Formalizing and extending C♯ type inference
(Work in progress)

Gavin Bierman
Microsoft Research Cambridge, UK
gmb@microsoft.com

Abstract

The current release of C♯ (version 2.0) introduces a number of new features intended to increase the expressivity of the language. The most significant is the addition of *generics*; classes, interfaces, delegates and methods can all be parameterized on types.

To make using generic methods easier, C♯ allows the programmer to drop the type arguments in method invocations, and the compiler implements "*type inference*" to reconstruct the arguments. Unfortunately this part of the published language specification is a little terse, and hence this feature can often behave in surprising ways for the programmer. Moreover, this process is quite different from the better known one implemented in Java 5.0. In this paper we attempt a formal reconstruction of the type inference process as it is currently implemented in C♯2.0. We also consider a number of proposed extensions to support new language features that will appear in C♯3.0.

1. Introduction

C♯2.0 introduced a number of new features to the language, most significant of which was the addition of generics. C♯2.0 permits classes, structs, interfaces, delegates and methods to be parameterized on types. Similar extensions were also added to version 5.0 of Java. The advantages of parametric polymorphism are well-known to users of functional programming languages such as Haskell and ML, and also to users of Eiffel and Ada. In the context of object-oriented languages such as C♯and Java, generics provide stronger compile-time type guarantees, require fewer explicit conversions between types and reduce the need for boxing operations and run-time type tests.

There has been considerable theoretical work on adding generics to object-oriented languages, such as PolyJ [11], NextGen [4], Pizza [13] and GJ [3]. One aspect of generics which has received less attention is the inference of type arguments when invoking a generic method. Both C♯ and Java support methods that are parameterized on types, which can appear in classes which may themselves be generic or non-generic. Consider the following code in C♯, taken from the language specification [7, §20.6.4].

```
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>
```

The main technical problem addressed in this paper is how compilers actually perform this type inference process. The processes used in C♯ and Java are actually quite different; this paper concentrates on the lesser-known process employed by C♯.

The type inference process for Java was considered in the context of GJ by Bracha et al. [3, §5]. The first version of GJ released for JSR14 was shown to be unsound by Jeffrey [9]. An alternative system was proposed by Odersky [12] and was subsequently included in the javac compiler. The latest version of the Java language specification gives a sixteen page formal description of the type inference process using a constraint system [6, §15.12.2.7-8].[1]

Interestingly, the GJ design appears to have been influenced by an assumption that there would be no way to explicitly specify a type argument list. Hence, the GJ inference process is intended to be as complete as possible. However, Java 5.0 *does* provide explicit syntax for supplying type argument lists.

Relatively little attention has been paid to the type inference process that appears in C♯2.0. In contrast to Java, the C♯ designers appear to have been focused on simplicity as opposed to completeness—the type inference process is described in a page and a half in the current language specification [§20.6.4].

However, a brief, informal (and, in this case, a particularly terse) specification is no substitute for formal rigour. In this paper we give a precise, formal description of both the type system of C♯2.0 and of the type inference process itself. One contribution of this paper is to show how a bidirectional type system[2] in the sense of Pierce and Turner [15] can be used and extended to describe both the type system and the type inference process. One advantage of a bidirectional type system is that it directly defines an implementation. The locality property of bidirectional systems such as the one presented here means that the implementation is quite simple. It is important to note that the Java inference process is *non*-local.

The main advantage of the formal description presented in this paper is that it enables the exploration of various extensions to the C♯ 2.0 type inference process. For example, the inclusion of λ-expressions in C♯ 3.0 will require an extension to type inference.

[1] It is interesting to note that, as formalized, solving this constraint system can lead to the generation of infinite types [6, p.465].

[2] This is sometimes referred to as "local type inference", although we prefer to use the term "bidirectional" to distinguish between the processes of type checking, type synthesis and inference of type arguments.

We show in this paper a number of possible extensions to handle λ-expressions.

This paper is organized as follows. In §2 we define a small, featherweight subset of C♯2.0, FC$_2^\sharp$, that we use in our formalization. In §3 we define the type system for FC$_2^\sharp$ as a bidirectional type system. In §3.3 we show how to extend this technique to capture type inference. In §4 we consider a number of extensions to the type inference process; relaxing the consistency condition (§4.1), handling λ-expressions (§4.2) and incorporating return type information (§4.3). We conclude in §5 and suggest some future work.

2. Featherweight C♯ 2.0

In this section we define a core fragment of C♯2.0 called Featherweight C♯2.0, or FC$_2^\sharp$ for short, that will be used in the formal specification of the type system and type inference process. This core fragment, whilst lightweight, has a similar computational feel to the full C♯ language and contains all its essential features, such as generic classes, delegates, state and mutable objects. It is similar in essence to core subsets of Java such as MJ [2] and ClassicJava [5]. It is important to note that FC$_2^\sharp$ is a completely valid subset of C♯ in that every FC$_2^\sharp$ program is literally an executable C♯ program.

2.1 FC$_2^\sharp$ syntax

A FC$_2^\sharp$ program consists of a sequence of zero or more delegate declarations, followed by a sequence of zero or more class declarations. Given an FC$_2^\sharp$ program we assume that there is a unique designated method within the standard class declarations that serves as the entry point (the `main` method). The grammar for FC$_2^\sharp$ programs is as follows.

FC$_2^\sharp$ programs:

$p ::= \overline{dd}\ \overline{cd}$	Program
$dd ::=$	Delegate Declaration
`public delegate` $\sigma\ D$`<`$\overline{X}$`>(`$\overline{\tau}\ \overline{x}$`)`	
$cd ::=$	Class Declaration
`public class` C`<`$\overline{X}$`>:`ι `{`$\overline{fd}\ \overline{md}$`}`	
$fd ::=$ `public` $\tau\ f$`;`	Field declaration
$md ::=$	Method Declaration
`public (virtual|override)` $\sigma\ m$`<`$\overline{X}$`>(`$\overline{\tau}\ \overline{x}$`){`$\overline{s}$`}`	

A class declaration consists of zero or more field declarations followed by zero or more method declarations. Methods must be defined either `virtual` or `override` and, for simplicity, we require all fields and methods be `public`. For conciseness, we do not model `static` 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.

The grammar for FC$_2^\sharp$ types is as follows.

Types:

$\sigma ::=$	Return type
τ	Denotable type
`void`	Void
$\tau ::=$	Denotable types
γ	Value type
ρ	Reference type
X	Type parameter
$\gamma ::=$	Value Type
`bool`	Boolean
`int`	Integer
$\rho ::=$	Reference Type
ι	Constructed type
D`<`$\overline{\tau}$`>`	Delegate type
$\iota ::= C$`<`$\overline{\tau}$`>`	Constructed Type

The two main categories of FC$_2^\sharp$ types are 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 the nullable types in our core fragment.

FC$_2^\sharp$ reference types include class types and delegate types. To simplify the presentation we don't consider arrays, and we write D to range over delegate types and C to range over class types. Following GJ [8] we permit the shorthand `C` for `C<>`. For simplicity we do not model constraints on generic parameters as supported in C♯2.0—they play no part in the type inference process and so are peripheral to the main concerns of this paper. We also assume a predefined superclass `object`.

Following FJ [8] we adopt an overloaded 'overbar' notation; for example, $\overline{\tau}\ \overline{f}$ is a shorthand for a possibly empty sequence $\tau_1\ f_1, \ldots, \tau_n\ f_n$.

FC$_2^\sharp$ expressions, as for C♯, are split into ordinary expressions and statement expressions. Statement expressions are expressions that can be used as statements. The grammars for both forms are as follows.

Expressions:

$e ::=$	Expression
b	Boolean
i	Integer
$e \oplus e$	Built-in operator
x	Variable
`null`	Null
$(\tau)e$	Cast
`delegate` $(\overline{\tau}\ \overline{x})${$\overline{s}$}	Anonymous method expression
$e.f$	Field access
se	Statement expression
$se ::=$	Statement expression
me`<`$\overline{\tau}$`>` ae	Explicit invocation
$me\ ae$	Implicit invocation
`new` $\iota\ ae$	Object/collection creation
x `=` e	Variable assignment
$me ::=$	Member access expression
$e.m$	Method access
$ae ::= (\overline{e})$	Argument expression

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 can not be used as a parameter of a method or delegate declaration.

FC$_2^\sharp$ statements are fairly standard and as follows.

Statements:

$s ::=$	Statement

;	Skip
se;	Expression statement
if (e) s **else** s	Conditional statement
$\tau\ x$ = e;	Explicitly-typed declaration
x = e;	Variable assignment
$e.f$ = e;	Field assignment
return e;	Return statement
return;	Empty return
{$\overline{s}$}	Block

In what follows we assume that $\mathsf{FC}_2^\sharp$ 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.

3. $\mathsf{FC}_2^\sharp$ bidirectional type system

In this section, we formalize the process of both typing $\mathsf{FC}_2^\sharp$ programs and also the inference of generic method type arguments. The main technical tool we use is a bidirectional type system [15, 14]. Such systems explicitly distinguish between type *checking* and type *synthesis*. We will see that this technique, whilst originating from studies of System $\mathrm{F}_{\leq}$, is actually rather well suited to commercial class-based languages.

