CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Transposing F to C♯: Expressivity of parametric polymorphism in an object-oriented language

Andrew Kennedy[1] and Don Syme[1]

[1] *Microsoft Research, Cambridge, U.K.*

SUMMARY

We present a type-preserving translation of System F (the polymorphic lambda calculus) into a forthcoming revision of the C♯ programming language supporting parameterized classes and polymorphic methods. The forthcoming revision of Java in JDK 1.5 also makes a suitable target. We formalize the translation using a subset of C♯ similar to Featherweight Java. We prove that the translation is fully type-preserving and that it preserves behaviour via the novel use of an environment-style semantics for System F.

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 dynamic dispatch for polymorphic methods that supports the encoding of the "first-class polymorphism" found in System F and recent extensions to ML and Haskell.

KEY WORDS: Polymorphism, Java, C♯, System F, Closure Conversion

1. Introduction

Parametric polymorphism is a well-known and well-studied feature of declarative programming languages such as Haskell and ML. 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 [9, 13] and ML [17]. The polymorphic lambda calculus, System F, and its higher-order variant F_ω [15] 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 [14, 3, 4], the forthcoming version of Java based on GJ [2], and the forthcoming version of C♯ [10, 6] also support the type-parameterization of static and virtual methods.

Received 5 Jan 2003

In this paper we pose the question "what's the difference?" and show that the combination of parameterized classes and polymorphic virtual methods is sufficient to encode the first-class notion of polymorphism supported by System F and extensions to ML and Haskell. We demonstrate this through a formal translation of a variant of System F into a subset of C♯ with polymorphism. The translation is fully type-preserving and preserves the evaluation behaviour of terms. When mentioning C♯ in the rest of this paper, we assume the inclusion of generics [10].

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♯, and in Section 4 we present the formal translation and a proof that it is fully type-preserving. Section 5 proves semantic soundness of the translation by first recasting System F in an environment-style semantics. Section 6 concludes.

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 * 'a Tree

fun tmap (f : 'a->'b) (t : 'a Tree) =
  case t of Leaf v => Leaf (f v)
          | Node(left,right) => Node (tmap f left, tmap f right)

val itree = Leaf 5
val stree = tmap Int.toString itree
```

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 (implicit 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♯, where parameterized classes and interfaces play the same role as parameterized types, and polymorphic methods play the role of polymorphic functions.

The corresponding Generic C♯ code suggests that parameterized classes alone 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> {
  public A val;
  public Leaf(A a) { val = a; }
}
class Node<A> : Tree<A> {
  public Tree<A> left;
  public Tree<A> right;
  public Node(Tree<A> l, Tree<A> r) { left = l; right = r; }
}

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

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

class Mapper<A,B> {
  public Tree<B> tmap(Arrow<A,B> f, Tree<A> tree) {
    if (tree is Leaf<A>) {
      return new Leaf<B>(f.apply((tree as Leaf<A>).val));
    } else {
      Node<A> node = tree as Node<A>;
      return new Node<B>(tmap(f, node.left), tmap(f, node.right));
    }
  }
}

Tree<int> itree = new Leaf<int>(5);
Tree<string> stree = new Mapper<int,string>().tmap(new IntToString(), itree);
```

Observe that parameterized datatypes and their constructors are translated directly into parameterized classes, and polymorphic functions are 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 contain free variables. Function values then correspond to object instances of such classes.

One case in such translations requires careful attention: core ML permits let-bound functions to be polymorphic, and closure conversion must take into account the cases where the bodies 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 $\forall\beta.(\alpha$ `->` $\beta)$ `->` β `list` and is used with type arguments `int` and `real`. Conversion to parameterized classes must translate `mark` into a class parameterized by both the type parameter β and the free type variable α:

```
class mark<A,B> : Arrow<Arrow<A,B>,List<B>> {
  List<A> l;
  string apply(Arrow<A,B> h) { ... }
}
class markTwice<A> : ... {
  ... apply() {
      ... new Pair<List<int>,List<double>>(
        new mark<A,int>(l).apply(f),
        new mark<A,float>(l).apply(g)) ...
  }
}
```

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 using polymorphic methods. For example, it is possible to translate a toy fragment of ML into C♯ with parameterized classes only.

1.2. First-class polymorphism

We now present an example illustrating how adding polymorphic methods to a system with parameterized classes leads to a system whose polymorphism is more expressive than found in core ML. Consider the following fragment of C♯:

```
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) { ... }
}

class IntComparer : IComparer<int> { ... }
class StringComparer : IComparer<string> { ... }

void Test(Sorter s, int[] ia, string[] sa) {
  s.Sort<int>(ia, new IntComparer());
  s.Sort<string>(sa, new StringComparer());
}
```

Here we define an interface* `Sorter` representing the type of *polymorphic sorters*, and whose single method is polymorphic in the element type of the 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 [9, 13] and ML [17] support a similar notion. For example, the above example can be rendered 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
```

*We could have used an abstract class in place of the interface

```
  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 array * ('a*'a->order) -> unit

