

Lost In Translation: Formalizing Proposed Extensions to C[#]

Gavin M. Bierman

Microsoft Research Cambridge, UK
gmb@microsoft.com

Erik Meijer

Microsoft Corporation, USA
emeijer@microsoft.com

Mads Torgersen

Microsoft Corporation, USA
madst@microsoft.com

Abstract

Current real-world software applications typically involve heavy use of relational and XML data and their query languages. Unfortunately object-oriented languages and database query languages are based on different semantic foundations and optimization strategies. The resulting “ROX (Relations, Objects, XML) impedance mismatch” makes life very difficult for developers.

Microsoft Corporation is developing extensions to the .NET framework to facilitate easier processing of non-object-oriented data models. Part of this project (known as “LINQ”) includes various extensions to the .NET languages to leverage this support.

In this paper we consider proposals for C[#]3.0, the next version of the C[#] programming language. We give both an informal introduction to the new language features, and a precise formal account by defining a translation from C[#]3.0 to C[#]2.0. This translation also demonstrates how these language extensions do not require any changes to the underlying CLR.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory

General Terms Languages, Theory

1. Introduction

Many programmers struggle with the problem of accessing and integrating data that is not natively defined using an object-oriented programming language (most commonly this means XML and relational data). This impedance mismatch between the data models typically results in fragile code that is subject to only very weak compile-time type checking.

Whilst this is an old problem [11, 20] it has been recently revisited in the context of commercial object-oriented lan-

guages. For example, XJ is an extension of Java 1.4 to support first-class XML [13], and C ω is an extension to C[#]1.1 to support both XML and relational data [7]. (As is common, these recent industrial proposals have been preceded by decades of academic work. We give some details of this prior work in §7.)

Microsoft Corporation has recently announced extensions to the .NET framework as well as to the .NET languages to support the integration of non-object data.¹ This project, called LINQ (Language INtegrated Query), proposes a very general approach to the impedance mismatch problem; defining a general pattern of operators to query data as well as providing four implementations of this pattern—LINQ over objects for querying in-memory collections of objects, LINQ over XML for querying in-memory XML data, LINQ over SQL for simple relational data and LINQ over Entities for querying data from the ADO.NET Entity Data Model [9]. On top of this pattern, both C[#] and Visual Basic will provide special *query expression* syntax, and Visual Basic will, in addition, add first class support for XML in the form of XML literals for constructing XML documents and so-called axis members for traversing XML documents. Importantly, none of these extensions will require any change to the underlying virtual machine (CLR).

In this paper we focus on the current proposal for the next version (3.0) of Microsoft’s C[#] programming language [14]. The new features to appear, whilst interesting in their own right, are essentially provided to facilitate a convenient style of LINQ programming. These features are not particularly tied to C[#]. The next version of Visual Basic (9.0) will have similar extensions. Indeed, they could be added to any class-based object-oriented language.

We believe that a formal, mathematical approach is essential to set a firm foundation for researchers, implementors and users of programming languages. For C[#]3.0 this is especially pertinent: all the new language features are essentially syntactic sugar (of a sophisticated nature). Our formalization of C[#]3.0 makes this observation precise: we define a formal translation of a fragment of C[#]3.0 into C[#]2.0. In other words, our translation provides a semantics of the new language features in terms of pre-existing language features. The exist-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA’07, October 21–25, 2007, Montréal, Québec, Canada.
Copyright © 2007 ACM 978-1-59593-786-5/07/0010...\$5.00.

¹<http://msdn.microsoft.com/netframework/future/linq>

tance of such a translation shows why the underlying CLR need not be changed.

This paper makes a number of contributions:

- We define an imperative, core fragment of C[#]3.0 called FC₃[#]. This fragment whilst reasonably small, contains all the essential features of C[#] including *all* of the new language features that will appear in C[#]3.0.
- We formalize the compilation of query expressions into standard query operators.
- We define an imperative, core fragment of C[#]2.0 called FC₂[#] (other fragments are much weaker, e.g. [17]).
- We formally specify a type-directed translation of FC₃[#] to FC₂[#]. This translation builds on the techniques of bidirectional type checking first used for local type inference in System F [25]. As far as we are aware this is the first such application of this technique.
- We prove two essential safety properties of this translation: type-preservation and conservativity.

The rest of the paper is organized as follows. In §2 we give an informal introduction to the new language features in C[#]3.0. In §3 we define formally a core fragment of C[#]3.0, that we dub FC₃[#]. In §4 we show how FC₃[#] query expressions can be compiled out into method calls. In §5 we define formally a translation from C[#]3.0 to C[#]2.0. In §5.1 we first define a target language for our translation: a core fragment of C[#]2.0 that we call FC₂[#]. We use a technique from bidirectional type checking to define translations from FC₃[#] to FC₂[#] in §5.2 and §5.3. In §5.4 we discuss the extensions to type inference in C[#]3.0. In §6 we consider some important safety properties of our translation. We review some related work in §7, before concluding in §8.

2. An introduction to C[#] 3.0

In this section we present the key ideas behind the new language extensions in C[#]3.0, and provide a number of examples to illustrate these ideas. This section should serve as a programmer's introduction to C[#]3.0. We assume the reader is familiar with C[#]/Java-like languages.

Our running example uses a variant of the CIA Worldfactbook Database² that contains a wealth of geographic information about the world's countries. But for the purposes of our example let us assume that we store with each country just its area and population. We represent this schema in C[#] using the following class declaration.

```
public class Country{
    public string Name;
    public double Area;
    public int Population;
}
```

²<https://www.cia.gov/library/publications/the-world-factbook>

The following statement uses a number of the new features in C[#]3.0 to define a small subset of the countries database that we will use as our running example.

```
var Countries = new []
{
    new Country{ Name = "Palau",
                Area = 458,
                Population = 16952},
    new Country{ Name = "Monaco",
                Area = 1.95,
                Population = 31719},
    new Country{ Name = "Belize",
                Area = 22960,
                Population = 219296},
    new Country{ Name = "Madagascar",
                Area = 587040,
                Population = 13670507}
};
```

Given this array, we can query and then output an array containing all those countries with a population less than 210,000 people, in descending order of their area, as follows.

```
var temp =
    from c in Countries
    where c.Population < 210000
    orderby c.Area descending
    select c.Name;

foreach (var t in temp) Console.WriteLine(t);
```

Let us review this code in detail to understand why it was so simple to write. First the definition of `Countries` was simplified through the use of the functional-style, *object initialization* syntax, e.g.

```
new Country{ Name="Monaco",
             Area=1.95,
             Population=31719}
```

The declaration of `Countries` also illustrates an *implicitly typed local declaration*. In other words the `var` keyword enables us to drop the type of the declaration. However, this does not imply that we have dropped the static typing discipline of C[#]; in fact, the compiler synthesizes a type for us (in this case it is `Country []`).

Likewise the declaration also uses an *implicitly typed array creation*. We have not supplied the element type for the array, but rely on the C[#]3.0 compiler to synthesize it for us (in this case it will be `Country`).

The local variable `temp` is initialized using a SQL-style *query expression* to generate the required subset of countries. The resemblance to SQL is intentional to allow C[#] programmers familiar with SQL to easily transition to using the compact query expressions to filter and process data. Again, the declaration is implicitly typed; in this case the compiler will infer the type `IEnumerable<string>`.

All the language extensions in C#3.0 are essentially sophisticated coatings of syntactic sugar that are compiled away using various type-directed translations into plain C#2.0. The query expression above is actually compiled to the following method calls:

```
var temp = Countries
    .Where((c) => c.Population < 210000)
    .OrderByDescending((c) => c.Area)
    .Select((c) => c.Name);
```

This translation also highlights another new language extension: *λ-expressions*. These are lightweight syntax for delegates; for example, the *λ-expression* `(int x) => x+1` is just a simpler way of writing the anonymous method expression `delegate(int x){ return x+1; }`. A key difference, as highlighted in the previous code, is that *λ-expressions* need not supply types for their arguments.

The methods `Where`, `OrderByDescending` and `Select` are three of a number of so-called *standard query operators* that are new to the LINQ framework. They are in essence new methods understood by objects that implement the `IEnumerable<T>` interface.

To facilitate this extension of `IEnumerable<T>`, C#3.0 provides a general means to extend existing types and constructed types with additional methods. These additional methods are known as *extension methods*. They are essentially static methods that can be invoked using instance method syntax. Extension methods are declared by specifying the (new) modifier `this` on the first parameter of the methods. Extension methods can only be declared in non-generic, non-nested static classes. For example, we could define a new method `PlayAnthem` to our pre-existing `Country` class as follows:

```
namespace Foo
{
    public static class Extension
    {
        public static
            void PlayAnthem(this Country c, Volume vol){...}
    }
}
```

Thus if we bring this namespace into scope with a `using Foo;` directive, we can invoke the extension method as if it were an instance method on `Country` objects, as follows.

```
var Blze = new Country{ Name = "Belize",
                       Area = 22960,
                       Population = 219296};

Blze.PlayAnthem(Volume.Loud);
```

Again, this language feature is actually sugar. Given a method invocation of the form $e.m(\bar{f})$, the compiler first checks to see if there is an instance method m supported

by the receiver object e . If not, then the compiler attempts to process it as an extension method invocation by translating it to the static method invocation $m(e, \bar{f})$. (The name m is resolved first using the closest enclosing namespace, and then successively outwards through nested namespaces, until reaching the containing compilation unit.)

When writing query expressions it is common to want to return more than one item per element, for example to return both the name *and* area of countries with a population greater than 32,000. To allow this C#3.0 introduces the new notion of an *anonymous type*.

```
var temp2 =
    from c in Countries
    where c.Population > 32000
    select new {
        MyName=c.Name,
        MyArea=c.Area
    };
```

An anonymous type is a nameless class type that inherits directly from `object`. An anonymous object initializer is of the form `new{f1=e1; ... fn=en}` and essentially creates an object of an unknown (to the programmer) type, with fields f_1, \dots, f_n which are initialized to e_1, \dots, e_n . The name of an anonymous type is automatically generated by the C#3.0 compiler and cannot be referenced in a program.

Within the same assembly, two anonymous initializers that specify a sequence of properties with the same names and types are guaranteed to produce instances of the same internal type. For example:

```
var city1 = new {City="London", GDPperCapita=31400};
var city2 = new {City="Paris", GDPperCapita=30100};
if (city1 == city2) Console.WriteLine("Same city!");
```

The comparison on the last line is permitted because `c1` and `c2` are of the same anonymous type.

We have seen that *λ-expressions* can be assigned to delegate types. However the extensibility features of the LINQ framework mean that it is often convenient to allow code to be emitted as data. The LINQ framework provides a new type `Expression<T>`, and C#3.0 allows a *λ-expression* to also be implicitly converted to an expression tree. For example:

```
Func<int,int> f = (x)=>x+1; //Code
Expression<Func<int,int>> e = (y) => y+1; //Data

Console.WriteLine(f(42)); //Prints 43
Console.WriteLine(e.Body.NodeType); //Prints Add
```

Expression trees are efficient in-memory representations of the *λ-expressions* and support a number of useful manipulation methods (such as `Body` and `NodeType` used above). The LINQ over SQL implementation uses these to translate expression trees to SQL statements that can be executed on the database engine. (This form of meta-programming can have

other interesting uses; see [27] for a related system built on the same API.)

To summarize, the new language extensions appearing in C[#]3.0 include:

- *Implicitly typed local variables*, which permit the type of local variables to be inferred from the expressions used to initialize them.
- *Extension methods*, which make it possible to extend existing types and constructed types with additional methods.
- *λ-expressions*, an evolution of anonymous methods that provides improved type inference and conversions to both delegate types and expression trees.
- *Object and collection initializers*, which ease construction and initialization of complex object instances and collections. In addition, there is support for implicitly typed arrays, a form of array creation and initialization that infers the element type of the array from an array initializer.
- *Anonymous types*, which are tuple types automatically inferred and created from object initializers.
- *Query expressions*, which provide a language integrated syntax for queries that is similar to relational and hierarchical query languages such as SQL, XQuery, and (list) comprehensions in Haskell and Python.
- *Expression trees*, which permit λ-expressions to be represented as data (expression trees) instead of as code (dellegates).