First, we need to introduce some additional notation. We write μ to range over *method types*, which are written $\forall\overline{X}.(\overline{\tau}) \to \sigma$. A *method group type*, which is just a sequence of method types, is written using the shorthand $\bigwedge_{i=1}^{n} \mu_i$, by which we mean $\mu_1 \wedge \mu_2 \wedge \cdots \wedge \mu_n$ (we will often drop the bounds when not important).

3.1 Subtyping

The subtyping rules for $\mathsf{FC}_2^\sharp$ are standard and are given below.

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

$$\frac{\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}: \iota}{C\texttt{<}\overline{\tau}\texttt{>} <: \iota[\overline{X} := \overline{\tau}]}\ [\mathsf{ST\text{-}Sub}] \qquad \frac{}{\tau <: \texttt{object}}\ [\mathsf{ST\text{-}Object}]$$

Rules [ST-Refl] and [ST-Trans] ensure that the subtyping relation is reflexive and transitive. The [ST-Object] rules ensures that **object** is the root of type hierarchy (values can be boxed as objects).

3.2 Bidirectional typing judgements for $\mathsf{FC}_2^\sharp$

Excluding type inference, there are two ways of typing a term: *checking* and *synthesizing*. Type checking is the process of determining whether a (given) term can be assigned a particular (given) type. Type synthesis is the process of automatically determining, or inferring, a type given a term. We capture these two processes using two relations which we outline below.

3.2.1 An introduction to type checking

The judgement form for type *checking* an expression is written $\Gamma \vdash e_1 : \tau \hookrightarrow e_{11}$ and should be read "given typing assumptions Γ, the expression e_1 can be type checked at type τ, which yields an expression e_{11}." (For now, the reader can ignore the yielded expression but the intention is that expression e_{11} results from inserting the inferred type arguments into the original expression e_1.)

Let us consider some rules for forming valid type checking judgements. The simplest rule is where the expression is an identifier:

$$\frac{\tau_1 <: \tau_2}{\Gamma, x : \tau_1 \vdash x : \tau_2 \hookrightarrow x}$$

To check whether an identifier x can be assigned a type τ_2 we first look in the environment to see what type we know for x. Assuming this type is τ_1, then the check succeeds if there is a conversion from τ_1 to τ_2. Here's a particular instance of the rule:

$$\frac{\texttt{int} <: \texttt{object}}{\Gamma, x : \texttt{int} \vdash x : \texttt{object} \hookrightarrow x}$$

In other words, if we know that x has type **int** we can conclude that it can be assigned type **object** as there is a conversion from the former to the later.

A particularly interesting type checking rule is for anonymous method expressions. The C$^\sharp$ language specification states that an anonymous method expression is "... a value with no type" [§21.3]. This is a little misleading; it *can* be assigned a type (in many cases, several!).[3] The rule for type checking an anonymous method expression is as follows.

$$\frac{dtype(D)(\overline{\tau}) = \overline{\tau_a} \to \sigma \qquad \Gamma, \overline{x} : \overline{\tau_a} \vdash \overline{s_1} : \sigma \hookrightarrow \overline{s_{11}}}{\Gamma \vdash \texttt{delegate(}\overline{\tau_a}\ \overline{x}\texttt{)\{}\overline{s_1}\texttt{\}} : D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \texttt{delegate(}\overline{\tau_a}\ \overline{x}\texttt{)\{}\overline{s_{11}}\texttt{\}}}$$

There are a number of points of interest in this rule. First, notice that the *only* type we can check an anonymous method expression against is a delegate type. Even the following code fails:

```
object zz = delegate (int z){return z;};
```

The rest of the rule has a quite straightforward reading. It makes use of an auxiliary function $dtype$, which is a map from delegate names to their associated type (which is a method type). In the rule, we use this function to determine the type of the delegate D, which is, say, $\forall\overline{X}.\overline{\tau_1} \to \sigma'$. We then substitute the given type arguments $\overline{\tau}$ for the type parameters $\overline{X}$ to get a type $\overline{\tau_a} \to \sigma$ (in the rule we use application as a shorthand). We then type check the statement sequence using the types $\overline{\tau_a}$ for the parameters $\overline{x}$ (there is an implicit assumption here that the number of parameters matches the number of types, i.e. $|\overline{\tau_a}| = |\overline{x}|$) and the return type σ.

3.2.2 An introduction to type synthesis

The judgement form for type *synthesis* of expressions is written $\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau$ and should be read "given typing assumptions Γ, the expression e_1 can be inferred to have type τ, yielding an expression e_{11}". (Again, for now, the reader can ignore the yielded expression.)

As before let us consider some of the rules for forming type synthesis judgements. The simplest rule, again, is where an expression is an identifier and is as follows.

$$\frac{}{\Gamma, x : \tau \vdash x \hookrightarrow x : \tau}$$

Thus to synthesize a type for the identifier x, we simply return the type assumption for x.

It is important to note that there are no rules for **null** expressions and anonymous method expressions. We can *not* synthesize (denotable) types for these expression forms.

[3] What the standard is perhaps attempting to capture is that there is no way to *synthesize* a type for an anonymous method expression.

The rules for forming type checking and type synthesis judgements are inter-defined. For example, the type checking rule for a field access expression is as follows.

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_1 \quad \mathit{ftype}(\tau_1, f) = \tau_2 \quad \tau_2 <: \tau_3}{\Gamma \vdash e_1 \,.\, f : \tau_3 \hookrightarrow e_{11} \,.\, f}$$

Here we are trying to type check an expression $e_1 \,.\, f$ at type τ_3. We first *synthesize* a type for the expression e_1, say τ_1. We then use an auxiliary function *ftype* to determine the declared type for the field f for the type τ_1, say τ_2. We conclude that the expression $e_1 \,.\, f$ can be assigned the type τ_3 if there is a conversion from τ_2 to τ_3.

3.2.3 Further details

The complete collection of rules for type checking and synthesis of expressions is given in Figures 1 and 2, respectively. For the most part, these rules are pretty straightforward, the main exception being the rules for typing method invocation expressions. These are explained in detail in §3.2.4.

We comment on a few of the rules in the figures. Rule [TC-IInv] highlights the role of the yielded expression. Here the yielded expression has the inferred type arguments inserted into the method invocation. In rule [TC-ObjCreation] we use some shorthand and write $\mathit{mtype}(\iota_1, \texttt{new}\ )$ to mean the method group type associated with the constructors for type ι_1.

The judgement form for type checking a statement is written $\Gamma \vdash s_1 : \sigma \hookrightarrow s_{11}$. The type σ is the expected return type of the statement. The key rule is then for `return` statements, and is as follows

$$\frac{\Gamma \vdash e_1 : \tau \hookrightarrow e_{11}}{\Gamma \vdash \texttt{return}\ e_1\texttt{;} : \tau \hookrightarrow \texttt{return}\ e_{11}\texttt{;}}\ \text{[TC-ReturnExp]}$$

We can also generalize this judgement to one that type checks a statement sequence in the obvious way. The rules for type checking statements and statement sequences are given in Figure 3.

3.2.4 Further details: method invocations

In this section we give the details of typing method invocation expressions where the type arguments have been explicitly given. We will build on these rules to define a type inference process (for method invocations where the type arguments have been omitted) in §3.3.

For brevity, we will describe type checking only—the details for type synthesis are almost identical. We will do this in a number of stages. First, for convenience, we introduce a new judgement form for *synthesizing* a type for a member access expression. In $\mathsf{FC}^\sharp_2$ a member access expression is always a invocation of a method, but in the full C$^\sharp$ language this is not always true as a field could have a generic delegate type, and so we could legitimately write, e.g. `e.f<int>(42)`. We have excluded this possibility in $\mathsf{FC}^\sharp_2$ but it would be straightforward to add. This new judgement form is written $\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i$, where me_1 is the member access expression and $\bigwedge \mu_i$ is the synthesized method group type. The single rule for forming such judgements is as follows.

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_1 \quad \mathit{mtype}(\tau_1, m) = \bigwedge \mu_i}{\Gamma \vdash e_1 \,.\, m \Uparrow e_{11} \,.\, m : \bigwedge \mu_i}$$

This rule is pretty straightforward: we first synthesize a type for the receiver, e_1, and then look up the declarations for method m for this synthesized receiver type. Again we use an auxiliary function mtype, which given a type τ and a method name m, returns the method group type. This method group type is then the synthesized type for the member access expression.

So far our rule for type checking a method invocation would be something like the following:

$$\frac{\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \quad ???}{\Gamma \vdash me_1\texttt{<}\overline{\tau}\texttt{>}\ ae : \tau_3 \hookrightarrow ???}$$

In other words we have generated a method group type for the invocation expression. Next we instantiate the method type parameters of the method types contained in the method group type with the explicit type arguments. We write this as $\bigwedge \mu_i(\overline{\tau})$. This operation returns a method group. Notice that this operation needs to first check the arity of the method types before instantiating the type parameters. For example consider the following method group type G:

$$\begin{array}{ll} \forall X.(X, \texttt{int}) \to \texttt{double} & \wedge \\ \forall X, Y.(X, Y) \to \texttt{double} & \wedge \\ \forall X, Y.(X, \texttt{object}) \to \texttt{double} & \wedge \\ \forall X, Y.(X, Y\,\texttt{[]}) \to \texttt{double} & \wedge \\ \forall X, Y, Z.(X, Y, Z) \to \texttt{double} & \end{array}$$

Hence $G(\texttt{object}) = (\texttt{object}, \texttt{int}) \to \texttt{double}$, whereas $G(\texttt{int}, \texttt{string}, \texttt{object}) = (\texttt{int}, \texttt{string}, \texttt{object}) \to \texttt{double}$.