(* type error: sorter is applied to two different kinds of arrays *)
fun test (sorter: 'a sorter, ia : int array, sa : string array) =
(
  sorter (ia, Int.compare);
  sorter (sa, String.compare)
)
```

2. System F with recursion

The above examples motivate a more rigorous investigation into the expressivity of polymorphic methods. Our approach is to take a variant of the expressive calculus System F [15] and to show that it can be compiled to Generic C♯ [10, 6], an object-oriented language with both parameterized classes and polymorphic methods. We focus on a translation which is fully type-preserving but which fails to support separate compilation, and briefly discuss a simpler translation which is only partially type-preserving but which supports separate compilation.

We consider an extension of System F with recursion and a call-by-value evaluation order. Its syntax, typing rules and big-step evaluation semantics are presented in Figure 1.

A typing environment Γ has the form $\Gamma = \overline{X}, \overline{x}{:}\overline{A}$ where free type variables in $\overline{A}$ are drawn from $\overline{X}$. A typing judgment $\Gamma \vdash M : A$ should be read "in the context of a typing environment Γ the term M has type A" with free type variables in M and A drawn from Γ. An evaluation judgment $M \Downarrow V$ should be read "closed term M evaluates to produce a closed value V".

We identify types and terms up to renaming of bound variables, and assume that names of variables are chosen so as to be different from names already bound by Γ. The notation $[B/X]A$ denotes the capture-avoiding substitution of B for X in A; likewise for $[B/A]M$ and $[V/x]M$. Observe that

- 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 [16] for proofs that these encodings are faithful).

Syntax:

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

Typing:

$$\frac{}{\Gamma, x{:}A \vdash x : A} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \qquad \frac{\Gamma, X \vdash M : A}{\Gamma \vdash \Lambda X.M : \forall X.A}$$

$$\frac{\Gamma \vdash M : \forall X.B}{\Gamma \vdash M\ A : [A/X]B} \qquad \frac{\Gamma, x{:}A, y{:}A \to B \vdash M : B}{\Gamma \vdash \texttt{rec}\ y(x{:}A){:}B.M : A \to B}$$

Evaluation:

$$\frac{M \Downarrow \Lambda X.M' \quad [A/X]M' \Downarrow V}{M\ A \Downarrow V} \qquad \frac{}{V \Downarrow V}$$

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

Figure 1. Syntax and semantics of System F with recursion

- Function values are by default recursive. Sometimes we will use the notation $\lambda x{:}A.M$ as shorthand for the equivalent $\texttt{rec}\ y(x{:}A){:}B.M$ (for fresh y and appropriate B).
- Evaluation does not occur under a Λ, but the body of a type abstraction is in general an arbitrary term; this contrasts with the 'value restriction' of ML [11].

Theorem 1 (System F evaluation preserves typing) *Suppose that* $\overline{X}, \overline{x}{:}\overline{A} \vdash M : A$ *and* $\vdash \overline{V} : [\overline{B}/\overline{X}]\overline{A}$. *If* $[\overline{V}/\overline{x}, \overline{B}/\overline{X}]M \Downarrow V$ *then* $\vdash V : [\overline{B}/\overline{X}]A$.

Proof. By induction over the structure of the evaluation derivation. □

3. C♯ minor

Our target language 'C♯ minor' is a small, purely-functional subset of Generic C♯ [10, 6]. Its syntax, typing rules and big-step evaluation semantics are presented in Figures 2 and 3. This formalisation is based on Featherweight GJ [8] and has similar aims: it is just enough for our purposes (in our case, a translation from System F) but does not "cheat" – valid programs in C♯ minor really are valid C♯ programs. The differences from Featherweight GJ are as follows:

- There are minor syntactic differences between Java and C♯: 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.
- We omit (down)cast expressions. These are considered in Section 4.7.

Like Featherweight GJ, the language does not include object identity and encapsulated state, which arguably are defining features of the object-oriented programming paradigm. However, it does include dynamic method dispatch, another defining feature of object-oriented programming, and the crucial ingredient in our translation from System F. Moreover, it is clear that the same translation works for larger subsets of C♯ (*e.g.* see [1] for a subset of non-generic Java that includes aliasing and state, which could be extended with parametric polymorphism).

For readers unfamiliar with the work on Featherweight GJ we summarise the language here; for more details see [8].

- A **type** (ranged over by T and U) is either a formal type parameter (ranged over by X and Y) or the type instantiation of a class (ranged over by C) written $C\texttt{<}\overline{T}\texttt{>}$ and ranged over by I. We abbreviate $C\texttt{<>}$ by C.
- A **class definition** cd consists of a class name C with formal type parameters $\overline{X}$, base class (superclass) $\underline{I}$, constructor definition kd, instance fields $\overline{T}\ \overline{f}$ and methods $\overline{md}$. Method names in $\overline{md}$ must be distinct *i.e.* there is no support for overloading.
- A **method definition** md consists of a method 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 **method qualifier** Q is either `public virtual`, denoting a new publically-accessible method that can be inherited or overridden in subclasses, or `public override`, denoting a method that overrides a method of the same name and type signature in the superclass.
- A **constructor** kd simply initializes the fields declared by the class and its superclass. Sometimes we omit constructors, as they are uniquely determined by the field declarations and superclass.
- An **expression** e can be a method parameter x, a field access $e\texttt{.}f$, the invocation of a virtual method at some type instantiation $e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)}$ or the creation of an object with initial field values $\texttt{new}\ I\texttt{(}\overline{e}\texttt{)}$. We abbreviate $e\texttt{.}m\texttt{<>(}\overline{e}\texttt{)}$ to $e\texttt{.}m\texttt{(}\overline{e}\texttt{)}$.

- A **value** v is a fully-evaluated expression, and has the form `new` $I(\overline{v})$.
- A **class table** $\mathcal{D}$ maps class names to class definitions. The distinguished class `object` is not listed in the table and is dealt with specially.

A typing environment E has the form $E = \overline{X}, \overline{x}{:}\overline{T}$ where free type variables in $\overline{T}$ are drawn from $\overline{X}$. Judgment forms are as follows:

- A typing judgment $E \vdash e : T$ states that "In the context of a typing environment E, the expression e has type T" with type variables in e and T drawn from E.
- A method well-formedness judgment $\vdash md\ ok\ in\ C\texttt{<}\overline{X}\texttt{>}$ states that "method definition md is valid in class $C\texttt{<}\overline{X}\texttt{>}$."
- A class well-formedness judgment $\vdash cd\ ok$ states that "class definition cd is valid".
- The judgment $e \Downarrow v$ states that "closed expression e evaluates to closed value v."

All of the judgment forms and helper definitions of Figures 2 and 3 assume a class table $\mathcal{D}$. When we wish to be more explicit, we annotate judgments and helpers with $\mathcal{D}$. We say that $\mathcal{D}$ is a *valid* class table if $\vdash^{\mathcal{D}} cd\ ok$ for each class definition cd in $\mathcal{D}$.

Theorem 2 (C♯ minor evaluation preserves typing) *Suppose that $\mathcal{D}$ is a valid class table, $\overline{X}, \overline{x}{:}\overline{T} \vdash^{\mathcal{D}} e : T$ and $\vdash^{\mathcal{D}} \overline{v} : [\overline{U}/\overline{X}]\overline{T}$. If $[\overline{v}/\overline{x}, \overline{U}/\overline{X}]e \Downarrow^{\mathcal{D}} v$ then $\vdash^{\mathcal{D}} v : [\overline{U}/\overline{X}]T$.*

Proof. By induction over the structure of the evaluation relation. □

4. Transposing F to C♯

In outline, the translation from System F to C♯ 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 parameterized class `Arrow` in C♯, whose definition is

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

Writing $A^\star$ for the translation of A, we have $(A \to B)^\star$ = `Arrow<`$A^\star, B^\star$`>`. In the full C♯ language, we would make the `app` method `abstract`. Given the lack of abstract methods in C♯ minor, we instead define a looping body; it is a property of the translation that only methods overriding `app` will be invoked.

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

Function values, *i.e.* λ-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

Syntax:

(class def)	cd	::=	$\texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\overline{T}\ \overline{f}\texttt{;}\ kd\ \overline{md}\}$
(constr def)	kd	::=	$\texttt{public}\ C\texttt{(}\overline{T}\ \overline{f}\texttt{)}\ \texttt{:}\ \texttt{base(}\overline{f}\texttt{)}\ \{\texttt{this.}\overline{f}\ \texttt{=}\ \overline{f}\texttt{;}\}$
(method qualifier)	Q	::=	$\texttt{public virtual} \mid \texttt{public override}$
(method def)	md	::=	$Q\ T\ m\texttt{<}\overline{X}\texttt{>(}\overline{T}\ \overline{x}\texttt{)}\ \{\ \texttt{return}\ e\texttt{;}\}$
(expression)	e	::=	$x \mid e\texttt{.}f \mid e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} \mid \texttt{new}\ I\texttt{(}\overline{e}\texttt{)}$
(value)	v, w	::=	$\texttt{new}\ I\texttt{(}\overline{v}\texttt{)}$
(type)	T, U	::=	$X \mid I$
(instantiated type)	I	::=	$C\texttt{<}\overline{T}\texttt{>}$
(typing environment)	E	::=	$\overline{X}, \overline{x} : \overline{T}$

Subtyping:

$$\frac{}{T <: T} \qquad \frac{T_1 <: T_2 \quad T_2 <: T_3}{T_1 <: T_3} \qquad \frac{}{X <: \texttt{object}} \qquad \frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\ldots\}}{C\texttt{<}\overline{T}\texttt{>} <: [\overline{T}/\overline{X}]I}$$

Typing:

$$\text{(ty-var)}\ \frac{}{E, x{:}T \vdash x : T} \qquad \text{(ty-fld)}\ \frac{E \vdash e : I \quad \mathit{fields}(I) = \overline{T}\ \overline{f}}{E \vdash e.f_i : T_i}$$

$$\text{(ty-sub)}\ \frac{E \vdash e : T \quad T <: U}{E \vdash e : U} \qquad \text{(ty-new)}\ \frac{\mathit{fields}(I) = \overline{T}\ \overline{f} \quad E \vdash \overline{e} : \overline{T}}{E \vdash \texttt{new}\ I\texttt{(}\overline{e}\texttt{)} : I}$$

$$\text{(ty-meth)}\ \frac{E \vdash e : I \quad E \vdash \overline{e} : [\overline{T}/\overline{X}]\overline{U} \quad \mathit{mtype}(I.m) = \texttt{<}\overline{X}\texttt{>}\overline{U} \to U}{E \vdash e\texttt{.}m\texttt{<}\overline{T}\texttt{>(}\overline{e}\texttt{)} : [\overline{T}/\overline{X}]U}$$

Method and Class Typing:

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

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

$$\frac{\begin{array}{l}\mathit{fields}(I) = \overline{U}\ \overline{g} \qquad \overline{f}\ \text{and}\ \overline{g}\ \text{disjoint} \\ \vdash \overline{md}\ \mathit{ok\ in}\ C\texttt{<}\overline{X}\texttt{>} \qquad kd = \texttt{public}\ C\texttt{(}\overline{U}\ \overline{g}, \overline{T}\ \overline{f}\texttt{)}\ \texttt{base(}\overline{g}\texttt{)}\ \{\texttt{this.}\overline{f}\texttt{=}\overline{f}\texttt{;}\ \}\end{array}}{\vdash \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\overline{T}\ \overline{f}\texttt{;}\ kd\ \overline{md}\}\ ok}$$

Figure 2. Syntax and typing rules for C♯ minor

Evaluation:

$$\text{(e-fld)}\ \frac{e \Downarrow \texttt{new}\ I\texttt{(}\overline{v}\texttt{)} \quad \mathit{fields}(I) = \overline{T}\ \overline{f}}{e\texttt{.}f_i \Downarrow v_i} \qquad \text{(e-new)}\ \frac{\overline{e} \Downarrow \overline{v}}{\texttt{new}\ I\texttt{(}\overline{e}\texttt{)} \Downarrow \texttt{new}\ I\texttt{(}\overline{v}\texttt{)}}$$

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

Field lookup:

$$\frac{}{\mathit{fields}(\texttt{object}) = \{\}} \qquad \frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\overline{U_1}\ \overline{f_1}\texttt{;}\ \mathit{kd}\ \overline{\mathit{md}}\} \quad \mathit{fields}([\overline{T}/\overline{X}]I) = \overline{U_2}\ \overline{f_2}}{\mathit{fields}(C\texttt{<}\overline{T}\texttt{>}) = \overline{U_2}\ \overline{f_2}, [\overline{T}/\overline{X}]\overline{U_1}\ \overline{f_1}}$$

Method lookup:

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\overline{U}\ \overline{f}\texttt{;}\ \mathit{kd}\ \overline{\mathit{md}}\} \qquad m \text{ not defined in } \overline{\mathit{md}}}{\begin{array}{c}\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = \mathit{mtype}([\overline{T_1}/\overline{X}]I.m) \\ \mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \mathit{mbody}([\overline{T_1}/\overline{X}]I.m\texttt{<}\overline{T_2}\texttt{>})\end{array}}$$

$$\frac{\mathcal{D}(C) = \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ I\ \{\overline{U_1}\ \overline{f}\texttt{;}\ \mathit{kd}\ \overline{\mathit{md}}\} \qquad Q\ U_2\ m\texttt{<}\overline{Y}\texttt{>(}\overline{U_2}\ \overline{x}\texttt{)}\ \{\texttt{return}\ e\texttt{;}\} \in \overline{\mathit{md}}}{\begin{array}{c}\mathit{mtype}(C\texttt{<}\overline{T_1}\texttt{>}.m) = [\overline{T_1}/\overline{X}](\texttt{<}\overline{Y}\texttt{>}\overline{U_2} \to U_2) \\ \mathit{mbody}(C\texttt{<}\overline{T_1}\texttt{>}.m\texttt{<}\overline{T_2}\texttt{>}) = \langle \overline{x}, [\overline{T_1}/\overline{X}, \overline{T_2}/\overline{Y}]e \rangle\end{array}}$$