3. FC₃[#]: A core fragment of C[#]3.0

In the rest of this paper we study formally the essence of C[#]3.0. We adopt a formal, mathematical approach and define a core calculus FC₃[#], similar to core subsets of Java such as FJ [16], MJ [8] and ClassicJava [12]. Whilst small, our core calculus supports all the essential object-oriented features (classes, overloading, inheritance, side-effects) as well as *all* the new features to appear in C[#]3.0, and yet still remains amenable to formal reasoning. In addition, FC₃[#] is a completely valid subset of C[#]3.0, in that every valid FC₃[#] program is literally an executable C[#]3.0 program.

In this section we define formally the syntax for FC₃[#] programs. In §4, subsequent sections we show how query expressions are compiled into method invocations and then in §5 we show how FC₃[#] programs are compiled into FC₂[#] (a featherweight fragment of C[#]2.0).

A FC₃[#] program consists of a sequence of zero or more of what we dub ‘extension class’ declarations (an extension class contains only extension methods) followed by one or more ‘standard’ class declarations. Given an FC₃[#] program we assume that there is a unique designated method within

the standard class declarations that serves as the entry point (the `main` method).

We have simplified matters in comparison to C[#]3.0 in that we do not support nesting of extension classes (nor, hence, the resolution of nested namespaces whilst compiling extension method invocation). Thus we assume that there is a single, implicit namespace (called `Extensions`, say) surrounding the extension classes and also that immediately before the ‘standard’ class declarations there is an implicit `using Extensions;` statement that imports all these extension methods. For regularity, we do not allow extension method declarations anywhere other than in extension classes.

FC₃[#] programs:

$p ::= \overline{ecd} \overline{cd}$	Program
$ecd ::=$	Extension class declaration
<code>public static class S {\overline{emd}}</code>	
$emd ::=$	Extension method declaration
<code>public static ϕ m<\overline{X}>(this τ x, $\overline{\tau}$ \overline{x}) {\overline{s}}</code>	
$cd ::=$	Class declaration
<code>public class C<\overline{X}>:C<$\overline{\tau}$> {\overline{fd} \overline{md} \overline{cmd}}</code>	
$fd ::=$ <code>public τ f;</code>	Field declaration
$md ::=$	Method declaration
<code>public virtual ϕ m<\overline{X}>($\overline{\tau}$ \overline{x}) {\overline{s}}</code>	
<code>public override ϕ m<\overline{X}>($\overline{\tau}$ \overline{x}) {\overline{s}}</code>	
$cmd ::=$	Constructor method declaration
<code>public C<\overline{X}>($\overline{\tau}$ \overline{x}) : this (\overline{e}) {\overline{s}}</code>	
<code>public C<\overline{X}>($\overline{\tau}$ \overline{x}) : base (\overline{e}) {\overline{s}}</code>	

An extension class declaration is simply a static class declaration that consists of a sequence of extension method declarations. As explained in §2, an extension method is a static method where the first parameter has a `this` annotation.

A ‘standard’ class declaration consists of zero or more field declarations, zero or more method declarations, and one or more constructor method declarations. Methods must be defined either `virtual` or `override` and, for simplicity, we require all methods be `public`. For conciseness, we do not model `static` methods (other than extension methods) and non-virtual instance methods, and we do not consider other modifiers such as `private` and `sealed`. However, we do support generic class declarations and generic method declarations that were included in C[#]2.0 [17]. Constructor methods are somewhat orthogonal to the concerns of this paper. For simplicity, they are treated essentially as normal methods with the distinguished name `.ctor`.

A distinctive feature of the new features in C[#]3.0 is that they do *not* require any further extensions to the type grammar of C[#]. This is in contrast to, say, the approach taken by C_w [7] where a number of new types were added to C[#]. Hence the

grammar for FC_3^\sharp types is identical to an equivalent fragment of $C^\sharp 2.0$, and is as follows.

Types:

$\phi ::=$	Return type
τ	Type
void	Void type
$\tau ::=$	Type
γ	Value type
ρ	Reference type
X	Type parameter
$\gamma ::=$	Value Type
bool	Boolean
int	Integer
$\rho ::=$	Reference Type
$C < \bar{\tau} >$	Class type
$D < \bar{\tau} >$	Delegate type
$\tau []$	Array type

In C^\sharp the type **void** can only appear as a return type for method and delegate declarations. The two main categories of FC_3^\sharp types are then value types and reference types. Value types include the base types; for simplicity we shall consider just two: **bool** and **int**. We do not include nullable types in our core fragment, although they are simple to add.

FC_3^\sharp reference types include arrays, class types and delegate types. To simplify the presentation, we only consider single dimension arrays, and we write D to range over delegate types and C to range over class types. Following GJ [16] we permit the shorthand C for $C < >$. For simplicity we do not model constraints on generic parameters.

As is usual with small calculi, we assume a number of pre-defined classes. In addition to the **object** class, we assume the classes $IEnumerable < T >^3$ and $Expression < T >$ which play an important role in LINQ.

For succinctness, we do not support user-defined delegate declarations in FC_3^\sharp . For the purposes of this paper, it is sufficient to simply assume a number of predefined delegates $Func$ that would be defined in C^\sharp as follows:

```

delegate R Func<R>();
delegate R Func<T1,R>(T1 arg);
delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);
...

```

FC_3^\sharp expressions are split into three categories: ordinary expressions, statement expressions, and query expressions. Statement expressions are expressions that can be used as statements. Query expressions are the new comprehension

syntax for LINQ queries. In §4 we detail how they are translated into standard query operators.

Expressions:

$e ::=$	Expression
b	Boolean
i	Integer
$e \oplus e$	Built-in operator
x	Variable
null	Null
$(\tau) e$	Cast
$e.f$	Field access
delegate $(\bar{\tau} \bar{x}) \{ \bar{s} \}$	Anonymous method expression
$(\bar{x}) \Rightarrow e$	Implicit λ -expression
$(\bar{\tau} \bar{x}) \Rightarrow e$	Explicit λ -expression
se	Statement expression
qe	Query expression
$se ::=$	Statement expression
$e(\bar{e})$	Delegate invocation
$e.m(\bar{e})$	Implicit method invocation
$e.m < \bar{\tau} > (\bar{e})$	Explicit method invocation
new $C < \bar{\tau} > (\bar{e})$	Object creation
new $C < \bar{\tau} > \{ \bar{f} = \bar{i}\bar{e} \}$	Object initialization
new $C < \bar{\tau} > \{ \bar{n}\bar{a}\bar{e} \}$	Collection initialization
new $\{ \bar{f} = \bar{e} \}$	Anonymous object creation
new $[] \{ \bar{e} \}$	Implicit array creation
new $\tau [] \{ \bar{e} \}$	Explicit array creation
$x = e$	Variable assignment
$ie ::=$	Object initializer expression
e	Expression
$\{ \bar{f} = \bar{i}\bar{e} \}$	Object initializer
$\{ \bar{n}\bar{a}\bar{e} \}$	Collection initializer

For simplicity, we assume only two classes of literals: booleans and integers. We assume a number of built-in primitive operators, such as $=$, $||$ and $\&\&$. In the grammar we write $e \oplus e$, where \oplus denotes an instance of one of these operators. We do not consider these operators further as their meaning is clear. We assume that x ranges over variable names, f ranges over field names and m ranges over method names (both virtual and extension). We assume that the set of variables includes the special variable **this**, which cannot be used as a parameter of a method declaration.

Following FJ [16] we adopt an overloaded ‘bar’ notation; for example, $\bar{\tau} \bar{f}$ is a shorthand for a possibly empty sequence $\tau_1 f_1, \dots, \tau_n f_n$.

Anonymous method expressions (AMEs) were introduced in $C^\sharp 2.0$, and provide a means to define a “nameless” method. They are unusual in that they are expressions that cannot *synthesize* a type—this notion will be defined later in §5—but they can (and must) be converted to a compatible delegate type. The body of an AME is treated like the body of a method, i.e. any **return** statements must respect the return type of the delegate type.

³ Rather than complicating FC_3^\sharp with the complexities of interfaces, we simply treat $IEnumerable$ as if it were a class.

$FC_3^\#$ supports the new, lightweight λ -expressions that will appear in $C^\#3.0$. We consider only those with expression bodies (those whose bodies are statement blocks are treated essentially as AMEs), but we do allow the formal parameter list to be either explicitly or implicitly typed.

As mentioned earlier, $FC_3^\#$ statement expressions are those expressions that can be used as a statement. This includes two forms of invocation expressions: applying a delegate to arguments and method invocation where we can either supply the type arguments or have them inferred [6].

$FC_3^\#$ supports all the new forms of statement expressions that will appear in $C^\#3.0$. This includes the new functional style of object and collection initializations. (In the latter, we write *nae* to range over any expression *except* for variable assignment expressions. This restriction ensures that object and collection initialization expressions are syntactically distinct.) In addition, $FC_3^\#$ supports implicitly typed array creation (where the array element type can be dropped), and anonymous object creation.

$FC_3^\#$ supports the two new forms of statements that will appear in $C^\#3.0$. These are implicitly typed declarations (signified by the keyword **var**) and the implicitly typed form of **foreach** statements. The grammar for $FC_3^\#$ statements is as follows.

Statements:

$s ::=$	Statement
<code>;</code>	Skip
<code>se;</code>	Expression statement
<code>if (e) s else s</code>	Conditional statement
<code>τ x = e;</code>	Explicit declaration
<code>var x = e;</code>	Implicit declaration
<code>e.f = e</code>	Field assignment
<code>return e;</code>	Return statement
<code>return;</code>	Empty return
<code>foreach(τ x in e) s</code>	Explicit foreach
<code>foreach(var x in e) s</code>	Implicit foreach
<code>{\bar{s}}</code>	Block

Finally, $FC_3^\#$ contains all the forms of query expressions that will appear in $C^\#3.0$.⁴ These are given by the following grammar.

Query Expressions:

$qe ::=$	Query expression
<code>from x in e qb</code>	Query expression
$qb ::=$	Query body
<code>\bar{qc} select e</code>	Select
<code>\bar{qc} group e by e</code>	Group
$qc ::=$	Query clause
<code>from x in e</code>	From clause

⁴Strictly speaking we have excluded *into* clauses and explicitly typed range variables, but they are purely shorthand and can be easily removed.

<code>let x = e</code>	Let clause
<code>where e</code>	Where clause
<code>join x in e on e equals e</code>	Join clause
<code>join x in e on e equals e into x</code>	Join-into clause
<code>orderby \bar{e} od</code>	Order clause

$od ::=$	Order direction
<code>ascending</code>	Ascending
<code>descending</code>	Descending

An $FC_3^\#$ query expression then consists of a from clause, followed by a number of supplementary query clauses, and ends with either a select clause or a group-by clause. As mentioned earlier, $C^\#3.0$ query expressions are simply convenient syntactic sugar for the use of the LINQ standard query operators. In the following section, we define the translation of an $FC_3^\#$ query expression into method calls. In $C^\#3.0$, order clauses are not required to have order direction annotations; if one is not supplied, it is assumed to be ascending. For simplicity, $FC_3^\#$ requires order directions on all order clauses.

In what follows we assume that $FC_3^\#$ programs are well-formed, e.g. no cyclic class hierarchies, correct method body construction, etc. These conditions can be easily formalized but we suppress the details for lack of space.

4. Compilation of query expressions

Query expressions are provided to enable programmers to succinctly write data processing expressions in a SQL-style syntax. However, they are just syntactic sugar. The $C^\#3.0$ compiler compiles them into standard query operators *before* typechecking the program. This provides programmers with great flexibility: *any* object can be queried using a query expression provided that its type supports the query operators that appear in the compilation of the query expression. (We detail this pattern in Appendix A.)

