qquad B \in \mathit{assoc\text{-}decl}(M_i),\ \text{for some } i}{B \in \mathit{assoc\text{-}decl}(I\texttt{<}\overline{T}\texttt{>}),\ \text{for all } \overline{T}}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : \overline{M}, K\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A}\ = \overline{V}; \overline{W}\ \overline{f};\ kd\ \overline{md}\}}{A_i \in \mathit{assoc\text{-}def}(C\texttt{<}\overline{T}\texttt{>}),\ \text{for all } \overline{T} \text{ and } i}$$

$$\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : \overline{M}, K\ \ldots \qquad B \in \mathit{assoc\text{-}def}(K)}{B \in \mathit{assoc\text{-}def}(C\texttt{<}\overline{T}\texttt{>}),\ \text{for all } \overline{T}} \qquad \frac{A \in \mathit{assoc\text{-}def}(T)}{A \in \mathit{assoc\text{-}decl}(T)}$$

$$\frac{\mathcal{D}(C\texttt{<}\overline{T}\texttt{>}, \texttt{this}) =\ \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \overline{M}, K\ \texttt{where}\ \overline{rd}\ \{\texttt{type}\ \overline{A} = \overline{W};\ \overline{U_1}\ \overline{f_1};\ kd\ \overline{md}\} \qquad \mathit{fields}(K) = \overline{U_2}\ \overline{f_2}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{U_2}\ \overline{f_2}, \overline{U_1}\ \overline{f_1}}$$

$$\frac{\mathcal{D}(I\texttt{<}\overline{T}\texttt{>}, P) = \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A};\ \texttt{require}\ \overline{rd_2};\ \overline{ms};\} \qquad V\ m(\overline{U}\ \overline{x}) \in \overline{ms}}{\langle m, \overline{U} \to V\rangle \in \mathit{msigs\text{-}decl}(I\texttt{<}\overline{T}\texttt{>}, P)}$$

$$\frac{\mathcal{D}(I\texttt{<}\overline{T}\texttt{>}, P) = \texttt{interface}\ I\texttt{<}\overline{T}\texttt{>} : \overline{M}\ \ldots \qquad \langle m, \overline{U} \to V\rangle \in \mathit{msigs\text{-}decl}(M_i, P),\ \text{for some } i}{\langle m, \overline{U} \to V\rangle \in \mathit{msigs\text{-}decl}(I\texttt{<}\overline{T}\texttt{>}, P)}$$

$$\frac{\mathcal{D}(C\texttt{<}\overline{T}\texttt{>}, \texttt{this}) = \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \overline{M}, K\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A}\ = \overline{V}; \overline{W}\ \overline{f};\ kd\ \overline{md}\} \qquad S\ m(\overline{U}\ \overline{x})\ \{\texttt{return}\ e;\} \in \overline{md}}{\langle m, \overline{U} \to S\rangle \in \mathit{msigs\text{-}def}(C\texttt{<}\overline{T}\texttt{>})}$$

$$\frac{\begin{array}{c}\mathcal{D}(C\texttt{<}\overline{T}\texttt{>}, \texttt{this}) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : \overline{M}, K\ \ldots \\ \langle m, \overline{U} \to S\rangle \in \mathit{msigs\text{-}def}(K)\end{array}}{\langle m, \overline{U} \to S\rangle \in \mathit{msigs\text{-}def}(C\texttt{<}\overline{T}\texttt{>})} \qquad \frac{\langle m, \overline{U} \to S\rangle \in \mathit{msigs\text{-}def}(T)}{\langle m, \overline{U} \to S\rangle \in \mathit{msigs\text{-}decl}(T, T)}$$

$$\frac{\mathcal{D}(C\texttt{<}T\texttt{>}, \texttt{this}) =\ \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \overline{P}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A} = \overline{V};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\} \qquad S\ m(\overline{U}\ \overline{x})\ \{\texttt{return}\ e;\} \in \overline{md}}{\mathit{mbody}(C\texttt{<}\overline{T}\texttt{>}.m) = \langle \overline{x}, e\rangle}$$

$$\frac{\begin{array}{c}\mathcal{D}(C\texttt{<}\overline{T}\texttt{>}, \texttt{this}) =\ \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \overline{M}, K\ \texttt{where}\ \overline{rd}\ \{\texttt{type}\ \overline{A} = \overline{V};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\} \\ m \text{ not defined in } \overline{md} \qquad \mathit{mbody}(K.m) = \langle \overline{x}, e\rangle\end{array}}{\mathit{mbody}(C\texttt{<}\overline{T}\texttt{>}.m) = \langle \overline{x}, e\rangle}$$

Figure 12: Lookup functions for associated types, fields, method signatures, and bodies.

CPRG-RIGHT-SUB

$$\frac{CT(E) = \texttt{class/interface}\ E\texttt{<}\overline{X}\texttt{>}\ \ldots \qquad U : E\texttt{<}\overline{T}\texttt{>} \in \mathit{constr}(V) \qquad R \in \mathit{constr}(E\texttt{<}\overline{X}\texttt{>})}{[\overline{T}/\overline{X}, U/\texttt{this}]R \in \mathit{constr}(V)}$$

CPRG-INHERIT

$$\frac{\mathcal{D}(E\texttt{<}\overline{T}\texttt{>}, \texttt{this}) = \texttt{class/interface}\ E\texttt{<}\overline{T}\texttt{>} : \overline{P}\ \ldots}{E\texttt{<}\overline{T}\texttt{>} : P_i \in \mathit{constr}(E\texttt{<}\overline{T}\texttt{>}),\ \text{for all } i}$$

CPRG-CLASS-DEF

$$\frac{\mathcal{D}(C\texttt{<}\overline{T}\texttt{>}, \texttt{this}) = \texttt{class}\ C\texttt{<}\overline{T}\texttt{>} : \overline{P}\ \texttt{where}\ \overline{rd}\ \{\texttt{type}\ \overline{A} = \overline{U};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\}}{\overline{rd} \subseteq \mathit{constr}(C\texttt{<}\overline{T}\texttt{>})}$$

CPRG-INTERF-DEF

$$\frac{\mathcal{D}(I\texttt{<}\overline{T}\texttt{>}, \texttt{this}) = \texttt{interface}\ I\texttt{<}\overline{T}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A};\ \texttt{require}\ \overline{rd_2};\ \overline{ms};\}}{\overline{rd_1}, \overline{rd_2} \subseteq \mathit{constr}(I\texttt{<}\overline{T}\texttt{>})}$$

CPRG-COMB-SUB

$$\frac{\mathit{constr}(T) = \overline{rd_1} \qquad \mathit{constr}(P) = \overline{rd_2}}{\mathit{constr}(T : P) = T : P, \overline{rd_1}, \overline{rd_2}}$$

CPRG-COMB-SAME

$$\frac{\mathit{constr}(T) = \overline{rd_1} \qquad \mathit{constr}(U) = \overline{rd_2}}{\mathit{constr}(T == U) = T == U, \overline{rd_1}, \overline{rd_2}}$$

$$\frac{}{G : P\ \mathit{propagable}} \qquad \frac{}{G == U\ \mathit{propagable}}$$

$$\frac{R \in \mathit{constr}(T) \qquad R\ \mathit{propagable}}{R \in \mathit{propag\text{-}c}(T)} \qquad \frac{R \in \mathit{constr}(Q) \qquad R\ \mathit{propagable}}{R \in \mathit{propag\text{-}c}(Q)}$$

Figure 13: Definitions of *constr* and *propag-c* used for constraint propagation.

