

Adding dynamic types to C[#]

Gavin Bierman¹, Erik Meijer², and Mads Torgersen²

¹ Microsoft Research

² Microsoft Corporation

{gmb,emeijer,madst}@microsoft.com

Abstract. Developers using statically typed languages such as C[#] and Java increasingly have to interoperate with APIs and object models defined in dynamic languages. This impedance mismatch results in code that is difficult to understand, awkward to analyze, and expensive to maintain. In this paper we describe new features in C[#]4.0 that support the safe combination of dynamically and statically typed code by deferring type checking of program fragments with static type `dynamic` until runtime. When executed, these dynamic code fragments are type-checked and resolved using the same rules as statically typed code. We formalize these features in a core fragment of C[#] and prove important safety properties.

1 Introduction

Real-world software applications are architected in several tiers. The increased use of JavaScript and other dynamic languages in web-based applications mean that the mid-tier software, typically written in a statically-typed language such as C[#] and Java, has to interoperate with dynamically-typed top-tier code and objects. There is clearly an impedance mismatch between these two data models, which leads to particularly awkward coding in the mid-tier code.

The Dynamic Language Runtime (DLR) is an API which runs on top of the Common Language Runtime (CLR). Its purpose is to enable efficient implementations of dynamic programming languages—for example, IronRuby and IronPython—on the CLR, but also to facilitate great interoperability between dynamic languages and statically-typed CLR languages such as C[#] [12]. Core to the DLR is the notion of *dynamic objects*; i.e., objects that can do their own name binding at runtime instead of having it done for them by a compiler. These are the currency of dynamic interoperation between languages.

In this paper we focus on new features in C[#]4.0 that improve interoperation with APIs and objects that are defined in dynamic languages and target the DLR. Whilst we focus on these new features in the context of C[#], none of them are especially tied to the language: the design principles, which we capture in our formalization, could easily be applied to any class-based object-oriented language.

These extensions to C[#] consist of a new type `dynamic` and changes to the type system to allow the safe coexistence of statically and dynamically typed code and data.³ The combination of static and dynamic type systems has been the focus of considerable

³ This combination of static and dynamic typing is sometimes referred to as *gradual typing*. We do not use this terminology to avoid confusion with Siek and Taha's particular approach [21].

previous work [1, 22, 3, 21, 16, 17] but as far as we are aware the particular approach described in this paper is novel.

We believe that a formal, mathematical approach is essential to set a precise foundation for researchers, implementors and users of programming languages. For C[#]4.0 this is especially true: the new language features require subtle changes to the type system, and build upon assumed behaviour supported by the runtime (in the actual implementation this behaviour is supported by the DLR). This can all be captured succinctly with fairly standard formalization techniques [18]. Moreover, they allow precise comparisons with previous approaches. We have found our formal approach to be useful not only in the design process but also in the production of natural language documentation.

This paper makes a number of contributions:

- We define an imperative, core fragment of C[#]4.0 called FC₄[#]. This fragment whilst reasonably small, contains all the essential features of C[#] (other fragments are too weak, e.g. [15]).
- We define C_{CLR}[#], which is the result of type-checking FC₄[#] programs and captures the same semantic information as the MSIL bytecode language that the actual C[#] compiler targets. As C_{CLR}[#] is more amenable to formal manipulation than MSIL we consider it to be of independent interest.
- We formally specify a type-directed translation of FC₄[#] to C_{CLR}[#]. This translation builds on the techniques of bidirectional type checking first used for local type inference in System F [19]. Of particular importance is the treatment of the `dynamic` type, which we believe to be a considerable improvement on other approaches [21] since our system maintains transitivity of the subtyping relationship.
- We are able to prove preservation of the translation.
- We give an operational semantics for C_{CLR}[#] for which we can establish type soundness.

The rest of the paper is organized as follows. In §2 we give an informal introduction to the support for dynamic types in C[#]4.0. §3 informally characterizes the C[#] type system. §4 formally describes FC₄[#], our core fragment of C[#]4.0, and §5 describes the target language C_{CLR}[#]. §6 shows the translation from FC₄[#] to C_{CLR}[#], which represents the compile-time binding of operations, and §7 gives the operational semantics of C_{CLR}[#], including the dynamic binding of operations. We review some related work in §8, before concluding in §9.

2 An introduction to dynamic types in C[#]4.0

In this section we give an informal introduction to the support of dynamic types in C[#]4.0, including a number of examples to illustrate the key ideas. We assume that the reader is familiar with C[#]/Java-like languages.

2.1 Example: JavaScript access in Silverlight

This example is a snippet of C[#]3.0 code running in Silverlight and calling into JavaScript objects running in a browser. Because JavaScript objects live outside the CLR type system, in C[#]3.0 methods and properties can only be accessed indirectly, through an interpretative string-based interface:

```

HtmlDocument doc = HtmlPage.Document;
HtmlWindow win = HtmlPage.Window;
string latitude, longitude, name, address;
...

ScriptObject map = win.CreateInstance("VEMap", "myMap");
map.Invoke("LoadMap");
map.Invoke("DeleteAllShapes");

var x = win.CreateInstance("VELatLong", latitude, longitude);
var pin = map.Invoke("AddPushpin", x);

doc.SetProperty("Title", "Information for: " + name);
((ScriptObject)pin).Invoke("SetTitle", name);
((ScriptObject)pin).Invoke("SetDescription", address);

map.Invoke("SetCenterAndZoom", x, 9);

```

Clearly this style of string-based interoperation is weak, fragile, difficult for tools to support and expensive to maintain. In C[#]4.0 we have a new type `dynamic`. The novelty is that the type system has been extended to allow access to any member of a dynamic object (just like in a dynamic language). The compiler inserts calls to the DLR to perform the familiar C[#] resolution rules at *runtime*. (Of course this means that we may get lookup errors as exceptions during the execution of the program, but the string-based interfaces had this property already.) Thus in C[#]4.0, we can rewrite the previous code and declare the `doc`, `win` and `map` variables to be of type `dynamic`.

```

dynamic doc = HtmlPage.Document;
dynamic win = HtmlPage.Window;
string latitude, longitude, name, address;
...

dynamic map = win.CreateInstance("VEMap", "myMap");
map.LoadMap();
map.DeleteAllShapes();

var x = win.CreateInstance("VELatLong", latitude, longitude);
var pin = map.AddPushpin(x);

doc.Title = "Information for: " + name;
pin.SetTitle(name);
pin.SetDescription(address);
map.SetCenterAndZoom(x, 9);

```

Notice how all the invocations of `Invoke` and `SetProperty` disappear in favour of ordinary method calls and member access. This works at runtime because objects such as `map` are dynamic objects that know how to correctly lookup members such as the `AddPushpin` method on the underlying JavaScript object. Of course the value returned from `map.AddPushpin()` again has the static type `dynamic`, allowing further dynamic invocations.

2.2 Example: COM interop

Many of the APIs on the Windows platform, such as Office and Windows 7 functionality such as the taskbar, location and sensors, are exposed as native COM components. Typically, COM components make heavy use of late binding since they are primarily used via dynamically typed scripting languages such as Visual Basic for Applications (VBA) or JavaScript. As a result, accessing COM components from previous versions of C# was notoriously painful.

Besides the introduction of dynamic types that are the topic of this paper, C# 4.0 also adds support for optional and named parameters, which is used in many dynamic languages in the absence of type-based overloading, and indexed properties. As a result, accessing COM components from C# 4.0 is as concise and convenient as accessing them from VBA:

```
var word = new Word.Application();
word.Visible = true;
word.Documents.Add();
...
word.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

Dynamic operations on COM objects are dispatched by a special DLR COM runtime binder that is shared among multiple languages including IronPython and IronRuby.

2.3 Example: Expando object

As the new dynamic features of C# 4.0 are built on top of the DLR we can use some of the features intended for dynamic languages directly in C#. For example, the DLR supports a class `ExpandoObject` which allows for the creation of instances that can have property members added and removed at runtime. This style of programming is popular in dynamic languages, as it is highly flexible and requires little declaration up front, allowing for rapid prototyping.

```
dynamic contact = new ExpandoObject();
contact.Name = "Erik";
contact.Phone = "425-555-0000";
contact.Address = new ExpandoObject();
contact.Address.Street = "101 Lakeside";
contact.Address.City = "Mercer Island";
contact.Address.State = "WA";
contact.Address.Zip = "68402";
```

Just by assigning to these properties they are brought into existence on the `ExpandoObject`. They can be examined like normal properties until they are explicitly removed, at which point the `ExpandoObject` will throw an error saying that it does not have such a member.

```
Console.WriteLine(contact.Address.Zip); // Prints the Zip
((IDictionary<string,object>)contact.Address).Remove("Zip");
Console.WriteLine(contact.Address.Zip); // Throws an error
```

2.4 Dynamic binding of ordinary objects

Dynamic objects are useful in that they implement their own lookup, but what happens when ordinary .NET objects are accessed dynamically? This is interesting because many dynamic calls will eventually return normal objects, but these will still have the type `dynamic`.

C#4.0 takes the approach that binding of any operation can take place dynamically, and that it will if any of its arguments or operands, not just the receiver, has the type `dynamic`. So in the following small example:

```
dynamic d;  
string last = d[d.Length-1];
```

There are no less than **four** dynamic operations: (1) to access the `Length` property; (2) to perform the minus operator; (3) to perform the indexing; and (4) to perform the implicit conversion to `string`. Each of these is looked up at runtime based on the actual runtime type of the `dynamic` value.

On the other hand, dynamic binding is not an either/or situation: Even though it happens at runtime, the binding is not necessarily based entirely on runtime information. In fact dynamic binding will only examine the runtime type of those contributing expressions that had type `dynamic` at compile time—for the remaining expressions only their static information will be used. Consider the following example:

```
public static void M(byte b, int i)  
{  
    Console.WriteLine("byte, int");  
}  
public static void M(short s, int i)  
{  
    Console.WriteLine("short, int");  
}  
static void Main(string[] args)  
{  
    short s = 42;  
    dynamic d = 7;  
    int i = 42;  
  
    M(s, 7);           //(1) short, int  
    M(42, 7);         //(2) byte, int  
    M(s, d);          //(3) short, int  
    M(42, d);         //(4) byte, int  
  