The compilation exploits the regular structure of query expressions. We demonstrate this with some examples that query a collection of customers defined as follows.

```
public class Customer
{
    public string Name;
    public string City;
    public int Age;
    public int Credit;
};
```

First, consider the simplest of examples:

```
var names =
    from c in Customers
    select c.Name;
```

This simply compiles into an instance of the `Select` query operator on the `Customers` object, as follows.

```
var names =
    Customers.Select((c)=>c.Name);
```

The compilation works by incrementally compiling the query clauses that follow the initial from clause in a similar way, until we get to the terminating select or group-by-clause. For example, consider the following example query expression that returns the names of all customers in descending order of their age that are in Redmond and have a credit limit of more than \$10,000.

```
var names =
    from c in Customers
    where c.City == "Redmond"
    where c.Credit > 10000
    orderby c.Age descending
    select c.Name;
```

The compilation process starts by considering the first query clause `where c.City="Redmond"` yielding the interim expression:

```
var names =
    from c in Customers
        .Where((c) => c.City == "Redmond")
    where c.Credit > 10000
    orderby c.Age descending
    select c.Name;
```

A similar step yields the interim expression:

```
var names =
    from c in Customers
        .Where((c) => c.City == "Redmond")
        .Where((c) => c.Credit > 10000)
    orderby c.Age descending
    select c.Name;
```

We now compile the orderby query clause to yield the interim expression:

```
var names =
    from c in Customers
        .Where((c) => c.City == "Redmond")
        .Where((c) => c.Credit > 10000)
        .OrderByDescending((c) => c.Age)
    select c.Name;
```

We now have a query of the simple form that we considered at the start of this section. The query is compiled to the following final form.

```
var names =
    Customers
        .Where((c) => c.City == "Redmond")
        .Where((c) => c.Credit > 10000)
        .OrderByDescending((c) => c.Age)
        .Select((c) => c.Name);
```

We specify formally the compilation of FC_3^\sharp query expressions in Figure 1.⁵

5. Translation of FC_3^\sharp to FC_2^\sharp

This section contains the main technical contribution of the paper: the formal details of a translation from FC_3^\sharp to FC_2^\sharp . This translation, whilst demonstrating that the new features in $C^\sharp 3.0$ do *not* require any CLR extensions, is actually quite subtle. It is for this reason that we believe that our formal approach is an appropriate one to take.

Our translation is expressed in the form of a bidirectional type system [25]. The original use of bidirectional type systems was to recover various omitted type annotations from context. We build on this approach and, in addition to various instances of similar local type inference, we also translate the new language features supported by $C^\sharp 3.0$ to simplify the LINQ style of programming.

More concretely, bidirectional type systems distinguish the two distinct phases of type *checking* and type *synthesis*. (There is also a phase of type *inference* that is used to generate type arguments for generic method invocations.) Type checking is the process of determining whether a given term can be assigned a particular given type. Type synthesis, on the other hand, is the process of automatically determining a type given a term. Synthesis mode is used when we do not know anything about the expected type of an expression; for example, the receiver subexpression in an invocation. The checking mode is used when the surrounding context determines the type of the expressions, and we only need to check whether the expression can be assigned the given type.

These two phases, whilst distinct, are actually interdefined. One particularly pleasant aspect of defining a bidirectional system is that it is very straightforward to read off an implementation from the definitions of these two relations.

As an example, consider the problem of translating a declaration. If it is explicitly typed, that is it is of the form: $\tau x = e;$, we clearly use type *checking* to see if the expression can be assigned the type τ . However, if it is an implicitly typed declaration of the form: `var y = f;`, we need to synthesize a type, τ' say, for the expression f . Then the implicitly typed declaration can be translated to an explicitly typed declaration $\tau' y = \dots$

The rest of this section is organized as follows. In §5.1 we discuss briefly our target language, FC_2^\sharp , which is a feather-weight fragment of $C^\sharp 2.0$. We describe the checking, synthesis and inference phases of our translation in §5.2, §5.3 and §5.4, respectively.

⁵The compiler may well perform some simple optimizations during the compilation of query expressions, in particular replacing generated instances of the subexpression `e.Select((x)=>x)` with `e`.

<pre> from x in e_1 select e_2 </pre>	$\stackrel{\text{def}}{=}$	$ e_1 .Select(x=> e_2)$
<pre> from x in e group x by k </pre>	$\stackrel{\text{def}}{=}$	$ e .GroupBy(x=>k)$
<pre> from x_1 in e_1 group e_2 by k </pre>	$\stackrel{\text{def}}{=}$	$ e .GroupBy(x=>k, x=> e_2)$
<pre> from x_1 in e_1 from x_2 in e_2 q </pre>	$\stackrel{\text{def}}{=}$	$\left\{ \begin{array}{ll} e_1 .SelectMany(x_2=> e_2 , (x_1, x_2)=> e_3) & \text{if } q \equiv \text{select } e_3 \\ \left \begin{array}{l} \text{from } z \text{ in } e. SelectMany(x_1=>e_2, (x_1, x_2)=>\text{new}\{x_1, x_2\}) \\ q[x_1 := z.x_1, x_2 := z.x_2] \end{array} \right. & \text{if } q \not\equiv \text{select } e_3, \text{ and} \\ & \text{where } z \text{ is fresh} \end{array} \right.$
<pre> from x_1 in e_1 let $x_2 = e_2$ q </pre>	$\stackrel{\text{def}}{=}$	$\left \begin{array}{l} \text{from } z \text{ in } e_1. Select(x_1=>\text{new}\{x_1, x_2=e_2\}) \\ q[x_1 := z.x_1, x_2 := z.x_2] \end{array} \right \quad \text{where } z \text{ is fresh}$
<pre> from x_1 in e_1 where e_2 q </pre>	$\stackrel{\text{def}}{=}$	$\left \begin{array}{l} \text{from } x_1 \text{ in } e_1. Where(x_1=>e_2) \\ q \end{array} \right $
<pre> from x_1 in e_1 join x_2 in e_2 on k_1 equals k_2 q </pre>	$\stackrel{\text{def}}{=}$	$\left\{ \begin{array}{ll} e_1 .Join(x_2=> e_2 , x_1=>k_1, x_2=>k_2, (x_1, x_2)=> e_3) & \text{if } q \equiv \text{select } e_3 \\ \left \begin{array}{l} \text{from } z \text{ in } e_1. Join(e_2, \\ \quad x_1=>k_1, \\ \quad x_2=>k_2, \\ \quad (x_1, x_2)=>\text{new}\{x_1, x_2\}) \\ q[x_1 := z.x_1, x_2 := z.x_2] \end{array} \right. & \text{if } q \not\equiv \text{select } e_3, \text{ and} \\ & \text{where } z \text{ is fresh} \end{array} \right.$
<pre> from x_1 in e_1 join x_2 in e_2 on k_1 equals k_2 into x_3 q </pre>	$\stackrel{\text{def}}{=}$	$\left\{ \begin{array}{ll} e_1 .GroupJoin(x_2=> e_2 , x_1=>k_1, x_2=>k_2, (x_1, x_3)=> e_3) & \text{if } q \equiv \text{select } e_3 \\ \left \begin{array}{l} \text{from } z \text{ in } e_1. GroupJoin(e_2, \\ \quad x_1=>k_1, \\ \quad x_2=>k_2, \\ \quad (x_1, x_3)=>\text{new}\{x_1, x_3\}) \\ q[x_1 := z.x_1, x_3 := z.x_3] \end{array} \right. & \text{if } q \not\equiv \text{select } e_3, \text{ and} \\ & \text{where } z \text{ is fresh} \end{array} \right.$
<pre> from x in e orderby k_1 od_1, k_2 od_2, \dots k_n od_n q </pre>	$\stackrel{\text{def}}{=}$	$\left \begin{array}{l} \text{from } x \text{ in } e. od_1 ^o(x=>k_1). \\ od_2 ^t(x=>k_2). \\ \dots \\ od_n ^t(x=>k_n) \\ q \end{array} \right \quad \text{where } \begin{array}{ll} \text{ascending} ^o & \stackrel{\text{def}}{=} \text{OrderBy} \\ \text{descending} ^o & \stackrel{\text{def}}{=} \text{OrderByDescending} \\ \text{ascending} ^t & \stackrel{\text{def}}{=} \text{ThenBy} \\ \text{descending} ^t & \stackrel{\text{def}}{=} \text{ThenByDescending} \end{array}$

Figure 1. Translation of query expressions

5.1 FC_2^\sharp : The target language fragment of $C^\sharp 2.0$

The target language for our translation is a corresponding featherweight fragment of $C^\sharp 2.0$, which we call FC_2^\sharp . Its syntax is essentially just a subset of the FC_3^\sharp syntax given in §3. Hence an FC_2^\sharp program is just a sequence of one or more class declarations. The syntax for FC_2^\sharp types is identical to that for FC_3^\sharp . The grammar for FC_2^\sharp expressions is identical except that λ -expressions and query expressions are removed. For convenience in the translation we add a new form of expression to FC_2^\sharp : a *block expression*, written $(\{\bar{s}\})^\tau$. This can be thought of as shorthand for the invocation expression $d()$ where the delegate d has been defined as `delegate() { \bar{s} }` and the `return` statements contained in \bar{s} can be implicitly converted to τ ; but we treat it here as if it were a new language form.

FC_2^\sharp statement expressions are the same as for FC_3^\sharp except that FC_2^\sharp does not support object initialization, anonymous object creations and implicit array creation—these are new features in $C^\sharp 3.0$. As our translation performs local type inference for implicitly typed method invocations, FC_2^\sharp does not have any form for implicit method invocation. Prelly to simplify the presentation of the translation, the syntax for explicit method invocations is also different. In FC_2^\sharp method invocations are of the form $e.M(\bar{e})$, where M is a *method descriptor* and is defined as follows.

Method Descriptor:

$$M ::= C \langle \overline{X_C} \rangle \langle \overline{\tau_C} \rangle :: m \langle \overline{X_m} \rangle \langle \overline{\tau_m} \rangle : (\overline{\tau_p}) \rightarrow \phi$$

The method descriptor is used to fully identify a specific method that is being called at a specific instantiation (both of the class within which it is defined, and of the method itself). These method descriptors are used inside the compiler to resolve overloading,⁶ and appear explicitly in MSIL, the bytecode language for the CLR (albeit with type parameters replaced by integers denoting their position) [28]. Consider the following class declaration.

```
class myC<T>
{
  public void myM<S>(T arg1, List<S> arg2){}
}
```

To invoke the method `myM` with a type argument `string` on an instance of (static) type `myC<int>`, the following method descriptor would be used.

$$myC<T>\langle int \rangle :: myM<S>\langle string \rangle : (T, List<S>) \rightarrow void$$

Subtyping (also known as implicit conversion) for FC_2^\sharp is standard. Interestingly, exactly the same rules are used for subtyping for FC_3^\sharp and are as follows.

⁶The C^\sharp overloading rules [14, §20.9.4] involve the formal parameter types both before and after instantiation, so the method descriptors store the formal parameter types pre-instantiation along with the instantiations.

Subtyping:

$$\frac{}{\tau_1 <: \tau_1} \text{[ST-Ref]} \quad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \text{[ST-Trans]}$$

$$\frac{\rho_1 <: \rho_2}{\rho_1 [] <: \rho_2 []} \text{[ST-ArrayCovar]} \quad \frac{}{\tau <: \mathbf{object}} \text{[ST-Obj]}$$

$$\frac{\mathbf{class} \ C_1 \langle \overline{X} \rangle : C_2 \langle \overline{\tau_2} \rangle}{C_1 \langle \overline{\tau_1} \rangle <: C_2 \langle \overline{\tau_2} \rangle [\overline{X} := \overline{\tau_1}]} \text{[ST-Sub]}$$

Rules [ST-Ref] and [ST-Trans] ensure that the subtyping relation is reflexive and transitive. Like Java, C^\sharp arrays are covariant, which is captured in the [ST-ArrayCovar] rule; although covariance is only supported for reference types. The rule [ST-Obj] ensures that `object` is the root of the type hierarchy.

