# Transposing F to C$^\sharp$
# (Preliminary report)

Andrew Kennedy Don Syme

Microsoft Research, Cambridge, U.K.
{akenn,dsyme}@microsoft.com

**Abstract**

We present a type-preserving translation of the polymorphic lambda calculus (System F) into an extension of the C$^\sharp$ programming language supporting parameterized classes and polymorphic methods.

We observe that whilst parameterized classes alone are sufficient to encode the parameterized datatypes and let-polymorphism of languages such as ML and Haskell, it is the presence of *polymorphic virtual* methods that allow the encoding of the "first-class polymorphism" found in System F and recent extensions to ML and Haskell.

## 1 Introduction

Parametric polymorphism is a well-known and well-explored feature of declarative programming languages such as Haskell, ML and Mercury. It ranges in expressive power from the parameterized types and let-polymorphism of core ML, through the parameterized modules of Standard ML and Caml, in which module parameters may themselves contain polymorphic values, right up to full support for first-class polymorphism found in extensions to Haskell [4, 7] and ML [11]. The polymorphic lambda calculus, System F, and its higher-order variant $F_\omega$ [9] are foundational calculi that capture the range of polymorphic constructs found in such languages.

Some object-oriented languages have introduced support for polymorphism. Eiffel supports parameterized classes, whilst extensions to Java such as Pizza, GJ and NextGen [8, 1, 2] and a proposed design for C$^\sharp$ [5] also support the type-parameterization of static and virtual methods.

In this paper we pose the question "what's the difference?" and show that the combination of parameterized classes and polymorphic virtual methods are sufficient to encode the first-class notion of polymorphism supported by System F and extensions to ML and Haskell. We demonstrate this by presenting a formal translation of a variant of System F into a subset of C$^\sharp$ with polymorphism. The translation is fully type-preserving; work is in progress to prove its semantic correctness using the method of logical relations. When mentioning C$^\sharp$ in the rest of this paper, we refer to the proposed design for generics [5].

The paper is structured as follows. We begin by discussing informally the various flavours of polymorphism. In Sections 2 and 3 we define typing and evaluation rules for System F and C$^\sharp$, and in Section 4 we present the formal translation. Section 5 discusses some future work.

### 1.1 Core ML polymorphism

The following piece of SML code typifies the use of polymorphism in the core language:

```
datatype 'a Tree = Leaf of 'a | Node of 'a Tree list

fun map (f : 'a->'b) (t : 'a Tree) =
case t of
  Leaf v => f v
| Node branches => Node (map f branches)

val stringtree = map Int.toString inttree
```

This code illustrates two aspects to polymorphism in ML: the type-parameterization of types (here, the type parameter `'a` to the type `Tree`) and of functions (type parameters `'a` and `'b` in the `map` function). Both aspects of polymorphism also feature in the object-oriented systems of Pizza, GJ, NextGen and Generic C$^\sharp$, where parameterized classes and interfaces play the same role as parameterized types, and parameterized methods play the role of polymorphic functions.

The following skeleton C$^\sharp$ code suggests that parameterized classes are sufficient to achieve a direct encoding of core ML into an object-oriented language with parametric polymorphism:

```
abstract class Tree<A> { ... }
class Leaf<A> : Tree<A> { ... }
class Node<A> : Tree<A> { ... }

interface Arrow<A,B> { B apply(A x); }

class IntToString : Arrow<int,string> {
  string apply(int x) { return x.ToString(); }
}

class Mapper<A,B> {
  Tree<B> map(Arrow<A,B> f, Tree<A> tree) {
    ... }
}

...
Tree<string> stringtree =
  new Mapper<string,int>().map(intToString, inttree);
```

The observation is that parameterized datatypes and their constructors can be translated directly into parameterized classes, and polymorphic functions must be translated into parameterized classes whose type parameters are the inferred type parameters of the function. Classes are required for parametric functions because, like any function value in core ML, these functions may be nested and thus contain free variables. Function values then correspond to objects of such classes.

One case in such translations requires careful attention: core ML permits "let-bound" functions to be polymorphic, and any closure conversion must take into account the cases where the body of such functions contain free type variables, i.e. type variables that are not type parameters of the function. For example, the following function extracts integer and floating point values for each element of a list:

```
fun markTwice (f:'a->int) (g:'a->real) (l:'a list) =
  let fun mark (h:'a->'b) = map h l
  in (mark f, mark g) end
```

Here the function `mark` is inferred to have the type scheme $\forall\beta.(\alpha$ `->` $\beta)$ `->` $\beta$ `list` and is used with type arguments `int` and `real`. A conversion into parameterized classes must translate `mark` into a class parameterized by both the type parameter $\beta$ and the free type variable $\alpha$:

```
class Mark<A,B> : Arrow<Arrow<A,B>,List<B>> {
  string apply(Arrow<A,B>) { ... }
}
class markTwice<A> : ... {
  ... apply(...) {
        new Mark<A,int>().apply(...)
        new Mark<A,float>().apply(...)
}
```

Such translations depend on the fact that in core ML polymorphic values are never passed around in their polymorphic form: instead they must be used locally at some (inferred) type instantiation. In this sense they are not "first-class" citizens of the language.