    M(i, 7);          //(5) FAIL at compile-time - no (int, int) overload  
    M(i, d);          //(6) FAIL at compile-time - no overload permits i  
    M(d, i);          //(7) FAIL at runtime - no (int,int) overload  
}
```

The first two cases show compile-time binding rules of C#'s overload resolution. The first prefers a precise match. The second, not finding a precise match, prefers the smallest type, `byte`, for the first argument. The third and fourth cases are similar but replace the literal `int` with the `dynamic` `d`, which has the *runtime* type `int`. The presence of this

`dynamic` expression causes the *entire* method invocation to be resolved dynamically, using the *runtime* type of `d`, but for the first argument the *compile-time* type is still used. In particular in the fourth case, not only is the compile-time type of `42` used (`int`), but also the fact that it is a literal, which allows it to be converted to any integral type that it will fit in, just like in the second case.

The remaining three cases illustrate error situations. Case 5 fails at compile-time because neither overload accepts an `int` as their first argument. Somewhat surprisingly, perhaps, case 6 also fails at compile-time for the same reason—the compiler knows enough to determine that the invocation could never succeed at runtime, regardless of the runtime type of `d`. The last case is allowed at compile-time, but fails to bind at runtime, because even though the runtime value of `d` is only 7, its runtime *type* is `int`.

2.5 Dynamic conversions

In C[#]4.0, expressions of type `dynamic` can be implicitly converted to any type. The conversion, like any other operation, will be bound at runtime, using the runtime type of the expression to determine if a conversion exists. So, for instance, the following:

```
dynamic d = "Hello World";  
int i = d;
```

is allowed at compile-time and only fails at runtime when a suitable conversion from `string` to `int` is not found. Note that although *expressions* of type `dynamic` can be implicitly converted to any type, it is *not* the case that `dynamic` is a subtype of `int`. As others have noticed [21], including this subtype rule collapses the subtyping relation. Our finer analysis allows us to maintain a transitive subtyping relation without the risk of cycles, while still allowing a smooth path back from `dynamic` to static through dynamically-bound implicit conversions.

Summary

In summary, C[#]4.0 offers a new type `dynamic`. At runtime this type is replaced by `object`, but it is treated specially by the compiler. All types that can be implicitly converted to `object` can be implicitly converted to `dynamic`. An expression that synthesizes type `dynamic` can be implicitly converted to any type (with a suitable runtime type test inserted). A method call that involves a subexpression of type `dynamic` (either the receiver or any argument) is treated as type-correct by the compiler, and the method call resolution is deferred until runtime. When resolving a method call at runtime we use the runtime type of any subexpression that was originally of type `dynamic` and the compile-time types of the remaining subexpressions. In all other respects the runtime resolution is *identical* to the compile-time resolution.⁴

3 An overview of the C[#] type system

Before we formalize the support of dynamic types we shall give a brief overview of the C[#] type system. At its heart, the C[#] type system is a bidirectional type system [19] which uses a variant of coercive subtyping [8]. We expand on these two points below.

⁴ In the C[#] system, the runtime and compile-time resolution is actually performed by the same code!

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 [4] although we do not consider it here.) Type checking is the process of determining whether a given term can be assigned a particular given type (the C[#] language specification [12] refers to this as type *conversion*). Type synthesis, on the other hand, is the process of automatically determining a type from a given term. Type synthesis is used when we do not know anything about the expected type of an expression; for example, the receiver subexpression in a method invocation. Type conversion 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 inter-defined. 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.

It is possible to see these two phases directly in C[#]. Consider the following two declarations.

```
T x = e;           // Type conversion
var y = e;        // Type synthesis
```

The first declaration uses the type checking phase to ensure that the expression e can be converted to the type T . In contrast, the second declaration, uses type synthesis to determine a type for e which is then used implicitly for the declaration of y .

These two phases are subtly different because there are C[#] expressions that do *not* synthesize types, and yet *can* be converted to a type. For example,