The FC_2^\sharp type system is defined as a bidirectional type system [25]. The key concept of bidirectional type systems is that we provide judgement forms for both type *checking* and type *synthesis*.

The type checking relation for expressions comes in two flavours: one for implicit conversions and one for explicit conversions (which arises in cast expressions [14, §6.2]). The first judgement form is written $\Gamma \vdash e \downarrow_i \tau$ and is read informally that “in context Γ , the FC_2^\sharp expression e can be implicitly converted to type τ .” The second judgement form is written $\Gamma \vdash e \downarrow_x \tau \rightsquigarrow$ and is read informally that “in context Γ , the FC_2^\sharp expression e can be explicitly converted to type τ .” A context, Γ , is a function from variables to types.

The type synthesis judgement form for expressions is written $\Gamma \vdash e \uparrow^s \tau$ and is read informally that “in context Γ , the FC_2^\sharp expression e synthesizes the type τ .”

To help familiarize the reader with bidirectional type systems, we give the rules for type checking and type synthesis of expressions in Figure 2. The rest of the rules (type synthesis of statement expressions and type checking of statements) are given in Appendix B.

The checking relations are built on top of the synthesis relation, so we consider that relation first. Most of these rules are routine. The rule [S₂-Cast] can be read as follows: a cast expression $(\tau_1)e_1$ synthesizes a type τ_1 provided that e_1 can be explicitly converted to type τ_1 . Rule [S₂-FieldAccess] makes use of an auxiliary function *f_{type}*, which is a map from a type and a field name to a type. Thus *f_{type}*(τ, f) returns the type of field f in type τ .

The implicit conversion checking relation requires two special cases to deal with the `null` constant and AMEs—these are expressions which can not synthesize a type. The [IC₂-Null] rule asserts that the `null` constant can be implicitly converted to any reference type. Rule [IC₂-AME] makes use of another auxiliary function *d_{type}*, which is a map from

$$\begin{array}{c}
\boxed{\Gamma \vdash e_1 \downarrow_i \tau_1} \\
\\
\frac{}{\Gamma \vdash \mathbf{null} \downarrow_i \rho} [\text{IC}_2\text{-Null}] \quad \frac{d\text{type}(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \phi \quad \Gamma, \bar{x}: \bar{\tau}_0 \vdash \bar{s} \downarrow \phi}{\Gamma \vdash \mathbf{delegate}(\bar{\tau}_0 \bar{x})\{\bar{s}\} \downarrow_i D \langle \bar{\tau} \rangle} [\text{IC}_2\text{-AME}] \quad \frac{\Gamma \vdash e \uparrow^s \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e \downarrow_i \tau_2} [\text{IC}_2\text{-Synth}] \\
\\
\boxed{\Gamma \vdash e_1 \downarrow_x \tau_1} \\
\\
\frac{\Gamma \vdash e \downarrow_i \tau}{\Gamma \vdash e \downarrow_x \tau} [\text{XC}_2\text{-ImpConv}] \quad \frac{\Gamma \vdash e \uparrow^s \tau_1 \quad \tau_1 <:^x \tau_2}{\Gamma \vdash e \downarrow_x \tau_2} [\text{XC}_2\text{-Synth}] \\
\\
\boxed{\Gamma \vdash e_1 \uparrow^s \tau_1} \\
\\
\frac{}{\Gamma \vdash b \uparrow^s \mathbf{bool}} [\text{S}_2\text{-Bool}] \quad \frac{}{\Gamma \vdash i \uparrow^s \mathbf{int}} [\text{S}_2\text{-Int}] \quad \frac{}{\Gamma, x: \tau \vdash x \uparrow^s \tau} [\text{S}_2\text{-Var}] \quad \frac{\Gamma \vdash e_1 \downarrow_x \tau_1}{\Gamma \vdash (\tau_1)e_1 \uparrow^s \tau_1} [\text{S}_2\text{-Cast}] \\
\\
\frac{\Gamma \vdash e_1 \uparrow^s \tau_1 \quad f\text{type}(\tau_1, f) = \tau_2}{\Gamma \vdash e_1.f \uparrow^s \tau_2} [\text{S}_2\text{-FieldAccess}]
\end{array}$$

Figure 2. Type checking and synthesis for FC_2^\sharp expressions

delegate names to their associated type. We write delegate types as function types in the System F sense; in general they are written $\forall \bar{X}. (\bar{\tau}) \rightarrow \phi$. For example, the following delegate declaration:

```
List<Y> delegate Map<X,Y>(Func<X,Y> f, List<X> xs);
```

would be represented as the type

$$\forall X, Y. (\text{Func} \langle X, Y \rangle, \text{List} \langle X \rangle) \rightarrow \text{List} \langle Y \rangle$$

In the rule we use type application for conciseness; we write $d\text{type}(D)(\bar{\tau}) = \bar{\tau}_1 \rightarrow \phi_1$ to mean first use $d\text{type}$ to determine the type of the delegate D , say $\forall \bar{X}. (\bar{\tau}_0) \rightarrow \phi_0$ and then substitute the types $\bar{\tau}$ for \bar{X} resulting in the type $\bar{\tau}_1 \rightarrow \phi_1$. The rule $[\text{IC}_2\text{-AME}]$ also reflects the restriction in C^\sharp , that AMEs can only be implicitly converted to (valid) delegate types; even the following code fails to typecheck.

```
object idFun = delegate (int x){ return x; };
```

The rule $[\text{IC}_2\text{-Synth}]$ can be read as follows: an expression e can be implicitly converted to τ_2 if it synthesizes a type τ_1 such that there is an implicit conversion from τ_1 to τ_2 .

The explicit conversion checking relation includes the implicit conversion checking relation; this corresponds to the allowing of redundant cast expressions in C^\sharp . The rule $[\text{XC}_2\text{-ImpConv}]$ captures this inclusion.

The rule $[\text{XC}_2\text{-Synth}]$ can be read as follows: an expression e can be explicitly converted to τ_2 if it synthesizes a type τ_1 such that there is an explicit conversion from τ_1 to τ_2 (written $\tau_1 <:^x \tau_2$). For our core calculus, this relation can be defined quite simply as follows.

$$\tau_1 <:^x \tau_2 \stackrel{\text{def}}{=} \tau_2 <: \tau_1$$

In the full language, things are more complicated as, for example, C^\sharp supports user-defined explicit conversions [14, §6.2.5].

5.2 Checking translation

The first translation we define is the so-called *checking translation*, or c -translation for short. This is defined over both expressions and statements. Analogous to type checking of FC_2^\sharp expressions, there are two c -translations of FC_3^\sharp expressions: one for implicit conversions and one for explicit conversions. The first judgement form is written $\Gamma \vdash e_1 \downarrow_i \tau \rightsquigarrow e_{11}$, and can be informally read as “in context Γ , the FC_3^\sharp expression e_1 can be implicitly converted to the type τ yielding a FC_2^\sharp expression e_{11} .” The rules for forming valid such judgements are given in Figure 3. As expected, there are a number of cases for dealing with expressions that can not synthesize a type (the \mathbf{null} constant, AMEs and the various forms of λ -expressions).

There are two versions of rules for λ -expressions depending on whether they are being implicitly converted to a delegate type (in which case they are translated to AMEs, i.e. code), or to the `Expression` type (in which case they are translated into data). In the latter case, the translation function, written $\ll - \gg$, takes the λ -expression as well as the context and produces an object that represents that λ -expression (essentially its abstract syntax tree). We do not give details of this translation for lack of space.

The second judgement form is written $\Gamma \vdash e_1 \downarrow_x \tau \rightsquigarrow e_{11}$, and can be informally read as “in context Γ , the FC_3^\sharp expression e_1 can be explicitly converted to the type τ yielding a FC_2^\sharp expression e_{11} .” This translation is defined in terms of the translation for implicit conversions. The rules for forming valid such judgements are also given in Figure 3.

$$\boxed{\Gamma \vdash e_1 \downarrow_i \tau \rightsquigarrow e_{11}}$$

$$\frac{}{\Gamma \vdash \mathbf{null} \downarrow_i \rho \rightsquigarrow \mathbf{null}} \text{[IC-Null]} \quad \frac{dtype(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \phi \quad \Gamma, \bar{x}: \bar{\tau}_0 \vdash \bar{s}_1 \downarrow \phi \rightsquigarrow \bar{s}_{11}}{\Gamma \vdash \mathbf{delegate}(\bar{\tau}_0 \bar{x})\{\bar{s}_1\} \downarrow_i D \langle \bar{\tau} \rangle \rightsquigarrow \mathbf{delegate}(\bar{\tau}_0 \bar{x})\{\bar{s}_{11}\}} \text{[IC-AME]}$$

$$\frac{dtype(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \tau_1 \quad \Gamma, \bar{x}_0: \bar{\tau}_0 \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_{11}}{\Gamma \vdash (\bar{x}_0) \Rightarrow e_1 \downarrow_i D \langle \bar{\tau} \rangle \rightsquigarrow \mathbf{delegate}(\bar{\tau}_0 \bar{x})\{\mathbf{return} e_{11};\}} \text{[IC-ImpLamDel]}$$

$$\frac{dtype(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \tau_1 \quad \Gamma, \bar{x}_0: \bar{\tau}_0 \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_{11}}{\Gamma \vdash (\bar{\tau}_0 \bar{x}_0) \Rightarrow e_1 \downarrow_i D \langle \bar{\tau} \rangle \rightsquigarrow \mathbf{delegate}(\bar{\tau}_0 \bar{x})\{\mathbf{return} e_{11};\}} \text{[IC-ExpLamDel]}$$

$$\frac{dtype(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \tau_1}{\Gamma \vdash (\bar{x}_0) \Rightarrow e_1 \downarrow_i \mathbf{Expression} \langle D \langle \bar{\tau} \rangle \rangle \rightsquigarrow \ll \Gamma \vdash (\bar{x}_0) \Rightarrow e_1 : \bar{\tau}_0 \rightarrow \tau_1 \gg} \text{[IC-ImpLamExpr]}$$

$$\frac{dtype(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \tau_1}{\Gamma \vdash (\bar{\tau}_0 \bar{x}_0) \Rightarrow e_1 \downarrow_i \mathbf{Expression} \langle D \langle \bar{\tau} \rangle \rangle \rightsquigarrow \ll \Gamma \vdash (\bar{\tau}_0 \bar{x}_0) \Rightarrow e_1 : \bar{\tau}_0 \rightarrow \tau_1 \gg} \text{[IC-ExpLamExpr]}$$

$$\frac{\Gamma \vdash e_1 \uparrow^s \tau_0 \rightsquigarrow e_{11} \quad \tau_0 <: \tau_1}{\Gamma \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_{11}} \text{[IC-Synth]}$$

$$\boxed{\Gamma \vdash e_1 \downarrow_x \tau \rightsquigarrow e_{11}}$$

$$\frac{\Gamma \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_{11}}{\Gamma \vdash e_1 \downarrow_x \tau_1 \rightsquigarrow e_{11}} \text{[XC-ImpConv]} \quad \frac{\Gamma \vdash e_1 \uparrow^s \tau_0 \rightsquigarrow e_{11} \quad \tau_0 <:^x \tau_1}{\Gamma \vdash e_1 \downarrow_x \tau_1 \rightsquigarrow e_{11}} \text{[XC-Synth]}$$

Figure 3. c -Translation of FC_3^\sharp expressions

The c -translation of FC_3^\sharp statements is given in Figure 4. These rules are pretty routine. The rules [C-EForeach1] and [C-EForeach2] together form a simplification of the actual rules used in $\text{C}^\sharp 2.0$ —we do not consider the complications of supporting foreach statements over objects that support the so-called collection pattern [14, §8.8.4]. In the rule [C-EForeach2] it is assumed that the synthesized type τ_2 is *not* an array type.