$$\frac{\Delta \vdash T ==_{\mathit{weak}} E\texttt{<}\overline{U}\texttt{>}}{E\texttt{<}\overline{U}\texttt{>} \in \mathit{canon\text{-}weak\text{-}inst}_\Delta(T)} \qquad \frac{X \in \Delta \qquad \Delta \vdash T ==_{\mathit{weak}} X}{X \in \mathit{canon\text{-}weak\text{-}tvar}_\Delta(T)} \qquad \frac{\Delta \vdash T ==_{\mathit{weak}} U :: A}{U :: A \in \mathit{canon\text{-}weak\text{-}assoc}_\Delta(T)}$$

$$\frac{}{\mathit{canon\text{-}weak}_\Delta(T) = \mathit{first}\begin{pmatrix}\mathit{order\text{-}set}(\mathit{canon\text{-}weak\text{-}inst}_\Delta(T))\ + \\ \mathit{order\text{-}set}(\mathit{canon\text{-}weak\text{-}tvar}_\Delta(T))\ + \\ \mathit{order\text{-}set}(\mathit{canon\text{-}weak\text{-}assoc}_\Delta(T))\end{pmatrix}}$$

CANON-TVAR

$$\frac{\mathit{canon\text{-}weak}_\Delta(T) = X \qquad X \in \Delta_v}{\mathit{canon}_\Delta(T) = X}$$

CANON-INSTANCE

$$\frac{\mathit{canon\text{-}weak}_\Delta(T) = E\texttt{<}\overline{U}\texttt{>} \qquad \mathit{canon}_\Delta(\overline{U}) = \overline{W}}{\mathit{canon}_\Delta(T) = E\texttt{<}\overline{W}\texttt{>}}$$

CANON-ASSOC

$$\frac{\mathit{canon\text{-}weak}_\Delta(T) = U :: A \qquad \mathit{canon}_\Delta(U) = S \qquad \Delta \vdash S \not<: C\texttt{<}\overline{W}\texttt{>}, \text{for all } C \text{ and } \overline{W}}{\mathit{canon}_\Delta(T) = S :: A}$$

CANON-ASSOC-DEF

$$\frac{\begin{array}{c}\mathit{canon\text{-}weak}_\Delta(T) = U :: A_i \qquad \mathit{canon}_\Delta(U) = S \qquad \Delta \vdash S <: C\texttt{<}\overline{W}\texttt{>}, \text{for some } C \text{ and } \overline{W} \\ \mathcal{D}(C\texttt{<}\overline{W}\texttt{>}, S) = \texttt{class}\ C\texttt{<}\overline{W}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A} = \overline{V};\ \ldots\} \qquad {V_i}' = \mathit{canon}_\Delta(V_i)\end{array}}{\mathit{canon}_\Delta(T) = {V_i}'}$$

$$\frac{\begin{array}{c}CT(I) = \texttt{interface}\ I\texttt{<}\overline{X}\texttt{>}\ldots \\ \mathit{assoc\text{-}decl}(I) = \overline{A}\end{array}}{\overline{X}, \overline{\texttt{this} :: A} \subseteq \mathit{full\text{-}assocs\text{-}tparams}(I)} \qquad \frac{\begin{array}{c}CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ldots \\ \mathit{assoc\text{-}def}(C) = \overline{A}\end{array}}{\overline{X}, \overline{\texttt{this} :: A} \subseteq \mathit{full\text{-}assocs\text{-}tparams}(C)}$$

$$\frac{T \in \mathit{full\text{-}assocs\text{-}tparams}(E) \qquad \mathit{env\text{-}for\text{-}body}(E) \vdash T :: A\ \mathit{ok}}{T :: A \in \mathit{full\text{-}assocs\text{-}tparams}(E)}$$

$$\frac{G \in \mathit{full\text{-}assocs\text{-}tparams}(E) \qquad \Delta = \mathit{env\text{-}for\text{-}body}(E) \qquad \mathit{canon}_\Delta(G) = T}{T \in \mathit{full\text{-}tparams}(E)}$$

Figure 14: Definitions of *canon* and *full-tparams* functions used in the translation.

$$\frac{canon_\Delta(V) = E\texttt{<}\overline{T}\texttt{>} \qquad CT(E) = \texttt{class/interface } E\texttt{<}\overline{X}\texttt{>}\ldots}{[\![V]\!]_\Delta = E\texttt{<}[\![[\overline{T}/\overline{X}]\,\mathit{ord\text{-}full\text{-}tparams}(E)]\!]_\Delta\texttt{>}} \qquad \frac{canon_\Delta(V) = G}{[\![V]\!]_\Delta = \mathit{translated\text{-}constrainable\text{-}type}(G)}$$

$$\frac{}{[\![T\ m(\overline{U}\ \overline{x});]\!]_\Delta = [\![T]\!]_\Delta\ m([\![\overline{U}]\!]_\Delta\ \overline{x});} \qquad \frac{}{[\![T\ m(\overline{U}\ \overline{x})\{\texttt{return } e;\}]\!]_\Delta = [\![T]\!]_\Delta\ m([\![\overline{U}]\!]_\Delta\ \overline{x})\{\texttt{return } [\![e]\!]_\Delta;\}}$$

$$\frac{CT(E) = \texttt{class/interface } E\texttt{<}\overline{X}\texttt{>}\ldots \qquad \Delta = \mathit{env\text{-}for\text{-}body}(E)}{\mathit{transl\text{-}env\text{-}for\text{-}body}(E) = [\![\Delta]\!]_\Delta, \mathit{transl\text{-}ord\text{-}full\text{-}tparams}(E)}$$

$$\frac{\Delta = \mathit{env\text{-}for\text{-}body}(I) \qquad \mathit{transl\text{-}ord\text{-}full\text{-}tparams}(I) = \overline{Y}}{\begin{array}{c}[\![\texttt{interface } I\texttt{<}\overline{X}\texttt{>} : \overline{M}\ \texttt{where}\ \overline{rd_1}\ \{\texttt{type}\ \overline{A};\ \texttt{require}\ \overline{rd_2};\ \overline{ms};\}]\!] = \\ \texttt{interface } I\texttt{<}\overline{Y}\texttt{>} : [\![\overline{M}]\!]_\Delta\ \texttt{where}\ [\![\Delta]\!]_\Delta\{[\![\overline{ms}]\!]_\Delta;\}\end{array}}$$

$$\frac{\Delta = \mathit{env\text{-}for\text{-}body}(C) \qquad \mathit{transl\text{-}ord\text{-}full\text{-}tparams}(C) = \overline{Y}}{[\![\texttt{class } C\texttt{<}\overline{X}\texttt{>} : \overline{P}\ \texttt{where}\ \overline{rd}\ \{\texttt{type}\ \overline{A} = \overline{U};\ \overline{W}\ \overline{f};\ kd\ \overline{md}\}]\!] = \texttt{class } C\texttt{<}\overline{Y}\texttt{>} : [\![\overline{P}]\!]_\Delta\ \texttt{where}\ [\![\Delta]\!]_\Delta\{[\![\overline{W}]\!]_\Delta\ \overline{f};\ [\![kd]\!]_\Delta\ [\![\overline{md}]\!]_\Delta\}}$$

$$\frac{}{[\![\texttt{new } K(\overline{e})]\!]_\Delta = \texttt{new } [\![K]\!]_\Delta([\![\overline{e}]\!]_\Delta)} \qquad \frac{}{[\![T(e)]\!]_\Delta = [\![T]\!]_\Delta([\![e]\!]_\Delta)} \qquad \frac{}{[\![e.f]\!]_\Delta = ([\![e]\!]_\Delta).f} \qquad \frac{}{[\![e.m(\overline{d})]\!]_\Delta = ([\![e]\!]_\Delta).m([\![\overline{d}]\!]_\Delta)}$$

$$\frac{}{[\![x]\!]_\Delta = x} \qquad \frac{[\![U]\!]_\Delta \neq \texttt{this}}{[\![U : P]\!]_\Delta = [\![U]\!]_\Delta : [\![P]\!]_\Delta} \qquad \frac{}{[\![\texttt{this} : T]\!]_\Delta = \emptyset} \qquad \frac{}{[\![T == U]\!]_\Delta = \emptyset}$$

Figure 15: Definition of the translation function $[\![\cdot]\!]_\Delta$ from FGJ+IAPR to FGJ+I.

used to represent associated types match in their uses and definitions.