Figure 3. Evaluation rules and helper definitions for C♯ minor

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 $\lambda x{:}Y \to X.x\ y$ with $y{:}Y$ would be `new C<X,Y>(y)` with

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

4.2. Recursion

Recursion in our extended version of System F is translated into self-reference through `this` in C♯. 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 parameterized `All` class in C♯, just as we used `Arrow` to translate function types. But this would require support for first-class type *functions* in C♯ such as the `F` parameter in the following definition:

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

Instead, we use multiple `All` classes to encode polymorphic types in the source. For example, the type $\forall X.X \to X$ is translated to a class

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

Again, we would use an abstract method for `tyapp` in the full C♯ language.

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

Polymorphic values, *i.e.* Λ-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. Again, the class must be parameterized by type variables in the context. For example, the term $\Lambda X.\lambda x{:}X.y$ with $y{:}Y$ is translated to `new C1<Y>(y)` where

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

Global class definitions ($\mathcal{G}$):

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

(Infinite family where freevars$(T) \setminus \{X\} = \overline{X}$ and identified up to renaming of $X, \overline{X}$)

$\texttt{class All}_{(\overline{X})\,T}\texttt{<}\overline{X}\texttt{>}\,\{\,\texttt{public virtual}\ T\ \texttt{tyapp<}X\texttt{>()\{return this.tyapp<}X\texttt{>();\}\}}$

Types:

$$\begin{array}{rcll} X^\star & = & X & \\ (A \to B)^\star & = & \texttt{Arrow<}A^\star\texttt{,}B^\star\texttt{>} & \\ (\forall X.A)^\star & = & \texttt{All}_{(\overline{X})\,T}\texttt{<}\overline{T}\texttt{>} & \text{where } X\text{-lifting of } A^\star \text{ is } \langle(\overline{X})\,T, \overline{T}\rangle \end{array}$$

Figure 4. Fully type-preserving translation: types

There are two difficulties with this scheme that need to be addressed. First, we must assign class names to polymorphic types. This can be done using a consistent "mangling" scheme that respects alpha-equivalence through the use of de Bruijn indices. For example, the type $\forall X.X \to (\forall Y.Y \to X)$ could be given the name `All!0toAll!0to!1` where `!`n denotes a type variable with de Bruijn index n.

The second problem is more fundamental: substitution of types for type parameters does not commute with the encoding. That is, we do not have $([B/X]A)^\star = [B^\star/X]A^\star$. For example, consider $\forall X.X \to Y$, translated as shown above to `All_X_XtoY<Y>`, and (substituting $\forall Z.Z$ for Y) 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 [9], is to define conversion functions in the target language between $([B/X]A)^\star$ and $[B^\star/X]A^\star$, and to apply the conversion function immediately after invoking the `tyapp` method in the translation of type application. We instead follow a method used by Läufer and Odersky [13] which ensures by definition 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<Y>` that captures *all* polymorphic types of the form $\forall X.X \to A$.

In the next two sections we formalize this translation, first for types and then for terms.

4.4. Translation of types

The translation of a type A is denoted by $A^\star$ and is defined in Figure 4. The class table $\mathcal{G}$ contains global definitions used in the translation. Type variables map to themselves and arrow types are encoded as discussed above. The translation of polymorphic types makes use of an operation that Odersky and Läufer call "lifting".

Definition 1 (Lifting) *The X-lifting of a C♯ type T is a pair $\langle(\overline{X})U, \overline{T}\rangle$ in which $(\overline{X})U$ is the abstracting out of maximal subterms $\overline{T}$ of T that do not contain X, replacing the subterms by type variables $\overline{X}$ such that $T = [\overline{T}/\overline{X}]U$.*

For example, the `X`-lifting of the type `Arrow<Arrow<X,Y>,Arrow<Y,Y>>` is the type abstraction (`Z1, Z2`)`Arrow<Arrow<X,Z1>,Z2>` together with the types `Y` and `Arrow<Y,Y>` which when substituted for variables `Z1` and `Z2` produce the original type.

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

Lemma 1. *$A^\star = B^\star$ iff $A = B$.*

Proof. Easy induction on structure of types, using the identification of `All` types up to renaming of type variables. □

Second, it commutes with substitution.

Lemma 2. *$([B/X]A)^\star = [B^\star/X]A^\star$.*

Proof. See [13]. □

4.5. Translation of terms

Figure 5 defines the translation of terms. The translation of a System F term M is given by a judgment

$$\Gamma; \psi \vdash M : A \rightsquigarrow e \text{ in } \mathcal{D}$$

which should be read "Under the context of typing environment Γ and function environment ψ, term M with type A translates to an expression e and additional class definitions $\mathcal{D}$". In the translation it is assumed that rules (tr-abs) and (tr-tyabs) generate fresh class names away from those in the global table $\mathcal{G}$.

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 of the function. To capture this in the translation the context contains both an ordinary typing environment Γ and "function environment" ψ defined by the grammar

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

in which the empty environment $\bullet$ is used when translating expressions that are not bodies of functions (closed terms or Λ-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 [12].

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

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

$$\text{(tr-argvar)}\ \frac{}{\Gamma; y(x{:}A){:}B \vdash x : A \rightsquigarrow x \ \mathsf{in}\ \{\}}$$

$$\text{(tr-funvar)}\ \frac{}{\Gamma; y(x{:}A){:}B \vdash y : A \to B \rightsquigarrow \texttt{this} \ \mathsf{in}\ \{\}}$$

$$\text{(tr-var)}\ \frac{}{\overline{X}, x_1{:}A_1, \ldots, x_n{:}A_n; \psi \vdash x_i : A_i \rightsquigarrow \texttt{this.}x_i \ \mathsf{in}\ \{\}}$$

$$\text{(tr-app)}\ \frac{\Gamma; \psi \vdash M : A \to B \rightsquigarrow e \ \mathsf{in}\ \mathcal{D} \qquad \Gamma; \psi \vdash N : A \rightsquigarrow e' \ \mathsf{in}\ \mathcal{D}'}{\Gamma; \psi \vdash M\ N : B \rightsquigarrow e\texttt{.app(}e'\texttt{)} \ \mathsf{in}\ \mathcal{D} \cup \mathcal{D}'}$$