The rules for declarations, [C-IDecSeq] and [C-EDecSeq] highlight the difference between c - and s -translations. We use an s -translation to synthesize the type for an implicitly typed declaration, and we use a c -translation for an explicitly typed declaration.

5.3 Synthesis translation

The second translation we define is the so-called *synthesis relation*, or s -translation for short. This is defined over just expressions; there are no rules for statements. Judgements are written $\Gamma \vdash e_1 \uparrow^s \tau_1 \rightsquigarrow e_{11}$, and can be informally read as “in context Γ , the FC_3^\sharp expression e_1 synthesizes the type τ_1 yielding a FC_2^\sharp expression e_{11} .” The rules for forming valid judgements are given in Figure 5.

The rules dealing with method invocation are the most complicated and so we shall explain them in detail. They make use of an auxiliary function $mtype$ that is a map from a type and a method name to a *method group*. Thus

$mtype(\tau, m)$ returns a method group that represents all the candidate methods called m that are accessible from type τ , i.e. it is a set of *method signatures* of the form $C \langle X_C \rangle \langle \bar{\tau}_C \rangle : m \langle X_m \rangle : (\bar{\tau}_p) \rightarrow \phi$. A method signature is essentially a method descriptor without the type arguments for the method (these have not been inferred or collected yet).

The rule [S-ExpMethInv] defines the s -translation of an explicit method invocation $e_1 . m \langle \bar{\tau}_1 \rangle (\bar{e}_2)$. First, we synthesize a type for e_1 and use the auxiliary function $mtype$ to build a method group MG. The next stage is to determine which of the methods in MG are actually applicable. For invocation expressions, this is quite straightforward: the applicable methods are those whose number of formal parameters matches the number of actual arguments and whose number of generic type parameters matches the number of supplied type arguments. We add the type argument list $\langle \bar{\tau}_1 \rangle$ to all the applicable methods from MG to form the *applicable method group*, AMG, which is hence a set of method descriptors.

Next we need to resolve this set using overloading resolution [14, §7.4.2]. Space prevents us from formalizing overloading resolution here, although it is relatively straightforward. Rather, we simply assume a function $bestMD$ that, when given an applicable method group, returns the best method descriptor (if it exists). Assuming that there is a best method descriptor, we c -translate the actual arguments \bar{e}_2 .

$$\boxed{\Gamma \vdash s_1 \downarrow \phi \rightsquigarrow s_{11}}$$

$$\begin{array}{c}
\frac{}{\Gamma \vdash ; \downarrow \phi \rightsquigarrow ;} \text{[C-Skip]} \quad \frac{\Gamma \vdash se_1 \uparrow^s \phi_1 \rightsquigarrow se_{11}}{\Gamma \vdash se_1 ; \downarrow \phi \rightsquigarrow se_{11};} \text{[C-ExpStatement]} \\
\\
\frac{\Gamma \vdash e_1 \downarrow_i \text{bool} \rightsquigarrow e_{11} \quad \Gamma \vdash s_1 \downarrow \phi \rightsquigarrow s_{11} \quad \Gamma \vdash s_2 \downarrow \phi \rightsquigarrow s_{21}}{\Gamma \vdash \text{if } (e_1) s_1 \text{ else } s_2 \downarrow \phi \rightsquigarrow \text{if } (e_{11}) s_{11} \text{ else } s_{21}} \text{[C-Cond]} \\
\\
\frac{\Gamma \vdash e_1 \uparrow^s \tau_1 \rightsquigarrow e_{11} \quad \text{ftype}(\tau_1, f) = \tau_2 \quad \Gamma \vdash e_2 \downarrow_i \tau_2 \rightsquigarrow e_{21}}{\Gamma \vdash e_1 . f = e_2 ; \downarrow \phi \rightsquigarrow e_{11} . f = e_{21}} \text{[C-FAss]} \\
\\
\frac{}{\Gamma \vdash \text{return}; \downarrow \text{void} \rightsquigarrow \text{return};} \text{[C-Return]} \quad \frac{\Gamma \vdash e_1 \downarrow_i \tau_0 \rightsquigarrow e_{11}}{\Gamma \vdash \text{return } e_1 ; \downarrow \tau_0 \rightsquigarrow \text{return } e_{11};} \text{[C-ReturnExp]} \\
\\
\frac{\Gamma \vdash e_1 \uparrow^s \tau_2 [] \rightsquigarrow e_{11} \quad x \notin \text{dom}(\Gamma) \quad \tau_1 <: \tau_2 \text{ or } \tau_2 <: \tau_1 \quad \Gamma, x : \tau_2 \vdash s_1 \downarrow \phi \rightsquigarrow s_{11}}{\Gamma \vdash \text{foreach } (\tau_1 x_1 \text{ in } e_1) s_1 \downarrow \phi \rightsquigarrow \text{foreach } (\tau_1 x_1 \text{ in } e_{11}) s_{11}} \text{[C-EForeach1]} \\
\\
\frac{\Gamma \vdash e_1 \uparrow^s \tau_2 \rightsquigarrow e_{11} \quad \tau_2 <: \text{IEnumerable} \langle \tau_3 \rangle \quad x \notin \text{dom}(\Gamma) \quad \tau_1 <: \tau_3 \text{ or } \tau_3 <: \tau_1 \quad \Gamma, x : \tau_3 \vdash s_1 \downarrow \phi \rightsquigarrow s_{11}}{\Gamma \vdash \text{foreach } (\tau_1 x_1 \text{ in } e_1) s_1 \downarrow \phi \rightsquigarrow \text{foreach } (\tau_1 x_1 \text{ in } e_{11}) s_{11}} \text{[C-EForeach2]} \\
\\
\frac{\Gamma \vdash e_1 \uparrow^s \tau_2 [] \rightsquigarrow e_{11} \quad x \notin \text{dom}(\Gamma) \quad \Gamma, x : \tau_2 \vdash s_1 \downarrow \phi \rightsquigarrow s_{11}}{\Gamma \vdash \text{foreach } (\text{var } x_1 \text{ in } e_1) s_1 \downarrow \phi \rightsquigarrow \text{foreach } (\tau_2 x_1 \text{ in } e_{11}) s_{11}} \text{[C-IForeach1]} \\
\\
\frac{\Gamma \vdash e_1 \uparrow^s \tau_2 \rightsquigarrow e_{11} \quad \tau_2 <: \text{IEnumerable} \langle \tau_3 \rangle \quad x \notin \text{dom}(\Gamma) \quad \Gamma, x : \tau_3 \vdash s_1 \downarrow \phi \rightsquigarrow s_{11}}{\Gamma \vdash \text{foreach } (\text{var } x_1 \text{ in } e_1) s_1 \downarrow \phi \rightsquigarrow \text{foreach } (\tau_3 x_1 \text{ in } e_{11}) s_{11}} \text{[C-IForeach2]}
\end{array}$$

$$\boxed{\Gamma \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}$$

$$\begin{array}{c}
\frac{\Gamma \vdash e_1 \uparrow^s \tau_1 \rightsquigarrow e_{11} \quad x \notin \text{dom}(\Gamma) \quad \Gamma, x : \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \text{var } x = e_1 ; \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1 x = e_{11} ; \overline{s_{11}}} \text{[C-IDecSeq]} \\
\\
\frac{\Gamma \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_{11} \quad x \notin \text{dom}(\Gamma) \quad \Gamma, x : \tau_1 \vdash \overline{s_1} \downarrow \phi \rightsquigarrow \overline{s_{11}}}{\Gamma \vdash \tau_1 x = e_1 ; \overline{s_1} \downarrow \phi \rightsquigarrow \tau_1 x = e_{11} ; \overline{s_{11}}} \text{[C-EDecSeq]}
\end{array}$$

Figure 4. *c*-Translation of FC_3^\sharp statements

The synthesized type of the method invocation is then the return type of the best method.

The rule [S-ImpMethInv] defines the *s*-translation of an implicit method invocation, i.e. where the type argument list is not supplied. In this case, we need to use type inference [14, §20.6.4] to infer the type argument list. This process is discussed more fully in §5.4 (including the breaking changes that will be made in $\text{C}^\sharp 3.0$) but for now we simply state the form of the type inference judgement. This is written $\Gamma; \overline{X} \vdash (\overline{e}) \rightsquigarrow (\overline{\tau}) \hookrightarrow \theta$, which can be informally read as “in context Γ , the argument list (\overline{e}) matches the formal parameter type list $(\overline{\tau})$ with free type parameters \overline{X} and infers the substitution θ .” A substitution is a function from type parameters to types.

Thus in the rule [S-ImpMethInv] we perform type inference for each method in the method group; the applicable method set then contains every method for which type inference

succeeds, along with the inferred type argument list. The rest of the rule is similar to [S-ExpMethInv].

Rule [S-ExpEMethInv] defines the *s*-translation of an extension method invocation of the form $e_1 . m \langle \overline{\tau_1} \rangle (\overline{e_2})$. First we synthesize a type for e_1 and check that there is no method m for that type. (Hence, extension methods have strictly lower precedence than regular instance methods.) We then use an auxiliary function *emtype* to get the method group containing the candidate extension methods. Recall from §2 that extension methods are defined in non-generic, static classes. Moreover, the first parameter is used to pass in the original receiver object. We employ some shorthand and write elements of the method group as $S : m \langle \overline{X}_m \rangle \langle \overline{\tau_1} \rangle : (\tau_r, \overline{\tau}_p) \rightarrow \phi$. This denotes an extension method m defined in static class S with type parameters \overline{X}_m where the **this** formal parameter is of type τ_r , the other parameters are of type $\overline{\tau}_p$ and the return type is ϕ . The