Next, we need to check the argument expression ae against the expected argument types in the method group. We introduce a new type checking judgement form, written $\Gamma \vdash ae : \bigwedge \mu_i \hookrightarrow \bigwedge \mu_j$. This means that the argument expression, ae, is systematically checked against the argument types of each method type in the method group. The result is the method group containing the method types that match.

The single rule for this judgement form is given in Figure 1 and is called [TC-MethodGroupType]. This is slightly awkward to formalize as what we need to capture is the set of successful method types from a method group type against which the argument expression can be type checked. We use a success set, S, which is used to index the *maximal* subset of successful method types.

Let us consider an example. We assume a class `Foo` which has static methods `m` with the method group type G given earlier. We shall consider type checking the following expression:

```
Foo.m<object,int>(null,42)
```

First, we instantiate the type parameters of the method group type with the given type arguments, `object` and `int`. Hence $G(\texttt{object}, \texttt{int})$ returns the method group type $M = (\texttt{object}, \texttt{int}) \to \texttt{double} \wedge (\texttt{object}, \texttt{object}) \to \texttt{double} \wedge (\texttt{object}, \texttt{int[]}) \to \texttt{double}$.

The argument expression `(null,42)` matches two of these method types, so we would expect to form the following type checking judgement.

$$\Gamma \vdash (\texttt{null},42) : M \rightsquigarrow (\texttt{object}, \texttt{int}) \to \texttt{double} \wedge (\texttt{object}, \texttt{object}) \to \texttt{double}$$

At this point C$^\sharp$ uses overloading resolution to determine which method would be called (if there is a single valid one). In this paper we do not formalize overloading resolution—it is defined precisely in the C$^\sharp$ language specification [§7.4.2]. For simplicity we simply postulate a (partial) function $\mathit{OloadRes}$ that returns the argument expression after being type checked against the chosen method and the return type of the chosen method.

To conclude, the type checking rule for a method invocation expression is as follows.

$\boxed{\Gamma \vdash e_1 : \tau \hookrightarrow e_{11}}$

$$\frac{\texttt{bool} <: \tau}{\Gamma \vdash b : \tau \hookrightarrow b}\ [\text{TC-Bool}] \qquad \frac{\texttt{int} <: \tau}{\Gamma \vdash i : \tau \hookrightarrow i}\ [\text{TC-Int}] \qquad \frac{\tau_1 <: \tau_2}{\Gamma, x : \tau_1 \vdash x : \tau_2 \hookrightarrow x}\ [\text{TC-Var}] \qquad \frac{\tau \text{ is ref}}{\Gamma \vdash \texttt{null} : \tau \hookrightarrow \texttt{null}}\ [\text{TC-Null}]$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_{11} \quad \tau_{11} <: \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash (\tau_1) e_1 : \tau_2 \hookrightarrow (\tau_1) e_{11}}\ [\text{TC-UpCast}] \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_{11} \quad \neg(\tau_{11} <: \tau_1) \quad \tau_1 <: \tau_{11} \quad \tau_1 <: \tau_2}{\Gamma \vdash (\tau_1) e_1 : \tau_2 \hookrightarrow (\tau_1) e_{11}}\ [\text{TC-DownCast}]$$

$$\frac{\mathit{dtype}(D)(\overline{\tau}) = \overline{\tau_a} \to \sigma \quad \Gamma, \overline{x} : \overline{\tau_a} \vdash \overline{s_1} : \sigma \hookrightarrow \overline{s_{11}}}{\Gamma \vdash \texttt{delegate}(\overline{\tau_a}\ \overline{x})\{\overline{s_1}\} : D\langle\overline{\tau}\rangle \hookrightarrow \texttt{delegate}(\overline{\tau_a}\ \overline{x})\{\overline{s_{11}}\}}\ [\text{TC-AnonMethExp}]$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_1 \quad \mathit{ftype}(\tau_1, f) = \tau_2 \quad \tau_2 <: \tau_3}{\Gamma \vdash e_1 . f : \tau_3 \hookrightarrow e_{11} . f}\ [\text{TC-FieldAccess}]$$

$\boxed{\Gamma \vdash se_1 : \tau \hookrightarrow se_{11}}$

$$\frac{\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \quad \Gamma \vdash ae_1 : \bigwedge \mu_i(\overline{\tau}) \hookrightarrow \bigwedge \mu_j \quad \mathit{OloadRes}(\bigwedge \mu_j, ae_1) = (ae_{11}, \sigma) \quad \sigma <: \tau_3}{\Gamma \vdash me_1\langle\overline{\tau}\rangle\ ae_1 : \tau_3 \hookrightarrow me_{11}\langle\overline{\tau}\rangle\ ae_{11}}\ [\text{TC-EInv}]$$

$$\frac{\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \quad \Gamma \vdash ae_1 \sim \bigwedge \mu_i \hookrightarrow \langle \bigwedge \mu_j, \overline{\mathcal{S}} \rangle \quad \mathit{OloadRes}(\bigwedge \mu_j, \overline{\mathcal{S}}, ae_1) = (ae_{11}, \overline{\tau}, \sigma) \quad \sigma <: \tau_3}{\Gamma \vdash me_1\ ae_1 : \tau_3 \hookrightarrow me_{11}\langle\overline{\tau}\rangle\ ae'}\ [\text{TC-IInv}]$$

$$\frac{\mathit{mtype}(\iota_1, \texttt{new}) = \bigwedge \mu_i \quad \Gamma \vdash ae_1 : \bigwedge \mu_i \hookrightarrow \bigwedge \mu_j \quad \mathit{OloadRes}(\bigwedge \mu_j, ae_1) = (ae_{11}, \texttt{void}) \quad \iota_1 <: \tau_1}{\Gamma \vdash \texttt{new}\ \iota_1\ ae_1 : \tau_1 \hookrightarrow \texttt{new}\ \iota_1\ ae_{11}}\ [\text{TC-ObjCreation}]$$

$$\frac{\Gamma, x_0 : \tau_0 \vdash e_1 : \tau_1 \hookrightarrow e_{11} \quad \tau_0 <: \tau_1}{\Gamma, x_0 : \tau_0 \vdash x_0 = e_1 : \tau_1 \hookrightarrow x_0 = e_{11}}\ [\text{TC-VarAssign}]$$

$\boxed{\Gamma \vdash ae_1 : \mu}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_{11} \quad \cdots \quad \Gamma \vdash e_n : \tau_n \hookrightarrow e_{n1}}{\Gamma \vdash (e_1, \ldots, e_n) : (\tau_1, \ldots, \tau_n) \to \sigma}\ [\text{TC-MethodType}]$$

$\boxed{\Gamma \vdash ae_1 : \bigwedge \mu_i \hookrightarrow \bigwedge \mu_j}$

$$\frac{\Gamma \vdash ae_1 : \mu_{S_1} \quad \cdots \quad \Gamma \vdash ae_1 : \mu_{S_k} \quad |S| = k \geq 1 \quad S \subseteq \{1, \ldots, n\}}{\Gamma \vdash ae_1 : \mu_1 \wedge \cdots \wedge \mu_n \hookrightarrow \mu_{S_1} \wedge \cdots \wedge \mu_{S_k}}\ [\text{TC-MethodGroupType}]$$

Figure 1. Type checking expressions

$$\frac{\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \qquad \mathit{OloadRes}(\bigwedge \mu_j, ae_1) = (ae_{11}, \sigma)}{\Gamma \vdash ae_1 : \bigwedge \mu_i(\overline{\tau}) \hookrightarrow \bigwedge \mu_j \quad \sigma <: \tau_3}$$
$$\Gamma \vdash me_1\langle\overline{\tau}\rangle\ ae_1 : \tau_3 \hookrightarrow me_{11}\langle\overline{\tau}\rangle\ ae_{11}$$

The rule for type synthesis of an explicit invocation expression is almost identical and as follows.

$$\frac{\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \qquad \Gamma \vdash ae_1 : \bigwedge \mu_i(\overline{\tau}) \hookrightarrow \bigwedge \mu_j \quad \mathit{OloadRes}(\bigwedge \mu_j, ae_1) = (ae_{11}, \tau_3)}{\Gamma \vdash me_1\langle\overline{\tau}\rangle\ ae_1 \hookrightarrow me_{11}\langle\overline{\tau}\rangle\ ae_{11} : \tau_3}$$

3.3 Type inference

Now we can build upon type checking and synthesis to define the type inference process. The key point is that when checking the argument expression against a method type, we need to also generate a *substitution*.[4] We refer to this process as type *matching*.

[4] A substitution is referred to as "type inferences" in the language specification [§20.6.4].