$$\text{(tr-tyapp)}\ \frac{\Gamma; \psi \vdash M : \forall X.B \rightsquigarrow e \ \mathsf{in}\ \mathcal{D}}{\Gamma; \psi \vdash M\ A : [A/X]B \rightsquigarrow e\texttt{.tyapp<}A^\star\texttt{>()} \ \mathsf{in}\ \mathcal{D}}$$

$$\text{(tr-abs)}\ \frac{\Gamma \uplus \psi; y(x{:}A){:}B \vdash M : B \rightsquigarrow e \ \mathsf{in}\ \mathcal{D} \quad \Gamma; \psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \ \mathsf{in}\ \{\}}{\Gamma; \psi \vdash \texttt{rec}\ y(x{:}A){:}B.M : A \to B \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \ \mathsf{in}\ \{C \mapsto cd\} \cup \mathcal{D}}$$

$\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}$
$C \notin \mathrm{dom}(\mathcal{D})$ and $cd =$
$\texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : (A \to B)^\star\ \{$
  $\overline{A}^\star\ \overline{x}\texttt{;}$
  $\texttt{public}\ C\texttt{(}\overline{A}^\star\ \overline{x}\texttt{)}\ \{\ \texttt{this.}\overline{x}\ \texttt{=}\ \overline{x}\texttt{;}\}$
  $\texttt{public override}\ B^\star\ \texttt{app(}A^\star\ x\texttt{)}\ \{\ \texttt{return}\ e\texttt{;}\ \}$
$\}$

$$\text{(tr-tyabs)}\ \frac{X, \Gamma \uplus \psi; \bullet \vdash M : A \rightsquigarrow e \ \mathsf{in}\ \mathcal{D} \quad \Gamma; \psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \ \mathsf{in}\ \{\}}{\Gamma; \psi \vdash \Lambda X.M : \forall X.A \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \ \mathsf{in}\ \{C \mapsto cd\} \cup \mathcal{D}}$$

$\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}$
$C \notin \mathrm{dom}(\mathcal{D})$ and $cd =$
$\texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : (\forall X.A)^\star\ \{$
  $\overline{A}^\star\ \overline{x}\texttt{;}$
  $\texttt{public}\ C\texttt{(}\overline{A}^\star\ \overline{x}\texttt{)}\ \{\ \texttt{this.}\overline{x}\ \texttt{=}\ \overline{x}\texttt{;}\}$
  $\texttt{public override}\ A^\star\ \texttt{tyapp<}X\texttt{>()}\ \{\ \texttt{return}\ e\texttt{;}\}$
$\}$

Figure 5. Fully type-preserving translation: terms

Observe that the translation is essentially defined by induction over the structure of the typing derivation of M. More precisely, $\Gamma;\psi \vdash M : A \rightsquigarrow e \text{ in } \mathcal{D}$ is defined when $\Gamma \uplus \psi \vdash M : A$.

4.6. Type preservation

We prove that the translation preserves types.

Theorem 3 (Type preservation for open terms) *Suppose* $\Gamma;\psi \vdash M : A_0 \rightsquigarrow e_0 \text{ in } \mathcal{D}_0$ *with* $\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}$ *and* $\Gamma;\psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \text{ in } \{\}$. *Then (a)* $\mathcal{D}_0 \cup \mathcal{G}$ *is a valid class table, and (b) for any valid class table* $\mathcal{D} \supseteq \mathcal{D}_0 \cup \mathcal{G}$ *and typing environment* E *of form* $E = \overline{X}, \ldots$ *such that* $E \vdash^{\mathcal{D}} \overline{e} : \overline{A}^\star$ *it is the case that* $E \vdash^{\mathcal{D}} e_0 : A_0^\star$.

Proof. It is easy to check that $\mathcal{G}$ is a valid class table. The proof then proceeds by induction over the structure of the translation derivation.

Cases (tr-argvar), (tr-funvar), (tr-var)

Result follows immediately from assumptions about $\mathcal{D}$ and E.

Case (tr-app)

We have a derivation that ends with the rule

$$\frac{\begin{array}{ll}\Gamma;\psi \vdash M : A \to B \rightsquigarrow e_1 \text{ in } \mathcal{D}_1 & (1)\\ \Gamma;\psi \vdash N : A \rightsquigarrow e_2 \text{ in } \mathcal{D}_2 & (2)\end{array}}{\Gamma;\psi \vdash M\ N : B \rightsquigarrow e_1\texttt{.app(}e_2\texttt{)} \text{ in } \mathcal{D}_1 \cup \mathcal{D}_2}$$

Suppose $\mathcal{D} \supseteq \mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{G}$. Applying the induction hypothesis to (1), we have that $\mathcal{D}_1 \cup \mathcal{G}$ is valid, and $E \vdash^{\mathcal{D}} e_1 : (A \to B)^\star$. Applying the induction hypothesis to (2), we have that $\mathcal{D}_2 \cup \mathcal{G}$ is valid, and $E \vdash^{\mathcal{D}} e_2 : A^\star$. For part (a) of the theorem, we use the fact (proof omitted) that if both $\mathcal{D}_1 \cup \mathcal{G}$ and $\mathcal{D}_2 \cup \mathcal{G}$ are valid (for disjoint $\mathcal{D}_1$ and $\mathcal{D}_2$), then so is $\mathcal{D}_1 \cup \mathcal{D}_2 \cup \mathcal{G}$. For part (b), the translation on types gives $(A \to B)^\star = \texttt{Arrow<}A^\star\texttt{,}B^\star\texttt{>}$. Then we can use rule (ty-meth) to obtain

$$\frac{E \vdash^{\mathcal{D}} e_1 : \texttt{Arrow<}A^\star\texttt{,}B^\star\texttt{>} \quad E \vdash^{\mathcal{D}} e_2 : A^\star \quad \mathit{mtype}_{\mathcal{D}}(\texttt{Arrow<}A^\star\texttt{,}B^\star\texttt{>.app}) = \texttt{<>}A^\star \to B^\star}{E \vdash^{\mathcal{D}} e_1\texttt{.app(}e_2\texttt{)} : B^\star}$$

as required.

Case (tr-tyapp)

We have a derivation that ends with the rule

$$\frac{\Gamma;\psi \vdash M : \forall X.B \rightsquigarrow e \text{ in } \mathcal{D}_0}{\Gamma;\psi \vdash M\ A : [A/X]B \rightsquigarrow e\texttt{.tyapp<}A^\star\texttt{>()} \text{ in } \mathcal{D}_0}$$

Applying the induction hypothesis to the premise of this rule, we have $E \vdash^{\mathcal{D}} e : (\forall X.B)^\star$. From the translation on polymorphic types given in Figure 4 we have $(\forall X.B)^\star = \texttt{All}_{(\overline{Y})U}\texttt{<}\overline{T}\texttt{>}$ where the X-lifting of $B^\star$ is $\langle(\overline{Y})U, \overline{T}\rangle$. The definition of `All` types gives us

$$\mathit{mtype}_{\mathcal{D}}(\texttt{All}_{(\overline{Y})U}\texttt{<}\overline{T}\texttt{>.tyapp}) = \langle X\rangle\texttt{()} \to [\overline{T}/\overline{Y}]U$$

Then from Definition 1 (Lifting) we know that $[\overline{T}/\overline{Y}]U = B^\star$. Finally, we can apply typing rule (ty-meth) to obtain a derivation of

$$E \vdash^{\mathcal{D}} e\texttt{.tyapp<}A^\star\texttt{>()} : [A^\star/X]B^\star$$

and using Lemma 2 we obtain the required result.

Case (tr-abs)

We have a derivation that ends with the rule

$$\frac{\begin{array}{ll}\Gamma \uplus \psi; y(x{:}A){:}B \vdash M : B \rightsquigarrow e \;\mathsf{in}\; \mathcal{D}_1 & (1)\\ \Gamma; \psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \;\mathsf{in}\; \{\} & (2)\end{array}}{\Gamma; \psi \vdash \texttt{rec}\ y(x{:}A){:}B.M : A \to B \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)}\ \mathsf{in}\ \{C \mapsto cd\} \cup \mathcal{D}_1}$$

$$\begin{array}{l}\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}\\ C \notin \mathrm{dom}(\mathcal{D}_1) \text{ and } cd =\\ \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : (A \to B)^\star\ \{\\ \quad \overline{A}^\star\ \overline{x}\texttt{;}\\ \quad \texttt{public override}\ B^\star\ \texttt{app(}A^\star\ x\texttt{)}\ \{\ \texttt{return}\ e\texttt{;}\ \}\ \}\end{array} \qquad (3)$$