```
Button x = null;   // null can be converted to type Button
var y = null;     // Fails as null does not synthesize a type.
```

The bidirectional type system also makes heavy use of a notion of subtyping. In C[#] this is achieved using coercive subtyping, by which we mean that when determining whether a type T is a subtype of type S , we generate a coercion C that when applied to a value of type T yields a value of type S . This means that both phases of the bidirectional type system return translated terms, which contain explicit coercions generated in the process of checking subtypes.

Finally, in the process of typing a program the C[#] type system also resolves some other implicit information including, importantly, calls to overloaded methods (including constructor methods). Thus in the process of typing a C[#] program we both resolve calls to overloaded methods and insert explicit coercion code. In other words, the typing of a C[#] program can also be seen as a type-directed translation to a target language, where only type correct programs can be translated. The target language in the actual C[#] compiler is MSIL, the bytecode language for the CLR; this paper introduces a higher-level target language, C[#]_{CLR}.

4 Source language: Featherweight C[#]4.0

In the rest of the paper we study the essence of C[#]4.0. We adopt a formal, mathematical approach and define a core calculus, FC[#]₄. Whilst small enough to remain amenable to formal reasoning, FC[#]₄ is a relatively large subset of C[#], certainly in comparison to

other core calculi such as FJ [14] and ClassicJava [11]. This is not only because we wish that our core calculus supports all the essential object-oriented features (classes, generics, overloading, inheritance, side-effects) but also because we wish to formalize *all* the additional complications of adding dynamic types to C^\sharp . We have, however, retained what we consider to be the fundamental property of FJ, namely that FC_4^\sharp is a completely valid subset of $C^\sharp 4.0$, i.e. every valid FC_4^\sharp program is literally an executable $C^\sharp 4.0$ program.

An FC_4^\sharp program consists of a sequence of one or more class declarations. Given an FC_4^\sharp program we assume that there is a unique designated method within the standard class declarations that serves as the entry point (the `main` method). Programs are defined as follows.

$p ::= \overline{cd}$	Program
$cd ::=$	Class declaration
<code>public class C<X>:C<σ> {fd md cmd}</code>	
$fd ::=$ <code>public σ f;</code>	Field declaration
$md ::=$	Method declaration
<code>public virtual σ m<X>(σ x) {s}</code>	
<code>public override σ m<X>(σ x) {s}</code>	
$cmd ::=$	Constructor method declaration
<code>public C<X>(σ x):this(ē){s}</code>	
<code>public C<X>(σ x):base(ē){s}</code>	

A 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`. To simplify matters, we require all methods to return a value, i.e. we do not model `void`-returning methods. For conciseness, we do not model `static` methods, extension methods (these have been formalized elsewhere [5]) 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. Although constructor methods interact with dynamic types (which is why we include them in our calculus) much of their complications are orthogonal to the concerns of the paper, so we simplify matters and treat them essentially as normal methods with the distinguished name `.ctor`.

$C^\sharp 4.0$ adds to the type grammar of C^\sharp a new reference type, `dynamic`. Thus the grammar for FC_4^\sharp types is as follows.

$σ ::=$	Type
$γ$	Value type
$ρ$	Reference type
X	Type parameter
$γ ::=$	Value Type
<code>bool</code>	Boolean
<code>int</code>	Integer
<code>byte</code>	Byte

$\rho ::=$	Reference Type
$C\langle\bar{\sigma}\rangle$	Class type (including <code>object</code> and <code>dynamic</code>)
$D\langle\bar{\sigma}\rangle$	Delegate type

The two main categories of $FC_4^\#$ types are value types and reference types. We simplify the treatment of $C^\#$ value types and drop both enumeration types and nullable types (although they are simple to add), and include just the simple types; indeed we shall consider just three: `bool`, `int` and `byte` (there is an interesting feature of the $C^\#$ type system regarding the latter two, which is why we include them).

$FC_4^\#$ reference types include class types and delegate types. We write D to range over delegate types and C to range over class types. Following GJ [14] we use the shorthand C for $C\langle\rangle$. We include two distinguished class types: `object` and `dynamic`. For simplicity we do not model constraints on generic parameters, and we do not include array types.

$FC_4^\#$ expressions are split into two categories: ordinary expressions and statement expressions. Statement expressions are expressions that can be used as statements. The grammar for expressions is as follows.

$e ::=$	Expression
b	Boolean
i	Integer
$e \oplus e$	Built-in operator
x	Variable
<code>null</code>	Null
$(\sigma)e$	Cast
$e.f$	Field access
<code>delegate</code> $(\bar{\sigma} \bar{x}) \{ \bar{s} \}$	Anonymous method expression
se	Statement expression
$se ::=$	Statement expression
$e(\bar{e})$	Delegate invocation
$e.m\langle\bar{\sigma}\rangle(\bar{e})$	Method invocation
<code>new</code> $C\langle\bar{\sigma}\rangle(\bar{e})$	Object creation
$x = e$	Variable assignment

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. 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 [14] we adopt an overloaded ‘bar’ notation; for example, $\bar{\sigma} \bar{f}$ is a shorthand for a possibly empty sequence $\sigma_1 f_1, \dots, \sigma_n f_n$.

Anonymous method expressions (AMEs) were introduced in $C^\#2.0$, and provide a means to define a “nameless” method.⁵ They are unusual in that they are expressions that cannot *synthesize* a type but they can (and must) be converted to a compatible

⁵ $C^\#3.0$ introduced syntactic sugar (“lambda expressions”) for these [5] but here we keep the unsugared form for simplicity.

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.

As mentioned earlier, FC_4^\sharp 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. FC_4^\sharp statements are standard and the grammar is as follows.

$s ::=$	Statement
;	Skip
se ;	Expression statement
if (e) s else s	Conditional statement
$\sigma x = e$;	Variable declaration
$e.f = e$;	Field assignment
return e ;	Return statement
{ \bar{s} }	Block

In what follows we assume that FC_4^\sharp 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. However, we assume that a correct program induces a number of important useful functions that are used in the typing rules. First, we assume an auxiliary function f_{type} , which is a map from a type and a field name to a type. Thus $f_{type}(\sigma, f)$ returns the type of field f in type σ . Second we assume an auxiliary function d_{type} , which is a map from delegate names to their associated type. We write delegate types as function types in the System F sense; in general they are written $\forall \bar{X}. (\bar{\sigma}_1) \rightarrow \sigma_2$. 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 rules we use type application for conciseness; we write $d_{type}(\mathbb{D})(\bar{\sigma}) = \bar{\sigma}_2 \rightarrow \sigma_3$ to mean first use d_{type} to determine the type of the delegate \mathbb{D} , say $\forall \bar{X}. (\bar{\sigma}_0) \rightarrow \sigma_1$ and then substitute the types $\bar{\sigma}$ for \bar{X} resulting in the type $\bar{\sigma}_2 \rightarrow \sigma_3$.

Finally, we assume an auxiliary function m_{type} that is a map from a type and a method name to a *method group*. Thus $m_{type}(\sigma, m)$ returns a method group that represents all the candidate methods called m that are accessible from type σ , i.e. it is a set of *method signatures* of the form $C\langle \bar{X}_C \rangle \langle \bar{\sigma}_C \rangle : : m\langle \bar{X}_m \rangle : (\bar{\sigma}_p) \rightarrow \sigma_1$.

As we have mentioned earlier, the treatment of the new dynamic types in C^\sharp is achieved by type-directed translation. Thus in the next section we define the target language, C_{CLR}^\sharp , and then in §6 we define precisely the translation of FC_4^\sharp to C_{CLR}^\sharp .

5 Target language

In this section, we define the target language of the typing of FC_4^\sharp programs. In reality, the C^\sharp compiler targets MSIL, the bytecode language for the CLR. However, this bytecode language is rather awkward to deal with mathematically so, instead, we define a C^\sharp -like target language that we call C_{CLR}^\sharp . Whilst at first glance it may look like FC_4^\sharp , it

is quite different. Four important distinguishing features of C_{CLR}^\sharp are: (1) All non-trivial conversions are *explicit* (or, equivalently, the “subtype” relation for C_{CLR}^\sharp is just the subclassing relation); (2) all method invocations have been fully resolved, so method invocations no longer involve simply method names, but complete method descriptors; (3) there are explicit operations to provide the new dynamic behaviour (in reality, these are simply calls to the appropriate DLR method, but for simplicity we shall treat them as if they are C_{CLR}^\sharp language constructs); and finally (4) there is *no dynamic* type in C_{CLR}^\sharp ; it is translated to **object**. To make this clear we write τ to range over target types, and θ to range over reference types that exclude **dynamic**.

C_{CLR}^\sharp expressions are given by the following grammar.

$E ::=$	Target expressions
b	Boolean
i	Integer
$E \oplus E$	Built-in operator
x	Variable
null	Null
$E.f$	Field access
delegate $(\bar{\tau} \bar{x}) \{ \bar{S} \}$	Anonymous method expression
CE	Conversion Expression
DE	Dynamic Expression
SE	Statement expression
$CE ::=$	Conversion Expression
ByteToInt (E)	Byte to Integer conversion
IntToByte (E)	Integer to Byte conversion