From these observations we claim informally that core ML may be directly encoded into parameterized classes without making use of using parameterized methods. For example, it is possible to translate a toy fragment of ML into C$^\sharp$ with parameterized classes only.

## 1.2 First-class polymorphism

We now show an example that indicates that adding parameterized methods to a system with parameterized classes leads to a system whose polymorphism is more expressive that found in core ML. Consider the following fragment of C$^\sharp$:

```
interface IComparer<T> {
  int Compare(T x, T y);
}

interface Sorter {
  void Sort<T>(T[] a, IComparer<T> c);
}

class QuickSort : Sorter {
  void Sort<T>(T[] a, IComparer<T> c) {
    ... }
}

class MergeSort : Sorter {
  void Sort<T>(T[] a, IComparer<T> c) {
    ... }
}
...
void Test(Sorter s, int[] ia, string[] sa) {
  s.Sort<int>(ia, IntComparer);
  s.Sort<string>(sa, StringComparer);
}
```

Here we define an interface[1] `Sorter` representing the type of *polymorphic sorters*, and whose single method is polymorphic in the element type of array to be sorted. This interface is then implemented by particular sorter classes such as `QuickSort` and `MergeSort`. Object instances of such sorters can then be passed around at run-time and applied at different type instantiations, as shown in the method `Test`. We call this "first-class polymorphism".

Recent variants of Haskell [4, 7] and ML [11] support a similar notion. For example, the above example can be encoded in Russo's extension to the module system of Standard ML.

```
signature Sorter = sig
  val Sort : 'a array * ('a*'a->order) -> unit
end
structure QuickSort :> Sorter = struct
  fun Sort(a, c) = ...
end
structure MergeSort :> Sorter = struct
  fun Sort(a, c) = ...
end
fun Test(s : [Sorter], ia, sa) =
let structure S as Sorter = s
in
  S.Sort(ia, Int.compare);
  S.Sort(sa, String.compare)
end;
```

Attempts to write the program in a similar fashion in Core ML fail, e.g.

```
type 'a sorter = ('a->'a->bool) -> 'a list -> unit

fun int_compare (n:int) (m:int) = n < m
fun string_compare (n:string) (m:string) = ...

// type error: sorter is applied to two
// different kinds of lists
fun test (sorter: 'a sorter) =
   sorter int_compare [ 2; 1; 3 ];
   sorter string_compare [ "c"; "a"; "b"; ]
```

## 2 System F with recursion

The above examples indicate that a more rigorous investigation into the expressivity of adding parameterized methods is required. Our technique is to take a variant of the expressive calculus System F [9] and to show that it can be compiled to an object-oriented language with both parameterized classes and methods. We mainly consider a translation which is type-preserving but which loses separate compilation, though later we also consider a simpler translation which is only partially type-preserving but which permits separate compilation.

Our source language is an extension of System F [9] with recursion and a call-by-value evaluation order. Its syntax, typing rules and big-step evaluation semantics are presented in Figure 1. Observe that

[1]We could have used an abstract class in place of the interface

- Terms of polymorphic type must be values (canonical forms). This matches the so-called "value restriction" of ML and simplifies the semantics slightly.

- There are no base types. The usual System F encodings can be used to support types such as `bool`, `nat` and $A$ `list` and operations over them (see [10] for proofs that these encodings are faithful).

- Function values are by default recursive. Sometimes we will use the notation $\lambda x{:}A.M$ as shorthand for the equivalent `rec` $y(x{:}A){:}B.M$ (for fresh $y$ and appropriate $B$).

**Property 1 (System F evaluation preserves typing).** *If $\Delta; E \vdash M : A$ and $\Delta; \emptyset \vdash V_i : A_i$ for each $x_i{:}A_i$ in $E$ then $M[V_i/x_i] \Downarrow V$ implies $\Delta; \emptyset \vdash V : A$.*

## 3 C$^\sharp$ minor

Our target language 'C$^\sharp$ minor' is a small, purely-functional subset of a new version of the C$^\sharp$ programming language with support for parametric polymorphism [5]. Its syntax, typing rules and big-step evaluation semantics are presented in Figure 2. This formalisation is based on Featherweight GJ [3] and has similar aims: it is just enough for our purposes (in this case, a translation from System F) but does not "cheat" – valid programs in C$^\sharp$ minor really are valid C$^\sharp$ programs. The differences from Featherweight GJ are as follows:

- There are minor syntactic differences between Java and C$^\sharp$: the use of '`:`' in place of `extends`, and `base` in place of `super`. Methods must be declared `virtual` explicitly, and are overridden explicitly using the keyword `override`. (In the full language, redeclaration of an inherited method as `virtual` introduces a new method without overriding the inherited one. Our subset does not support this.)

- For simplicity we omit bounds on type parameters.

- We include a separate rule for subsumption instead of including subtyping judgments in multiple rules.