For part (a) of the theorem, consider $E' = \{\overline{X}, x{:}A^\star, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>}\}$ and $\Gamma' = \Gamma \uplus \psi$ and $\psi' = y(x{:}A){:}B$ and $\mathcal{D}' = \{C \mapsto cd\} \cup \mathcal{D}_1 \cup \mathcal{G}$. Clearly we have

$$\begin{array}{llll}\Gamma'; \psi' \vdash x : A \rightsquigarrow x \;\mathsf{in}\; \{\} & \text{and} & E' \vdash^{\mathcal{D}'} x : A^\star & \text{by (ty-var)}\\ \Gamma'; \psi' \vdash y : A \to B \rightsquigarrow \texttt{this} \;\mathsf{in}\; \{\} & \text{and} & E' \vdash^{\mathcal{D}'} \texttt{this} : (A \to B)^\star & \text{by (ty-sub)}\\ \Gamma'; \psi' \vdash \overline{x} : \overline{A} \rightsquigarrow \texttt{this.}\overline{x} \;\mathsf{in}\; \{\} & \text{and} & E' \vdash^{\mathcal{D}'} \texttt{this.}\overline{x} : \overline{A^\star} & \text{by (ty-fld)}\end{array}$$

which fulfils the assumptions required to apply the induction hypothesis to (1), to get $E' \vdash^{\mathcal{D}'} e : B^\star$. Together with the fact that $mtype_{\mathcal{D}'}(\texttt{Arrow<}A^\star\texttt{,}B^\star\texttt{>.app}) = \texttt{<>}A^\star \to B^\star$ this gives us method validity for `app` in (3) (see rules in Figure 2) and class validity for cd then follows.

For part (b), from (2) and assumptions on E and $\mathcal{D}$ we have $E \vdash^{\mathcal{D}} \overline{e} : \overline{A^\star}$. We can then apply typing rules (ty-new) and (ty-sub) to obtain $E \vdash^{\mathcal{D}} \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} : (A \to B)^\star$ as required.

Case (tr-tyabs)

We have a derivation ending with the rule

$$\frac{\begin{array}{ll}X, \Gamma \uplus \psi; \bullet \vdash M : A \rightsquigarrow e \;\mathsf{in}\; \mathcal{D}_1 & (1)\\ \Gamma; \psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \;\mathsf{in}\; \{\} & (2)\end{array}}{\Gamma; \psi \vdash \Lambda X.M : \forall X.A \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)}\ \mathsf{in}\ \{C \mapsto cd\} \cup \mathcal{D}_1}$$

$$\begin{array}{l}\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}\\ C \notin \mathrm{dom}(\mathcal{D}_1) \text{ and } cd =\\ \texttt{class}\ C\texttt{<}\overline{X}\texttt{>} : (\forall X.A)^\star\ \{\\ \quad \overline{A}^\star\ \overline{x}\texttt{;}\\ \quad \texttt{public override}\ A^\star\ \texttt{tyapp<}X\texttt{>()}\ \{\ \texttt{return}\ e\texttt{;}\}\ \}\end{array} \qquad (3)$$

Global 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:

$$\frac{\Gamma;\psi \vdash M : \forall X.B \rightsquigarrow e \text{ in } \mathcal{D}}{\Gamma;\psi \vdash M\ A : [A/X]B \rightsquigarrow \texttt{((}([A/X]B)^\star\texttt{)(}e\texttt{.tyapp<}A^\star\texttt{>())} \text{ in } \mathcal{D}}$$

Figure 6. Partial type-preserving translation

For part (a) of the theorem, consider $E' = \{X, \overline{X}, \texttt{this}{:}C\texttt{<}\overline{X}\texttt{>}\}$ and $\Gamma' = X, \Gamma \uplus \psi$ and $\psi' = \bullet$ and $\mathcal{D}' = \{C \mapsto cd\} \cup \mathcal{D}_1 \cup \mathcal{G}$. Clearly we have

$$\Gamma';\psi' \vdash \overline{x} : \overline{A} \rightsquigarrow \texttt{this.}\overline{x} \text{ in } \{\} \quad \text{and} \quad E' \vdash^{\mathcal{D}'} \texttt{this.}\overline{x} : \overline{A}^\star \quad \text{by (ty-fld)}$$

which fulfils the assumptions required to apply the induction hypothesis to (1), to get $E' \vdash^{\mathcal{D}'} e : A^\star$. Together with the fact that $mtype_{\mathcal{D}'}((\forall X.A)^\star\texttt{.tyapp}) = \texttt{<}X\texttt{>()} \to A^\star$ this gives us method validity for `tyapp` in (3) and class validity for cd then follows.

For part (b), from (2) and assumptions on E and $\mathcal{D}$ we have $E \vdash^{\mathcal{D}} \overline{e} : \overline{A}^\star$. We then apply typing rules (ty-new) and (ty-sub) to obtain $E \vdash^{\mathcal{D}} \texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} : (\forall X.A)^\star$ as required. □

This is a rather general formulation of type preservation; specializing to closed terms produces a more direct statement.

Corollary 1 (Type preservation for closed terms) *If* $\vdash M : A \rightsquigarrow e$ in $\mathcal{D}$ *then* $\mathcal{D} \cup \mathcal{G}$ *is a valid class table and* $\vdash^{\mathcal{D}\cup\mathcal{G}} e : A^\star$.

4.7. A partial type-preserving translation

The translation above assumes the existence of an infinite collection of `All` classes. If we wished to *compile* System F to C♯, we would instead generate such classes on demand. However, this breaks separate compilation; or to put it another way, the translation is not compositional. For example, consider separately translating the terms

$$\begin{array}{rl} & f = \lambda x{:}(\forall X.X \to X).x\ A\ V \\ \text{and} & g = \Lambda X.\lambda x{:}X.x \end{array}$$

In System F, f can be applied to g, but separate translation would generate two distinct classes for the type $\forall X.X \to X$ and so the translation of $f\ g$ would be ill-typed.

Syntax:

$$\text{(expression)} \quad e \quad ::= \quad \ldots \mid \texttt{(}T\texttt{)}\,e$$

Typing:

$$\frac{E \vdash e : T}{E \vdash \texttt{(}U\texttt{)}\,e : U}$$

Evaluation:

$$\frac{e \Downarrow \texttt{new}\ I\texttt{(}\overline{v}\texttt{)} \quad I <: T}{\texttt{(}T\texttt{)}\,e \Downarrow \texttt{new}\ I\texttt{(}\overline{v}\texttt{)}}$$

Figure 7. Downcasts for C$^\sharp$ minor

Compositionality and separate compilation can be attained if one is willing to sacrifice full type preservation. Figure 6 presents changes to the 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 syntax and semantics of this cast construct are formalized in Figure 7. Note that "bad" casts simply fail to evaluate; all casts introduced by the translation will succeed.

This new translation on types has the property that if $A = B$ then $A^\star = B^\star$ but not the converse, as witnessed by any pair of distinct System F $\forall$-types.

5. Semantic soundness

In addition to type preservation we would also like to know that it is sound semantically: that it preserves observable *behaviour*. For our variant of System F, we take observable behaviour to be the convergence of closed terms. (If we added constants to the language then evaluation to a literal value could be taken as observation instead; but convergence is enough.) For the translation to preserve behaviour means that if M translates to e then $M \Downarrow$ iff $e \Downarrow$.