Box $[\gamma](E)$	Boxing conversion
Unbox $[\gamma](E)$	Unboxing conversion
Downcast $[\rho](E)$	Downcast
$DE ::=$	Dynamic Expression
Convert $[\sigma](E)$	Dynamic type test
MemberAccess $[f](E: \sigma)$	Dynamic field selection
DInvoke $(E: \sigma, \bar{E}: \bar{\sigma})$	Dynamic delegate invocation
ObjectCreate $[\rho](\bar{E}: \bar{\sigma})$	Dynamic object creation
MInvoke $[m](E: \sigma, \bar{E}: \bar{\sigma})$	Dynamic method invocation
$MD ::=$	Target method descriptor
$C \langle \bar{X}_C \rangle \langle \bar{\tau}_C \rangle : : m \langle \bar{X}_m \rangle \langle \bar{\tau}_m \rangle : (\bar{\tau}_p) \rightarrow \tau_r$	
$SE ::=$	Statement expression
$E(\bar{E})$	Delegate invocation
$E.MD(\bar{E})$	Method invocation
new $MD(\bar{E})$	Object creation
$x = E$	Variable assignment

Notice that there are no cast expressions in C_{CLR}^\sharp , these have either been translated into explicit conversion calls or removed. There are two new syntactic categories: conversion expressions and dynamic expressions. The former includes operations to convert the representation of literals, and also boxing and unboxing operations which are an important feature of C^\sharp type system by which a value type can be converted to and from the **object** type. Dynamic expressions are used to denote the operations that

support the new dynamic behaviour in $C^{\#}4.0$. Their meaning will become clear when we define the operational semantics of $C^{\#}_{CLR}$ in §7.

A method descriptor fully identifies 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). Method descriptions replace method names after overloading resolution,⁶ and appear explicitly in MSIL (albeit with type parameters replaced by integers denoting their position) [26].

$C^{\#}_{CLR}$ statements are given by the following grammar.

$S ::=$	Statement
<code>;</code>	Skip
<code>SE;</code>	Expression statement
<code>if (E) S else S</code>	Conditional statement
<code>τ x = E;</code>	Variable declaration
<code>E.f = E;</code>	Field assignment
<code>return E;</code>	Return statement
<code>{\bar{s}}</code>	Block
<code>Assign[f](E: σ, E: σ);</code>	Dynamic field assignment

The type conversion relation for $C^{\#}_{CLR}$ is very straightforward, as it is essentially just the subclassing relation and a rule that any reference type can be converted to `object`. This relation is written $\tau_1 \leq \tau_2$ and its simple definition is omitted for lack of space.

The type system for $C^{\#}_{CLR}$ is also defined as a bidirectional system. Thus we have two typing relations: a type conversion relation and a type synthesis relation. The type conversion relation is written $\Gamma \triangleright E \leq \tau$ and is read informally that “in context Γ , the $C^{\#}_{CLR}$ expression E can be converted to type τ .” A context, Γ , is a function from variables to types. The rules are as follows.

$$\begin{array}{c}
\text{[CLR-Byte]} \frac{0 \leq i \leq 255}{\Gamma \triangleright i \leq \text{byte}} \quad \text{[CLR-Null]} \frac{}{\Gamma \triangleright \text{null} \leq \theta} \\
\text{[CLR-AME]} \frac{|dtype(D)(\bar{\tau})|^* = \bar{\tau}_0 \rightarrow \tau_1 \quad \Gamma, \bar{x}: \bar{\tau}_0 \triangleright \bar{S}_1 \leq \tau_1}{\Gamma \triangleright \text{delegate}(\bar{\tau}_0 \bar{x}) \{ \bar{S}_1 \} \leq D \langle \bar{\tau} \rangle} \\
\text{[CLR-Synth]} \frac{\Gamma \triangleright e_1 \uparrow \tau_0 \quad \tau_0 \leq \tau_1}{\Gamma \triangleright e_1 \leq \tau_1}
\end{array}$$

We make use of a function that translates a $FC^{\#}_4$ type into $C^{\#}_{CLR}$ type, i.e. it replaces occurrences of the type `dynamic` with `object`, and in all other respects is the identity function. For example $|C \langle \text{int}, \text{dynamic} \rangle|^* = C \langle \text{int}, \text{object} \rangle$. The type synthesis relation is written $\Gamma \triangleright E \uparrow \tau$ and is read informally that “in context Γ , the $C^{\#}_{CLR}$ expression E synthesizes type τ .” The rules are as follows.

⁶ The $C^{\#}$ overloading rules [12, §7.4.3] 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.

$$\begin{array}{c}
\text{[CLR-S-Int]} \frac{}{\Gamma \triangleright i \uparrow \mathbf{int}} \qquad \text{[CLR-S-Bool]} \frac{}{\Gamma \triangleright b \uparrow \mathbf{bool}} \\
\text{[CLR-S-Var]} \frac{}{\Gamma, x: \tau \triangleright x \uparrow \tau} \quad \text{[CLR-S-Field]} \frac{\Gamma \triangleright E_1 \uparrow \tau_1 \quad |f\text{type}(\tau_1, f)|^* = \tau_2}{\Gamma \triangleright E_1.f \uparrow \tau_2} \\
\text{[CLR-S-DellInv]} \frac{\Gamma \triangleright E_1 \uparrow D\langle\bar{\tau}\rangle \quad |d\text{type}(D)(\bar{\tau})|^* = \bar{\tau}_1 \rightarrow \tau_2 \quad \Gamma \triangleright \bar{E}_2 \leq \bar{\tau}_1}{\Gamma \triangleright E_1(\bar{E}_2) \uparrow \tau_2} \\
\text{[CLR-S-New]} \frac{MD = \mathbf{C}\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::\mathbf{ctor}:(\bar{\tau}_p) \quad \Gamma \triangleright \bar{e}_1 \leq \bar{\tau}_p[\bar{X}_C := \bar{\tau}_C]}{\Gamma \triangleright \mathbf{new} MD(\bar{E}_1) \uparrow \mathbf{C}\langle\bar{\tau}_C\rangle} \\
\text{[CLR-S-VarAssign]} \frac{\Gamma, x_1: \tau_1 \triangleright E_1 \leq \tau_1}{\Gamma, x_1: \tau_1 \triangleright x_1 = E_1 \uparrow \tau_1} \\
\text{[CLR-S-MethInv]} \frac{MD = \mathbf{C}\langle\bar{X}_C\rangle\langle\bar{\tau}_C\rangle::\mathbf{m}\langle\bar{X}_m\rangle\langle\bar{\tau}_1\rangle:(\bar{\tau}_p) \rightarrow \tau_r \quad \Gamma \triangleright E_1 \leq \mathbf{C}\langle\bar{\tau}_C\rangle \quad \Gamma \triangleright \bar{E}_2 \leq \bar{\tau}_p[\bar{X}_C, \bar{X}_m := \bar{\tau}_C, \bar{\tau}_1]}{\Gamma \triangleright E_1.MD(\bar{E}_2) \uparrow \tau_r} \\
\text{[CLR-S-B2]} \frac{\Gamma \triangleright E \leq \mathbf{byte}}{\Gamma \triangleright \mathbf{ByteToInt}(E) \uparrow \mathbf{int}} \quad \text{[CLR-S-I2B]} \frac{\Gamma \triangleright E \leq \mathbf{int}}{\Gamma \triangleright \mathbf{IntToByte}(E) \uparrow \mathbf{byte}} \\
\text{[CLR-S-Box]} \frac{\Gamma \triangleright E \leq \gamma}{\Gamma \triangleright \mathbf{Box}[\gamma](E) \uparrow \mathbf{object}} \quad \text{[CLR-S-Unbox]} \frac{\Gamma \triangleright E \leq \mathbf{object}}{\Gamma \triangleright \mathbf{Unbox}[\gamma](E) \uparrow \gamma} \\
\text{[CLR-S-Downcast]} \frac{\Gamma \triangleright E \uparrow \tau \quad \tau \leq |\rho|^*}{\Gamma \triangleright \mathbf{Downcast}[\rho](E) \uparrow |\rho|^*} \quad \text{[CLR-S-DConv]} \frac{\Gamma \triangleright E \leq \mathbf{object}}{\Gamma \triangleright \mathbf{Convert}[\sigma](E) \uparrow |\sigma|^*} \\
\text{[CLR-S-DMemAcc]} \frac{\Gamma \triangleright E \leq \mathbf{object}}{\Gamma \triangleright \mathbf{MemberAccess}[f](E) \uparrow \mathbf{object}} \\
\text{[CLR-S-DDellInv]} \frac{\Gamma \triangleright E_0 \leq |\sigma_0|^* \quad \sigma_0 = \mathbf{dynamic} \text{ or } D\langle\bar{\sigma}\rangle \quad \Gamma \triangleright \bar{E}_1 \leq |\bar{\sigma}_1|^*}{\Gamma \triangleright \mathbf{DInvoke}(E_0: \sigma_0, \bar{E}_1: \bar{\sigma}_1) \uparrow \mathbf{object}} \\
\text{[CLR-S-DNew]} \frac{\Gamma \triangleright \bar{E} \leq |\bar{\sigma}|^*}{\Gamma \triangleright \mathbf{ObjectCreate}[\rho](\bar{E}: \bar{\sigma}) \uparrow |\rho|^*} \\
\text{[CLR-S-DMethInv]} \frac{\Gamma \triangleright E_0 \leq |\sigma_0|^* \quad \sigma_0 = \mathbf{dynamic} \text{ or } C\langle\bar{\sigma}\rangle \quad \Gamma \triangleright \bar{E}_1 \leq |\bar{\sigma}_1|^*}{\Gamma \triangleright \mathbf{MInvoke}[m](E_0: \sigma_0, \bar{E}_1: \bar{\sigma}_1) \uparrow \mathbf{object}}
\end{array}$$

The type conversion relation for statements is written $\Gamma \triangleright S \leq \tau$, and is as follows.

$$\begin{array}{c}
\text{[CLR-Skip]} \frac{}{\Gamma \triangleright ; \leq \tau} \quad \text{[CLR-ExpStatement]} \frac{\Gamma \triangleright SE_1 \uparrow \tau_1}{\Gamma \triangleright SE_1 ; \leq \tau} \\
\text{[CLR-Cond]} \frac{\Gamma \triangleright E_1 \leq \text{bool} \quad \Gamma \triangleright S_1 \leq \tau \quad \Gamma \triangleright S_2 \leq \tau}{\Gamma \triangleright \text{if } (E_1) S_1 \text{ else } S_2 \leq \tau} \\
\text{[CLR-FAss]} \frac{\Gamma \triangleright E_1 \uparrow \tau_1 \quad |f\text{type}(\tau_1, f)|^* = \tau_2 \quad \Gamma \triangleright E_2 \leq \tau_2}{\Gamma \triangleright E_1.f = E_2 ; \leq \tau} \\
\text{[CLR-FAssDyn]} \frac{\Gamma \triangleright E_1 \leq |\sigma_1|^* \quad \Gamma \triangleright E_2 \leq |\sigma_2|^*}{\Gamma \triangleright \text{Assign}[f](E_1 : \sigma_1, E_2 : \sigma_2) ; \leq \tau} \\
\text{[CLR-ReturnExp]} \frac{\Gamma \triangleright E_1 \leq \tau_1}{\Gamma \triangleright \text{return } E_1 ; \leq \tau_1} \\
\text{[CLR-Seq]} \frac{\Gamma \triangleright E_1 \leq \tau_1 \quad x \notin \text{dom}(\Gamma) \quad \Gamma, x : \tau_1 \triangleright \overline{S_1} \leq \tau}{\Gamma \triangleright \tau_1 x = E_1 ; \overline{S_1} \leq \tau}
\end{array}$$

6 Translation

This section contains one of the main technical contributions of the paper: the formal details of a translation from FC_4^\sharp to $\text{C}_{\text{CLR}}^\sharp$. This translation is actually quite subtle and we believe justifies our formal approach.

6.1 Type conversions

In this section we consider type conversions (or subtyping) in C^\sharp . Type conversions are classified into *implicit* and *explicit* conversions, which determines whether an explicit cast is required. For example, `byte` can be implicitly converted to `int` and so an expression of type `byte` can be used where an expression of type `int` is expected. However, an expression of type `int` requires an explicit cast to be used where an expression of type `byte` is expected.

At the heart of both type conversion relations is the subclass relation which is defined by the programmer in the class declarations. We write $C_1 \langle \overline{\sigma_1} \rangle : C_2 \langle \overline{\sigma_2} \rangle$ for the subclass relation, which is defined as follows.