Translating some type T starts by finding a canonical form for T with the $canon_\Delta$ function, shown in Figure 14. This function selects one of the types that, based on Δ, can be proven equal to T. The definition of $canon_\Delta$ is a bit involved. We first define the relation $==_{weak}$, which is a weaker form of the type equality relation $==$. The definition is obtained by substituting $==_{weak}$ for $==$ in the rules TE-REFL, TE-FROM-ENV, TE-TRANS, TE-SYM, TE-ASSOC-CRG, and TE-PARAM-CRG. Note that the substitution obviously does not apply to the $==$ on the left-hand side of $\vdash$ in the rule TE-FROM-ENV. Hence, $==_{weak}$ is the same as $==$, except that the rule TE-ASSOC-TYPE is excluded from the former.

The helper function *canon-weak*$_\Delta$ works by ordering equivalent types into a list, with all instances first, followed by all type variables, followed by all associated types; and picking the first element of this sequence. For this, we assume a function *order-set* that transforms a set of types into an ordered list, using some arbitrary but fixed order, e.g, lexicographical ordering based on the alphabetical ordering of the characters in the type expressions; and a function *first* that selects the first element from a list. The symbol $+$ denotes list concatenation. If the first element of this list is a type variable, it is the final canonical form ($canon_\Delta$) of the input type. For an instance, the canonical form is obtained by canonicalizing the type arguments. The translation of associated types, however, is more intricate. The two rules for associated types compensate for the TE-ASSOC-TYPE rule missing from the definition of $==_{weak}$, but in a way that would correspond to only applying TE-ASSOC-TYPE from left to right. A more straightforward definition without this restriction would be possible, we believe, but the current definition guarantees that canonicalization preserves the *ok* property of types, which makes the safety proofs of the translation easier (see the proof of Lemma 9).

The *full-tparams* function collects the set of all associated types and type variables used in the definition of a given class or interface, recursing to its constraints and ancestors. The translated class or interface needs a type parameter for each element in this set. We can again use *order-set* to generate an ordered list from the set. We define *ord-full-tparams*(E) = *order-set*(*full-tparams*(E)). To get a valid list of type parameter names, we also assume the existence of a *translated-constrainable-type* helper function that maps constrainable types into type variable names. The naming scheme can be arbitrary, as long as each distinct type is mapped into a distinct name, and to the same name every time. The *transl-ord-full-tparams* function, used in the translation rules, then applies *translated-constrainable-type* to each element of the ordered list obtained as the result of *ord-full-tparams*.

With the above tools the translation guarantees that the type argument list of the translation of any instance $E\texttt{<}\overline{T}\texttt{>}$ will match the type parameter list of the translation of the definition of the class or interface E.

Note that it is possible to write a class or interface definition with an inconsistent set of constraints. A class with inconsistent constraints can never be successfully instantiated. In particular, every environment in the program (generated by class and interface definitions) must satisfy the consistency rule that $\Delta \vdash E\texttt{<}\overline{T}\texttt{>} == F\texttt{<}\overline{U}\texttt{>}$ implies $E = F$ and $\Delta \vdash \overline{T} == \overline{U}$. We define programs violating this rule to be invalid; the translation is not guaranteed to work for invalid programs. Detecting inconsistencies that are troublesome for the translation can be accomplished by checking each class or interface definition prior to translation. The outline of an algorithm to perform this checking is as follows: (1) let Δ_1 be the environment generated from the class or interface using *env-for-body*; (2) generate a new environment Δ_2 by replacing each type in Δ_1 by its canonical form (computed using *canon*); (3) check that the canonical form of each instance $E\texttt{<}\overline{T}\texttt{>}$ in Δ_1 is some other instance of E; and (4) check that the transitive closure of the same-type constraints in Δ_2 is not inconsistent. Inconsistencies originating from subtype constraints that require a class to derive from two unrelated classes or interfaces are not detected by the above algorithm. Inconsistent constraints of this kind will, however, remain inconsistent after translation, and are thus harmless.

5.4 Properties of the formalization

In order to show that our language extensions are type-safe and that our translation of these extensions into standard Generic C# is correct, we prove several properties of the extensions. We have defined translations on types, constraints, expressions, definitions, and environments; here, we show that these translations are consistent with each other and with vanilla FGJ+I. The semantics of our extensions are defined by translation into FGJ+I followed by evaluation in that system; the fact that our translation preserves program type behavior and typing then shows that our extensions are type-safe, as long as FGJ+I itself is type-safe. We start by establishing a number of minor results, and conclude with the theorems that state the main results about our language extensions. In the proofs, we assume that all classes and interfaces in the class table CT are *ok* by the rules CLASS-DEF-OK and INTERFACE-DEF-OK.

LEMMA 1.
a) If $\Delta \vdash T == U$, then $\sigma\Delta \vdash \sigma T == \sigma U$.
b) If $\Delta \vdash T <: U$, then $\sigma\Delta \vdash \sigma T <: \sigma U$.

PROOF. a) This proof is by induction on the derivation that $\Delta \vdash T == U$. For TE-REFL, TE-SYM, TE-TRANS, TE-ASSOC-CRG, TE-TPARAM-CRG, and TE-ASSOC-TYPE, the result is trivial. For TE-FROM-ENV, substitution on all constraints in the environment provides the desired result.

b) This proof is by induction on the derivation that $\Delta \vdash T <: U$. For S-REFL, S-TRANS, and S-VIA-EQ, the result is trivial (S-VIA-EQ uses part (a) of this lemma). For S-FROM-ENV, the substitution on constraints in the environment provides the desired result. For S-CLASS, $\mathcal{D}(\sigma(E\texttt{<}\overline{V}\texttt{>}), \sigma(\texttt{this})) = \mathcal{D}(E\texttt{<}\sigma\overline{V}\texttt{>}, \sigma(\texttt{this})) = \sigma\mathcal{D}(E\texttt{<}\overline{V}\texttt{>}, \texttt{this})$ by the definitions of substitution and $\mathcal{D}$. Thus, the base classes and interfaces in $\overline{P}$ are also substituted appropriately, and so the subtype relationship still holds after applying σ. □

LEMMA 2. *If $\Delta \vdash T$ ok, and $\Delta \vdash \sigma$ ok, then $\sigma\Delta \vdash \sigma T$ ok.*

PROOF. The proof is by induction on the structure of T. If T is a type variable, it is either left alone by σ or is redefined to an *ok* type, since σ is *ok*. If T is a class or interface instance (say, $E\texttt{<}\overline{V}\texttt{>}$), all of the substituted type arguments $\sigma\overline{V}$ are *ok* by the induction hypothesis. They meet their bounds from E by Lemma 1 (the bounds in E have the correct substitution applied by WF-CLASS-INSTANCE and WF-INTERFACE-INSTANCE), and so $\sigma(E\texttt{<}\overline{V}\texttt{>}) = E\texttt{<}\sigma\overline{V}\texttt{>}$ is *ok*. Otherwise, T is an associated type $U :: A$. Thus, $\sigma T = \sigma U :: A$. It is known that σU is *ok* by the induction hypothesis, so all that remains is to show that σU has a member type A. Since $U :: A$ is *ok* in its unsubstituted form, Δ must imply some constraint $U : F\texttt{<}\ldots\texttt{>}$, where $F\texttt{<}\ldots\texttt{>}$ has an associated type A. Thus, it must be true that $\Delta \vdash U <: F\texttt{<}\ldots\texttt{>}$, and so $\Delta \vdash \sigma U <: F\texttt{<}\ldots\texttt{>}$ by Lemma 1. Therefore, σU has an associated type A, and so $\sigma U :: A = \sigma T$ is *ok* in Δ. □

The next lemma shows that if we have some environment Δ where an instance is well-formed, some type U that would be well-formed in a class or interface definition, is well-formed in Δ after substituting the instance's type arguments into any occurrences of type parameters in U.