A first attempt at defining a type matching rule might be a declarative rule such as the following.

$$\frac{\mathrm{dom}(\mathcal{S}) = \overline{X} \quad \Gamma \vdash e_1 : \mathcal{S}(\tau_1) \hookrightarrow e_{11} \ \cdots\ \Gamma \vdash e_n : \mathcal{S}(\tau_n) \hookrightarrow e_{n1}}{\Gamma \vdash (e_1, \ldots, e_n) \sim \forall \overline{X}.(\tau_1, \ldots, \tau_n) \to \sigma \hookrightarrow \mathcal{S}}$$

This rule states that to match the argument expression against the argument types of the method type, we 'simply' find a substitution for all the method type parameters, $\overline{X}$, such that each individual component expression, e_i, can be type checked against the resulting argument type, $\mathcal{S}(\tau_i)$.

Of course, matters are not quite so simple. The problem is finding the substitution. Moreover, there may be several such substitutions. Are they all valid? Which one should be chosen? Is there always a best one to chose? The process of determining this substitution is the essence of the type inference process.

As mentioned earlier, C♯2.0 adopts a fairly simple but elegant solution, albeit one that is not quite so well known. In the rest of this section we formalize precisely the C♯2.0 approach, but along the way we point out some of the weaknesses. In later sections we will propose extensions to address some of these weaknesses.

$\boxed{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_1}$

$$\frac{}{\Gamma \vdash b \hookrightarrow b : \texttt{bool}}\ [\text{TS-Bool}] \qquad \frac{}{\Gamma \vdash i \hookrightarrow i : \texttt{int}}\ [\text{TS-Int}] \qquad \frac{}{\Gamma, x : \tau \vdash x \hookrightarrow x : \tau}\ [\text{TS-Var}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_{11}}{\Gamma \vdash (\tau_1) e_1 \hookrightarrow e_{11} : \tau_1}\ [\text{TS-Cast}] \qquad \frac{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_1 \qquad \mathit{ftype}(\tau_1, f) = \tau_2}{\Gamma \vdash e_1 . f \hookrightarrow e_{11} . f : \tau_2}\ [\text{TS-FieldAccess}]$$

$\boxed{\Gamma \vdash se_1 \hookrightarrow se_2 : \tau_1}$

$$\frac{\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \quad \Gamma \vdash ae_1 : \bigwedge \mu_i(\overline{\tau}) \hookrightarrow \bigwedge \mu_j \quad \mathit{OloadRes}(\bigwedge \mu_j, ae_1) = (ae_{11}, \tau_3)}{\Gamma \vdash me_1 \texttt{<}\overline{\tau}\texttt{>}\ ae_1 \hookrightarrow me_{11} \texttt{<}\overline{\tau}\texttt{>}\ ae_{11} : \tau_3}\ [\text{TS-EInv}]$$

$$\frac{\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \quad \Gamma \vdash ae_1 \sim \bigwedge \mu_i \hookrightarrow \langle \bigwedge \mu_j, \overline{\mathcal{S}} \rangle \quad \mathit{OloadRes}(\bigwedge \mu_j, \overline{\mathcal{S}}, ae_1) = (ae_{11}, \overline{\tau}, \tau_3)}{\Gamma \vdash me_1\ ae_1 \hookrightarrow me_{11} \texttt{<}\overline{\tau}\texttt{>}\ ae_{11} : \tau_3}\ [\text{TS-IInv}]$$

$$\frac{\mathit{mtype}(\iota_1, \texttt{new}) = \bigwedge \mu_i \quad \Gamma \vdash ae_1 : \bigwedge \mu_i \hookrightarrow \bigwedge \mu_j \quad \mathit{OloadRes}(\bigwedge \mu_j, ae_1) = (ae_{11}, \texttt{void})}{\Gamma \vdash \texttt{new}\ \iota_1\ ae_1 \hookrightarrow \texttt{new}\ \iota_1\ ae_{11} : \iota_1}\ [\text{TS-ObjCreation}]$$

$$\frac{\Gamma, x_0 : \tau_0 \vdash e_1 : \tau_0 \hookrightarrow e_{11}}{\Gamma, x_0 : \tau_0 \vdash x_0 = e_1 \hookrightarrow x_0 = e_{11} : \tau_0}\ [\text{TS-VarAssign}]$$

Figure 2. Type synthesis for expressions

$\boxed{\Gamma \vdash s_1 : \sigma \hookrightarrow s_{11}}$

$$\frac{}{\Gamma \vdash ; : \tau \hookrightarrow ;}\ [\text{TC-Skip}] \qquad \frac{\Gamma \vdash se_1 \hookrightarrow se_{11} : \tau_1}{\Gamma \vdash se_1 ; : \sigma \hookrightarrow se_{11} ;}\ [\text{TC-ExpStatement}] \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \hookrightarrow e_{11} \quad \Gamma \vdash s_1 : \sigma \hookrightarrow s_{11} \quad \Gamma \vdash s_2 : \sigma \hookrightarrow s_{21}}{\Gamma \vdash \texttt{if}\ (e_1)\ s_1\ \texttt{else}\ s_2 : \sigma \hookrightarrow \texttt{if}\ (e_{11})\ s_{11}\ \texttt{else}\ s_{21}}\ [\text{TC-Cond}]$$

$$\frac{\Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_1 \quad \mathit{ftype}(\tau_1, f) = \tau_2 \quad \Gamma \vdash e_2 : \tau_2 \hookrightarrow e_{21}}{\Gamma \vdash e_1 . f = e_2 ; : \sigma \hookrightarrow e_{11} . f = e_{21}}\ [\text{TC-FAss}]$$

$$\frac{}{\Gamma \vdash \texttt{return} ; : \texttt{void} \hookrightarrow \texttt{return} ;}\ [\text{TC-Return}] \qquad \frac{\Gamma \vdash e_1 : \tau \hookrightarrow e_{11}}{\Gamma \vdash \texttt{return}\ e_1 ; : \tau \hookrightarrow \texttt{return}\ e_{11} ;}\ [\text{TC-ReturnExp}]$$

$\boxed{\Gamma \vdash \overline{s_1} : \sigma \hookrightarrow \overline{s_{11}}}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \hookrightarrow e_{11} \quad x \notin \mathit{dom}(\Gamma) \quad \Gamma, x : \tau_1 \vdash \overline{s_1} : \sigma \hookrightarrow \overline{s_{11}}}{\Gamma \vdash \tau_1\ x = e_1 ;\ \overline{s_1} : \sigma \hookrightarrow \tau_1\ x = e_{11} ;\ \overline{s_{11}}}\ [\text{TC-EDecSeq}] \qquad \frac{\Gamma \vdash s_1 : \sigma \hookrightarrow s_{11} \quad \Gamma \vdash \overline{s_2} : \sigma \hookrightarrow \overline{s_{21}}}{\Gamma \vdash s_1\ \overline{s_2} : \sigma \hookrightarrow s_{11}\ \overline{s_{21}}}\ [\text{TC-StSeq}]$$

Figure 3. Type checking of statements

We introduce a notion of *type matching* for $\mathsf{FC}_2^\sharp$; whose judgement form for expressions is written $\Gamma; \Delta \vdash e_1 \sim \tau_1 \hookrightarrow \mathcal{S}$. This is read "given typing assumptions Γ, the expression e_1 type matches a type τ_1 with 'free' method type parameters contained in Δ, yielding a substitution $\mathcal{S}$." A substitution is simply a (partial) function from a method type parameter to a type.

For example, we would expect the following to be a valid type matching judgement.

$$\Gamma; X \vdash \mathsf{42} \sim X \hookrightarrow \{X \mapsto \texttt{int}\}$$

The alert reader will have noticed that this judgement form is algorithmic as opposed to declarative. A declarative version is possible, and will be given in a future version of this paper. We stick with the algorithmic version here as it follows quite closely the actual language specification and so will aid comparison.

The rules for type matching expressions are given in Figure 4. There are three special cases ([TM-Null], [TM-Closed], and [TM-AnonMethExp]) and one general rule ([TM-OpenExp]). The rule [TM-Null] simply checks that the type is a reference type and returns an empty substitution. The rule [TM-Closed] covers the case when type matching an expression against a closed type. In this case, we simply return an empty substitution. The rule [TM-AnonMethExp] reflects the (rather strict) restriction that anonymous method expressions in C$^\sharp$2.0 can be passed as arguments to method invocations, but they do not contribute to the type inference process.

The [TM-OpenExp] rule is where substitutions are actually generated. Assuming that the type to be matched, τ_1, contains free variables, we first synthesize a type for the expression e_1, say τ_2. We then need to 'match' τ_1 with τ_2.