$$\frac{}{\rho : \rho} \frac{\text{class } C_1 \langle \overline{X} \rangle : C_2 \langle \overline{\sigma_2} \rangle \quad C_2 \langle \overline{\sigma_2} \rangle [\overline{X} := \overline{\sigma_1}] : C_3 \langle \overline{\sigma_3} \rangle}{C_1 \langle \overline{\sigma_1} \rangle : C_3 \langle \overline{\sigma_3} \rangle}$$

The implicit type conversion relation is written $\sigma_1 <_i \sigma_2 \rightsquigarrow C$ where σ_1 and σ_2 are FC_4^\sharp types and C is a conversion. A $\text{C}_{\text{CLR}}^\sharp$ conversion is represented as a linear context, i.e. a $\text{C}_{\text{CLR}}^\sharp$ expression with a single hole in it, which we write ‘ \bullet ’. The intention is that if $\sigma_1 <_i \sigma_2 \rightsquigarrow C$ then the $\text{C}_{\text{CLR}}^\sharp$ expression C is the code which converts a value of type $|\sigma_1|^*$ into a value of type $|\sigma_2|^*$. We write $C[E]$ to denote the context C with the hole replaced by the expression E . The implicit type conversion relation is defined as follows.

$$\begin{array}{c}
\text{[IC-RefI]} \frac{}{\sigma_1 <:_i \sigma_1 \rightsquigarrow \bullet} \qquad \text{[IC-ByteToInt]} \frac{}{\text{byte} <:_i \text{int} \rightsquigarrow \text{ByteToInt}(\bullet)} \\
\text{[IC-Val-Obj]} \frac{}{\gamma <:_i \text{object} \rightsquigarrow \text{Box}[\gamma](\bullet)} \qquad \text{[IC-Ref-Obj]} \frac{}{\rho <:_i \text{object} \rightsquigarrow \bullet} \\
\text{[IC-Sub]} \frac{C_1 <\overline{\sigma_1}> : C_2 <\overline{\sigma_2}>}{C_1 <\overline{\sigma_1}> <:_i C_2 <\overline{\sigma_2}> \rightsquigarrow \bullet} \qquad \text{[IC-Dynamic]} \frac{\sigma <:_i \text{object} \rightsquigarrow C}{\sigma <:_i \text{dynamic} \rightsquigarrow C}
\end{array}$$

Rule [IC-RefI] states that all types can be implicitly converted to themselves. Rule [IC-ByteToInt] states that any value of type `byte` can be converted to a value of type `int`, using the `ByteToInt(-)` expression. Rule [IC-Val-Obj] captures the property that any value of a value type can be implicitly boxed to type `object`. Rule [IC-Ref-Obj] states that any reference type (either class or delegate) can be implicitly converted to type `object`. In other words, these two rules imply that `object` is the top type. Rule [IC-Sub] states that one class can be implicitly converted to another if it is a subclass. Note that these first five rules are *unchanged* from C#3.0; the only new rule is [IC-Dynamic]. This rule simply states that a type can be implicitly converted to `dynamic` if it can be implicitly converted to `object`.

It is important to note that we do *not* have a rule that states that the type `dynamic` can be implicitly converted to any type. Clearly adding such a rule means that any two types are implicitly convertible!

The explicit type conversion relation is written $\sigma_1 <:_x \sigma_2 \rightsquigarrow C$, and is defined as follows.

$$\begin{array}{c}
\text{[XC-RefI]} \frac{}{\sigma_1 <:_x \sigma_1 \rightsquigarrow \bullet} \qquad \text{[XC-IntToByte]} \frac{}{\text{int} <:_x \text{byte} \rightsquigarrow \text{IntToByte}(\bullet)} \\
\text{[XC-ObjVal]} \frac{}{\text{object} <:_x \gamma \rightsquigarrow \text{Unbox}[\gamma](\bullet)} \qquad \text{[XC-ObjRef]} \frac{\rho \neq \text{dynamic}}{\text{object} <:_x \rho \rightsquigarrow \text{Downcast}[\rho](\bullet)} \\
\text{[XC-Down]} \frac{C_1 <\overline{\sigma_1}> : C_2 <\overline{\sigma_2}>}{C_2 <\overline{\sigma_2}> <:_x C_1 <\overline{\sigma_1}> \rightsquigarrow \text{Downcast}[C_1 <\overline{\sigma_1}>](\bullet)} \qquad \text{[XC-IC]} \frac{\sigma_1 <:_i \sigma_2 \rightsquigarrow C}{\sigma_1 <:_x \sigma_2 \rightsquigarrow C}
\end{array}$$

Interestingly, this relation is almost unchanged from C#3.0; the only slight amendment is a precondition in the rule [XC-ObjRef] which resolves the potential ambiguity when determining whether `object` can be explicitly converted to `dynamic` (the preferred conversion is the identity conversion).

Proposition 1 (Type conversion relations are functions).

1. If $\sigma_1 <:_i \sigma_2 \rightsquigarrow C_1$ and $\sigma_1 <:_i \sigma_2 \rightsquigarrow C_2$ then $C_1 = C_2$.
2. If $\sigma_1 <:_x \sigma_2 \rightsquigarrow C_1$ and $\sigma_1 <:_x \sigma_2 \rightsquigarrow C_2$ then $C_1 = C_2$.

6.2 Term conversions

The type checking relation for expressions, as for type conversions, comes in two flavours: one for implicit conversions and one for explicit conversions. The first judgement form is written $\Gamma \vdash e <:_i \sigma \rightsquigarrow E$ and is read informally that “in context Γ , the FC_4^\sharp expression e can be implicitly converted to type σ yielding $\text{C}_{\text{CLR}}^\sharp$ expression E .”

The second judgement form is written $\Gamma \vdash e <:_x \sigma \rightsquigarrow E$ and is read informally that “in context Γ , the FC_4^\sharp expression e can be explicitly converted to type σ yielding $\text{C}_{\text{CLR}}^\sharp$ expression E .”

The rules for implicit conversion of expressions are as follows.

$$\begin{array}{c}
\text{[IC-Byte]} \frac{0 \leq i \leq 255}{\Gamma \vdash i <:_i \text{byte} \rightsquigarrow i} \quad \text{[IC-Null]} \frac{}{\Gamma \vdash \text{null} <:_i \rho \rightsquigarrow \text{null}} \\
\text{[IC-AME]} \frac{d\text{type}(\text{D})(\bar{\sigma}) = \bar{\sigma}_1 \rightarrow \sigma_2 \quad \Gamma, \bar{x}: \bar{\sigma}_1 \vdash \bar{s}_1 <:_i \sigma_2 \rightsquigarrow \bar{S}_1}{\Gamma \vdash \text{delegate}(\bar{\sigma}_0 \bar{x}) \{\bar{s}_1\} <:_i \text{D} \langle \bar{\sigma} \rangle \rightsquigarrow \text{delegate}(\bar{\sigma}_0 \bar{x}) \{\bar{S}_1\}} \\
\text{[IC-Synth]} \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \quad \sigma_0 \neq \text{dynamic} \quad \sigma_0 <:_i \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <:_i \sigma_1 \rightsquigarrow C[E_1]} \\
\text{[IC-Dynamic]} \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1}{\Gamma \vdash e_1 <:_i \sigma_1 \rightsquigarrow \text{Convert}[\sigma_1](E_1: \text{dynamic})}
\end{array}$$

The rule [IC-Synth] forms the heart of this relation: an expression e_1 can be implicitly converted to type σ_1 if it synthesizes a type σ_0 which is implicitly convertible to σ_1 . However, C^\sharp includes some other special-case rules for implicit conversion of expressions. First, we need to add special rules to deal with the `null` literal and for AMEs. As mentioned earlier, these are both expression forms that do *not* synthesize types but clearly they can both be implicitly converted to appropriate types. In the case of rule [IC-Null], the `null` literal can be implicitly converted to any reference type. Rule [IC-AME] states that an AME can be converted to a delegate type if its contents satisfy the component types of the delegate type. C^\sharp includes special implicit conversion rules for small literal values. Thus rule [IC-Byte] states that a positive integer literal that fits into 8 bits can be considered of type `byte`.

To support dynamic typing, we need only add an extra rule: [IC-Dynamic]. This rule states that an expression e_1 can be implicitly converted to a type σ_1 if it synthesizes a type `dynamic`. In this case we need to yield code to perform the type test at *run-time*. This is the role of the target expression $\text{Convert}[\sigma_1](E_1: \text{dynamic})$; it performs a runtime type-test on the target expression E_1 .

This rule highlights the major design principle in adapting the $\text{C}^\sharp 3.0$ rules to support dynamic types: all rules that traditionally synthesized types of subexpressions now have special cases for when the synthesized type is `dynamic`.

The two rules for the explicit conversion of expressions are as follows.

$$\begin{array}{c}
\text{[XC-Synth]} \frac{\Gamma \vdash e_1 \uparrow \sigma_0 \rightsquigarrow E_1 \quad \sigma_0 \neq \text{dynamic} \quad \sigma_0 <:_x \sigma_1 \rightsquigarrow C}{\Gamma \vdash e_1 <:_x \sigma_1 \rightsquigarrow C[E_1]} \\
\text{[XC-Dynamic]} \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1}{\Gamma \vdash e_1 <:_x \sigma_1 \rightsquigarrow \text{Convert}[\sigma_1](E_1: \text{dynamic})}
\end{array}$$

These rules are similar in that if the expression synthesizes the type `dynamic` then we delay the type-test until runtime (rule [XC-Dynamic]), and if not we use the explicit conversion relation defined in §6.1 (rule [XC-Synth]).

We also need to type check statements, primarily to ensure that any `return` statements satisfy the expected return type. Thus we only need one judgement form, which

we write $\Gamma \vdash s <_i \sigma \rightsquigarrow S$ and is read informally that “in context Γ , the FC_4^\sharp statement s can be implicitly converted to type σ yielding $\text{C}_{\text{CLR}}^\sharp$ statement S .” The rules for the implicit conversion of statements are as follows.

$$\begin{array}{c}
\text{[C-Skip]} \frac{}{\Gamma \vdash ; <_i \sigma \rightsquigarrow ;} \quad \text{[C-ExpStatement]} \frac{\Gamma \vdash se_1 \uparrow \sigma_1 \rightsquigarrow SE_1}{\Gamma \vdash se_1 ; <_i \sigma \rightsquigarrow SE_1;} \\
\text{[C-Cond]} \frac{\Gamma \vdash e_1 <_i \text{bool} \rightsquigarrow E_1 \quad \Gamma \vdash s_1 <_i \sigma \rightsquigarrow S_1 \quad \Gamma \vdash s_2 <_i \sigma \rightsquigarrow S_2}{\Gamma \vdash \text{if } (e_1) s_1 \text{ else } s_2 <_i \sigma \rightsquigarrow \text{if } (E_1) S_1 \text{ else } S_2} \\
\text{[C-FAss]} \frac{\Gamma \vdash e_1 \uparrow \sigma_1 \rightsquigarrow E_1 \quad \sigma_1 \neq \text{dynamic} \quad \text{ftype}(\sigma_1, f) = \sigma_2 \quad \Gamma \vdash e_2 <_i \sigma_2 \rightsquigarrow E_2}{\Gamma \vdash e_1.f=e_2 ; <_i \sigma \rightsquigarrow E_1.f=E_2;} \\
\text{[C-FAssDyn]} \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1 \quad \Gamma \vdash e_2 \uparrow^+ \sigma_2 \rightsquigarrow E_2}{\Gamma \vdash e_1.f=e_2 ; <_i \sigma \rightsquigarrow \text{Assign}[f](E_1:\text{dynamic}, E_2:\sigma_2);} \\