Note that the translation is complex enough that translation does *not* commute with evaluation, even disregarding choice of names for classes. That is to say, if M translates to e and $M \Downarrow V$ and $e \Downarrow v$ then it is not necessarily true that V translates to v. Consider, for example, the term $(\Lambda X.\lambda x{:}X.\lambda y{:}X.x)\,A\,V$. If this term is translated into C$^\sharp$ minor it evaluates to a value of the form $\texttt{new}\ C\texttt{<}A^\star\texttt{>(}v\texttt{)}$ but the translation of the evaluated System F term $\lambda y{:}A.\,V$ has the form $\texttt{new}\ C'\texttt{<>()}$.

We prove semantic soundness in two stages. First we recast the semantics of System F in terms of environments and closure values instead of substitutions, and prove that this semantics is equivalent to the original one. Then we show that under the environment-style semantics, translation *does* commute with evaluation, given an appropriate definition of translation for closure values. Combining these two results applied to convergence behaviour of closed terms gives the soundness result we desire.

Syntax:

$$\begin{array}{rrcl} \text{(closure values)} & \gamma & ::= & \langle \rho, V \rangle \\ \text{(environments)} & \rho & ::= & \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\} \end{array}$$

Typing:

$$\frac{\vdash \overline{\gamma} : [\overline{B}/\overline{X}]\overline{A} \qquad \overline{X}, \overline{x}{:}\overline{A} \vdash V : A}{\vdash \langle \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}, V \rangle : [\overline{B}/\overline{X}]A}$$

Evaluation:

$$\text{(e-var)}\ \frac{}{\rho \vdash x \Downarrow_e \rho(x)} \qquad \text{(e-val)}\ \frac{}{\rho \vdash V \Downarrow_e \langle \rho, V \rangle}$$

$$\text{(e-tyapp)}\ \frac{\rho \vdash M \Downarrow_e \langle \rho', \Lambda X.M' \rangle \quad X \mapsto \rho A, \rho' \vdash M' \Downarrow_e \gamma}{\rho \vdash M\ A \Downarrow_e \gamma}$$

$$\text{(e-app)}\ \frac{\rho \vdash M \Downarrow_e \langle \rho', \texttt{rec}\ y(x{:}A){:}B.M' \rangle \quad \rho \vdash N \Downarrow_e \gamma \quad x{:}A \mapsto \gamma, y{:}A \to B \mapsto \langle \rho', \texttt{rec}\ y(x{:}A){:}B.M' \rangle, \rho' \vdash M' \Downarrow_e \gamma'}{\rho \vdash M\ N \Downarrow_e \gamma'}$$

Figure 8. Environment-style semantics of System F with recursion

Figure 8 presents the new semantics. An environment $\rho = \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}$ maps type variables to closed types and value variables to closure values; then a closure value $\langle \rho, V \rangle$ pairs an environment with a System F value whose free type variables and free variables are bound by the environment. Programming languages are sometimes specified in this style (see, for example, the specification of Standard ML [11]) but the application to a polymorphic language in the style of System F appears to be novel.

Evaluation is given by a judgment $\rho \vdash M \Downarrow_e \gamma$ that should be read "Under environment ρ the term M evaluates to closure value γ". The notation ρA, used in rule (e-tyapp), denotes the application on A of the substitution defined by the mapping from type variables to types present in ρ.

Theorem 4 (System F environment-style evaluation preserves typing) *If* $\overline{X}, \overline{x}{:}\overline{A} \vdash M : A$ *and* $\vdash \overline{\gamma} : [\overline{B}/\overline{X}]\overline{A}$ *then* $\{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\} \vdash M \Downarrow_e \gamma$ *implies* $\vdash \gamma : [\overline{B}/\overline{X}]A$.

Proof. Straightforward induction over the structure of the evaluation derivation. □

To relate the environment-style semantics back to the substitution-style semantics we define an operator $[\cdot]^\circ$ that maps closures to closed values and maps environments to substitutions on type variables and value variables.

$$\begin{array}{rcl} \langle \rho, V \rangle^\circ & = & \rho^\circ\, V \\ \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}^\circ & = & [\overline{B}/\overline{X}, \overline{\gamma}^\circ/\overline{x}] \end{array}$$

$$\text{(tr-absclos)}\ \frac{\vdash^{\mathcal{D}} \overline{\gamma} \rightsquigarrow \overline{v} \quad \overline{X}, \overline{x}{:}\overline{A}; y(x{:}A){:}B \vdash M : B \rightsquigarrow e \ \mathsf{in}\ \mathcal{D}' \quad \mathcal{D}' \subseteq \mathcal{D}}{\vdash^{\mathcal{D}} \langle \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}, \texttt{rec}\ y(x{:}A){:}B.M\rangle \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{B}^{\star}\texttt{>(}\overline{v}\texttt{)}}$$

$$\begin{array}{l}\mathcal{D}(C) = \\ \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ (A \to B)^{\star}\ \{ \\ \quad \overline{A}^{\star}\ \overline{x}\texttt{;} \\ \quad \texttt{public}\ C\texttt{(}\overline{A}^{\star}\ \overline{x}\texttt{)}\ \{\ \texttt{this.}\overline{x}\ \texttt{=}\ \overline{x}\texttt{;}\} \\ \quad \texttt{public override}\ B^{\star}\ \texttt{app(}A^{\star}\ x\texttt{)}\ \{\ \texttt{return}\ e\texttt{;}\ \} \\ \}\end{array}$$

$$\text{(tr-tyabsclos)}\ \frac{\vdash^{\mathcal{D}} \overline{\gamma} \rightsquigarrow \overline{v} \quad X, \overline{X}, \overline{x}{:}\overline{A}; \bullet \vdash M : A \rightsquigarrow e \ \mathsf{in}\ \mathcal{D}' \quad \mathcal{D}' \subseteq \mathcal{D}}{\vdash^{\mathcal{D}} \langle \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}, \Lambda X.M\rangle \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{B}^{\star}\texttt{>(}\overline{v}\texttt{)}}$$

$$\begin{array}{l}\mathcal{D}(C) = \\ \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ (\forall X.A)^{\star}\ \{ \\ \quad \overline{A}^{\star}\ \overline{x}\texttt{;} \\ \quad \texttt{public}\ C\texttt{(}\overline{A}^{\star}\ \overline{x}\texttt{)}\ \{\ \texttt{this.}\overline{x}\ \texttt{=}\ \overline{x}\texttt{;}\} \\ \quad \texttt{public override}\ A^{\star}\ \texttt{tyapp<}X\texttt{>()}\ \{\ \texttt{return}\ e\texttt{;}\} \\ \}\end{array}$$

Figure 9. Fully type-preserving translation: closure values

Lemma 3 (Equivalence of two styles of semantics for System F) *$\rho \vdash M \Downarrow_e \gamma$ iff $\rho^{\circ} M \Downarrow \gamma^{\circ}$.*

Proof. By induction over the structure of the evaluation derivations. □

We now show that evaluation under the environment-style semantics commutes with translation. Translation on closure values is defined in Figure 9; the judgment $\vdash^{\mathcal{D}} \gamma \rightsquigarrow v$ should be read "Closure value γ translates to v using classes drawn from $\mathcal{D}$". Unlike the translation on terms, classes are not *generated*; this is a requirement because evaluation of C♯ expressions does not create new classes.