LEMMA 3. *If $\Delta \vdash E\texttt{<}\overline{T}\texttt{>}$ ok-constraint, and $\Delta \vdash W$ ok, and $\Delta \vdash W <: E\texttt{<}\overline{T}\texttt{>}$, and $\sigma = [\overline{T}/\overline{X}, \overline{W}/\texttt{this}]$, and $CT(E) =$ $\texttt{class/interface}$ $E\texttt{<}\overline{X}\texttt{>}$ $\ldots$, and env-for-body$(E) \vdash U$ ok, then $\Delta \vdash \sigma U$ ok.*

PROOF. Applying Lemma 2 to *env-for-body*$(E) \vdash U$ *ok* (an assumption) gives $\sigma($*env-for-body*$(E)) \vdash \sigma U$ *ok*. Under the assumption on CT, *env-for-body*(E) consists of some type parameter names (which are all *ok* after substitution because $\Delta \vdash \overline{T}$ *ok*), plus the requirements *propag-c*$(E\texttt{<}\overline{X}\texttt{>})$ and $\texttt{this} : E\texttt{<}\overline{X}\texttt{>}$, and thus $\sigma($*propag-c*$(E\texttt{<}\overline{X}\texttt{>})), \sigma(\texttt{this} : E\texttt{<}\overline{X}\texttt{>}) \vdash \sigma U$ *ok* follows. Because of the definition of σ, this is the same as $\sigma($*propag-c*$(E\texttt{<}\overline{X}\texttt{>})), W : E\texttt{<}\overline{T}\texttt{>} \vdash \sigma U$ *ok*. We know that $\Delta \vdash W : E\texttt{<}\overline{T}\texttt{>}$ by assumption, and so it remains to show that $\Delta \vdash \sigma($*propag-c*$(E\texttt{<}\overline{X}\texttt{>}))$ *satisfied*. This is true since it follows from $\Delta \vdash W <: E\texttt{<}\overline{T}\texttt{>}$ (using CPRG-INHERIT and CPRG-RIGHT-SUB) that $\sigma($*constr*$(E\texttt{<}\overline{X}\texttt{>}))$ is a subset of *constr*(W), and so $\sigma($*propag-c*$(E\texttt{<}\overline{X}\texttt{>}))$ is a subset of *propag-c*(W). □

LEMMA 4. *$\Delta \vdash T ==$ canon$_\Delta(T)$.*

PROOF. The proof is by induction on the derivation of *canon*$_\Delta(T)$ (i.e., the algorithm specified in the rules for computing *canon*$_\Delta(T)$). By the definition of *canon-weak*, $\Delta \vdash T ==_{weak}$ *canon-weak*$_\Delta(T)$. Because $\Delta \vdash T ==_{weak} U$ implies $\Delta \vdash T == U$, $\Delta \vdash T ==$ *canon-weak*$_\Delta(T)$ then follows. Based on this observation, the CANON-TVAR case is trivial. By the induction hypothesis, the type arguments are equal to their canonical forms in the CANON-INSTANCE case; thus, the main result follows from TE-TPARAM-CRG. The CANON-ASSOC case is by the induction hypothesis and TE-ASSOC-CRG, using similar reasoning. In the CANON-ASSOC-DEF case, the fact that $\Delta \vdash T == U :: A$, for some U and A, follows from *canon-weak*$_\Delta(T) = U :: A$. By the induction hypothesis and TE-ASSOC-CRG, $\Delta \vdash T == S :: A$, where S inherits from some class that binds some type V to the associated type A. Thus, by TE-ASSOC-TYPE, TE-SYM, and TE-TRANS, $\Delta \vdash T == V$, from which the result follows by the induction hypothesis and from the fact that *canon*$(T) =$ *canon*(V) by CANON-ASSOC-DEF. □

LEMMA 5. *$\Delta \vdash T == U$ if and only if canon$_\Delta(T) =$ canon$_\Delta(U)$.*

PROOF.
($\longleftarrow$) A consequence of Lemma 4, and the TE-SYM and TE-TRANS rules.
($\longrightarrow$) The proof is by induction on the derivation that $\Delta \vdash T == U$. For TE-REFL, TE-TRANS, and TE-SYM, the property follows directly from the induction hypothesis and properties of equivalence (=) on types. For TE-FROM-ENV, $\Delta \vdash T ==_{weak} U$ follows from $\Delta \vdash T == U$ because $==_{weak}$ includes TE-FROM-ENV. Thus, *canon-weak*$_\Delta(T)$ is the same as *canon-weak*$_\Delta(U)$ by the definition of *canon-weak*, and thus by the definition of *canon*, it is also the case that *canon*$_\Delta(T) =$ *canon*$_\Delta(U)$. For TE-TPARAM-CRG, the property follows from CANON-INSTANCE and the induction hypothesis. TE-ASSOC-CRG uses the definition of *canon* for associated types (the CANON-ASSOC and CANON-ASSOC-DEF rules), and the induction hypothesis. The TE-ASSOC-TYPE case follows from the CANON-ASSOC-DEF rule. □

The following lemma states that the name of the class or an interface is preserved in the translation:

LEMMA 6. *Assume $E \in dom(CT)$, and $\Delta \vdash \overline{T}$ ok.* $[\![E\texttt{<}\overline{T}\texttt{>}]\!]_\Delta = E\texttt{<}\overline{U}\texttt{>}$ *for some* $\overline{U}$.

PROOF. Because Δ is consistent, $\Delta \vdash E\texttt{<}\overline{T}\texttt{>} == F\texttt{<}\overline{U}\texttt{>}$ implies $E = F$. Thus $canon_\Delta(E\texttt{<}\overline{T}\texttt{>}) = E\texttt{<}\overline{U}\texttt{>}$ for some $\overline{U}$. The lemma then follows from the definition of translation for instantiated classes or interfaces. $\square$

THEOREM 1.
a) If $\Delta \vdash T == U$, then $[\![T]\!]_\Delta$ and $[\![U]\!]_\Delta$ are the same type in FGJ+I.
b) Assume $E \in dom(CT)$, $\Delta = env\text{-}for\text{-}body(E)$, and $\Delta' = transl\text{-}env\text{-}for\text{-}body(E)$. If $\Delta \vdash T <: P$, then $\Delta' \vdash [\![T]\!]_\Delta <: [\![P]\!]_\Delta$.