- The typing rules for casts are simpler because (a) C$^\sharp$ supports exact run-time types, and (b) our use of a big-step evaluation relation avoids the need for the "stupid cast" rule used by Featherweight GJ [3, §2.2].

For readers unfamiliar with the work on GJ we summarise the language here; for more details see [3].

- A **type** (ranged over by $T$, $U$ and $V$) is either a formal type parameters (ranged over by $X$ and $Y$) or the type instantiation of a named class (ranged over by $N$) written $C$`<`$\overline{T}$`>`. We abbreviate $C$`<>` by $C$.

- A **class definition** $CL$ consists of a class name $C$ with formal type parameters $\overline{X}$, base class (superclass) $N$, constructor definition $K$, instance fields $\overline{T}\ \overline{f}$ and methods $\overline{M}$.

- A **method definition** $M$ consists of a qualifier $Q$, a return type $T$, name $m$, formal type parameters $\overline{X}$, formal argument names $\overline{x}$ and types $\overline{T}$, and body consisting of an expression $e$.

- A **constructor** $K$ simply initializes the fields declared by the class and its superclass; as they are determined solely by the field declarations we will often omit them for clarity. (They are there just so that C$^\sharp$ minor syntax is valid C$^\sharp$ syntax.)

- An **expression** $e$ can be a method parameter $x$, a field access $e.f$, the invocation of a virtual method at some type instantiation $e.m$`<`$\overline{T}$`>(`$\overline{e}$`)`, the creation of an object with initial field values `new` $N$`(`$\overline{e}$`)` or a type cast `(`$T$`)`$e$. A **value** is an expression in canonical form.

- A **class table** $CT$ maps class names to class definitions. The distinguished class `object` is not listed in the table and is dealt with specially. A C$^\sharp$ minor program '$e$ **in** $CT$' consists of an expression $e$ in the presence of a class table $CT$.

A typing judgment $CT; \Delta; \Gamma \vdash e : T$ states that "In the context of a class table $CT$, type parameters $\Delta$ and environment $\Gamma$ mapping variables to types, the expression $e$ has type $T$. We usually omit $CT$ from the context, though to be precise it should appear in all the judgment forms and helper definitions of Figure 2.

The judgment $e \Downarrow^{CT} v$ states that "Under the class table $CT$ expression $e$ evaluates to $v$." As with typing judgments we often omit $CT$ when it is apparent from the context.

**Property 2 (C$^\sharp$ minor evaluation preserves typing).** *Suppose $CT; \Delta; \Gamma \vdash e : T$ and $CT; \Delta; \emptyset \vdash v_i : T_i$ for each $x_i{:}T_i$ in $\Gamma$. If $e[v_i/x_i] \Downarrow^{CT} v$ then $CT; \Delta; \emptyset \vdash v : T$.*

## 4 Transposing F to C$^\sharp$

In outline, the translation from System F to C$^\sharp$ minor works as follows. There are three main aspects: support for functions, support for recursion, and support for polymorphism.

### 4.1 Functions

Function types in System F are translated into instantiations of a special class `Arrow` in C$^\sharp$, whose signature is

```
class Arrow<X,Y> {
  public virtual Y app(X x);
}
```

Writing $A^\star$ for the translation of $A$, we have $(A \to B)^\star =$ `Arrow<`$A^\star, B^\star$`>`.

Function application is simply invocation of the `app` method. For example, $x\ y$ translates to `x.app(y)`.

Function values, *i.e.* $\lambda$-terms, are translated into *closures*, where a closure value is represented by an instance of a class extending an `Arrow` type whose method `app` implements the body of the function and whose fields contain the free variables of the function. In general, functions contain free *type* variables in addition to free term variables, so we must close over these too. This is done by parameterizing the closure class on the type variables in the context. For example, the translation of $\lambda x{:}X \to Y.x\ y$ with $y{:}X$ would be `new C<X,Y>(y)` with

```
class C<X,Y> : Arrow<Arrow<X,Y>,Y> {
  X y;
  public C(X y) { this.y = y; }
  public override Y app(Arrow<X,Y> x) {
    return x.app(this.y);
  }
}
```

**Syntax:**

$$\begin{array}{rlcl}
\text{(types)} & A, B & ::= & X \mid A \to B \mid \forall X.A \\
\text{(terms)} & M, N & ::= & x \mid M\ N \mid \mathtt{rec}\ y(x{:}A){:}B.M \mid M\ A \mid \Lambda X.V \\
\text{(values)} & V, W & ::= & \mathtt{rec}\ y(x{:}A){:}B.M \mid \Lambda X.V
\end{array}$$

**Typing:**

$$\frac{}{\Delta; E \vdash x : E(x)} \qquad \frac{\Delta; E \vdash M : A \to B \qquad \Delta; E \vdash N : A}{\Delta; E \vdash M\ N : B} \qquad \frac{\Delta; E, x : A, y : A \to B \vdash M : B}{\Delta; E \vdash \mathtt{rec}\ y(x{:}A){:}B.M : A \to B}$$

$$\frac{\Delta; E \vdash M : \forall X.B \qquad \Delta \vdash A}{\Delta; E \vdash M\ A : [A/X]B} \qquad \frac{\Delta, X; E \vdash V : A}{\Delta; E \vdash \Lambda X.V : \forall X.A}$$