This 'matching' is captured by a function that we write $\mathit{Match}(\Delta, \tau_p, \tau_a)$, and which returns a substitution, $\mathcal{S}$, if one exists. This substitution satisfies the equality $\mathcal{S}(\tau_p) = \tau_a$ (hence Match really implements a simple form of one-way unification). The set Δ contains

$\boxed{Match(\Delta, \tau_p, \tau_a)}$

$$
\begin{array}{rcl}
Match(\Delta, X, \tau) & \stackrel{\text{def}}{=} & \begin{array}{ll} \{X \mapsto \tau\} & \text{if } X \in \Delta \\ \emptyset & \text{otherwise} \end{array} \\
Match(\Delta, \tau_1\,\texttt{[]}, \tau_2\,\texttt{[]}) & \stackrel{\text{def}}{=} & Match(\Delta, \tau_1, \tau_2) \\
Match(\Delta, \rho_1, \tau_2) & \stackrel{\text{def}}{=} & \{\overline{Y} \mapsto \overline{\tau_1}\} \\
 & & \begin{array}{ll} \text{where} & \exists!\overline{\tau_1}.\tau_2 <:^i \rho_1[\overline{Y} := \overline{\tau_1}] \\ \text{and} & \overline{Y} = ftv(\rho_1) \cap \Delta \end{array}
\end{array}
$$

$\boxed{\Gamma; \Delta \vdash e_1 \sim \tau_1 \hookrightarrow \mathcal{S}}$

$$\frac{}{\Gamma; \Delta \vdash \texttt{null} \sim \rho \hookrightarrow \emptyset}\ [\text{TM-Null}]$$

$$\frac{ftv(\tau_1) \cap \Delta = \emptyset}{\Gamma; \Delta \vdash e_1 \sim \tau_1 \hookrightarrow \emptyset}\ [\text{TM-Closed}]$$

$$\frac{}{\Gamma; \Delta \vdash \texttt{delegate(}\overline{\tau_a}\ \overline{x}\texttt{)\{}\overline{s_1}\texttt{\}} \sim D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \emptyset}\ [\text{TM-AnonMethExp}]$$

$$\frac{ftv(\tau_1) \cap \Delta \neq \emptyset \quad \Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_2 \quad Match(\Delta, \tau_1, \tau_2) = \mathcal{S}}{\Gamma; \Delta \vdash e_1 \sim \tau_1 \hookrightarrow \mathcal{S}}\ [\text{TM-OpenExp}]$$

$\boxed{\Gamma \vdash ae_1 \sim \mu_1 \hookrightarrow \mathcal{S}}$

$$\frac{\begin{array}{l} \Gamma; \overline{X} \vdash e_1 \sim \tau_1 \hookrightarrow \mathcal{S}_1 \\ \cdots \\ \Gamma; \overline{X} \vdash e_n \sim \tau_n \hookrightarrow \mathcal{S}_n \\ \text{dom}(\mathcal{S}_1) \cup \cdots \cup \text{dom}(\mathcal{S}_n) = \overline{X} \\ Consistent(\mathcal{S}_1, \ldots, \mathcal{S}_n) \end{array}}{\Gamma \vdash (e_1, \ldots, e_n) \sim \forall\overline{X}.(\tau_1, \ldots, \tau_n) \to \sigma \hookrightarrow \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n}\ [\text{TM-MethodType}]$$

$\boxed{\Gamma \vdash ae_1 \sim \bigwedge \mu_i \hookrightarrow \langle \bigwedge \mu_j, \overline{\mathcal{S}} \rangle}$

$$\frac{\Gamma \vdash ae_1 \sim \mu_{S_1} \hookrightarrow \mathcal{S}_1 \quad \cdots \quad \Gamma \vdash ae_1 \sim \mu_{S_k} \hookrightarrow \mathcal{S}_k \quad |S| = k \geq 1 \quad S \subseteq \{1, \ldots, n\}}{\Gamma \vdash ae_1 \sim \mu_1 \wedge \cdots \wedge \mu_n \hookrightarrow \langle (\mu_{S_1} \wedge \cdots \wedge \mu_{S_k}), (\mathcal{S}_1, \ldots, \mathcal{S}_k) \rangle}\ [\text{TM-MethodGroupType}]$$

Figure 4. Type matching of expressions

the generic type parameters of the method to ensure that we only generate substitutions for the generic type parameters.

The definition of $Match$ is also given in Figure 4; it is recursively defined over the structure of its second argument, the type parameter τ_p. For completeness we also give the clause for array types. This recursive definition is a direct formalization of the iterative process described in the language specification [§20.6.4]. The third clause is least obvious:[5] it covers the case where the parameter type is a constructed type. It is perhaps best motivated by an example. Imagine that we have a class `C<X>` that extends `D<X,int>` and a class `D<A,B>` that extends `E<B,A>`. Consider the case where we are matching a parameter type `E<X,Y>` against an argument type `C<string>`. The purpose of the third clause is to unwind the class declarations to produce the substitution $\{X \mapsto \texttt{int}, Y \mapsto \texttt{string}\}$. The language specification insists that this unwinding only use *standard implicit conversions*, which we write as $<:^i$, although in $\text{FC}^\sharp_2$ this relation is identical to the subtyping relation.[6]

[5] It is formalized declaratively here to match the description in the language specification. An algorithmic version is possible but is omitted for lack of space.

[6] In the full C♯ language the difference is that the standard implicit conversions specifically exclude user-defined implicit conversions.

The uniqueness condition in the third clause of the definition of the $Match$ function arises because of generic interfaces (so it could have been dropped for this fragment). For example, consider the following:

```
interface I<X>{...}
class C:I<int>,I<object>{...}
void m0<X>(I<X> arg1){...}

m0(new C());                          //FAILS
```

Type checking the method invocation of `m0` will result in the following call to the $Match$ function: $Match(X, \texttt{C}, \texttt{I<}X\texttt{>})$. Without the uniqueness requirement in the third clause we would discover two instantiations for X; namely, `int` and `object`. Hence, C♯ rejects this method invocation.

We can lift type matching to argument expressions; judgements are of the form $\Gamma \vdash ae_1 \sim \mu \hookrightarrow \mathcal{S}_1$. This is read that "given type assumptions Γ, the argument expression ae_1 can be type matched against the method type μ, yielding a substitution $\mathcal{S}_1$".

The single rule for this judgement is [TM-MethodType] in Figure 4. This rule has an intuitive operational reading. The given argument expression is of the form $(e_1, \ldots, e_n)$ and the method type is $\forall\overline{X}.(\tau_1, \ldots, \tau_n) \to \sigma$. We then type match each individual argument expression e_i against the parameter type τ_i (where the

set of free method type parameters that we are trying to instantiate is $\{\overline{X}\}$) in turn, yielding a substitution S_i.

C$^\sharp$ imposes two conditions on the collection of substitutions $\mathcal{S}_i$. The first is that collectively they provide instantiations for all the method type parameters. In other words the domain of the collective substitution must be equal to the type parameters of the method type. This condition on the collective substitution is referred to as *completeness* in the language specification.

The second condition is referred to as *consistency*. This turns out to be a very strict condition in practice. It essentially requires that if we have inferred multiple substitutions for a type parameter, then these substitutions must be *identical*. Consider the following method signature and code fragment:

```
void myfoo<T>(T arg1, T arg2);

myfoo("hello",new object());      //FAILS
myfoo<object>("hello",new object()); //CHECKS
```

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

In the rule [TM-MethodType] we denote this consistency check using a predicate $Consistent$. We omit its rather routine definition in the interests of space.

Finally, we lift type matching of an argument expression against a method type, to a method *group* type. These judgements are of of the form $\Gamma \vdash ae \sim \bigwedge \mu_i \hookrightarrow \langle \bigwedge \mu_j, \overline{\mathcal{S}} \rangle$. This means that the argument expression ae is systematically type matched against each method type in the method group. The result is a pair of (1) a method group containing the method types that matched and for each of these (2) the resulting substitution. The single rule for this judgement form is [TM-MethodGroupType] in Figure 4.

3.4 Conclusion

We have now completely, and formally, specified the type system and type inference process of our core fragment, FC$^\sharp_2$, of C$^\sharp$2.0! We have utilized the bidirectional approach and defined type checking and type synthesis judgements. To define type inference we needed to add a third judgement: type matching.

4. Extending C$^\sharp$2.0 type inference

4.1 Relaxing consistency

As demonstrated in the previous section, the consistency condition imposed on inferred substitutions is quite restrictive. In this section, we give a simple relaxation of the consistency condition that is both in the spirit of C$^\sharp$, and simple to implement.

We shall first give the intuitions behind the proposal and give the formalization later. We will consider array types in this section for completeness. We shall use the following method signatures in our informal discussion:

```
void m1<X>(X arg1, X arg2);
void m2<X>(X arg1, list<X> arg2);
void m3<X>(list<X> arg1, list<X> arg2);
void m4<X>(list<X> arg1, X arg2, X arg3);
```

First, we need to introduce notation to represent multiple substitutions for method type parameters. We shall write these, for example: $\{X \mapsto \{\texttt{int}, \texttt{object}\}\}$. We shall write $\uplus$ to denote multiple substitution composition. For example, $\{X \mapsto \texttt{int}\} \uplus \{X \mapsto \texttt{object}\} = \{X \mapsto \{\texttt{int}, \texttt{object}\}\}$. (We shall refer to a substitution where every type parameter is mapped to a single type as a *simple* substitution.)

The next concept we use exists already in C$^\sharp$(for example, in the typing of implicit arrays): the notion of a *best* type from a given set of types. The best type is simply one from the set which all the other types in the set can be converted to.

In resolving our multiple substitutions we shall use this notion of best. The idea is that we consider all the substitutions for each type parameter and pick the best representative from the set. If there is not a single best type, then type inference fails.