\text{[C-ReturnExp]} \frac{\Gamma \vdash e_1 <_i \sigma \rightsquigarrow E_1}{\Gamma \vdash \text{return } e_1 ; <_i \sigma \rightsquigarrow \text{return } E_1;} \\
\text{[C-Seq]} \frac{\Gamma \vdash e_1 <_i \sigma_1 \rightsquigarrow E_1 \quad x \notin \text{dom}(\Gamma) \quad \Gamma, x:\sigma_1 \vdash \overline{s_1} <_i \sigma \rightsquigarrow \overline{S_1}}{\Gamma \vdash \sigma_1 x = e_1 ; \overline{s_1} <_i \sigma \rightsquigarrow |\sigma_1|^* x = E_1 ; \overline{S_1}}
\end{array}$$

These rules are unchanged from $\text{C}^\sharp 3.0$. The only new rule is [C-FAssDyn], which follows the pattern of earlier rules. As the receiving expression synthesizes the type `dynamic`, we package up the components along with their compile-time types so the field assignment will be checked and performed at runtime. We make use of an extended synthesis relation (written \uparrow^+), which we will define and explain in the following section.

6.3 Type synthesis

The heart of the C^\sharp type system is the type synthesis phase. This is only defined over expressions; there is no notion of type synthesis for statements. Judgements are written $\Gamma \vdash e_1 \uparrow \sigma_1 \rightsquigarrow E_1$, and can be informally read as “in context Γ , the FC_4^\sharp expression e_1 synthesizes the type σ_1 yielding a $\text{C}_{\text{CLR}}^\sharp$ expression E_1 .” The rules that are unchanged from $\text{C}^\sharp 3.0$ are given in Fig. 1, and those that involve the addition of the `dynamic` type are given in Fig. 2.

Space prevents us from a full description of all these rules. Instead we shall describe just the process of synthesizing a type for a method invocation. This is the most complicated part of the type synthesis process and involves the rules [S-MInv], [MInvDyn1] and [S-MInvDyn2]. The intention is to synthesize a type for the FC_4^\sharp expression $e_1.m\langle\overline{\sigma_1}\rangle(\overline{e_2})$.⁷ The first step is to synthesize a type, σ_1 , for subexpression e_1 . If it synthesizes the `dynamic` type, we use rule [S-MInvDyn1]: we synthesize types for the arguments to the invocation and yield a $\text{C}_{\text{CLR}}^\sharp$ dynamic expression. The intention is that at runtime this dynamic expression will use the runtime type of the expression

⁷ For simplicity, in FC_4^\sharp we require that all generic method invocations are passed type argument lists. In C^\sharp proper this can be omitted in which case an inference phase is performed to infer the type argument list [4].

$$\begin{array}{c}
\text{[S-Bool]} \frac{}{\Gamma \vdash b \uparrow \mathbf{bool} \rightsquigarrow b} \quad \text{[S-Int]} \frac{}{\Gamma \vdash i \uparrow \mathbf{int} \rightsquigarrow i} \quad \text{[S-Var]} \frac{}{\Gamma, x: \sigma \vdash x \uparrow \sigma \rightsquigarrow x} \\
\text{[S-Cast]} \frac{\Gamma \vdash e_1 <:_{\times} \sigma_1 \rightsquigarrow E_1}{\Gamma \vdash (\sigma_1) e_1 \uparrow \sigma_1 \rightsquigarrow E_1} \\
\text{[S-Field]} \frac{\Gamma \vdash e_1 \uparrow \sigma_1 \rightsquigarrow E_1 \quad \sigma_1 \neq \mathbf{dynamic} \quad \text{ftype}(\sigma_1, f) = \sigma_2}{\Gamma \vdash e_1.f \uparrow \sigma_2 \rightsquigarrow E_1.f} \\
\text{[S-Dellnv]} \frac{\Gamma \vdash e_1 \uparrow \mathbf{D}\langle \bar{\sigma} \rangle \rightsquigarrow E_1 \quad \text{dtype}(\mathbf{D})(\bar{\sigma}) = \bar{\sigma}_1 \rightarrow \sigma_2 \quad \Gamma \vdash \bar{e}_2 <:; \bar{\sigma}_1 \rightsquigarrow \bar{E}_2}{\Gamma \vdash e_1(\bar{e}_2) \uparrow \sigma_2 \rightsquigarrow E_1(\bar{E}_2)} \\
\text{[S-New]} \frac{\text{AMG} \stackrel{\text{def}}{=} \{ \mathbf{C}\langle \bar{X}_C \rangle \langle \bar{\sigma}_C \rangle :: \dots \mathbf{ctor}: (\bar{\sigma}_p) \mid \\ \mathbf{C}\langle \bar{X}_C \rangle \langle \bar{\sigma}_C \rangle :: \dots \mathbf{ctor}: (\bar{\sigma}_p) \in \text{CMG}, \\ |\bar{\sigma}_p| = |\bar{e}_1|, \Gamma \vdash \bar{e}_1 <:; \bar{\sigma}_p[\bar{X}_C := \bar{\sigma}_c] \} \\ \Gamma \vdash \text{best}(\text{AMG}, \bar{e}_1) \rightsquigarrow \text{md} = \mathbf{C}\langle \bar{X}_C \rangle \langle \bar{\sigma}_C \rangle :: \dots \mathbf{ctor}: (\bar{\sigma}_p) \\ \Gamma \vdash \bar{e}_1 <:; \bar{\sigma}_p[\bar{X}_C := \bar{\sigma}_c] \rightsquigarrow \bar{E}_1}{\Gamma \vdash \mathbf{new} \mathbf{C}\langle \bar{\sigma} \rangle(\bar{e}_1) \uparrow \mathbf{C}\langle \bar{\sigma} \rangle \rightsquigarrow \mathbf{new} |\text{md}|^*(\bar{E}_1)} \\
\text{[S-VarAssign]} \frac{\Gamma, x_1: \sigma_1 \vdash e_1 <:; \sigma_1 \rightsquigarrow E_1}{\Gamma, x_1: \sigma_1 \vdash x_1 = e_1 \uparrow \sigma_1 \rightsquigarrow x_1 = E_1} \\
\text{[S-MInv]} \frac{\Gamma \vdash e_1 \uparrow \sigma \rightsquigarrow E_1 \quad \sigma \neq \mathbf{dynamic} \quad \text{CMG} \stackrel{\text{def}}{=} \text{mtype}(\sigma, m) \\ \text{AMG} \stackrel{\text{def}}{=} \{ \mathbf{C}\langle \bar{X}_C \rangle \langle \bar{\sigma}_C \rangle :: \dots \mathbf{m}\langle \bar{X}_m \rangle \langle \bar{\sigma}_1 \rangle: (\bar{\sigma}_p) \rightarrow \sigma_r \mid \\ \mathbf{C}\langle \bar{X}_C \rangle \langle \bar{\sigma}_C \rangle :: \dots \mathbf{m}\langle \bar{X}_m \rangle \langle \bar{\sigma}_1 \rangle: (\bar{\sigma}_p) \rightarrow \sigma_r \in \text{CMG}, \\ |\bar{X}_m| = |\bar{\sigma}_1|, \Gamma \vdash \bar{e}_2 <:; \bar{\sigma}_p[\bar{X}_C := \bar{\sigma}_C, \bar{X}_m := \bar{\sigma}_1] \} \\ \Gamma \vdash \text{best}(\text{AMG}, \bar{e}_2) \rightsquigarrow \text{md} = \mathbf{C}\langle \bar{X}_C \rangle \langle \bar{\sigma}_C \rangle :: \dots \mathbf{m}\langle \bar{X}_m \rangle \langle \bar{\sigma}_1 \rangle: (\bar{\sigma}_p) \rightarrow \sigma_r \\ \Gamma \vdash \bar{e}_2 <:; \bar{\sigma}_p[\bar{X}_C := \bar{\sigma}_C, \bar{X}_m := \bar{\sigma}_1] \rightsquigarrow \bar{E}_2}{\Gamma \vdash e_1.\mathbf{m}\langle \bar{\sigma}_1 \rangle(\bar{e}_2) \uparrow \sigma_r[\bar{X}_C := \bar{\sigma}_C, \bar{X}_m := \bar{\sigma}_1] \rightsquigarrow E_1.\text{md}^*(\bar{E}_2)}
\end{array}$$

Fig. 1. Type synthesis of FC_4^{d} expressions, part I

$$\begin{array}{c}
\text{[S-FieldDyn]} \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1}{\Gamma \vdash e_1.f \uparrow \text{dynamic} \rightsquigarrow \text{MemberAccess}[f](E_1:\text{dynamic})} \\
\text{[S-DInvDyn1]} \frac{\Gamma \vdash e \uparrow \text{D}\langle\bar{\sigma}\rangle \rightsquigarrow E \quad \text{dtype}(\text{D})(\bar{\sigma}) = \bar{\sigma}_2 \rightarrow \sigma_3 \quad \exists i.1 \leq i \leq n.\sigma_i = \text{dynamic}}{\Gamma \vdash e_1 \uparrow^+ \sigma_1 \rightsquigarrow E_1 \cdots \Gamma \vdash e_n \uparrow^+ \sigma_n \rightsquigarrow E_n \quad \Gamma \vdash e(e_1, \dots, e_n) \uparrow \text{dynamic} \rightsquigarrow \text{DInvoke}(E:\text{D}\langle\bar{\sigma}\rangle, (E_1:\sigma_1, \dots, E_n:\sigma_n))} \\
\text{[S-DInvDyn2]} \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1 \quad \Gamma \vdash \bar{e}_2 \uparrow^+ \bar{\sigma}_2 \rightsquigarrow \bar{E}_2}{\Gamma \vdash e_1(\bar{e}_2) \uparrow \text{dynamic} \rightsquigarrow \text{DInvoke}(E_1:\text{dynamic}, \bar{E}_2:\bar{\sigma}_2)} \\
\text{[S-NewDyn]} \frac{\text{CMG} \stackrel{\text{def}}{=} \text{mtype}(\text{C}\langle\bar{\sigma}\rangle, \text{.ctor}) \quad \Gamma \vdash e_1 \uparrow^+ \sigma_1 \rightsquigarrow E_1 \cdots \Gamma \vdash e_n \uparrow^+ \sigma_n \rightsquigarrow E_n \quad \exists j.1 \leq j \leq n.\sigma_j = \text{dynamic} \quad \text{AMG} \stackrel{\text{def}}{=} \{\text{C}\langle\bar{X}_C\rangle\langle\bar{\sigma}_C\rangle::\text{.ctor}:(\bar{\sigma}') \mid \text{C}\langle\bar{X}_C\rangle\langle\bar{\sigma}_C\rangle::\text{.ctor}:(\bar{\sigma}') \in \text{CMG}, |\bar{\sigma}'| = n, \Gamma \vdash \bar{e}_i <_i \bar{\sigma}'_i[\bar{X}_C := \bar{\sigma}_C] i \in 1..n\} \quad |\text{AMG}| \geq 1}{\Gamma \vdash \text{new C}\langle\bar{\sigma}\rangle(e_1, \dots, e_n) \uparrow \text{C}\langle\bar{\sigma}\rangle \rightsquigarrow \text{ObjectCreate}[\text{C}\langle\bar{\sigma}\rangle](E_1:\sigma_1, \dots, E_n:\sigma_n)} \\
\text{[S-MInvDyn1]} \frac{\Gamma \vdash e_1 \uparrow \text{dynamic} \rightsquigarrow E_1 \quad \Gamma \vdash \bar{e}_2 \uparrow^+ \bar{\sigma} \rightsquigarrow \bar{E}_2}{\Gamma \vdash e_1.m\langle\bar{\sigma}_1\rangle(\bar{e}_2) \uparrow \text{dynamic} \rightsquigarrow \text{MInvoke}[m](E_1:\text{dynamic}, \bar{E}_2:\bar{\sigma})} \\
\text{[S-MInvDyn2]} \frac{\Gamma \vdash e \uparrow \sigma \rightsquigarrow E \quad \text{CMG} \stackrel{\text{def}}{=} \text{mtype}(\sigma, m) \quad \Gamma \vdash e_1 \uparrow^+ \sigma_1 \rightsquigarrow E_1 \cdots \Gamma \vdash e_n \uparrow^+ \sigma_n \rightsquigarrow E_n \quad \exists j.1 \leq j \leq n.\sigma_j = \text{dynamic} \quad \text{AMG} \stackrel{\text{def}}{=} \{\text{C}\langle\bar{X}_C\rangle\langle\bar{\sigma}_C\rangle::m\langle\bar{X}_m\rangle\langle\bar{\sigma}_1\rangle:(\bar{\sigma}') \rightarrow \sigma_r \mid \text{C}\langle\bar{X}_C\rangle\langle\bar{\sigma}_C\rangle::m\langle\bar{X}_m\rangle\langle\bar{\sigma}_1\rangle:(\bar{\sigma}') \rightarrow \sigma_r \in \text{CMG}, |\bar{X}_m| = |\bar{\sigma}_1|, |\bar{\sigma}'| = n, \Gamma \vdash \bar{e}_i <_i \bar{\sigma}'_i[\bar{X}_C := \bar{\sigma}_C, \bar{X}_m := \bar{\sigma}_1] i \in 1..n\} \quad |\text{AMG}| \geq 1}{\Gamma \vdash e.m\langle\bar{\sigma}_1\rangle(e_1, \dots, e_n) \uparrow \text{dynamic} \rightsquigarrow \text{MInvoke}[m](E:\sigma, (E_1:\sigma_1, \dots, E_n:\sigma_n))}
\end{array}$$

Fig. 2. Type synthesis of $\text{C}_{\text{CLR}}^\sharp$ expressions, part II

e_1 to resolve the method invocation (if possible). Clearly the overall synthesized type in this case is `dynamic`.

In synthesizing the types for the arguments we used a modified version of the type synthesis relation. This relation is written $\Gamma \vdash e \uparrow^+ \sigma \rightsquigarrow E$ and is defined as follows.

$$\frac{}{\Gamma \vdash i \uparrow^+ \mathbf{int}^l \rightsquigarrow i} \quad \frac{}{\Gamma \vdash \mathbf{null} \uparrow^+ \mathbf{object} \rightsquigarrow \mathbf{null}} \quad \frac{\Gamma \vdash e_1 \uparrow \sigma \rightsquigarrow E_1}{\Gamma \vdash e_1 \uparrow^+ \sigma \rightsquigarrow E_1}$$