**Evaluation:**

$$\frac{M \Downarrow \Lambda X.V}{M\ A \Downarrow [A/X]V} \qquad \frac{}{V \Downarrow V} \qquad \frac{M \Downarrow \mathtt{rec}\ y(x{:}A){:}B.M' \qquad N \Downarrow V \qquad [\mathtt{rec}\ y(x{:}A){:}B.M'/y,\ V/x]M' \Downarrow W}{M\ N \Downarrow W}$$

Figure 1: Syntax and semantics of System F with recursion

## 4.2 Recursion

Recursion in our extended version of System F is translated into self-reference through `this` in C$^\sharp$. For example, the translation of `rec` $y(x{:}X){:}X.y(x)$ would be `new C<X>()` with

```
class C<X> : Arrow<X,X> {
  public override X app(X x) {
    return this.app(x);
  }
}
```

## 4.3 Polymorphism

Ideally, to translate polymorphic types in System F we would use a special `All` class in C$^\sharp$, just as we used `Arrow` to translate function types. But this would require support for first-class type *functions* in C$^\sharp$ (see later).

So instead, multiple `All` classes are used to encode polymorphic types in the source. For example, the type $\forall X.X \to X$ is translated to a class with signature

```
class All_X_XtoX {
  public virtual Arrow<X,X> tyapp<X>();
}
```

Similarly, the type $\forall X.X \to Y$ is translated to a class

```
class All_X_XtoY<Y> {
  public virtual Arrow<X,Y> tyapp<X>();
}
```

Type application translates to invocation of the `tyapp` method with a particular instantiation. For example, $x$ `int` translates to `x.tyapp<int>()`.

Polymorphic values, *i.e.* $\Lambda$-terms, are translated into instances of a class extending an `All` type whose polymorphic method `tyapp` implements the body of the term and whose fields contain its free variables. For example, the term $\Lambda X.\lambda x{:}X.y$ with the polymorphic type just described is translated to `new C1(y)` where

```
class C1<Y> : All_XtoX {
  public C1(Y y) { this.y = y; }
  public override Arrow<X,Y> tyapp<X>() {
    return new C2<X,Y>(y);
  }
}
class C2<X,Y> : Arrow<X,Y> {
  public C2(Y y) { this.y = y; }
  public override Y app(X x) {
    return y;
  }
}
```

There are two difficulties with this scheme that need to be addressed. The first is *naming*: assigning class names to polymorphic types. This can be solved by any consistent "mangling" scheme that respects alpha-equivalence *e.g.* through the use of de Brujn indices.

The second problem is more fundamental: substitution of types for type parameters does not commute with the encoding. That is, we do not have $(A[B/X])^\star = A^\star[B^\star/X]$. For example, consider $\forall X.X \to Y$, translated as shown above to `All_X_XtoY<Y>`, and (applying $Y \mapsto \forall Z.Z$) the type $\forall X.X \to \forall Z.Z$, translated to `All_X_XtoAll_Z_Z`.

There are two ways out. The first, described by Jones [4], is to define conversion functions in the target language between $(A[B/X])^\star$ and $A^\star[B^\star/X]$. We instead follow a method used by Läufer and Odersky [7] which ensures that substitution commutes with translation. Instead of defining a separate `All` class for every polymorphic type, parameterize the `All` classes so that each captures a *family* of polymorphic types sharing the same pattern of occurrences of the bound type variable. For example, the translation of $\forall X.X \to \forall Z.Z$ is `All_X_XtoY<All_X_X>`, making use of the class `All_X_XtoY` that captures *all* polymorphic types of the form $\forall X.X \to A$.

## 4.4 The translation in detail

Figure 3 presents the translation. We discuss the translation on types and terms in turn.

**Syntax:**

$$
\begin{array}{rlll}
\text{(class def)} & CL & ::= & \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ N\ \{\overline{T}\ \overline{f}\texttt{;}\ K\ \overline{M}\} \\
\text{(constr def)} & K & ::= & \texttt{public}\ C\texttt{(}\overline{T}\ \overline{f}\texttt{)}\ \texttt{:}\ \texttt{base(}\overline{f}\texttt{)}\ \{\texttt{this.}\overline{f}\ \texttt{=}\ \overline{f}\texttt{;}\} \\
\text{(method qualifier)} & Q & ::= & \texttt{public virtual}\ \mid\ \texttt{public override} \\
\text{(method def)} & M & ::= & Q\ T\ m\texttt{<}\overline{X}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\ \texttt{return}\ e\texttt{;}\} \\
\text{(expression)} & e & ::= & x\ \mid\ e\texttt{.}f\ \mid\ e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)}\ \mid\ \texttt{new}\ N\texttt{(}\overline{e}\texttt{)}\ \mid\ \texttt{(}T\texttt{)}e \\
\text{(value)} & v & ::= & \texttt{new}\ N\texttt{(}\overline{v}\texttt{)} \\
\text{(type)} & T, U, V & ::= & X\ \mid\ N \\
\text{(non-var type)} & N & ::= & C\texttt{<}\overline{T}\texttt{>}
\end{array}
$$