PROOF. a) The translation of a type is defined as the translation of its canonicalization (as computed by *canon*). Thus, the theorem follows directly from Lemma 5.

b) The proof is by induction on the derivation of $\Delta \vdash T <: P$, with case analysis on the last rule used. There are five subtyping rules: the cases for S-REFL and S-TRANS are straightforward, and the case for S-FROM-ENV follows because *transl-env-for-body*(E) includes translations of all constraints in *env-for-body*(E). The case for S-VIA-EQ follows from Theorem 1(a). In the final case, S-CLASS, let $T = F\texttt{<...>}$. Lemma 6 ensures that the translation of T is of the form $F\texttt{<...>}$. The case, and thus the theorem, follows from the definition of the translation of class and interface definitions, which preserves inheritance declarations. $\square$

LEMMA 7. *If $\Delta \vdash T ==_{weak} U$, $\Delta \vdash \Delta$ ok, and either $\Delta \vdash T$ ok or $\Delta \vdash U$ ok, then both $\Delta \vdash T$ ok and $\Delta \vdash U$ ok.*

PROOF. The proof is by induction on the structure of the derivation that $\Delta \vdash T ==_{weak} U$, with case analysis on the last rule used. The cases TE-REFL, TE-TRANS, TE-SYM, and TE-TPARAM-CRG are trivial. TE-FROM-ENV follows from $\Delta \vdash \Delta$ *ok*, because all constraints in Δ are *ok* in Δ, and so all types mentioned in those constraints are *ok* in Δ by the definition of a constraint being *ok*. For TE-ASSOC-CRG, let T be $V :: A$ and U be $W :: A$. Assume without loss of generality that $V :: A$ is *ok*. That means that V is *ok*, and so W is *ok* by the induction hypothesis. Therefore, W is a subtype of everything V is a subtype of by S-VIA-EQ, and in particular it is a subtype of whatever contains the associated type A. Therefore, W has the associated type A and W is *ok* so $W :: A$ is *ok*. This completes the last case and the proof. $\square$

LEMMA 8.
If $\Delta \vdash T$ ok and $\Delta \vdash \Delta$ ok, then $\Delta \vdash canon\text{-}weak_\Delta(T)$ ok.

PROOF. Follows directly from Lemma 7 and from $\Delta \vdash T ==_{weak} canon\text{-}weak_\Delta(T)$, which is obvious by the definition of $canon\text{-}weak_\Delta$. $\square$

LEMMA 9.
If $\Delta \vdash T$ ok and $\Delta \vdash \Delta$ ok, then $\Delta \vdash canon_\Delta(T)$ ok.

PROOF. The proof is by induction on the derivation of $canon_\Delta(T)$. If the last rule used is CANON-TVAR, the statement follows from $\Delta \vdash canon\text{-}weak_\Delta(T)$ *ok*, which is true by Lemma 8. The CANON-INSTANCE case follows from the induction hypothesis and from Lemma 4. Similarly, the induction hypothesis together with Lemma 4 proves the CANON-ASSOC case. In the final case CANON-ASSOC-DEF, S in the premise of the rule is *ok* in Δ by induction hypothesis, and thus also for $C\texttt{<}\overline{W}\texttt{>}$ in the rule, $\Delta \vdash C\texttt{<}\overline{W}\texttt{>}$ *ok*, and thus by CLASS-DEF-OK and Lemma 3, $\Delta \vdash V_i$ *ok*. $\square$

$$\frac{\overline{T}\ proper}{C\texttt{<}\overline{T}\texttt{>}\ proper} \qquad \frac{\overline{T}\ proper \qquad assoc\text{-}decl(I\texttt{<}\overline{T}\texttt{>}) = \emptyset}{I\texttt{<}\overline{T}\texttt{>}\ proper}$$

$$\frac{}{G\ proper}$$

Figure 16: The definition of *proper* types.

In the next lemma we use the definition of *proper* types, shown in Figure 16. Intuitively, proper types exclude all types that are, or contain, concept interfaces (interfaces that have associated types).

LEMMA 10. *If $\Delta \vdash T$ ok and $\Delta \vdash \Delta$ ok, then $canon_\Delta(T)$ proper.*

PROOF. The proof is by induction on the derivation of $canon_\Delta(T)$. The case for CANON-TVAR gives a proper type trivially. For CANON-ASSOC, we know by the induction hypothesis that the base of the resulting associated type is proper, and that this base does not inherit from any class (or associated type with interfaces, by Lemma 9); thus, the produced associated type is proper. For CANON-INSTANCE, the type arguments of the produced type are proper by the induction hypothesis, and the produced type is *ok* by Lemma 9; thus, the produced type itself is proper. In the CANON-ASSOC-DEF case, the type from the first part of the rule serves as input to *canon* again, and so the resulting type is proper by the induction hypothesis. $\square$

LEMMA 11. *If $\Delta \vdash T <: U$, $\Delta \vdash T$ ok, $\Delta \vdash U$ ok, U inherits from some class instance, and $\Delta \vdash V$ ok, then $\Delta \vdash [T/\texttt{this}]V == [U/\texttt{this}]V$.*

PROOF. The proof is by induction on the structure of V. If V is an instance, all of its type arguments are *ok*, and so by the induction hypothesis, this lemma is satisfied for them; the result then follows for V by TE-TPARAM-CRG. If V is a type variable, V is not affected by the substitutions for `this`, and so the result follows by TE-REFL. For associated types of `this`, the desired property is that $\Delta \vdash T :: A == U :: A$, which is true by CLASS-DEF-OK. For associated types of other types the induction hypothesis ensures that the base type of V satisfies the lemma, and thus the result is true for all of V by TE-ASSOC-CRG and the definition of substitution. $\square$

LEMMA 12. *If $CT(E) =$ `class`/`interface` $E\texttt{<}\overline{X}\texttt{>}$ ..., and $\Delta \vdash E\texttt{<}\overline{W}\texttt{>}$ ok-constraint, then $\Delta \vdash constr(E\texttt{<}\overline{W}\texttt{>})$ semi-ok and satisfied.*

PROOF. Given an arbitrary constraint Q in *constr*$(E\texttt{<}\overline{W}\texttt{>})$, the proof will be by induction on the structure of the derivation that $Q \in constr(E\texttt{<}\overline{W}\texttt{>})$. If Q is from CPRG-CLASS-DEF or CPRG-INTERFACE-DEF, it is trivially *semi-ok* and *satisfied* because it is a member of $[\overline{W}/\overline{X}]propag\text{-}c(E)$ by the definitions of those rules. From $\Delta \vdash E\texttt{<}\overline{W}\texttt{>}$ *ok*, we know that $\Delta \vdash [\overline{W}/\overline{X}]propag\text{-}c(E)$. If Q is from CPRG-INHERIT, Q is satisfied by the subtyping rule S-CLASS, and is *semi-ok* by the rules for a class or interface definition, and its instances, being *ok-constraint*. If Q is from CPRG-RIGHT-SUB, there are two types T and $F\texttt{<}\overline{V}\texttt{>}$ (where F accepts type parameters $\overline{Y}$), and a constraint R, such that Q is of the form σR, where $\sigma = [\overline{V}/\overline{Y}, T/\texttt{this}]$. By the induction hypothesis, $T : F\texttt{<}\overline{V}\texttt{>}$ is *semi-ok* and *satisfied* in Δ; R is *semi-ok* and *satisfied* in *env-for-body*(F) by CLASS/INTERFACE-DEF-OK. Thus,