This is already sufficient to type simple invocations (that currently fail to type in C$^\sharp$2.0) such as:

```
m1(42, new object()); // Infers <object>
m1(42, "hello");      // Fails - no best type
```

However, as it stands, this is *unsound*! It is too expressive; the following succeeds but shouldn't.

```
m3(new List<int>(), new List<object>()); //Shouldn't work
```

As it stands, we would infer the type argument `<object>`. The problem is that C$^\sharp$ constructed types are *invariant*—this has been ignored so far. The solution is quite simple: we record whether the substitution applies to a type variable that is in a position where a conversion can apply (such as the top level of a method signature) or not (such as inside a constructed type). We annotate the target types of a substitution accordingly, so they are of the form $\{X \mapsto \tau^a\}$, where a is either $<:$ (a "convertible substitution") or $=$ (an "equality substitution"). Consider the following invocation:

```
m1(42, new object()); //SUCCEEDS
```

We generate the substitution: $\{X \mapsto \{\texttt{int}^{<:}, \texttt{object}^{<:}\}\}$. The modified consistency rule is that if we have only convertible substitutions then we resolve them by finding the single best type. In this case, the best type is `object`, so the inference process succeeds.

Consider the following:

```
m3(new List<int>(), new List<object>()); //FAILS
```

Now we infer the substitution: $\{X \mapsto \{\texttt{int}^{=}, \texttt{object}^{=}\}\}$. The modified consistency rule is that the equality substitutions must be identical (this is the current rule for C$^\sharp$2.0). Hence this invocation fails the type inference process.

Consider the following:

```
m2(42, new List<object>());
```

We infer the substitution: $\{X \mapsto \{\texttt{int}^{<:}, \texttt{object}^{=}\}\}$, i.e. a convertible *and* an equality substitution. The modified consistency rule is that the convertible target types must all be convertible to the (single) equality target type. The example above then succeeds as `int` converts to `object`. So we infer the type argument `<object>`.

Consider another similar example:

```
m4(new List<object>(), 42, "hello");
```

We infer the substitution: $\{X \mapsto \{\texttt{int}^{<:}, \texttt{object}^{=}, \texttt{string}^{<:}\}\}$. As both `int` and `string` convert to `object` we infer the type parameter `<object>`.

Some further examples are as follows:

```
m1(42, new int?(84));                    //Infers <int?>
m1(42, "hello");                         //FAILS
m3(new List<int>(), new List<object>()); //FAILS
m3(new List<Button>(), new List<Control>()); //FAILS
```

In C♯ arrays are only covariant for reference types. This slightly complicates the type inference process. We thus add a new annotation for the target type of a substitution: $\circ$, which denotes a covariant substitution. For example, consider the following method signatures:

```
void m5<X>(X[] arg1, X[] arg2);
void m6<X>(list<X[]> arg1, list<X[]> arg2);
void m7<X>(list<X>[] arg1, list<X>[] arg2);
```

Consider the following invocation:

```
m5(new string[]{}, new object[]{});
```

We infer the substitution: $\{X \mapsto \{\texttt{string}^\circ, \texttt{object}^\circ\}\}$. We add to our consistency rule to first process the covariant substitutions. If, as above, we generate covariant substitutions whose target types are *all* reference types, then we rewrite them as convertible substitutions, and continue as described before. Hence this invocation above would succeed and we would infer the type argument `<object>`.

However, consider the following invocation:

```
m5(new int[]{}, new object[]{});
```

We infer the substitution $\{X \mapsto \{\texttt{int}^\circ, \texttt{object}^\circ\}\}$. Here we have generated covariant substitutions where one of the target types is a value type. In this case we rewrite all the covariant substitutions as equality substitutions (as no conversion exists here). Hence, the invocation above would fail. Likewise the following invocation would also fail:

```
m5(new sbyte[]{}, new byte[]{});
```

The following would succeed, and infer the type argument `<int>`.

```
m5(new int[]{}, new int[]{});
```

Some further examples are as follows:

```
m6(new List<string[]>(), new List<object[]>()) //FAILS
m7(new List<string>[]{}, new List<object>[]{});
                                  //Infers <object>
```

Extending the formalization from the previous section is quite straightforward. We parameterize the type matching function, $Match$, with an annotation, a.

$$
\begin{array}{rcll}
Match^a(\Delta, X, \tau) & \stackrel{\text{def}}{=} & \{X \mapsto \tau^a\} & \text{if } X \in \Delta \\
 & & \emptyset & \text{otherwise} \\
Match^a(\Delta, \tau_1\,[\,], \tau_2\,[\,]) & \stackrel{\text{def}}{=} & Match^\circ(\Delta, \tau_1, \tau_2) & \\
Match^a(\Delta, \rho_1, \tau_2) & \stackrel{\text{def}}{=} & \{\overline{Y} \mapsto \overline{\tau_1^=}\} & \\
 & & \text{where} \quad \exists!\overline{\tau_1}.\tau_2 <:^i \rho_1[\overline{Y} := \overline{\tau_1}] & \\
 & & \text{and} \quad \overline{Y} = ftv(\rho_1) \cap \Delta &
\end{array}
$$

The type matching rule [TM-OpenExp] is also modified to the following.

$$
\frac{ftv(\tau_1) \cap \Delta \neq \emptyset \quad \Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_2 \quad Match^{<:}(\Delta, \tau_1, \tau_2) = \mathcal{S}}{\Gamma; \Delta \vdash e_1 \sim \tau_1 \hookrightarrow \mathcal{S}}
$$

The final change is to replace the $Consistent$ predicate with a partial function (which we shall also call $Consistent$). This takes as input a *multiple* substitution and returns the corresponding *simple* substitution. We also refer to the action of this function as "resolving" a multiple substitution. For example,

$$
\begin{array}{rl}
Consistent(\{X \mapsto \{\texttt{int}^{<:}, \texttt{object}^{=}, \texttt{string}^{<:}\}\}) & = \{X \mapsto \texttt{object}\} \\
Consistent(\{Y \mapsto \{\texttt{int}^{=}, \texttt{object}^{=}\}\}) & = \text{undefined} \\
Consistent(\{Z \mapsto \{\texttt{string}^{\circ}, \texttt{object}^{=}\}\}) & = \{X \mapsto \texttt{object}\}
\end{array}
$$

It is fairly straightforward to turn the informal description of the modified consistency rule given above into a formal definition, although we do not do so here for lack of space.

4.2 λ-expressions

The next version of C♯(version 3.0) will contain a number of new language features, mostly to support the LINQ framework [10]. Interestingly all of these new features can be explained in terms of a type-directed translation from C♯3.0 to C♯2.0; the details of this translation will appear in a forthcoming paper [1].

One new feature, the λ-expression, whilst essentially syntactic sugar, actually interacts with the type inference process. Hence, type inference needs to be changed in the C♯3.0 compiler. In this section we show how our treatment of $\mathsf{FC}_2^\sharp$ can be extended quite simply with the addition of λ-expressions.

First we extend $\mathsf{FC}_2^\sharp$ with new syntax for λ-expressions. We follow C♯3.0 and allow the parameter list of a λ-expression to be explicitly or implicitly typed.

$e ::=$	Expression
...	
$(\overline{\tau}\ \overline{x})$ => e	Explicitly typed lambda
$(\overline{x})$ => e	Implicitly typed lambda

C♯3.0 includes a predefined delegate family, `Func`, which is used in connection with λ-expressions. Similarly, we shall assume the following global delegate definitions:

```
delegate R Func<T1,R>(T1 arg);
delegate R Func<T1,T2,R>(T1 arg1, T2 arg2);
delegate R Func<T1,T2,T3,R>(T1 arg1, T2 arg2, T3 arg3); ...
```

Type *checking* λ-expressions is relatively straightfoward; the new rules are as follows:

$$
\frac{dtype(D)(\overline{\tau}) = \overline{\tau_0} \to \tau_1 \quad \Gamma, \overline{x_0} : \overline{\tau_0} \vdash e_1 : \tau_1 \hookrightarrow e_{11}}{\Gamma \vdash (\overline{x_0})\texttt{ => } e_1 : D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \texttt{delegate}\ (\overline{\tau_0}\ \overline{x_0})\texttt{\{return}\ e_{11}\texttt{;\}}}
$$

$$
\frac{dtype(D)(\overline{\tau}) = \overline{\tau_0} \to \tau_1 \quad \Gamma, \overline{x_0} : \overline{\tau_0} \vdash e_1 : \tau_1 \hookrightarrow e_{11}}{\Gamma \vdash (\overline{\tau_0}\ \overline{x_0})\texttt{ => } e_1 : D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \texttt{delegate}(\overline{\tau_0}\ \overline{x_0})\texttt{\{return}\ e_{11}\texttt{;\}}}
$$

Again we observe that, like for anonymous method expressions, λ-expressions can only be checked against delegate types.

Type *synthesis* for λ-expressions is also simple: no (denotable) type can be synthesized! This explains why the following is not type correct in C♯3.0.

```
var myIdFun = (x) => x;  //Type error!
```

Implicit declarations (those marked **var**) simply synthesize types for the defining expression. In this case no type can be synthesized, so the declaration is type incorrect.