**Subtyping:**

$$
\frac{}{T <: T}
\qquad
\frac{T <: U \qquad U <: V}{T <: V}
\qquad
\frac{}{T <: \texttt{object}}
\qquad
\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ N\ \{\ldots\}}{C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]N}
$$

**Typing:**

$$
\frac{}{\Delta;\Gamma \vdash x : \Gamma(x)}
\qquad
\frac{\Delta;\Gamma \vdash e : N \quad \mathit{fields}(N) = \overline{T}\ \overline{f}}{\Delta;\Gamma \vdash e.f_i : T_i}
\qquad
\frac{\Delta;\Gamma \vdash e : T \quad T <: U}{\Delta;\Gamma \vdash e : U}
\qquad
\frac{\Delta;\Gamma \vdash e : T \quad U <: T}{\Delta;\Gamma \vdash \texttt{(}U\texttt{)}e : U}
$$

$$
\frac{\mathit{fields}(N) = \overline{T}\ \overline{f} \quad \Delta;\Gamma \vdash \overline{e} : \overline{T}}{\Delta;\Gamma \vdash \texttt{new}\ N\texttt{(}\overline{e}\texttt{)} : N}
\qquad
\frac{\Delta;\Gamma \vdash e : N \quad \Delta;\Gamma \vdash \overline{e} : [\overline{T}/\overline{X}]\overline{U} \quad \mathit{mtype}(N.m) = \texttt{<}\overline{X}\texttt{>}\overline{U} \to U}{\Delta;\Gamma \vdash e.m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} : [\overline{T}/\overline{X}]U}
$$

**Method and Class Typing:**

$$
\frac{\mathit{mtype}(N.m)\ \text{not defined} \quad \overline{X}, \overline{Y}; \overline{x}{:}\overline{T}, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>} \vdash e : T}{C\texttt{<}\overline{X}\texttt{>} : N \vdash \texttt{public virtual}\ T\ m\texttt{<}\overline{Y}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\}}
$$

$$
\frac{\mathit{mtype}(N.m) = \texttt{<}\overline{Y}\texttt{>}\overline{T} \to T \quad \overline{X}, \overline{Y}; \overline{x}{:}\overline{T}, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>} \vdash e : T}{C\texttt{<}\overline{X}\texttt{>} : N \vdash \texttt{public override}\ T\ m\texttt{<}\overline{Y}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\}}
$$

$$
\frac{\mathit{fields}(N) = \overline{U}\ \overline{g} \quad C\texttt{<}\overline{X}\texttt{>} : N \vdash \overline{M} \quad K = \texttt{public}\ C\texttt{(}\overline{U}\ \overline{g}, \overline{T}\ \overline{f}\texttt{)}\ \texttt{base(}\overline{g}\texttt{)}\ \{\texttt{this.}\overline{f}\texttt{=}\overline{f}\texttt{;}\ \}}{\vdash \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ N\ \{\overline{T}\ \overline{f}\texttt{;}\ K\ \overline{M}\}}
$$

**Field lookup:**

$$
\frac{}{\mathit{fields}(\texttt{object}) = \{\}}
\qquad
\frac{CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ N\ \{\overline{U}_1\ \overline{f}\texttt{;}\ K\ \overline{M}\} \quad \mathit{fields}([\overline{T}/\overline{X}]N) = \overline{V}\ \overline{g}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{V}\ \overline{g}, [\overline{T}/\overline{X}]\overline{U}\ \overline{f}}
$$

**Method lookup:**

$$
\frac{\begin{array}{l} CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ N\ \{\overline{U}\ \overline{f}\texttt{;}\ K\ \overline{M}\} \\ m\ \text{not defined in}\ \overline{M} \end{array}}{\begin{array}{l} \mathit{mtype}(C\texttt{<}\overline{T}\texttt{>}.m) = \mathit{mtype}([\overline{T}/\overline{X}]N.m) \\ \mathit{mbody}(C\texttt{<}\overline{T}\texttt{>}.m\texttt{<}\overline{U}\texttt{>}) = \mathit{mbody}([\overline{T}/\overline{X}]N.m\texttt{<}\overline{U}\texttt{>}) \end{array}}
\qquad
\frac{\begin{array}{l} CT(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ N\ \{\overline{U}\ \overline{f}\texttt{;}\ K\ \overline{M}\} \\ Q\ V\ m\texttt{<}\overline{Y}\texttt{>(}\overline{V}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\} \in \overline{M} \end{array}}{\begin{array}{l} \mathit{mtype}(C\texttt{<}\overline{T}\texttt{>}.m) = [\overline{T}/\overline{X}](\texttt{<}\overline{Y}\texttt{>}\overline{V} \to V) \\ \mathit{mbody}(C\texttt{<}\overline{T}\texttt{>}.m\texttt{<}\overline{U}\texttt{>}) = \langle \overline{x}, [\overline{T}/\overline{X}, \overline{U}/\overline{Y}]e \rangle \end{array}}
$$