This relation is a small extension of the normal type synthesis relation, which serves two purposes: (1) To allow the `null` literal to synthesize a type (`object`), and (2) to record whether an expression that synthesizes the type `int` is an integer literal or not. Hence, assuming a variable d of type `dynamic` then the expressions $d.m(42)$ and $d.m(40+2)$ and $d.m(\mathbf{null})$ all synthesize the type `dynamic`, but yield the C_{CLR}^\sharp expressions $\text{MInvoke}[m](d: \mathbf{dynamic}, 42: \mathbf{int}^l)$, $\text{MInvoke}[m](d: \mathbf{dynamic}, 40 + 2: \mathbf{int})$ and $\text{MInvoke}[m](d: \mathbf{dynamic}, \mathbf{null}: \mathbf{object})$, respectively. However, the expression $d.m(\text{delegate}(\mathbf{int} \ x)\{ \text{return } x; \})$ fails to synthesize a type as we are unable to synthesize a type for the argument.

Let us return to synthesizing a type for the invocation expression $e_1.m \langle \overline{\sigma_1} \rangle (\overline{e_2})$, where e_1 synthesizes a type σ_1 which is *not* `dynamic`. We now wish to consider whether any of the invocation arguments are of type `dynamic`. We use the extended synthesis relation to synthesize types for the invocation arguments. If one or more of them synthesizes the type `dynamic` we use rule [S-MInvDyn2] to synthesize the overall type `dynamic` and yield a C_{CLR}^\sharp dynamic expression. (In fact, the rule [S-MInvDyn2] contains a slight optimization, but one which will appear in $C^\sharp 4.0$. It is checked to see if there is at least one potential applicable method. If there is not, then there is no point delaying matters until runtime as we are certain that the method invocation will fail.)

If neither of these two cases hold, then we consider the method invocation in the same way as for $C^\sharp 3.0$, using rule [S-MInv]. Thus from σ_1 we generate the Candidate Method Group (CMG) for method m . This is simply the set of method descriptors for every method m accessible from type σ_1 . We then generate the Applicable Method Group (AMG), which is essentially all the methods from the CMG with the correct number of type parameters and whose argument types are applicable, i.e. the arguments $\overline{e_2}$ can be implicitly converted to the argument types. Next we need to resolve this set using overloading resolution [12, §7.4.3]. We omit a formalization and simply assume a function *best* that returns the best method descriptor (if it exists) from a given applicable method group given also a context and an argument list. Assuming that there is a best method descriptor, we then implicitly convert the arguments, synthesize the return type and yield the appropriate C_{CLR}^\sharp expression (noting that it records the method descriptor—with occurrences of `dynamic` replaced with `object`—not just the method name).

6.4 Formal properties

In this section we briefly mention the key property of our translations of FC_4^\sharp to C_{CLR}^\sharp . We do not give any details of the proof; they are all quite routine and appear in a supporting technical report.

The main technical result is that the translation of FC_4^\sharp into C_{CLR}^\sharp is type-preserving. In other words, if there is a translation then the resulting C_{CLR}^\sharp fragment is well-typed.

Theorem 1 (Preservation of typing by translation).

1. If $\Gamma \vdash e_1 <:_i \sigma_1 \rightsquigarrow E_1$ then $|\Gamma|^* \triangleright E_1 \leq |\sigma_1|^*$
2. If $\Gamma \vdash e_1 <:_x \sigma_1 \rightsquigarrow E_1$ then $|\Gamma|^* \triangleright E_1 \uparrow |\sigma_1|^*$
3. If $\Gamma \vdash s_1 <:_i \sigma_1 \rightsquigarrow S_1$ then $|\Gamma|^* \triangleright S_1 \leq |\sigma_1|^*$
4. If $\Gamma \vdash e_1 \uparrow \sigma_1 \rightsquigarrow E_1$ then $|\Gamma|^* \triangleright E_1 \uparrow |\sigma_1|^*$
5. If $\Gamma \vdash e_1 \uparrow^+ \sigma_1 \rightsquigarrow E_1$ then $|\Gamma|^* \triangleright E_1 \leq |\sigma_1|^*$

Proof. By simultaneous induction over the translation relations.

7 Operational semantics

In this section we define the operational semantics of $\mathbf{C}_{\text{CLR}}^\sharp$ and show in particular how the dynamic expressions reuse the compile-time typing and resolution rules at runtime. We follow closely the MJ operational semantics [7, 6] and define evaluation in terms of a transition relation between configurations, rather than using evaluation contexts.

A configuration is a four-tuple, written $\langle H, ST, F, FS \rangle$, where H is a heap, S is a stack, F is a frame and FS is a frame stack. A heap is a map from object identifiers (ranged over by o) to heap objects. A heap object is a pair of a type and a field function, which is a map from field names to runtime values. A runtime value, r , is either a value, the `null` literal, or an object identifier. A value is either an integer or a boolean literal. A stack is essentially a map from variables to object identifiers. However, to model correctly the block-structured scoping of \mathbf{C}^\sharp it is actually a list of list of functions from variables to object identifiers. A frame, F , is either a statement, a sequence of statements, or an expression. A frame stack is essentially the program context in which the frame is currently being evaluated. More precisely, it is a list of expressions or statements containing a single hole. We refer the reader to earlier work for more details [7, 6].

We define a binary transition relation between configurations, which is written $\langle H_1, ST_1, F_1, FS_1 \rangle \rightsquigarrow \langle H_2, S_2, F_2, FS_2 \rangle$. Given the space restrictions we omit the transition rules for the standard constructs of $\mathbf{C}_{\text{CLR}}^\sharp$ as they are almost identical to the corresponding rules for MJ [7, 6]. We give the transition rules for just the conversion expressions and the dynamic expressions of $\mathbf{C}_{\text{CLR}}^\sharp$. The transition rules for conversion expressions are as follows.⁸

$$\begin{array}{c}
 \text{[E-Box]} \frac{o \notin \text{dom}(H) \quad H' \stackrel{\text{def}}{=} H \uparrow [o \mapsto \langle \gamma, \{\text{value} \mapsto v\} \rangle]}{\langle H, ST, \text{Box}[\gamma](v), FS \rangle \rightsquigarrow \langle H', ST, o, FS \rangle} \\
 \\
 \text{[E-Unbox]} \frac{H(o) = \langle \gamma, \{\text{value} \mapsto v\} \rangle}{\langle H, ST, \text{Unbox}[\gamma](o), FS \rangle \rightsquigarrow \langle H, ST, v, FS \rangle} \\
 \\
 \text{[E-ByteToInt]} \frac{}{\langle H, ST, \text{ByteToInt}(i), FS \rangle \rightsquigarrow \langle H, ST, i, FS \rangle} \\
 \\
 \text{[E-IntToByte]} \frac{}{\langle H, ST, \text{IntToByte}(i), FS \rangle \rightsquigarrow \langle H, ST, [i]_8, FS \rangle} \\
 \\
 \text{[E-Downcast]} \frac{H(o) = \langle \theta, v \rangle \quad \theta \leq |\rho|^*}{\langle H, ST, \text{Downcast}[\rho](o), FS \rangle \rightsquigarrow \langle H, ST, o, FS \rangle}
 \end{array}$$

⁸ We write $[i]_8$ for the 8-bit truncation of the integer i .

The intention of the dynamic expressions is that they perform compile-time typing and resolution at runtime. To be able to reuse the FC_4^\sharp judgements from earlier, we need to make a few extensions. First, we extend contexts to additionally map object identifiers to types; and write $|H|$ to denote the translation of a heap, H , into a context. Secondly, we add a new form of expression to FC_4^\sharp , which we call a *payload component*, and is written $r:\sigma$ where r is a $\text{C}_{\text{CLR}}^\sharp$ runtime value. Finally, we need to add type synthesis and type checking rules for a payload component expression. The type synthesis rules for a payload expression are as follows.

$$\begin{array}{c} \text{[S-PayODyn]} \frac{}{\Gamma, o: \tau \vdash o: \text{dynamic} \uparrow \tau \rightsquigarrow o} \quad \text{[S-PayIntDyn]} \frac{}{\Gamma \vdash \underline{i}: \text{dynamic} \uparrow \text{int} \rightsquigarrow \underline{i}} \\ \text{[S-PayStatic]} \frac{\sigma \neq \text{dynamic}}{\Gamma \vdash o: \sigma \uparrow \sigma \rightsquigarrow o} \end{array}$$

The rule [S-PayODyn] states that if the payload had the static type `dynamic`, then at runtime we look up the runtime type from the context. Rule [S-PayIntDyn] states that if the payload had the static type `dynamic` and is an integer literal, then we synthesize the type `int`. The rule [S-PayStatic] states that a payload expression whose compile-time type, σ , was not `dynamic` synthesizes simply σ . Notice that the payload expression `null:dynamic` does *not* synthesize a type.

The rules for the implicit conversion of payload expressions are as follows.

$$\begin{array}{c} \text{[C-PayODyn]} \frac{\tau <_{;i} \sigma \rightsquigarrow C}{\Gamma, o: \tau \vdash o: \text{dynamic} <_{;i} \sigma \rightsquigarrow C[o]} \\ \text{[C-PayNullDyn]} \frac{}{\Gamma \vdash \text{null}: \text{dynamic} <_{;i} \rho \rightsquigarrow \text{null}} \\ \text{[C-PayIntDyn]} \frac{\text{int} <_{;i} \sigma \rightsquigarrow C}{\Gamma \vdash \underline{i}: \text{dynamic} <_{;i} \sigma \rightsquigarrow C[\underline{i}]} \quad \text{[C-PayIntLit]} \frac{1 \leq \underline{i} \leq 255}{\Gamma \vdash \underline{i}: \text{int}^l <_{;i} \text{byte} \rightsquigarrow \underline{i}} \\ \text{[C-PayStatic]} \frac{\sigma_1 \neq \text{dynamic} \quad \sigma_1 <_{;i} \sigma_2 \rightsquigarrow C}{\Gamma \vdash r: \sigma_1 <_{;i} \sigma_2 \rightsquigarrow C[r]} \end{array}$$

The rule [C-PayODyn] states that if the payload expression has the static type `dynamic`, then it can be implicitly converted to a type σ if its runtime type σ_r can be implicitly converted to σ . The rule [C-PayIntDyn] states that if the payload had the static type `dynamic` and is an integer literal, then it can be converted to a type σ if the type `int` can be converted to σ . The rule [C-PayIntLit] applies in conjunction with our extended synthesis rules (defined in §6.3). If the payload expression had the static type `intl` then it was an integer literal at compile-time. Thus the payload expression can be implicitly converted to type `byte` if the integer literal is positive and small enough. The rule [C-PayStatic] states that a payload expression whose compile-time type, σ_1 , was not `dynamic` can be implicitly converted to a type σ_2 , if the type σ_1 can be implicitly converted to σ_2 .