$$\begin{array}{c}
\boxed{\Gamma \vdash e_1 \uparrow^s \phi_1 \rightsquigarrow e_{11}} \quad \frac{}{\Gamma \vdash b \uparrow^s \mathbf{bool} \rightsquigarrow b} \text{[S-Bool]} \quad \frac{}{\Gamma \vdash i \uparrow^s \mathbf{int} \rightsquigarrow i} \text{[S-Int]} \quad \frac{}{\Gamma, x: \tau \vdash x \uparrow^s \tau \rightsquigarrow x} \text{[S-Var]} \\
\\
\frac{\Gamma \vdash e_1 \downarrow_x \tau_1 \rightsquigarrow e_{11}}{\Gamma \vdash (\tau_1)e_1 \uparrow^s \tau_1 \rightsquigarrow (\tau_1)e_{11}} \text{[S-Cast]} \quad \frac{\Gamma \vdash e_1 \uparrow^s \tau_1 \rightsquigarrow e_{11} \quad \text{ftype}(\tau_1, f) = \tau_2}{\Gamma \vdash e_1.f \uparrow^s \tau_2 \rightsquigarrow e_{11}.f} \text{[S-FieldAccess]} \\
\\
\frac{\Gamma \vdash e_1 \uparrow^s D\langle\bar{\tau}\rangle \rightsquigarrow e_{11} \quad \text{dtype}(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \phi \quad \Gamma \vdash \bar{e}_2 \downarrow_i \bar{\tau}_0 \rightsquigarrow \bar{e}_{21}}{\Gamma \vdash e_1(\bar{e}_2) \uparrow^s \phi \rightsquigarrow e_{11}(\bar{e}_{21})} \text{[S-DellInv]} \\
\\
\text{AMG} \stackrel{\text{def}}{=} \{C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::m\langle\bar{X}_m\rangle\langle\bar{\tau}_i\rangle: (\bar{\tau}_p) \rightarrow \phi \mid C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::m\langle\bar{X}_m\rangle: (\bar{\tau}_p) \rightarrow \phi \in \text{MG}, \\
\Gamma; \bar{X}_m \vdash \bar{e}_2 \sim \bar{\tau}_p[\bar{X}_C := \bar{\tau}_C] \hookrightarrow \theta, \theta \equiv [\bar{X}_m \mapsto \bar{\tau}_i]\} \\
\frac{M = C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::m\langle\bar{X}_m\rangle\langle\bar{\tau}_i\rangle: (\bar{\tau}_p) \rightarrow \phi \stackrel{\text{def}}{=} \text{bestMD}(\text{AMG}) \quad \Gamma \vdash \bar{e}_2 \downarrow_i \bar{\tau}_p[\bar{X}_C := \bar{\tau}_C, \bar{X}_m := \bar{\tau}_i] \rightsquigarrow \bar{e}_{21}}{\Gamma \vdash e_1.m(\bar{e}_2) \uparrow^s \phi[\bar{X}_C := \bar{\tau}_C, \bar{X}_m := \bar{\tau}_i] \rightsquigarrow e_{11}.M(\bar{e}_{21})} \text{[S-ImpMethInv]} \\
\\
\text{AMG} \stackrel{\text{def}}{=} \{C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::m\langle\bar{X}_m\rangle\langle\bar{\tau}_1\rangle: (\bar{\tau}_p) \rightarrow \phi \mid C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::m\langle\bar{X}_m\rangle: (\bar{\tau}_p) \rightarrow \phi \in \text{MG}, \\
|\bar{X}_m| = |\bar{\tau}_1|, |\bar{\tau}_p| = |\bar{e}_{21}|\} \\
\frac{M = C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::m\langle\bar{X}_m\rangle\langle\bar{\tau}_1\rangle: (\bar{\tau}_p) \rightarrow \phi \stackrel{\text{def}}{=} \text{bestMD}(\text{AMG}) \quad \Gamma \vdash \bar{e}_2 \downarrow_i \bar{\tau}_p[\bar{X}_C := \bar{\tau}_C, \bar{X}_m := \bar{\tau}_1] \rightsquigarrow \bar{e}_{21}}{\Gamma \vdash e_1.m\langle\bar{\tau}_1\rangle(\bar{e}_2) \uparrow^s \phi[\bar{X}_C := \bar{\tau}_C, \bar{X}_m := \bar{\tau}_1] \rightsquigarrow e_{11}.M(\bar{e}_{21})} \text{[S-ExpMethInv]} \\
\\
\text{AMG} \stackrel{\text{def}}{=} \{S::m\langle\bar{X}_m\rangle\langle\bar{\tau}_i\rangle: (\tau_r, \bar{\tau}_p) \rightarrow \phi \mid S::m\langle\bar{X}_m\rangle: (\tau_r, \bar{\tau}_p) \rightarrow \phi \rightarrow \in \text{MG}, \\
\Gamma; \bar{X}_m \vdash (e_1, \bar{e}_2) \sim (\tau_r, \bar{\tau}_p) \hookrightarrow \theta, \theta \equiv [\bar{X}_m \mapsto \bar{\tau}_i]\} \\
\frac{M = S::m\langle\bar{X}_m\rangle\langle\bar{\tau}_i\rangle: (\tau_r, \bar{\tau}_p) \rightarrow \phi \stackrel{\text{def}}{=} \text{bestMD}(\text{AMG}) \quad \Gamma \vdash (e_1, \bar{e}_2) \downarrow_i (\tau_r, \bar{\tau}_p)[\bar{X}_m := \bar{\tau}_i] \rightsquigarrow (e_{11}, \bar{e}_{21})}{\Gamma \vdash e_1.m(\bar{e}_2) \uparrow^s \phi[\bar{X}_m := \bar{\tau}_i] \rightsquigarrow S.M(e_{11}, \bar{e}_{21})} \text{[S-ImpEMethInv]} \\
\\
\text{AMG} \stackrel{\text{def}}{=} \{S::m\langle\bar{X}_m\rangle\langle\bar{\tau}_1\rangle: (\tau_r, \bar{\tau}_p) \rightarrow \phi \mid S::m\langle\bar{X}_m\rangle: (\tau_r, \bar{\tau}_p) \rightarrow \phi \rightarrow \in \text{MG}, |\bar{X}_m| = |\bar{\tau}_1|, |\bar{\tau}_p| = |\bar{e}_{21}|\} \\
\frac{M = S::m\langle\bar{X}_m\rangle\langle\bar{\tau}_1\rangle: (\tau_r, \bar{\tau}_p) \rightarrow \phi \stackrel{\text{def}}{=} \text{bestMD}(\text{AMG}) \quad \Gamma \vdash (e_1, \bar{e}_2) \downarrow_i (\tau_r, \bar{\tau}_p)[\bar{X}_m := \bar{\tau}_1] \rightsquigarrow (e_{11}, \bar{e}_{21})}{\Gamma \vdash e_1.m\langle\bar{\tau}_1\rangle(\bar{e}_2) \uparrow^s \phi[\bar{X}_m := \bar{\tau}_1] \rightsquigarrow S.M(e_{11}, \bar{e}_{21})} \text{[S-ExpEMethInv]} \\
\\
\text{MG} \stackrel{\text{def}}{=} \text{mtype}(C\langle\bar{\tau}\rangle, \text{.ctor}) \\
\text{AMG} \stackrel{\text{def}}{=} \{C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::\text{.ctor}: (\bar{\tau}_p) \mid C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::\text{.ctor}: (\bar{\tau}_p) \in \text{MG}, |\bar{\tau}_p| = |\bar{e}_{11}|\} \\
\frac{C\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::\text{.ctor}: (\bar{\tau}_p) \stackrel{\text{def}}{=} \text{bestMD}(\text{AMG}) \quad \Gamma \vdash \bar{e}_1 \downarrow_i \bar{\tau}_p[\bar{X}_C := \bar{\tau}_C] \rightsquigarrow \bar{e}_{11}}{\Gamma \vdash \mathbf{new} C\langle\bar{\tau}\rangle(\bar{e}_1) \uparrow^s C\langle\bar{\tau}\rangle \rightsquigarrow \mathbf{new} C\langle\bar{\tau}\rangle(\bar{e}_{11})} \text{[S-ObjCreation]} \\
\\
\frac{\text{fields}(C\langle\bar{\tau}\rangle) = \{\bar{f}: \bar{\tau}_1\} \quad \Gamma \vdash \bar{ie}_1 \downarrow_i \bar{\tau}_1 \rightsquigarrow \bar{ie}_{11}}{\Gamma \vdash \mathbf{new} C\langle\bar{\tau}\rangle\{\bar{f} = \bar{ie}_1\} \uparrow^s C\langle\bar{\tau}\rangle \rightsquigarrow (\{C\langle\bar{\tau}\rangle x = \mathbf{new} C\langle\bar{\tau}\rangle(); x.\bar{f} = \bar{ie}_{11}; \mathbf{return} x;\})^{C\langle\bar{\tau}\rangle}} \text{[S-ObjInit]} \\
\\
\frac{C\langle\bar{\tau}\rangle <: \mathbf{I}Enumerable\langle\bar{\tau}_1\rangle \quad \Gamma \vdash \bar{nae}_1 \downarrow_i \bar{\tau}_1 \rightsquigarrow \bar{nae}_{11}}{\Gamma \vdash \mathbf{new} C\langle\bar{\tau}\rangle\{\bar{nae}_1\} \uparrow^s C\langle\bar{\tau}\rangle \rightsquigarrow (\{C\langle\bar{\tau}\rangle x = \mathbf{new} C\langle\bar{\tau}\rangle(); x.\mathbf{add}(\bar{nae}_{11}); \mathbf{return} x;\})^{C\langle\bar{\tau}\rangle}} \text{[S-ColnInit]} \\
\\
\frac{\Gamma \vdash \bar{e}_1 \uparrow^s \bar{\tau}_1 \rightsquigarrow \bar{e}_{11} \quad \text{anontype}(\{\bar{f}: \bar{\tau}_1\}) += C}{\Gamma \vdash \mathbf{new} \{\bar{f} = \bar{e}_1\} \uparrow^s C \rightsquigarrow \mathbf{new} C(\bar{e}_{11})} \text{[S-AnonObjCreation]} \\
\\
\frac{CT = \{\tau_i \mid \Gamma \vdash e_i \uparrow^s \tau_i \rightsquigarrow e_{i1}, 1 \leq i \leq n\} \quad \tau_b = \text{best}(CT) \quad \Gamma \vdash e_1 \downarrow_i \tau_b \rightsquigarrow e_{12} \cdots \Gamma \vdash e_n \downarrow_i \tau_b \rightsquigarrow e_{n2}}{\Gamma \vdash \mathbf{new} []\{e_1, \dots, e_n\} \uparrow^s \tau_b [] \rightsquigarrow \mathbf{new} \tau_b []\{e_{12}, \dots, e_{n2}\}} \text{[S-ImpArray]} \\
\\
\frac{\Gamma \vdash \bar{e}_1 \downarrow_i \tau \rightsquigarrow \bar{e}_{11}}{\Gamma \vdash \mathbf{new} \tau []\{\bar{e}_1\} \uparrow^s \tau [] \rightsquigarrow \mathbf{new} \tau []\{e_{11}\}} \text{[S-ExpArray]} \quad \frac{\Gamma, x_1: \tau_1 \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_{11}}{\Gamma, x_1: \tau_1 \vdash x_1 = e_1 \uparrow^s \tau_1 \rightsquigarrow x_1 = e_{11}} \text{[S-VarAssign]}
\end{array}$$

Figure 5. *s*-Translation of expressions

process of building the applicable method group and overloading resolution is the same as in rule [S-ExpMethInv].

Rule [S-ImpEMethInv] defines the s -translation of an extension method invocation where the type argument list has been omitted. This works similarly as the explicit method invocation case, except that we use type inference to generate the type argument list for the applicable method group.

The s -translation of object creation expressions is defined in rule [S-ObjCreation]. As mentioned in §3, we treat constructors as methods with the special internal name `.ctor`. So, the rule is similar to that for explicit method invocation expressions.

The [S-ObjInit] rule shows how the new C#3.0 object initialization expression is just compiled into a block expression that first initializes the object and then assigns to the fields imperatively. The rule makes use of a c -translation on initialization expressions which is defined as follows.

c -translation of initializer expressions:

$$\frac{\Gamma \vdash \mathbf{new} \tau \{\bar{f} = \bar{i}e\} \downarrow_i \tau \rightsquigarrow e}{\Gamma \vdash \{\bar{f} = \bar{i}e\} \downarrow_i \tau \rightsquigarrow e} \text{ [C-ObjIE]}$$

$$\frac{\Gamma \vdash \mathbf{new} \tau \{\bar{n}ae\} \downarrow_i \tau \rightsquigarrow e}{\Gamma \vdash \{\bar{n}ae\} \downarrow_i \tau \rightsquigarrow e} \text{ [C-ColnIE]}$$

The rule [S-ColnInit] similarly translates a collection initialization expression into a block expression that first creates the object, and then uses the `add` method to systematically add the (translated) elements to the collection. This rule uses a small simplification for succinctness: we write as an assumption $C \langle \bar{\tau} \rangle <: \text{IEnumerable} \langle \tau_1 \rangle$, by which we mean there exists a (single) type τ_1 such that the subtyping judgement is true.

The rule [S-AnonObjCreation] defines the s -translation of an anonymous object creation expression. As has been discussed earlier, anonymous objects are represented by privately named classes. Moreover, two anonymous object initializers with matching field names and synthesized types will produce objects of the same internal class. For example, the following code is type correct.

```
var point1 = new { x=32, y=32 };
var point2 = new { x=0, y=-1 };
if (point1 == point2) Console.WriteLine("Equal");
```

Our rule makes use of a global auxiliary side-effecting function, *anontype*, which is a map from a set of field-type pairs to the private class name that will represent instances of these objects. This function either generates a fresh class name for a set of field-type pairs that has not been seen before, or returns the pre-existing private class name. We use the `+=` notation to highlight this caching behaviour.