As mentioned earlier, λ-expressions interact with the type inference process. Recall that rule [TM-AnonMethExp] captures the restriction that anonymous method expressions do not contribute substitutions to the type inference process. Whilst λ-expressions are nothing more than syntactic sugar for anonymous method expressions, their bodies are *expressions*, and C♯3.0 leverages that fact to enable λ-expressions to generate substitutions.[7] Thus we could add the following type *matching* rule for explicitly-typed λ-expressions.

$$\frac{dtype(D)(\overline{\tau}) = \overline{\tau_1} \to \tau \quad \Gamma, \overline{x} : \overline{\tau_1}; \Delta \vdash e_1 \sim \tau \hookrightarrow \mathcal{S}}{\Gamma; \Delta \vdash \texttt{(}\overline{\tau_1}\ \overline{x}\texttt{) => } e_1 \sim D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \mathcal{S}}$$

In fact, we can be more flexible. The rule above insists that the explicit parameter type is identical to the delegate argument type. We could be more liberal and use the following rule.

$$\frac{dtype(D)(\overline{\tau}) = \overline{\tau_2} \to \tau \quad \Gamma, \overline{x} : \overline{\tau_1}; \Delta \vdash e_1 \sim \tau \hookrightarrow \mathcal{S}_1 \quad \overline{Match}^{=}(\Delta, \overline{\tau_2}, \overline{\tau_1}) = \mathcal{S}_2}{\Gamma; \Delta \vdash \texttt{(}\overline{\tau_1}\ \overline{x}\texttt{) => } e_1 \sim D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \mathcal{S}_1 \uplus \mathcal{S}_2}$$

Some further examples of using this more liberal type matching rule for explicitly-typed lambda expressions are as follows.

```
void m8<X>(Func<X,X> arg1);
void m9<X>(Func<X,X> arg1, X arg2);

m8((object x) => x);         //Infers <object>
m8((object x) => "hello");   //Infers <object>
m9((object x) => x, 42);     //Infers <object>
m9((int x) => x, new object()); //FAILS
```

The real problem is type matching implicitly-typed λ-expressions. Matters are quite straightforward if all the argument types are closed; the type matching rule is then as follows.

$$\frac{dtype(D)(\overline{\tau}) = \overline{\tau_1} \to \tau \quad ftv(\overline{\tau_1}) \cap \Delta = \emptyset \quad \Gamma, \overline{x} : \overline{\tau_1}; \Delta \vdash e_1 \sim \tau \hookrightarrow \mathcal{S}}{\Gamma; \Delta \vdash \texttt{(}\overline{x}\texttt{) => } e_1 \sim D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \mathcal{S}}$$

How do we deal with open argument types? We can't simply type match the body of the λ-expression because we will then have type assumptions with unbound type parameters. There are a number of possibilities, using a more global constraint-based system, for example, but many of these (non-local) techniques interact rather badly with overloading and subtyping. This is certainly an area requiring further research.

The current proposal for C♯3.0 is to make the type inference process *iterative*. Thus type matching an implicitly-typed λ-expression with an open argument type simply postpones. We consider all the other arguments and generate an intermediate substitution. We then apply this to any postponed type matches to see if any will become enabled (if the argument types have become closed). We then continue this process until either all the arguments have been type matched or where we do not make progress, in which case type inference will fail.

It turns out that this iterative type inference process can be expressed with only a small change to the formalization so far. The first change we make from our formalization in the previous sections is to the notion of type matching. We break matching up into two stages: First we introduce a notion of partial matching. We write this judgement as follows: $\Gamma; \Delta \vdash e \rhd \tau \hookrightarrow \mathcal{S}$, which is meant to be read as "partially type matching e against type τ yields substitutions $\mathcal{S}$".[8]

The rules for partial type matching are given in Figure 5. They are essentially the same as the type matching rules from the previous sections. The main difference is the treatment of implicitly-typed λ-expressions. The rule [PTM-ILambdaExp1] defines partial type matching for λ-expressions where the argument type is closed; the rule [PTM-ILambdaExp2] where the argument type is open. In this case the substitution is (for the moment) empty.

The other judgement we dub complete matching, which is written $\Gamma; \Delta \vdash e_1 \sim \tau \hookrightarrow \mathcal{S}$, which is read as "matching e_1 against type τ completes and yields a substitution $\mathcal{S}$". The rules for these judgement forms are also given in Figure 5.

The key rules are [TM3-ClosedArgExp] and [TM3-OpenArgExp]. They recursively encode the iterative type matching process described earlier. The recursion is driven by the set of type parameters Δ. If the set is empty, then the [TM3-ClosedArgExp] rule applies, i.e. there is no matching to be done.

If the set Δ is non-empty then the rule [TM3-OpenArgExp] applies. This uses the partial type matching rules to generate an intermediate substitution, $\mathcal{S}$. If this substitution is empty, then we have made no progress and type inference fails. If it is non-empty then we apply the $Consistent$ function to yield a simple substitution. We apply this substitution to the argument types and remove the type parameters that have now been substituted for, from Δ. We then recurse using this new type parameter set and argument type.

It is relatively straightforward to see that these two rules define a terminating recursive function to perform type inference.

The resulting type system is quite expressive, and should be attractive to the reader familiar to functional languages such as Haskell or ML. Consider the following example method signature and expression :

```
void m10<X,Y,Z>(Func<X,Y> arg1, Func<Y,Z> arg2, X arg3);

m10((x)=>x, (y)=>y, 42)
```

In the first phase of type inference the first and second arguments do not contribute substitutions as their argument types are open. The third argument contributes the substitution $\{X \mapsto \texttt{int}\}$. In phase two we apply the substitution and so the first parameter type becomes `Func<int,Y>` and hence we can apply rule [PTM-ILambdaExp1]. This produces the substitution $\{Y \mapsto \texttt{int}\}$. The second argument does not contribute a substitution to phase two. In phase three we apply the substitution and so the second parameter type becomes `Func<int,Z>` and we can apply rule [PTM-ILambdaExp1]. This produces the substitution $\{Z \mapsto \texttt{int}\}$ and the type inference process then succeeds producing the type argument list `<int,int,int>`.

4.3 Return types

In rule [TM3-MethodType] of Figure 5, the return type of the method type plays no part in the generation of substitutions. There are circumstances where that is a serious restriction. Consider the following method signature and assignment.

```
class Foo
{
  public static List<T> Nil<T>(){return new List<T>();};
```

[7] There is no reason why C♯ could not enable anonymous method expressions to generate subsitutions. This appears to have been an implementation decision.

[8] It is only partial in the sense that it includes λ-expressions which can not be currently typed.

$\boxed{\Gamma; \Delta \vdash e_1 \rhd \tau_1 \hookrightarrow \mathcal{S}}$

$$\frac{}{\Gamma; \Delta \vdash \texttt{null} \rhd \rho \hookrightarrow \emptyset}\ \text{[PTM-Null]}$$

$$\frac{\mathit{ftv}(\tau_1) \cap \Delta = \emptyset}{\Gamma; \Delta \vdash e_1 \rhd \tau_1 \hookrightarrow \emptyset}\ \text{[PTM-ClosedExp]}$$

$$\frac{}{\Gamma; \Delta \vdash \texttt{delegate(}\overline{\tau_a}\ \overline{x}\texttt{)\{}\overline{s_1}\texttt{\}} \rhd D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \emptyset}\ \text{[PTM-AnonMethExp]}$$

$$\frac{\mathit{ftv}(\tau_1) \cap \Delta \neq \emptyset \quad \Gamma \vdash e_1 \hookrightarrow e_{11} : \tau_2 \quad \mathit{Match}^{<:}(\Delta, \tau_1, \tau_2) = \mathcal{S}}{\Gamma; \Delta \vdash e_1 \rhd \tau_1 \hookrightarrow \mathcal{S}}\ \text{[PTM-OpenExp]}$$

$$\frac{\mathit{dtype}(D)(\overline{\tau}) = \overline{\tau_2} \to \tau \quad \Gamma, \overline{x} : \overline{\tau_1}; \Delta \vdash e_1 \rhd \tau \hookrightarrow \mathcal{S}_1 \quad \mathit{Match}^{=}(\Delta, \overline{\tau_2}, \overline{\tau_1}) = \mathcal{S}_2}{\Gamma; \Delta \vdash \texttt{(}\overline{\tau_1}\ \overline{x}\texttt{) => } e_1 \rhd D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \mathcal{S}_1 \uplus \mathcal{S}_2}\ \text{[PTM-ELambdaExp]}$$

$$\frac{\mathit{dtype}(D)(\overline{\tau}) = \overline{\tau_2} \to \tau \quad \mathit{ftv}(\overline{\tau_2}) \cap \Delta = \emptyset \quad \Gamma, \overline{x} : \overline{\tau_2}; \Delta \vdash e_1 \rhd \tau \hookrightarrow \mathcal{S}}{\Gamma; \Delta \vdash \texttt{(}\overline{x}\texttt{) => } e_1 \rhd D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \mathcal{S}}\ \text{[PTM-ILambdaExp1]}$$

$$\frac{\mathit{dtype}(D)(\overline{\tau}) = \overline{\tau_2} \to \tau \quad \mathit{ftv}(\overline{\tau_2}) \cap \Delta \neq \emptyset}{\Gamma; \Delta \vdash \texttt{(}\overline{x}\texttt{) => } e_1 \rhd D\texttt{<}\overline{\tau}\texttt{>} \hookrightarrow \emptyset}\ \text{[PTM-ILambdaExp2]}$$

$$\frac{\Gamma; \Delta \vdash e_1 \rhd \tau_1 \hookrightarrow \mathcal{S}_1 \quad \cdots \quad \Gamma; \Delta \vdash e_n \rhd \tau_n \hookrightarrow \mathcal{S}_n}{\Gamma; \Delta \vdash (e_1, \ldots, e_n) \rhd (\tau_1, \ldots, \tau_n) \hookrightarrow \mathcal{S}_1 \uplus \cdots \uplus \mathcal{S}_n}\ \text{[PTM-ArgExp]}$$