The transition rules for $\text{C}_{\text{CLR}}^\sharp$ dynamic expressions are as follows.

$$\text{[E-Convert]} \frac{|H| \vdash o: \sigma_1 <_{;i} \sigma_2 \rightsquigarrow E}{\langle H, ST, \text{Convert}[\sigma_2](o: \sigma_1), FS \rangle \rightsquigarrow \langle H, ST, E, FS \rangle}$$

$$\begin{array}{c}
\text{[E-DMemAcc]} \frac{|H| \vdash (o: \sigma). f <:; \mathbf{object} \rightsquigarrow E}{\langle H, ST, \mathbf{MemberAccess}[f](o: \sigma), FS \rangle \rightarrow \langle H, ST, E, FS \rangle} \\
\text{[E-DNew]} \frac{|H| \vdash \mathbf{new} C \langle \bar{\sigma} \rangle (r: \sigma') <:; \mathbf{object} \rightsquigarrow E}{\langle H, ST, \mathbf{ObjectCreate}[C \langle \bar{\sigma} \rangle](r: \sigma'), FS \rangle \rightarrow \langle H, ST, E, FS \rangle} \\
\text{[E-DDellInv]} \frac{|H| \vdash o: \sigma (r: \sigma') <:; \mathbf{object} \rightsquigarrow E}{\langle H, ST, \mathbf{DInvoke}(o: \sigma, r: \sigma'), FS \rangle \rightarrow \langle H, ST, E, FS \rangle} \\
\text{[E-DMethInv]} \frac{|H| \vdash (o: \sigma). m (r: \sigma') <:; \mathbf{object} \rightsquigarrow E}{\langle H, ST, \mathbf{MInvoke}[m](o: \sigma, r: \sigma'), FS \rangle \rightarrow \langle H, ST, E, FS \rangle} \\
\text{[E-DFAss]} \frac{|H| \vdash (o: \sigma). f = (o': \sigma') <:; \mathbf{object} \rightsquigarrow E}{\langle H, ST, \mathbf{Assign}[f](o: \sigma, o': \sigma'), FS \rangle \rightarrow \langle H, ST, E, FS \rangle}
\end{array}$$

These rules capture the essence of the design for supporting dynamic types in $\mathbf{C}^\#4.0$. Consider the rule [E-DMethInv]. The $\mathbf{C}_{\text{CLR}}^\#$ dynamic expression $\mathbf{MInvoke}[m](o: \sigma, r: \sigma')$ arose from the compilation of a method invocation in $\mathbf{FC}_4^\#$ where one of the components synthesized the type `dynamic`. Thus at this stage we use the *compile-time* implicit conversion relation to re-type check the method invocation, except now on the runtime values. However, our use of payload expressions means that we only use the runtime types of those expressions whose compile-time type was `dynamic` (for the others we use their compile-time type). The result of the implicit conversion of the method invocation expression yields a new $\mathbf{C}_{\text{CLR}}^\#$ expression, E , which is the result of the transition.⁹

Formal properties It is possible to show type soundness for $\mathbf{C}_{\text{CLR}}^\#$ using the familiar technique of proving preservation and progress properties. Naturally, we need to extend the notions of typing to configurations; again the details follow those in earlier work [7]. The key property is the following.

Theorem 2 (Type preservation for $\mathbf{C}_{\text{CLR}}^\#$).
If $\triangleright \langle H_1, ST_1, F_1, FS_1 \rangle \leq \tau_1$ and $\langle H_1, ST_1, F_1, FS_1 \rangle \rightarrow \langle H_2, ST_2, F_2, FS_2 \rangle$ then $\triangleright \langle H_2, ST_2, F_2, FS_2 \rangle \leq \tau_1$.

Proof. Most of the details are routine [7]. The new cases of interest deal with the $\mathbf{C}_{\text{CLR}}^\#$ dynamic expressions. Consider the transition step [E-DMethInv], for example. According to the typing rule [CLR-S-DMethInv] the dynamic method invocation expression must be of type `object`. But, by application of Theorem 1, we have that E is also of type `object`, and so this case is done. All the other cases of $\mathbf{C}_{\text{CLR}}^\#$ dynamic expressions are similar.

8 Comparison with existing work

There has been considerable work in the area of adding some form of dynamic typing to a statically typed language. Early work by Abadi et al. [1] proposed adding a new

⁹ If the implicit conversion fails then the actual runtime system would throw an exception, although in our formalization we class this as a known stuck state of the transition system.

Dynamic type along with a constructor `dynamic` that packages a value (along with its type) into a value of type `Dynamic`, and a `typecase` construct for inspecting the runtime type tag of a dynamic value. This proposal seems overly explicit in practice where, for example, migrating code between dynamic and static checking would force wholesale changes in existing type signatures. Indeed, much of the value of the proposal described in this paper is its use of *implicit* conversions to manage the migration between dynamically and statically typed code.

Much closer to our work is the proposal for gradual typing by Siek and Taha for an object-based language [21], following on from their earlier work on functional languages [20]. They propose a new type, written `?` (equivalent to our `dynamic` type) and replace the familiar notion of type equality with a non-transitive ‘consistency’ relation. A key difference is that they consider a structural object type system whereas we work within a nominal system. It would be interesting to see if the techniques described in this paper could apply to a structural type system of the sort considered by Siek and Taha.

Independent to our work, Wrigstad et al. [25] have also proposed adding dynamic types to a Java-like, statically-typed language, Thorn. There appears to be much in common although they offer additionally an intermediate step between static and dynamic types. A variable typed with a so-called *like type*, written `like C`, is checked statically against the type `C`, but also all values bound to the variable are dynamically checked.

There is a good deal of work in the other direction, i.e. adding static typing capabilities to a dynamic language [9, 2, 13, 24]. Such systems aim to reduce the amount of runtime type checking of dynamically typed code with the hope of improving the overall performance. It would be interesting to consider whether such techniques could be applied to the dynamically typed portions of `C#4.0` code.

Another interesting area is the issue of blame tracking when interoperating between statically and dynamically typed code [23, 10]. We intend to explore to what extent existing work can apply to the support of dynamic types in `C#`.

9 Conclusions

In this paper we describe new features in `C#4.0` that enable the safe co-existence of statically and dynamically typed code. This allows more natural interoperation with dynamic languages, and offers a form of gradual typing. Thus the `C#4.0` developer will be able to write portions of code as if using a dynamic language and rely on the DLR to provide efficient implementation. Furthermore, this “dynamic” code can then be translated into statically typed code later; all within the same language.

This gradual migration of code from dynamically to statically typed has been the subject of considerable academic study. In this paper we have formalized the support of dynamic types in `C#`, and also shown how it can be achieved with a relatively straightforward extension to the existing type system.

References

1. M. Abadi, L. Cardelli, B.C. Pierce, and G.D. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of POPL*, 1989.

2. A. Aiken, E.L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proceedings of POPL*, 1994.
3. C. Anderson and S. Drossopoulou. BabyJ: From object based to class based programming via types. In *Proceedings of WOOD*, 2003.
4. G.M. Bierman. Formalizing and extending C^\sharp type inference. In *Proceedings of FOOL*, 2007.
5. G.M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to C^\sharp . In *Proceedings of OOPSLA*, 2007.
6. G.M. Bierman and M.J. Parkinson. Effects and effect inference for a core Java calculus. In *Proceedings of WOOD*, 2003.
7. 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.
8. V. Breazu-Tannen, T. Coquand, C.A. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and computation*, 93(1):172–221, 1991.
9. R. Cartwright and M. Fagan. Soft typing. In *Proceedings of PLDI*, 1991.
10. R. Findler and P. Wadler. Well-typed programs can't be blamed. In *Proceedings of ESOP*, 2009.
11. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, 1998.
12. A. Hejlsberg, M. Torgersen, S. Wiltamuth, and P. Golde. *The C[‡] Programming Language*. Addison-Wesley, third edition, 2009.
13. F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.
14. 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.
15. A. Kennedy and D. Syme. Transposing F to C^\sharp . *Concurrency and Computation*, 16(7), 2004.
16. K. Knowles, A. Tomb, J. Gronski, S.N. Freund, and C. Flanagan. SAGE: Unified hybrid checking for first-class types, general refinement types and *Dynamic*. Technical report, UCSC, 2007.
17. E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA Workshop on Revival of Dynamic Languages*, 2004.
18. B.C. Pierce. *Types and programming languages*. MIT Press, 2002.
19. B.C. Pierce and D.N. Turner. Local type inference. In *Proceedings of POPL*, 1998.
20. J. Siek and W. Taha. Gradual typing for functional languages. In *Proceedings of Scheme and Functional Programming Workshop*, 2006.
21. J. Siek and W. Taha. Gradual typing for objects. In *Proceedings of ECOOP*, 2007.
22. S. Thatte. Quasi-static typing. In *Proceedings of POPL*, 1990.
23. S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: From scripts to programs. In *Proceedings of DSL*, 2006.
24. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Proceedings of POPL*, 2007.
25. T. Wrigstad, F. Zappa Nardelli, S. Lebesne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proceedings of POPL*, 2010.
26. D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET common language runtime. In *Proceedings of POPL*, 2004.