The rule [S-ImpArray] first attempts to synthesize types for all the element expressions and collects these types into a set (recall that not all expressions can synthesize a type). It then uses an auxiliary function *best* which is a (partial) function that maps a set of types, to the single, unique type from the input set to which the rest of the input set can be converted to (if such a type exists). If this “best” type does exist, it is used in the translation to an explicit array initialization, provided that all the element expressions can be converted to this best type. This models the following behaviour in C#3.0.

```
var ex1 = new []{"hello", new object()};
// Creates an object array
var ex2 = new []{new Button(), null};
// Creates a Button array
var ex3 = new []{new Button(), "hello"};
// Fails - no best type
var ex4 = new []{null,null};
// Fails - no best type
```

It is important to notice that there are no rules for the `null` constant, anonymous method expressions, or λ -expressions. These are expressions that *can not* have types synthesized for them. This reflects some important restrictions in C#3.0. For example, none of the following declarations are type correct (as no type can be synthesized):

```
var a1 = delegate(int x){ return x+1; };
var b1 = (int y) => y+1;
var c1 = (z) => new object();
var d1 = null;
```

whereas the following are all type correct:

```
Func<int,int> a2 = delegate(int x){ return x+1; };
var b2 = (Func<int,int>)(int x) => x+1;
var c2 = (Func<int,object>)((z) => new object());
var d2 = (Button)null;
```

5.4 Type inference

C#2.0 supports generic methods that can appear in classes which may themselves be generic or non-generic. Consider the following code in C#2.0 [14, §20.6.4].

```
public class Chooser
{
    static Random rand = new Random();

    public static T Choose<T>(T first, T second)
    {
        return (rand.Next(2) == 0) ? first : second;
    }
}
```

In C# a generic method invocation can explicitly specify a type argument list, or it can omit the type argument list and rely on a process known as “type inference” to determine the type arguments automatically. Hence, given the code

above it is possible to make the following invocations of the `Choose` method:

```
int i = Chooser.Choose(5, 213);
// Calls Choose<int>
string s = Chooser.Choose("foo", "bar");
// Calls Choose<string>
```

Whilst extremely useful, there are a number of restrictions in the support of type inference in C#2.0. The first restriction is known as “completeness”, and it ensures that a substitution must be found for every type parameter. Consider the following method declaration and code fragment:

```
static void myfoo1<T>(int arg1){}

myfoo1(42); // FAILS
myfoo1<object>(42); // Ok
```

The first call fails as nothing can be inferred for the type parameter `T`; and thus inference is incomplete. The second succeeds as a type argument is provided for `T`.

The second restriction in C#2.0 is known as “consistency”, and it ensures that if multiple substitutions have been inferred for a type parameter, these substitutions must be *identical*. Consider the following method declaration and code fragment:

```
static void myfoo2<T>(T arg1, T arg2){}

myfoo2("hello",new object());
// FAILS in C# 2.0
myfoo2<object>("hello",new object());
// Ok
```

The first method invocation fails as the type inference process will generate two substitutions: $\{T \mapsto \text{string}\}$ and $\{T \mapsto \text{object}\}$. As these substitutions are *not* identical, then type inference fails. The second invocation succeeds, which demonstrates that an inference could have been made.

The third restriction in C#2.0 is that arguments that are anonymous method expressions do not participate in the type inference process. Consider the following method declaration and code fragment:

```
static void myfoo3<T,U>(Func<T,U> arg1, U arg2){}

myfoo3(delegate(int x){return x;}, 42); // FAILS
```

This code fails because nothing is inferred from the first argument as it is an AME, and we only infer from the second argument the substitution $\{U \mapsto \text{int}\}$. Hence, we have not inferred anything for the type parameter `T`, and type inference is incomplete.

Space prevents us from a deeper analysis of type inference (some further discussions can be found elsewhere [6]) but we shall make two important observations that have an impact on the work reported here. First, we observe that not

allowing AMEs to participate in type inference would have catastrophic consequences for the LINQ style of programming. In §4 we detailed how query expressions are compiled into standard query operator calls *before typechecking*. Most of these make heavy use of λ -expressions. As λ -expressions are just syntactic sugar for AMEs, this would mean that most queries would compile to method invocations that would fail type inference! In conclusion, to support the LINQ style of programming it is necessary to extend the rather limited form of type inference in C#2.0 for C#3.0. For example, C#3.0 infers the type argument list $\langle \text{int}, \text{int} \rangle$ for the invocation `myfoo3((int x) => x, 42)`.

In addition, the *consistency* restriction in the C#2.0 type inference process has been relaxed. If a set of possible types has been inferred for a type parameter, then C#3.0 simply requires that there is a unique best type from this set to which all the other members of the set can be converted to. This best type (if it exists) is then the inferred type. Hence the following expression typechecks in C#3.0, inferring the type argument `object` for the expression `myfoo2("hello",new object())`.

It is important to note that the enhancement to the type inference process that will appear in C#3.0 represents a *breaking change* to the language. Consider the following overloaded method declarations and code fragment:

```
static void breaking<T>(Func<int,T> arg1, int arg2){}
static void breaking<T>(Func<int,int> arg1, T arg2){}

breaking(delegate(int x){ return x; }, 42);
```

In C#2.0, this invocation is resolved to the second method, as the first fails type inference. However, in C#3.0 this invocation is ambiguous as the first method now passes type inference (inferring the substitution $\{T \mapsto \text{int}\}$) and neither method is considered better than the other in the overloading resolution phase.

A second observation is that there is an awkward interaction between type inference and overloading resolution; consider the following overloaded method declarations and code fragment:

```
static void mybar<T>(int a, T b){}
static void mybar<T>(int a, int b){}

mybar(42,42); // Picks method 1
mybar<int>(42,42); // Picks method 2
```

In the first invocation, type inference succeeds for the first overload of `mybar`, but is incomplete for the second. Hence, the compiler infers the type argument `int` and selects the first overload. If, as in the second invocation, the programmer supplies the inferred type argument explicitly the compiler will actually select the second overload of `mybar`! This is because both methods are applicable, and the tie-breaking

rules for overloading determine that the second method has a better type signature.

This impacts on our translation of FC_3^\sharp as we need to deal with the changes to type inference and yet wish the translation to preserve the semantics! There are a number of possible solutions but the most compact is to extend FC_2^\sharp with explicit method descriptors. Whilst this means that FC_2^\sharp is not a strict subset of $C^\sharp 2.0$ (much like FGJ is not a strict subset of GJ [16]), it is actually quite straightforward to translate a FC_2^\sharp program into an equivalent valid $C^\sharp 2.0$ program.

6. Properties

In this section we briefly mention some properties of our translations of FC_3^\sharp to FC_2^\sharp . We do not give any details of the proofs; they are all quite routine and will appear in a forthcoming technical report.

Our main technical result is that the translation of FC_3^\sharp into FC_2^\sharp is type-preserving. In other words, if there is a translation then the resulting FC_2^\sharp fragment is well-typed.

THEOREM 1 (Preservation of typing).

1. *If $\Gamma \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_{11}$ then $\Gamma \vdash e_{11} \downarrow_i \tau_1$*
2. *If $\Gamma \vdash e_1 \downarrow_x \tau_1 \rightsquigarrow e_{11}$ then $\Gamma \vdash e_{11} \downarrow_x \tau_1$*
3. *If $\Gamma \vdash s_1 \downarrow \phi \rightsquigarrow s_{11}$ then $\Gamma \vdash s_{11} \downarrow \phi$*
4. *If $\Gamma \vdash e_1 \uparrow^s \phi_1 \rightsquigarrow e_{11}$ then $\Gamma \vdash e_{11} \uparrow^s \phi_1$*

Proof. By simultaneous induction over the translation relations.

Moreover, if we extend FC_3^\sharp with block expressions and method descriptor invocations, then we are able to show that our translations are essentially the identity function on well-typed FC_2^\sharp programs.

THEOREM 2 (Conservativity).

1. *If $\Gamma \vdash e_1 \downarrow_i \tau_1$ then $\Gamma \vdash e_1 \downarrow_i \tau_1 \rightsquigarrow e_1$*
2. *If $\Gamma \vdash e_1 \downarrow_x \tau_1$ then $\Gamma \vdash e_1 \downarrow_x \tau_1 \rightsquigarrow e_1$*
3. *If $\Gamma \vdash s_1 \downarrow \phi$ then $\Gamma \vdash s_1 \downarrow \phi \rightsquigarrow s_1$*
4. *If $\Gamma \vdash e_1 \uparrow^s \phi_1$ then $\Gamma \vdash e_1 \uparrow^s \phi_1 \rightsquigarrow e_1$*

Proof. By simultaneous induction over the typing relations.

7. Related work

There has been several decades of research on integrating databases and programming languages. Atkinson and Buneman [3] gave an influential survey in the late 1980s and Paton et al. [24] gave an updated survey roughly a decade later. Given the space constraints we cannot provide a thorough survey of all this work, but hopefully provide enough to give some historical context for LINQ/C^{sharp}3.0.

Cook and Ibrahim [10] categorize solutions to the impedance mismatch problem according to their interface style: either

orthogonal persistence or *explicit query execution*. LINQ is firmly in the latter category.

Orthogonal persistence allows run time objects to persist beyond a single program execution. Typically objects persist as long as they can be referenced by some persistent root object. The beauty of such systems is that persistent data is programmed transparently—it is treated like normal transient data. This makes code succinct but places a huge burden on the language runtime. Examples of persistent languages include Napier88 [23], OPJ [21], PJama [4], and Thor [19].

Languages providing explicit query execution allow queries to be written in some specialized query language; the main advantage being that programmers can interact directly with the database engine. One simple way to provide a query language is to use a preprocessor. SQLJ [26], for example, allows SQL commands to be embedded directly in Java code using the prefix `#sql`. SQLJ compilation consists of two stages; first to pre-process the embedded SQL, and second the ‘pure Java’ compilation.

An alternative, and dominant, approach is to provide a standardized API through which a programming language can access a database engine. For .NET this might be ADO.NET and for Java this might be JDBC. Unfortunately, these APIs typically offer only weak compile-time guarantees as most database commands are passed as strings.

A number of proposals have been made for better language-database query integration (see, e.g. [1, 2, 22, 18, 15, 5]). In spite of the obvious advantages of these languages, it appears that their acceptance has been hampered by the fact that they are “different” from more mainstream application languages, such as Java and C^{sharp}. For example, HaskellDB [18] proposes extensions to the lazy functional language, Haskell; and TL [22] is a hybrid functional/imperative language with advanced type and module systems. LINQ builds on much of this previous research experience and transfers it to a modern, commercial virtual machine, programming language and IDE (Visual Studio) setting, as well as providing a sophisticated built-in object-relational mapping tool.

8. Conclusions

The ROX impedance mismatch problem is a recurring headache for today’s developers. Rather than use a collection of ad-hoc tools and pre-processors, the next version of Visual Studio (code-named ‘Orcas’) will contain extensions to the .NET framework to support the integration of non-object data. The LINQ framework provides a very general pattern of operators to query data and provides four implementations of this pattern: one for in-memory objects, one for in-memory XML data, one for external relational data, and one for external data that adheres to the ADO.NET Entity Data Model.

In addition, both VB.NET and C[#] will be extended to provide a convenient form of LINQ programming. In this paper we have studied formally the new features to appear in C[#]. We have shown how the new query expressions of C[#]3.0 can be compiled into method calls and how the rest of the new C[#]3.0 language features can be translated into C[#]2.0. Our translation uses techniques from bidirectional type checking, and has been proven sound.

Acknowledgements We are grateful to both the C[#] and VB teams for their hard work and support. We thank Andrew Kennedy, Eric Lippert and Claudio Russo for various discussions about our formalization, and also to the anonymous reviewers who suggested a number of improvements.