$\Delta \vdash F\texttt{<}\overline{V}\texttt{>}$ *ok* and it must be shown that $\Delta \vdash \sigma R$ *semi-ok* and *satisfied*. By WF-CLASS-INSTANCE or WF-INTERF-INSTANCE and the equivalence between *env-for-body*$_c$ and *propag-c*, $\Delta \vdash \sigma(\textit{env-for-body}(F))$ *satisfied*. By Lemmas 1 and 2, $\sigma(\textit{env-for-body}(F)) \vdash \sigma R$ *semi-ok* and *satisfied*; thus $\Delta \vdash Q$ *semi-ok* and *satisfied*. □

LEMMA 13. *If $E \in dom(CT)$ and $\Delta = \textit{env-for-body}(E)$, then $\Delta \vdash \Delta$ ok.*

PROOF. The environment Δ is *ok* whenever each of its members (types and constraints) is *ok*. All of the type parameters in Δ are *ok* trivially, just by being members of Δ. Thus, the only interesting case is those members of Δ which are constraints. Let Q be a constraint from Δ. The environment Δ contains exactly the constraints resulting from *env-for-body*(E). This function can only produce constraints from two sources: constraints on `this` and constraints that come from *propag-c* and thus from *constr*. The constraints on `this` are *semi-ok* by SEMI-WF-THIS-CONSTR. The other kinds of constraints are also included in *constr*$(E\texttt{<}\overline{X}\texttt{>})$, and so are *semi-ok* in Δ by Lemma 12. All of these constraints are propagable, and so their being *semi-ok* implies that they are *ok*. Thus, all elements of Δ are *ok* in Δ, and so $\Delta \vdash \Delta$ *ok*. □

LEMMA 14.
Assume $CT(E)$ = `class/interface` $E\texttt{<}\overline{X}\texttt{>}\ldots$ and $\Delta \vdash \Delta$ ok. If $\Delta \vdash E\texttt{<}\overline{T}\texttt{>}$ ok-constraint and there exists some type W so that $\Delta \vdash W ok$ and $\Delta \vdash W <: E\texttt{<}\overline{T}\texttt{>}$, then every member of $[\overline{T}/\overline{X}, W/\texttt{this}]\textit{ord-full-tparams}(E)$ is ok in Δ.

PROOF. From the definitions of the functions *full-tparams* and *ord-full-tparams* and from Lemma 9, each element of *ord-full-tparams*(E) is *ok* in the environment *env-for-body*(E). The result is then an immediate application of Lemma 3. □

LEMMA 15. *If $CT(F)$ = `class/interface` $F\texttt{<}\overline{Y}\texttt{>}\ \ldots$, $\Delta_1 = \textit{env-for-body}(F)$, $\Delta_1 \vdash F\texttt{<}\overline{U}\texttt{>}$ ok, and $\Delta_2 \vdash T$ ok, then $[[\![[\overline{U}/\overline{Y}]\,\textit{ord-full-tparams}(F)]\!]_{\Delta_1}/$ $\textit{transl-ord-full-tparams}(F)][\![T]\!]_{\Delta_2} = [\![[\overline{U}/\overline{Y}]T]\!]_{\Delta_1}$.*

PROOF. Let V be $canon_{\Delta_2}(T)$. The proof is by induction on the structure of V, which is either a constrainable type or an instance; this is because V is *proper* by Lemmas 13 and 10. If V is constrainable, $[\![T]\!]_{\Delta_2}$ is a type in *transl-ord-full-tparams*(F), and so V is its corresponding entry in *ord-full-tparams*(F). Therefore, the left side of the equality above reduces to $[\![[\overline{U}/\overline{Y}]V]\!]_{\Delta_1}$, which is exactly the right side. If V is an instance type, assume it has the form $E\texttt{<}\overline{W}\texttt{>}$. The left side of the desired equality is $E\texttt{<}[[\![[\overline{U}/\overline{Y}]\textit{ord-full-tparams}(F)]\!]_{\Delta_1}/\textit{transl-ord-full-tparams}(F)]$ $[\![\overline{W}]\!]_{\Delta_2}\texttt{>}$. By the induction hypothesis, the inside of this can be replaced with the right side of the equality, and this produces $E\texttt{<}[\![[\overline{U}/\overline{Y}]\overline{W}]\!]_{\Delta_1}\texttt{>}$. This, after applying the translation rule for instances, is exactly the same as the right side of the desired equality. □

THEOREM 2.
Assume env-for-body$(E) = \Delta$ and transl-env-for-body$(E) = \Delta'$. If $\Delta \vdash T ok$, then $\Delta' \vdash [\![T]\!]_\Delta$ ok.

PROOF. The proof is by induction on the structure of $canon_\Delta(T)$, which can either be a constrainable type or an instance. Lemma 13 shows that $\Delta \vdash \Delta$ *ok*; Lemma 9 then shows that $\Delta \vdash canon_\Delta(T)$ *ok*. If $canon_\Delta(T)$ is a constrainable type, its translation is a type variable, which is *ok* in Δ' because Δ' is the result of *transl-env-for-body*.

The more challenging case is when $canon_\Delta(T)$ is an instance. If $canon_\Delta(T)$ is an instance, its type arguments are valid types because $canon_\Delta(T)$ is *ok*. Let $F\texttt{<}\overline{U}\texttt{>}$ be $canon_\Delta(T)$, and let $\overline{V}$ be $[\overline{U}/\overline{Y}]\,\textit{ord-full-tparams}(F)$, where $\overline{Y}$ are the declared type parameters of F. All of $\overline{V}$ is *ok* in Δ by Lemma 14.

Given that all of $\overline{V}$ is *ok* in Δ, $[\![\overline{V}]\!]_\Delta$ is *ok* in Δ' by the induction hypothesis (allowed because the type arguments of an instance returned by *canon* are the results of recursive applications of *canon*, by the definition of *canon*). By the definition of the translation function and the assumption of $canon_\Delta(T)$ being an instance, $[\![T]\!]_\Delta$ is an instance, and in particular is $F\texttt{<}[\![\overline{V}]\!]_\Delta\texttt{>}$.

The final part of showing that $F\texttt{<}[\![\overline{V}]\!]_\Delta\texttt{>}$ is *ok* in Δ' is to show that $[\![\overline{V}]\!]_\Delta$ meets its constraints from the translated definition of F. For this, let Δ_2 be *env-for-body*(F), and let Δ_2' be $[\![\Delta_2]\!]_{\Delta_2}$ (i.e., the constraints from the translation of F). It is known that $\Delta \vdash [\overline{U}/\overline{Y}]\Delta_2$ because $F\texttt{<}\overline{U}\texttt{>}$ is *ok* in Δ; this gives $\Delta' \vdash [\![[\overline{U}/\overline{Y}]\Delta_2]\!]_\Delta$ by Theorem 1. By Lemma 15, this is equivalent to $\Delta' \vdash [[\![[\overline{U}/\overline{Y}]\,\textit{ord-full-tparams}(F)]\!]_\Delta/$ $\textit{transl-ord-full-tparams}(F)]\Delta_2'$. This becomes $\Delta' \vdash [[\![\overline{V}]\!]_\Delta/$ $\textit{transl-ord-full-tparams}(F)]\Delta_2'$ by the definition of $\overline{V}$. This statement says exactly that the new type arguments $[\![\overline{V}]\!]_\Delta$ meet the constraints Δ_2' in the translation of F.