**Evaluation:**

$$
\frac{e \Downarrow \texttt{new}\ N\texttt{(}\overline{v}\texttt{)} \quad \mathit{fields}(N) = \overline{T}\ \overline{f}}{e\,\texttt{.}f_i \Downarrow v_i}
\qquad
\frac{\overline{e} \Downarrow \overline{v}}{\texttt{new}\ N\texttt{(}\overline{e}\texttt{)} \Downarrow \texttt{new}\ N\texttt{(}\overline{v}\texttt{)}}
\qquad
\frac{e \Downarrow \texttt{new}\ N\texttt{(}\overline{v}\texttt{)} \quad N <: T}{\texttt{(}T\texttt{)}\,e \Downarrow \texttt{new}\ N\texttt{(}\overline{v}\texttt{)}}
$$

$$
\frac{e \Downarrow \texttt{new}\ N\texttt{(}\overline{w}\texttt{)} \quad \mathit{mbody}(N.m\texttt{<}\overline{T}\texttt{>}) = \langle \overline{x}, e' \rangle \quad \overline{e} \Downarrow \overline{v} \quad [\overline{v}/\overline{x}, \texttt{new}\ N\texttt{(}\overline{w}\texttt{)}/\texttt{this}]e' \Downarrow v}{e\,\texttt{.}\,m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \Downarrow v}
$$

Figure 2: Syntax and semantics for C$^\sharp$ minor

### 4.4.1 Types

The translation of a type $A$ is denoted by $A^\star$. Type variables map to themselves and arrow types are encoded as discussed above. The translation of polymorphic types makes use of an operation called "lifting". Following [7] we define the $X$-lifting of a C$^\sharp$ type $T$ to be a type $T'$ paired with a substitution $\sigma = \overline{X} \mapsto \overline{T}$ such that

- $\sigma T' = T$; and
- $\overline{T}$ are maximal subterms of $T$ that do not contain $X$.

The translation satisfies two important properties. First, it does not lose any type information, justifying the term "fully type-preserving". Second, it commutes with substitution.

**Property 3.** *$A^\star = B^\star$ iff $A = B$.*

**Property 4.** *$(A[B/X])^\star = A^\star[B^\star/X]$.*

### 4.4.2 Terms

The translation of a System F term $M$ is given by a judgment

$$\Delta; E; F \vdash M : A \leadsto e \textbf{ in } CT$$

in which the target of the translation is a C$^\sharp$ minor program, written '$e$ **in** $CT$', where $e$ is a C$^\sharp$ minor expression and $CT$ is a set of class definitions. When translating the body $M$ of a function value `rec` $y(x{:}A){:}B.M$ it is necessary to distinguish three kinds of variable: the argument $x$, the function $y$ itself, or a free variable. To capture this in the translation the context contains both an ordinary environment $E$ and "function environment" $F$ defined by the grammar

$$F ::= \bullet \mid y(x{:}A){:}B$$

in which the empty environment $\bullet$ is used when translating terms that are not bodies of functions (programs or $\Lambda$-abstractions), and $y(x{:}A){:}B$ denotes an environment used when translating functions in which $x{:}A$ is the argument and $y{:}A \to B$ is the function. A similar split-context technique is used in treatments of typed closure conversion for functional languages [6].

The translation of function and type abstractions makes use of an operation $E \uplus F$ that pushes the bindings from $F$ into $E$. It is defined as follows:

$$\begin{array}{lcl} E \uplus \bullet & = & E \\ E \uplus y(x{:}A){:}B & = & E \uplus \{x{:}A, y : A \to B\} \end{array}$$

Observe that the translation is essentially defined by induction over the structure of the typing derivation of $M$. More precisely, $\Delta; E; F \vdash M : A \leadsto e$ **in** $CT$ is defined when $\Delta; E \uplus F \vdash M : A$.

## 4.5 Correctness

We prove that the translation preserves typability.

**Theorem 1 (Type preservation for open terms).** *Suppose $\Delta = \overline{X}$ and $E = \overline{x} : \overline{A}$.*

1. *If $\Delta; E \vdash M : A_0$ and $\Delta; E; \bullet \vdash M : A_0 \leadsto e$ **in** $CT$ then for any $CT'$ and $T$ such that $fields^{CT \uplus CT'}(T) = \overline{A^\star}\ \overline{x}$ it is the case that $CT \uplus CT'; \Delta; \texttt{this} : T \vdash e : {A_0}^\star$.*
2. *Suppose that $F = y(x{:}A){:}B$. If $\Delta; E \uplus F \vdash M : A_0$ and $\Delta; E; F \vdash M : A_0 \leadsto e$ **in** $CT$ then for any $CT'$ and $T$ such that $fields^{CT \uplus CT'}(T) = \overline{A^\star}\ \overline{x}$ and $CT \uplus CT' \vdash T <: (A \to B)^\star$ it is the case that $CT \uplus CT'; \Delta; \texttt{this} : T, x : A^\star \vdash e : {A_0}^\star$.*

*Proof.* By induction over the structure of the translation derivation. □

This is a rather general statement of the result, dealing with the two varieties of function environment for $F$ and incorporating $CT'$ and $T$ so that the induction hypothesis is sufficiently strong for cases (abs) and (tyabs) to go through. Specializing to closed terms produces a more direct statement.