$\boxed{\Gamma; \Delta \vdash (e_1, \ldots, e_n) \sim (\tau_1, \ldots, \tau_n) \hookrightarrow \mathcal{S}}$

$$\frac{}{\Gamma; \emptyset \vdash (e_1, \ldots, e_n) \sim (\tau_1, \ldots, \tau_n) \hookrightarrow \emptyset}\ \text{[TM3-ClosedArgExp]}$$

$$\frac{\begin{array}{l}\Delta \neq \emptyset \\ \Gamma; \Delta \vdash (e_1, \ldots, e_n) \rhd (\tau_1, \ldots, \tau_n) \hookrightarrow \mathcal{S} \\ \mathcal{S} \neq \emptyset \\ \mathcal{S}_c = \mathit{Consistent}(\mathcal{S}) \\ \Gamma; \Delta - \mathrm{dom}(\mathcal{S}_c) \vdash (e_1, \ldots, e_n) \sim \mathcal{S}_c[(\tau_1, \ldots, \tau_n)] \hookrightarrow \mathcal{S}_f\end{array}}{\Gamma; \Delta \vdash (e_1, \ldots, e_n) \sim (\tau_1, \ldots, \tau_n) \hookrightarrow \mathcal{S}_c \cup \mathcal{S}_f}\ \text{[TM3-OpenArgExp]}$$

$\boxed{\Gamma \vdash ae_1 \sim \mu_1 \hookrightarrow \mathcal{S}}$

$$\frac{\Gamma; \overline{X} \vdash (e_1, \ldots, e_n) \sim (\tau_1, \ldots, \tau_n) \hookrightarrow \mathcal{S}}{\Gamma \vdash (e_1, \ldots, e_n) \sim \forall \overline{X}.(\tau_1, \ldots, \tau_n) \to \sigma \hookrightarrow \mathcal{S}}\ \text{[TM3-MethodType]}$$

$\boxed{\Gamma \vdash ae_1 \sim \bigwedge \mu_i \hookrightarrow \langle \bigwedge \mu_j, \overline{\mathcal{S}} \rangle}$

$$\frac{\Gamma \vdash ae_1 \sim \mu_{S_1} \hookrightarrow \mathcal{S}_1 \quad \cdots \quad \Gamma \vdash ae_1 \sim \mu_{S_k} \hookrightarrow \mathcal{S}_k \quad |S| = k \geq 1 \quad S \subseteq \{1, \ldots, n\}}{\Gamma \vdash ae_1 \sim \mu_1 \wedge \cdots \wedge \mu_n \hookrightarrow \langle \mu_{S_1} \wedge \cdots \wedge \mu_{S_k}, (\mathcal{S}_1, \ldots, \mathcal{S}_k) \rangle}\ \text{[TM3-MethodGroupType]}$$

Figure 5. Type matching of argument expressions in C♯3.0

```
}

List<int> empty = Foo.Nil(); \\FAILS
```

The invocation of `Nil` will fail type inference as we can never infer a type for `T` from the (empty) list of arguments. But, in this case, we could use the fact that the result of the method invocation is immediately assigned to the type `List<int>` to infer the type argument `<int>`.

To implement this, we first need to extend the type *checking* rule for implicit method invocation to feed in the expected assigned type (in this case, τ_3) to the type matching relation.

$$\frac{\begin{array}{l}\Gamma \vdash me_1 \Uparrow me_{11} : \bigwedge \mu_i \\ \Gamma \vdash ae_1 \sim \bigwedge \mu_i : \tau_3 \hookrightarrow \langle \bigwedge \mu_j, \overline{\mathcal{S}} \rangle \\ \mathit{OloadRes}(\bigwedge \mu_j, \overline{\mathcal{S}}, ae_1) = (ae_{11}, \overline{\tau}, \sigma) \\ \sigma <: \tau_3\end{array}}{\Gamma \vdash me_1\ ae_1 : \tau_3 \hookrightarrow me_{11}\texttt{<}\overline{\tau}\texttt{>}\ ae_{11}}$$

We now define a variant of the type matching judgement that takes in addition a return type. For example, the variant judgement for type matching an argument expression against a method type would be of the form $\Gamma \vdash ae_1 \sim \mu : \tau \hookrightarrow \mathcal{S}_1$. The rule for this judgement would be as follows:

$$\frac{\begin{array}{l}\Gamma; \mathit{ftv}(\overline{\tau}) \cap \overline{X} \vdash ae_1 \sim (\tau_1, \ldots, \tau_n) \hookrightarrow \mathcal{S}_1 \\ \mathcal{S}_2 = \mathit{Match}^{=}(\overline{X}, \tau_{21}, \sigma) \\ \mathcal{S}_3 = \mathit{Consistent}(\mathcal{S}_1 \uplus \mathcal{S}_2) \\ \mathrm{dom}(\mathcal{S}_3) = \overline{X}\end{array}}{\Gamma \vdash ae_1 \sim \forall \overline{X}.(\tau_1, \ldots, \tau_n) \rightarrow \sigma : \tau_{21} \rightsquigarrow \mathcal{S}_3}$$

In other words, we generate substitutions $\mathcal{S}_1$ from the argument expression for the free type parameters in the argument types that are method type parameters. We then see if matching the expected return type τ_{21} against the actual return type of the method σ generates a new substitution. This is then combined with substitution $\mathcal{S}_1$, and checked for consistency and for completeness.

So, in the example given above, the expression `Foo.Nil()` would type and the inferred type argument would be `<int>`.

5. Conclusions and work in progress

In this paper we have attempted a formal reconstruction of the type inference process as it is implemented in C♯2.0. We used a bidirectional type system, and extended this with a new typing judgement, type matching, to capture the (local) type inference process implemented in C♯.

One advantage of our formalization is that it is a good setting to consider extensions to the type inference process itself, and to see the impact of adding new language features. We showed how to relax the strict consistency conditions currently implemented in C♯2.0. We also considered the difficulties of adding λ-expressions in the style of C♯3.0. We gave a refactoring of our type matching judgements that implements an iterative type inference process.

As mentioned in the introduction, most work on type inference has studied the problem for Java. The Java 5.0 type inference process is more complete than its counterpart in C♯2.0 but it is a *non*-local process. On the other hand, it is more complicated. A fuller comparison of the two type inference processes remains future work.

Work in progress We are currently working on a considerably more flexible type inference process for λ-terms. Consider the following method signature

```
void foo<X,Y>(X arg1, Func<Y,X> arg2, Y arg3){...}
```

and an implicit invocation `foo(42,(x)=>new object(),42)`. Using the type inference process discussed in this paper, the first phase would deduce the substitution $\{X \mapsto \texttt{int}, Y \mapsto \texttt{int}\}$, from the first and third arguments. The second argument, the λ-expression, does *not* contribute as in the first phase its argument type is open. The rule [TM3-OpenArgExp] applies the substitutions, so we have effectively fixed type parameter X to be `int`. The subsequent type check of the λ-expression argument will fail. However, if we had not fixed the substitution but postponed it until we had considered the second argument, we could have deduced the substitution of `object` for X, and hence the invocation would succeed. Devising an inference process that is sensitive to the dependencies of type parameters induced by function types is current work in progress.

In addition, we are also developing declarative versions of our rules, which builds on the work of Odersky et al. [14] on *coloured* local type inference.

Acknowledgements I am grateful to Erik Meijer, Andrew Kennedy, Claudio Russo, Mads Torgersen, Eric Lippert and the anonymous referees for their insightful comments on this work.

References

[1] G.M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing C♯ 3.0. Unpublished draft paper, To appear.

[2] 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.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to Java. In *Proceedings of OOPSLA*, 1998.

[4] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java programming language. In *Proceedings of OOPSLA*, 1998.

[5] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of POPL*, 1998.

[6] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005.

[7] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C♯ Programming Language*. Addison-Wesley, second edition, 2006.

[8] 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.

[9] A. Jeffrey. Generic Java type inference is unsound. Note sent to `types` mailing list, December 2001.

[10] E. Meijer, B. Beckman, and G.M. Bierman. LINQ: Reconciling objects, relations and XML in the .NET framework. In *Proceedings of SIGMOD*, 2006.

[11] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *Proceedings of POPL*, 1997.

[12] M. Odersky. Inferred type instantiation for GJ. Note sent to `types` mailing list, January 2002.

[13] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of POPL*, 1997.

[14] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *Proceedings of POPL*, 2001.

[15] B.C. Pierce and D.N. Turner. Local type inference. In *Proceedings of POPL*, 1998.