That completes the proof of the case for those T where $canon_\Delta(T)$ is an instance. Since *canon* can only return constrainable types and instances, the full theorem is now proven. □

THEOREM 3.
Assume $E \in dom(CT)$, $\Delta = \textit{env-for-body}(E)$, and $\Delta' = \textit{transl-env-for-body}(E)$. If $\Delta; \Gamma \vdash e : T$, then $\Delta'; [\![\Gamma]\!]_\Delta \vdash [\![e]\!]_\Delta : [\![T]\!]_\Delta$

PROOF. The proof is by induction on the derivation of $\Delta; \Gamma \vdash e : T$, with case analysis on the last rule used: TY-VAR is trivial; TY-SUB follows from Theorem 1(a); TY-NEW from Theorem 2; TY-FLD from the translation of class definitions; TY-METH from Theorem 1(a) and the translations of method signatures and definitions; TY-UCAST from Theorems 1(a) and 2; TY-DCAST from Theorems 1(a) and (b), and 2; and TY-SCAST from Theorems 1(a) and 2. □

LEMMA 16.
Assume $E \in dom(CT)$, $\Delta = \textit{env-for-body}(E)$, and $\Delta' = \textit{transl-env-for-body}(E)$. If $\Delta \vdash ms$ ok, then $\Delta' \vdash [\![ms]\!]_\Delta$ ok. If $\Delta \vdash md$ ok, then $\Delta' \vdash [\![md]\!]_\Delta$ ok.

PROOF. Trivial based on Theorems 2 and 3. □

THEOREM 4.
If `class` $C\texttt{<}\overline{X}\texttt{>}\ldots$ *ok in* FGJ+IAPR, *then* $[\![$`class` $C\texttt{<}\overline{X}\texttt{>}\ldots]\!]$ *ok in* FGJ+I. *If* `interface` $I\texttt{<}\overline{X}\texttt{>}\ldots$ *ok in* FGJ+IAPR, *then* $[\![$`interface` $I\texttt{<}\overline{X}\texttt{>}\ldots]\!]$ *ok in* FGJ+I.

PROOF. A class or an interface is *ok*, both in FGJ+IAPR and in FGJ+I, if all of its ancestors, constraints, fields, and methods are *ok* in the environments generated from the type parameters and constraints of the class/interface. The above theorems and lemmas show that translation of each of these sub-parts, including translation of environments, preserves well-formedness. The theorem follows. □

Theorem 5. *Assume* $\emptyset;\emptyset \vdash e : T$, $CT' = [\![CT]\!]$ *and* $e' = [\![e]\!]_\emptyset$. *If* $\langle CT, e\rangle$ *is a valid program in* FGJ+IAPR, *then* CT' *ok and* $\emptyset;\emptyset \vdash e' : [\![T]\!]_\emptyset$. *That is,* $\langle CT', e'\rangle$ *is a valid program in* FGJ+I.

Proof. Follows from Theorems 3 and 4. $\square$

6. RELATED WORK

The associated type extension described here can be seen as a type-safe variation of virtual types. Several other related formalisms and language features have been reported. Closest to ours is *nested inheritance* [21], a Java extension, that can be translated to standard Java with techniques similar to those described in this paper. Nested inheritance associates nested types with classes, rather than with individual objects as the original virtual type systems [9,10] do. Compared to our work, nested inheritance does not consider type parameters or binding member types to existing types, but rather, nested types must be bound to newly defined classes.

Virtual types are often viewed as an alternative to parameterized types. For instance, virtual types, as described in [9, 10], do not include a mechanism for type parameterization (beyond that of virtual types themselves); Thorup and Torgersen argue that type parameterization is beneficial even in the presence of virtual types [20]. Our formalism analogously combines type parameterization and associated types, and argues for the importance of constraint propagation in this combination. C# includes type parameters as its main mechanism for generics; our proposal adds to that, and does not intend to replace it.

Original virtual types require run-time type checks for full type safety. By introducing suitable restrictions (see, e.g. [10, 22]), several systems improve the static type safety of virtual types, while preserving the property of associating nested types to objects, not classes. These restrictions are similar to the requirement we impose that once defined, associated types are not redefined in subclasses. The MyType construct [19] allows a method to refer to the dynamic type of the **this** object, providing a solution to the binary method problem [23] by allowing method parameter types to vary covariantly in a statically type-safe way. MyType can be emulated with an extra type parameter, which resembles the technique we use for translating associated types to type parameters.

MyType was later extended into *type groups*, which, instead of just the single MyType, allow the definition of groups of types (either nested in a class or separate), which can be changed within subclasses, updating all references between types in the group appropriately [24, 25]. Family polymorphism [26] also allows groups of nested types to be inherited, but imposes restrictions based on object identity: the member types of two objects can only be used as the same type if the two objects are provably the same; this limitation was also pointed out in [21]. This restricts the interoperability of associated types of two distinct parameters of a generic function.

A recent formalization and improvement of family polymorphism, νObj, allows nested types to be aliases for existing types in addition to newly defined types. Furthermore, two nested types can be constrained to be the same [27]. νObj is presented as a foundational calculus for examining properties and behavior of nested types, and the Scala programming language is a practical implementation of these ideas [28]. Although supporting types as members of other types, the default model for these languages is that types are members of objects.

Structural virtual types allow virtual types to be used for some of the tasks that parameterized types are typically used for [20]. For example, two subclasses of the same class, with the same nested type definitions, are viewed as the same type. Also, a subtyping relation among parameterized types, such that the constraints of a parameterized class enter into the subtype relationship, is automatically defined: a polymorphic class is itself a type; adding more restrictions to its parameters produces a subtype of the original polymorphic type; instances of that polymorphic class are also subtypes of the uninstantiated class.

The work on *nested types* (distinct from nested inheritance) explores allowing constructions like ML signatures and structures, as well as objects, in one unified framework [29]. This work defines a formalization of records with type members and same-type constraints, but not in combination with parameterized types.

Support for associated types is also found in a few languages which are not predominantly object-oriented. C++ supports associated types with member **typedef**s, or with type functions known as *traits classes* [30]. ML modules can contain both types and values (including functions), and thus provide a direct representation for associated types [31]. However, an ML module is an entity by itself, distinct from a type; thus, types cannot be directly associated with other types.

Haskell's *functional dependencies* among type class parameters offer partial support for associated types [32]. Influenced by our results in [8], a more direct mechanism for associated types has been suggested [33]. This language extension adds associated types to type classes, but does not allow existing types to be bound as associated types, and does not allow constraints requiring particular pairs of associated types to be the same. These restrictions are lifted in subsequent work on *associated type synonyms* of the same authors [34]. The Haskell work does not consider the effects of an object-oriented type system, including subtype relationships, on associated types.

In contrast to systems related to associated types, work on constraint propagation appears infrequently in the literature. We are not aware of work formally defining constraint propagation, though the Cecil programming language includes it as a feature [35, § 4.2]. Also, Java's wildcard types allow a limited form of constraint propagation [36,37]. Systems which include full type inference for type constraints, such as type classes in Haskell [38], include the equivalent of constraint propagation, but often also introduce restrictions, such as forbidding a function name to be used in more than one type class.

One unique characteristic of our work is the combination of associated types and constraint propagation. Associated types can be bound to existing types, not only to newly created classes. Furthermore, we define associated types and constraint propagation as extensions to mainstream object-oriented languages with constrained generics, and describe translation of the extensions to such a language. The translation precisely describes a correspondence between type parameters and associated types, and demonstrates how the results directly apply to the generics of C# and Java.

7. CONCLUSION

A high degree of parameterization on types, typical for libraries built with generic programming, stretches the practical limits of generics in languages such as Java or C#. In particular, the number of type parameters, and the amount of code required to express constraints on type parameters, can easily become excessive. Associated types, as described in this paper, can significantly improve the situation: in contrast to type parameters, associated types allow types to be encapsulated in data structures. Note that associated types and parameterized types are not mutually exclusive features; our work demonstrates that these features are useful together, and describes their interaction precisely.