**Corollary 1 (Type preservation for closed terms).** *If $\vdash M : A \leadsto e$ **in** $CT$ then $CT \vdash e : A^\star$.*

Odersky and Laüfer [7] prove a similar result for their encoding into an extension of Haskell. Of course we would also like to know that the semantics is preserved by the translation, which for our source language means the convergence behaviour of closed terms.

**Theorem 2 (Semantic preservation for closed terms).** *If $\vdash M : A \leadsto e$ **in** $CT$ then $M \Downarrow$ iff $e \Downarrow^{CT}$.*

The proof of this result is work-in-progress and involves constructing a type-indexed logical relation between source and target terms. Recursion is dealt with by relating finite approximants of a recursive function in System F to approximants in the target. See [6] for a similar result for closure-conversion (for predicative polymorphism and without recursion) and [10] for constructions on relations that deal with the impredicative polymorphism of System F.

## 4.6 A partial type-preserving translation

The translation above is not compositional because of the need for global generation of `All` classes. From the point of view of *compiling* System F to C$^\sharp$, this implies that it does not support separate compilation.

Compositionality can be attained if one is willing to sacrifice type preservation. Figure 4 presents an alternative translation that retains the structure of function types but represents all polymorphic types by the same `All` class, relying on a runtime-checked type-cast operation after type application. The type and semantic preservation results from the previous section also go through for this translation.

Alternatively, if C$^\sharp$ supported type *functions* as type parameters, then all polymorphic types can be represented as instantiations of a single class:

```
class All<F,X> {
  public virtual F<X> tyapp<X>();
}
```

## 5 Future work

We have shown how the combination of parameterized classes and polymorphic methods in object-oriented languages provides enough power to accurately encode System F. This helps to characterize why the polymorphic virtual methods present in these languages lead to systems that

**Class definitions:**

$\texttt{class Arrow<}X, Y\texttt{> \{ public virtual } Y \texttt{ app(}X\ x\texttt{) \{return this.app(x);\}\}}$

Infinite family indexed by $X$ and $T$ where $\overline{X} = (\text{type variables free in } T) - X$
$\texttt{class All}^{X}_{T}\texttt{<}\overline{X}\texttt{> \{ public virtual } T \texttt{ tyapp<}X\texttt{>()\{return this.tyapp<}X\texttt{>();\}\}}$

**Types:**

$$\begin{array}{rcl}
X^\star & = & X \\
(A \to B)^\star & = & \texttt{Arrow<}A^\star\texttt{,}B^\star\texttt{>} \\
(\forall X.A)^\star & = & \texttt{All}^{X}_{T}\texttt{<}\overline{T}\texttt{>} \quad \text{where } \mathit{lift}_X A^\star = \langle T, \overline{X} \mapsto \overline{T}\rangle
\end{array}$$

**Terms:**

$$\text{(arg)}\ \frac{}{\Delta; E; y(x{:}A){:}B \vdash x : A \leadsto x \ \textbf{in}\ \{\}} \qquad \text{(rec)}\ \frac{}{\Delta; E; y(x{:}A){:}B \vdash y : A \to B \leadsto \texttt{this}\ \textbf{in}\ \{\}}$$

$$\text{(var)}\ \frac{}{\Delta; x_1{:}A_1, \ldots, x_n{:}A_n; F \vdash x_i : A_i \leadsto \texttt{this.}x_i\ \textbf{in}\ \{\}}$$

$$\text{(app)}\ \frac{\Delta; E; F \vdash M : A \to B \leadsto e\ \textbf{in}\ CT \qquad \Delta; E; F \vdash N : A \leadsto e'\ \textbf{in}\ CT'}{\Delta; E; F \vdash M\ N : B \leadsto e\texttt{.app(}e'\texttt{)}\ \textbf{in}\ CT \cup CT'}$$

$$\text{(tyapp)}\ \frac{\Delta; E; F \vdash M : \forall X.B \leadsto e\ \textbf{in}\ CT}{\Delta; E; F \vdash M\ A : [A/X]B \leadsto e\texttt{.tyapp<}A^\star\texttt{>()}\ \textbf{in}\ CT}$$

$$\text{(abs)}\ \frac{\Delta; E \uplus F; y(x{:}A){:}B \vdash M : B \leadsto e\ \textbf{in}\ CT \quad \Delta; E; F \vdash \overline{x} : \overline{A} \leadsto \overline{e}\ \textbf{in}\ \{\}}{\Delta; E; F \vdash \texttt{rec}\ y(x{:}A){:}B.M : A \to B \leadsto \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)}\ \textbf{in}\ CT \uplus \{C \mapsto CL\}}$$