References

- [1] A. Albano, G. Ghelli, and R. Orsini. Types for databases: the Galileo experience. In *Proceedings of DBPL*, 1989.
- [2] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *Journal of Very Large Data Bases*, 4(3):403–444, 1995.
- [3] M.P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–170, 1987.
- [4] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.
- [5] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of ICFP*, 2003.
- [6] G.M. Bierman. Formalizing and extending C[#] type inference. In *Proceedings of FOOL*, 2007.
- [7] G.M. Bierman, E. Meijer, and W. Schulte. The essence of data access in C^ω. In *Proceedings of ECOOP*, 2005.
- [8] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge, 2003.
- [9] J.A. Blakeley, D. Campbell, S. Muralidhar, and A. Nori. The ADO.NET entity framework: Making the conceptual level real. *SIGMOD Record*, 35(4):32–39, 2006.
- [10] W.R. Cook and A.H. Ibrahim. Programming languages and databases: What’s the problem? Unpublished paper, 2005.
- [11] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of ACM SIGMOD*, 1984.
- [12] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, 1998.
- [13] M. Harren, M. Raghavachari, O. Shmueli, M.G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: Facilitating XML processing in Java. In *Proceedings of WWW*, 2005.
- [14] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C[#] Programming Language*. Addison-Wesley, second edition, 2006.
- [15] H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language. In *Proceedings of WebDB*, 2000.
- [16] A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
- [17] A. Kennedy and D. Syme. Transposing F to C[#]. *Concurrency and Computation*, 16(7), 2004.
- [18] D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of Conference on Domain-Specific Languages*, 1999.
- [19] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A.C. Myers, M. Day, and L. Shriram. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of SIGMOD*, 1996.
- [20] D. Maier. Representing database programs as objects. In *Proceedings of DBPL*, 1987.
- [21] A. Marquez, S. Blackburn, G. Mercer, and J.N. Zigman. Implementing orthogonally persistent Java. In *Proceedings of POS*, 2000.
- [22] F. Matthes, S. Müßig, and J.W. Schmidt. Persistent polymorphic programming in Tycoon: An introduction. Technical report, University of Glasgow, 1994.
- [23] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. The Napier88 reference manual. Technical report, University of Glasgow, 1989.
- [24] N. Paton, R. Cooper, H. Williams, and P. Trinder. *Database programming languages*. Prentice Hall, 1996.
- [25] B.C. Pierce and D.N. Turner. Local type inference. In *Proceedings of POPL*, 1998.
- [26] J. Price. *Java programming with Oracle SQLJ*. O’Reilly, 2001.
- [27] D. Syme. Leveraging .NET meta-programming components from F#: Integrated queries and interoperable heterogeneous execution. In *Proceedings of ML Workshop*, 2006.
- [28] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *Proceedings of POPL*, 2004.

A. The query expression pattern

In §4 we showed how C[#]3.0 query expressions are compiled into a sequence of method invocations. Moreover, this compilation occurs *before* typechecking. As we pointed out, this means that *any* type that supports a pattern of methods can support query expressions. In fact, there is considerable flexibility in how a type can support this query expression pattern: methods can be implemented as instance methods or extensions methods, and the methods can request delegates or expressions as λ -expressions are convertible to both.

The recommended query expression pattern for an arbitrary generic type C<T> is quite large. The subset that is required for the translation of C[#]3.0 query expressions is given in Figure 6. (Note, it is possible to implement the pattern for non-

generic types as well.) The remaining methods of the pattern (which are not shown here) can only be called directly as methods from C^\sharp , but other languages may choose to target these with query expression syntax.

The *standard query operators* are an implementation of the query expression pattern for any type that implements the `IEnumerable<T>` interface. This implementation will be included in the base class library.

B. Featherweight $C^\sharp 2.0$: further details

In §5.1 we introduced our target language, FC_2^\sharp , which is intended to be a core featherweight fragment of $C^\sharp 2.0$. We recall here that FC_2^\sharp , unlike FC_3^\sharp , is not a precise subset of $C^\sharp 2.0$, in that the method invocations use method descriptors, and we also introduced a new syntactic form of block expressions. The latter is just for convenience, and could easily be removed (albeit at the cost of a more ugly translation). The former was used to circumvent a problem arising from an interaction between overloading resolution and type inference (this was explained in more detail in §5.4). Again, this could be compiled away to leave standard method invocations but at the expense of clarity.

In §5.1 we gave the type checking and type synthesis rules for expressions. In Figure 7 we give the rest of the rules, namely type synthesis of statement expressions, and type checking of statements.

Given the translations of §5, these rules are pretty straightforward. The two new rules are worth noting. The rule [S₂-MethInv], simply uses the types contained in the method descriptor to synthesize the type of the invocation. The rule [S₂-BlockExp] can be read as follows: the block expression $(\{\bar{s}\})^\tau$ synthesizes the type τ provided that the statements \bar{s} type check at type τ (i.e. any expressions that are contained in `return` statements can be implicitly converted to type τ).

```

class C
{
    public C<T> Cast<T>();
}
class C<T>
{
    public C<T> Where(Func<T,bool> predicate);
    public C<U> Select<U>(Func<T,U> selector);
    public C<U> SelectMany<U,V>(Func<T,C<U>> selector, Func<T,C<U>,V> resultSelector);
    public C<V> Join<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,U,V> resultSelector);
    public C<V> GroupJoin<U,K,V>(C<U> inner, Func<T,K> outerKeySelector,
        Func<U,K> innerKeySelector, Func<T,C<U>,V> resultSelector);
    public O<T> OrderBy<K>(Func<T,K> keySelector);
    public O<T> OrderByDescending<K>(Func<T,K> keySelector);
    public C<G<K,T>> GroupBy<K>(Func<T,K> keySelector);
    public C<G<K,E>> GroupBy<K,E>(Func<T,K> keySelector,
        Func<T,E> elementSelector);
}
class O<T> : C<T>
{
    public O<T> ThenBy<K>(Func<T,K> keySelector);
    public O<T> ThenByDescending<K>(Func<T,K> keySelector);
}
class G<K,T> : C<T>
{
    public K Key { get; }
}

```

Figure 6. The LINQ query expression pattern

$$\boxed{\Gamma \vdash se_1 \uparrow^s \phi_1}$$

$$\frac{\Gamma \vdash e_1 \uparrow^s D \langle \bar{\tau} \rangle \quad d\text{type}(D)(\bar{\tau}) = \bar{\tau}_0 \rightarrow \phi \quad \Gamma \vdash \bar{e}_2 \downarrow_i \bar{\tau}_0}{\Gamma \vdash e_1(\bar{e}_2) \uparrow^s \phi} [\text{S}_2\text{-DellInv}] \quad \frac{M = C \langle \bar{X}_C \rangle \langle \bar{\tau}_C \rangle :: m \langle \bar{X}_m \rangle \langle \bar{\tau}_1 \rangle : (\bar{\tau}_p) \rightarrow \phi \quad \Gamma \vdash e_1 \downarrow_i C \langle \bar{\tau}_C \rangle \quad \Gamma \vdash \bar{e}_2 \downarrow_i \bar{\tau}_p [\bar{X}_C := \bar{\tau}_C, \bar{X}_m := \bar{\tau}_1]}{\Gamma \vdash e_1.M(\bar{e}_2) \uparrow^s \phi [\bar{X}_C := \bar{\tau}_C, \bar{X}_m := \bar{\tau}_1]} [\text{S}_2\text{-MethInv}]$$

$$\text{MG} \stackrel{\text{def}}{=} m\text{type}(C \langle \bar{\tau} \rangle, \text{.ctor})$$

$$\text{AMG} \stackrel{\text{def}}{=} \{C \langle \bar{X}_C \rangle \langle \bar{\tau}_C \rangle :: \text{.ctor} : (\bar{\tau}_p) \mid C \langle \bar{X}_C \rangle \langle \bar{\tau}_C \rangle :: \text{.ctor} : (\bar{\tau}_p) \in \text{MG}, |\bar{\tau}_p| = |\bar{e}_1|\}$$

$$\frac{C \langle \bar{X}_C \rangle \langle \bar{\tau}_C \rangle :: \text{.ctor} : (\bar{\tau}_p) \stackrel{\text{def}}{=} \text{bestMD}(\text{AMG}) \quad \Gamma \vdash \bar{e}_1 \downarrow_i \bar{\tau}_p [\bar{X}_C := \bar{\tau}_C]}{\Gamma \vdash \text{new } C \langle \bar{\tau} \rangle (\bar{e}_1) \uparrow^s C \langle \bar{\tau} \rangle} [\text{S}_2\text{-ObjCreation}]$$

$$\frac{\Gamma \vdash \bar{e}_1 \downarrow_i \tau}{\Gamma \vdash \text{new } \tau [] \{\bar{e}_1\} \uparrow^s \tau []} [\text{S}_2\text{-ExpArray}] \quad \frac{\Gamma, x_1 : \tau_1 \vdash e_1 \downarrow_i \tau_1}{\Gamma, x_1 : \tau_1 \vdash x_1 = e_1 \uparrow^s \tau_1} [\text{S}_2\text{-VarAssign}] \quad \frac{\Gamma \vdash \bar{s} \downarrow \tau}{\Gamma \vdash (\{\bar{s}\})^\tau \uparrow^s \tau} [\text{S}_2\text{-BlockExp}]$$

$$\boxed{\Gamma \vdash s_1 \downarrow \phi_1}$$

$$\frac{}{\Gamma \vdash ; \downarrow \phi} [\text{C-Skip}] \quad \frac{\Gamma \vdash se_1 \uparrow^s \phi_1}{\Gamma \vdash se_1 ; \downarrow \phi} [\text{C}_2\text{-ExpStatement}] \quad \frac{\Gamma \vdash e_1 \downarrow_i \text{bool} \quad \Gamma \vdash s_1 \downarrow \phi \quad \Gamma \vdash s_2 \downarrow \phi}{\Gamma \vdash \text{if } (e_1) s_1 \text{ else } s_2 \downarrow \phi} [\text{C}_2\text{-Cond}]$$

$$\frac{\Gamma \vdash e_1 \uparrow^s \tau_1 \quad f\text{type}(\tau_1, f) = \tau_2 \quad \Gamma \vdash e_2 \downarrow_i \tau_2}{\Gamma \vdash e_1.f = e_2 ; \downarrow \phi} [\text{C}_2\text{-FAss}] \quad \frac{}{\Gamma \vdash \text{return} ; \downarrow \text{void}} [\text{C}_2\text{-Return}] \quad \frac{\Gamma \vdash e_1 \downarrow_i \tau_0}{\Gamma \vdash \text{return } e_1 ; \downarrow \tau_0} [\text{C}_2\text{-ReturnExp}]$$

$$\frac{\Gamma \vdash e_1 \uparrow^s \tau_2 [] \quad x \notin \text{dom}(\Gamma) \quad \tau_1 <: \tau_2 \text{ or } \tau_2 <: \tau_1 \quad \Gamma, x : \tau_2 \vdash s_1 \downarrow \phi}{\Gamma \vdash \text{foreach } (\tau_1 x_1 \text{ in } e_1) s_1 \downarrow \phi} [\text{C}_2\text{-EForEach1}] \quad \frac{\Gamma \vdash e_1 \uparrow^s \tau_2 \quad \tau_2 <: \text{IEnumerable} \langle \tau_3 \rangle \quad x \notin \text{dom}(\Gamma) \quad \tau_1 <: \tau_3 \text{ or } \tau_3 <: \tau_1 \quad \Gamma, x : \tau_3 \vdash s_1 \downarrow \phi}{\Gamma \vdash \text{foreach } (\tau_1 x_1 \text{ in } e_1) s_1 \downarrow \phi} [\text{C}_2\text{-EForEach2}]$$

$$\boxed{\Gamma \vdash \bar{s}_1 \downarrow \phi_1}$$

$$\frac{\Gamma \vdash e_1 \downarrow_i \tau_1 \quad x \notin \text{dom}(\Gamma) \quad \Gamma, x : \tau_1 \vdash \bar{s}_1 \downarrow \phi}{\Gamma \vdash \tau_1 x = e_1 ; \bar{s}_1 \downarrow \phi} [\text{C}_2\text{-EDecSeq}]$$

Figure 7. Remaining type synthesis and type checking rules for FC_2^\sharp