Associated types allow functional dependencies between types to be expressed well and encapsulated within interfaces, but do not directly encapsulate constraints on associated types or type parameters similarly. The full benefits of associated types depend on the other proposed extension: constraint propagation. These two extensions can be implemented separately, but together they allow a very concise expression of constraints for generic methods and classes, while still allowing separate type-checking.

The paper gives a rigorous description of both associated types and constraint propagation in a form that is directly applicable to mainstream object-oriented languages supporting F-bounded polymorphism, such as C# and Java. The proposed language features provide a significant improvement in the level of support for generic programming in these languages without fundamental changes to the languages or their implementations. We suggest a translation from a language with the extensions to one without them, but obviously a direct implementation is also feasible. The paper presents an idealized language, FGJ+IAPR, in the style of Featherweight Generic Java, to study the extensions formally, and proves their type-safety in this setting. We have implemented an experimental type checker for FGJ+IAPR, and a translation to C# [39]; work is also underway to implement the proposed extensions for the full C# language.

8. ACKNOWLEDGMENTS

This work was supported by NSF grant EIA-0131354 and a grant from the Lilly Endowment. The second author was supported by a Department of Energy High Performance Computer Science Fellowship. We are grateful to Jeremy Siek and Amr Sabry for their helpful comments on this work, and for Andrew Kennedy and Don Syme for early discussions on the topic. A number of anonymous reviewers made excellent suggestions that improved the paper.

9. REFERENCES

[1] Stepanov, A., Lee, M.: The Standard Template Library. Technical Report HPL-94-34(R.1), Hewlett-Packard Laboratories (1994) `http://www.hpl.hp.com/techreports`.

[2] Stepanov, A.: The Standard Template Library — how do you build an algorithm that is both generic and efficient? Byte Magazine **20** (1995)

[3] An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., Rauchwerger, L.: STAPL: An adaptive, generic parallel C++ library. In: Languages and Compilers for Parallel Computing. Volume 2624 of Lecture Notes in Computer Science. Springer (2001) 193–208

[4] Siek, J., Lee, L.Q., Lumsdaine, A.: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

[5] Siek, J., Lumsdaine, A.: A modern framework for portable high performance numerical linear algebra. In: Modern Software Tools for Scientific Computing. Birkhäuser (1999)

[6] Pitt, W.R., Williams, M.A., Steven, M., Sweeney, B., Bleasby, A.J., Moss, D.S.: The Bioinformatics Template Library–generic components for biocomputing. Bioinformatics **17** (2001) 729–737

[7] Microsoft Corporation: C# Version 2.0 Specification, March 2005 Draft. (2005) `http://msdn.microsoft.-com/vcsharp/programming/language`.

[8] Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A comparative study of language support for generic programming. In: OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, New York, NY, USA, ACM Press (2003) 115–134

[9] Madsen, O.L., Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press (1989) 397–406

[10] Thorup, K.K.: Genericity in Java with virtual types. In: ECOOP '97. Volume 1241 of Lecture Notes in Computer Science. (1997) 444–471

[11] Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM Press (1989) 273–280

[12] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23** (2001) 396–450

[13] Kennedy, A., Syme, D.: Transposing F to C#: Expressivity of polymorphism in an object-oriented language. Concurrency and Computation: Practice and Experience **16** (2004) 707–733

[14] Jazayeri, M., Loos, R., Musser, D., Stepanov, A.: Generic Programming. In: Report of the Dagstuhl Seminar on Generic Programming, Schloss Dagstuhl, Germany (1998)

[15] Kapur, D., Musser, D.: Tecton: a framework for specifying and verifying generic system components. Technical Report RPI–92–20, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180 (1992)

[16] Austern, M.H.: Generic programming and the STL: Using and extending the C++ Standard Template Library. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1998)

[17] Silicon Graphics, Inc.: SGI Implementation of the Standard Template Library. (2004) `http://www.sgi.com/tech/stl/`.

[18] International Organization for Standardization: ISO/IEC 14882:1998: Programming languages — C++, Geneva, Switzerland (1998)

[19] Bruce, K.B., Fiech, A., Petersen, L.: Subtyping is not a good "match" for object-oriented languages. In: ECOOP '97. Volume 1241 of Lecture Notes in Computer Science., Springer-Verlag (1997) 104–127

[20] Thorup, K.K., Torgersen, M.: Unifying genericity — combining the benefits of virtual types and parameterized classes. In: ECOOP '99. Volume 1628 of Lecture Notes in Computer Science., Springer-Verlag (1999) 186–204

[21] Nystrom, N., Chong, S., Myers, A.C.: Scalable extensibility via nested inheritance. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, ACM Press (2004) 99–115

[22] Torgersen, M.: Virtual types are statically safe. In: FOOL 5: Workshop on Foundations of Object-Oriented Languages. (1998) `http://pauillac.inria.fr/~remy/fool/`.

[23] Bruce, K.B., Cardelli, L., Castagna, G., Eifrig, J., Smith, S.F., Trifonov, V., Leavens, G.T., Pierce, B.C.: On binary methods. Theory and Practice of Object Systems **1** (1995) 221–242

[24] Bruce, K.B., Odersky, M., Wadler, P.: A statically safe alternative to virtual types. In: ECOOP '98. Volume 1445. (1998) 523–549

[25] Bruce, K.B., Vanderwaart, J.C.: Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In: Mathematical Foundations of Programming Semantics. Volume 20 of Electronic Notes in Theoretical Computer Science. (2000)

[26] Ernst, E.: Family polymorphism. In: ECOOP '01. Volume 2072 of Lecture Notes in Computer Science., Springer (2001) 303–326

[27] Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: ECOOP '03. Volume 2743 of Lecture Notes in Computer Science., Springer-Verlag (2003) 201–224

[28] Odersky, M., Altherr, P., Cremet, V., Emir, B., et al.: The Scala Language Specification. Programming Methods Laboratory, EPFL. Version 1.0 edn. (2005) `http://scala.epfl.ch/docu/files/ScalaReference.pdf`.

[29] Odersky, M., Zenger, C.: Nested types. In: FOOL 8: Workshop on Foundations of Object-Oriented Languages. (2001)

[30] Myers, N.: A new and useful technique: "traits". C++ Report **7** (1995) 32–35

[31] Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press (1990)

[32] Jones, M.P.: Type classes with functional dependencies. In: ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems. Volume 1782 of Lecture Notes in Computer Science., New York, NY, Springer-Verlag (2000) 230–244

[33] Chakravarty, M.M.T., Keller, G., Peyton Jones, S., Marlow, S.: Associated types with class. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2005) 1–13

[34] Chakravarty, M.M.T., Keller, G., Peyton Jones, S.: Associated type synonyms. In: ICFP '05: Proceedings of the International Conference on Functional Programming, New York, NY, USA, ACM Press (2005) 241–253

[35] Chambers, C., the Cecil Group: The Cecil Language: Specification and Rationale, Version 3.1. University of Washington, Computer Science and Engineering. (2002) `http://www.cs.washington.edu/research/projects/cecil/`.

[36] Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: ACM Symposium on Applied computing, ACM Press (2004) 1289–1296

[37] Torgersen, M., Ernst, E., Hansen, C.P.: Wild FJ. In: FOOL 12: Workshop on Foundations of Object-Oriented Languages. (2005)

[38] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad-hoc. In: ACM Symposium on Principles of Programming Languages, ACM (1989) 60–76

[39] `http://research.cs.tamu.edu/jarvi/csharp` (2005)