$E \uplus F = \overline{x}{:}\overline{A}$ and $\Delta = \overline{X}$
$C \notin \mathrm{dom}(CT)$ and $CL =$
$\texttt{class}\ C\texttt{<}\overline{X}\texttt{> : } (A \to B)^\star\ \{$
  $\overline{A}^\star\ \overline{x}\texttt{;}$
  $\texttt{public}\ C\texttt{(}\overline{A}^\star\ \overline{x}\texttt{) \{ this.}\overline{x}\texttt{ = }\overline{x}\texttt{;\}}$
  $\texttt{public override}\ B^\star\ \texttt{app(}A^\star\ x\texttt{)}$
  $\texttt{\{ return } e\texttt{; \}}$
$\}$

$$\text{(tyabs)}\ \frac{\Delta, X; E \uplus F; \bullet \vdash V : A \leadsto e\ \textbf{in}\ CT \quad \Delta; E; F \vdash \overline{x} : \overline{A} \leadsto \overline{e}\ \textbf{in}\ \{\}}{\Delta; E; F \vdash \Lambda X.V : \forall X.A \leadsto \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)}\ \textbf{in}\ CT \uplus \{C \mapsto CL\}}$$

$E \uplus F = \overline{x}{:}\overline{A}$ and $\Delta = \overline{X}$
$C \notin \mathrm{dom}(CT)$ and $CL =$
$\texttt{class}\ C\texttt{<}\overline{X}\texttt{> : } (\forall X.A)^\star\ \{$
  $\overline{A}^\star\ \overline{x}\texttt{;}$
  $\texttt{public}\ C\texttt{(}\overline{A}^\star\ \overline{x}\texttt{) \{ this.}\overline{x}\texttt{ = }\overline{x}\texttt{;\}}$
  $\texttt{public override}\ A^\star\ \texttt{tyapp<}X\texttt{>()}$
  $\texttt{\{ return } e\texttt{;\}}$
$\}$

Figure 3: Fully type-preserving translation

**Class definitions:**

```
class Arrow<X, Y> { public virtual Y app(X x) {return this.app(x);}}

class All { public virtual object tyapp<X>(){return this.tyapp<X>();}}
```

**Types:**

$$\begin{array}{rcl} X^\star & = & X \\ (A \to B)^\star & = & \texttt{Arrow<}A^\star\texttt{,}B^\star\texttt{>} \\ (\forall X.A)^\star & = & \texttt{All} \end{array}$$

**Terms:**

$$\frac{\Delta; E; F \vdash M : \forall X.B \rightsquigarrow e \textbf{ in } CT}{\Delta; E; F \vdash M\ A : [A/X]B \rightsquigarrow (([A/X]B)^\star)(e\texttt{.tyapp<}A^\star\texttt{>()}) \textbf{ in } CT}$$

Figure 4: Partial type-preserving translation

are more expressive than both core ML and systems with only polymorphic classes.

We plan to investigate further a number of aspects:

- The fully type-preserving translation does not make use of run-time types. However, their presence in C$^\sharp$ does mean that it should be easy to extend the translation to deal with extensions of System F with dynamic types.
- We believe that the technique of deferring some type-checking to run-time, as used in Figure 4, can be extended to support an encoding of System $F_\omega$.
- In a sense the main trickiness in the translation is not the "paradigm-shift" between functional and object-oriented but rather between a system supporting structural type equivalence and one based on names (see, for example, [4, 7] where the same issue arises). We have been able to encode function types using a parameterized class, and have suggested three ways of encoding polymorphic types: through multiple classes (breaking separate compilation), through run-time checking (breaking type preservation) and through higher kinds (changing the target language). It would be interesting to study the relative power of structural and nominal type equivalence in a more general framework.

## References

[1] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA).* ACM, October 1998.

[2] R. Cartwright and G. L. Steele. Compatible genericity with run-time types for the Java programming language. In *Object-Oriented Programming: Systems, Languages, Applications (OOPSLA)*, Vancouver, October 1998. ACM.

[3] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1999.

[4] M. P. Jones. First-class polymorphism with type inference. In *ACM Symposium on Principles of Programming Languages*, pages 483–496. ACM, 1997.

[5] A. J. Kennedy and D. Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Programming Language Design and Implementation.* ACM, 2001.

[6] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[7] M. Odersky and K. Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages.* ACM, 1996.

[8] M. Odersky, P. Wadler, G. Bracha, and D. Stoutamire. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages*, pages 146–159. ACM, 1997.

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

[10] A. M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000. A preliminary version appeared in *Proceedings, Second Workshop on Higher Order Operational Techniques in Semantics (HOOTS II), Stanford CA, December 1997*, Electronic Notes in Theoretical Computer Science 10, 1998.

[11] C. V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4), Winter 2000.