Consider evaluation of the System F term given earlier: we have

$$\vdash (\Lambda X.\lambda x{:}X.\lambda y{:}X.x)\ A\ V \Downarrow_e \langle \{X \mapsto A, x{:}X \mapsto \langle \{\}, V\rangle\}, \lambda y{:}X.x\}\rangle.$$

Translating this closure value using the rules of Figure 9 matches the C♯ minor evaluation of the translation of the original term, as we desired.

As with type preservation, the general statement of the theorem is rather complicated because of the need for a sufficiently strong induction hypothesis. To simplify the statement

we introduce an auxiliary judgment that relates environments for the source language to substitutions on the target language.

Definition 2. *We write* $\rho \triangleright^{\mathcal{D}}_{\Gamma;\psi} S$ *to mean "Under class table* $\mathcal{D}$ *the substitution* S *interprets environment* ρ *for translation context* $\Gamma;\psi$*". For* $\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}$ *it holds when* $S(\overline{X}) = \rho(\overline{X})^\star$ *and* $\Gamma;\psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e}$ in $\mathcal{D}$ *and there exists* $\overline{v}$ *such that* $S(\overline{e}) \Downarrow^{\mathcal{D}} \overline{v}$ *and* $\vdash^{\mathcal{D}} \rho(\overline{x}) \rightsquigarrow \overline{v}$.

The theorem is then as follows.

Theorem 5. *Suppose that* $\Gamma;\psi \vdash M_0 : A_0 \rightsquigarrow e_0$ in $\mathcal{D}_0$. *Then for any valid class table* $\mathcal{D} \supseteq \mathcal{D}_0 \cup \mathcal{G}$, *environment* ρ *and substitution* S *such that* $\rho \triangleright^{\mathcal{D}}_{\Gamma;\psi} S$:
(1) If $\rho \vdash M_0 \Downarrow_e \gamma_0$ *then there exists* v_0 *such that* $Se_0 \Downarrow^{\mathcal{D}} v_0$ *and* $\vdash^{\mathcal{D}} \gamma_0 \rightsquigarrow v_0$.
(2) If $Se_0 \Downarrow^{\mathcal{D}} v_0$ *then there exists* γ_0 *such that* $\rho \vdash M_0 \Downarrow_e \gamma_0$ *and* $\vdash^{\mathcal{D}} \gamma_0 \rightsquigarrow v_0$.

Proof.

Part (1). By induction over the structure of the derivation of $\rho \vdash M_0 \Downarrow_e \gamma_0$.

Case (e-var)

Let $M_0 = x$ so $\rho \vdash x \Downarrow_e \rho(x)$. The result follows immediately from Definition 2.

Case (e-val)

Let $M_0 = \Lambda X.M$ so $\rho \vdash \Lambda X.M \Downarrow_e \langle \rho, \Lambda X.M \rangle$. Suppose $\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}$. The translation derivation must end with the following instance of (tr-tyabs):

$$\frac{\begin{array}{ll} X, \Gamma \uplus \psi; \bullet \vdash M : A \rightsquigarrow e \text{ in } \mathcal{D}' & (1) \\ \Gamma;\psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \text{ in } \{\} & (2) \end{array}}{\begin{array}{l} \Gamma;\psi \vdash \Lambda X.M : \forall X.A \rightsquigarrow \texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \text{ in } \{C \mapsto cd\} \cup \mathcal{D}' \\ \quad C \notin \mathrm{dom}(\mathcal{D}') \text{ and } cd = \\ \quad \texttt{class } C\texttt{<}\overline{X}\texttt{> : (}\forall X.A\texttt{)}^\star \texttt{ \{} \\ \qquad \overline{A}^\star\ \overline{x}\texttt{;} \\ \qquad \texttt{public override } A^\star \texttt{ tyapp<}X\texttt{>() \{ return } e\texttt{;\} \}} \end{array}} \quad (3)$$

From Definition 2 we have $S(\texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)}) = \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}S(\overline{e})\texttt{)}$. Hence by evaluation rule (e-new), premise (2) and Definition 2 we obtain

$$\frac{S(\overline{e}) \Downarrow^{\mathcal{D}} \overline{v}}{\texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}S(\overline{e})\texttt{)} \Downarrow^{\mathcal{D}} \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}\overline{v}\texttt{)}}$$

where $\vdash^{\mathcal{D}} \rho(\overline{x}) \rightsquigarrow \overline{v}$. Now using premise (1) and side-condition (3) from the translation derivation we can apply translation rule (tr-tyabsclos) to get

$$\frac{\vdash^{\mathcal{D}} \rho(\overline{x}) \rightsquigarrow \overline{v} \quad X, \Gamma \uplus \psi; \bullet \vdash M : A \rightsquigarrow e \text{ in } \mathcal{D}'}{\vdash^{\mathcal{D}} \langle \rho, \Lambda X.M \rangle \rightsquigarrow \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}\overline{v}\texttt{)}}$$

as required.

Case (e-tyapp)

Let $M_0 = M\ A$. The evaluation derivation must end with the rule

$$\frac{\begin{array}{ll}\rho \vdash M \Downarrow_e \langle \rho', \Lambda X.M' \rangle & (1)\\ X \mapsto \rho A, \rho' \vdash M' \Downarrow_e \gamma_0 & (2)\end{array}}{\rho \vdash M\ A \Downarrow_e \gamma_0}$$

and the translation derivation must end with the following instance of (tr-tyapp):

$$\frac{\Gamma; \psi \vdash M : \forall X.B \rightsquigarrow e \ \mathsf{in}\ \mathcal{D}_0}{\Gamma; \psi \vdash M\ A : [A/X]B \rightsquigarrow e\texttt{.tyapp<}A^\star\texttt{>()}\ \mathsf{in}\ \mathcal{D}_0}$$

Applying the induction hypothesis to the subderivation ending with (1), there exists some v such that $S(e) \Downarrow^{\mathcal{D}} v$ and $\vdash^{\mathcal{D}} \langle \rho', \Lambda X.M' \rangle \rightsquigarrow v$. Now suppose $\rho' = \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}$. The closure translation must have been derived using rule (tr-tyabsclos):

$$\frac{\vdash^{\mathcal{D}} \overline{\gamma} \rightsquigarrow \overline{v} \quad X, \overline{X}, \overline{x}{:}\overline{A}; \bullet \vdash M' : A \rightsquigarrow e' \ \mathsf{in}\ \mathcal{D}' \quad \mathcal{D}' \subseteq \mathcal{D}}{\vdash^{\mathcal{D}} \langle \rho', \Lambda X.M' \rangle \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{B}^\star\texttt{>(}\overline{v}\texttt{)}}$$

$$\begin{array}{l}\mathcal{D}(C) = \\ \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ (\forall X.A)^\star\ \{ \\ \quad \overline{A}^\star\ \overline{x}\texttt{;} \\ \quad \texttt{public override}\ A^\star\ \texttt{tyapp<}X\texttt{>()}\ \{\ \texttt{return}\ e'\texttt{;}\}\ \}\end{array}$$

It is easy to see that

$$X \mapsto \rho A, \rho' \ \triangleright^{\mathcal{D}}_{X, \overline{X}, \overline{x}:\overline{A}; \bullet}\ [(\rho A)^\star/X, \overline{B}^\star/\overline{X}, \texttt{new}\ C\texttt{<}\overline{B}^\star\texttt{>(}\overline{v}\texttt{)/this}]$$

so we can apply the induction hypothesis to the subderivation ending with (2) to show that there exists some v_0 such that

$$[(\rho A)^\star/X, \overline{B}^\star/\overline{X}, \texttt{new}\ C\texttt{<}\overline{B}^\star\texttt{>(}\overline{v}\texttt{)/this}]e' \Downarrow^{\mathcal{D}} v_0.$$

and $\vdash^{\mathcal{D}} \gamma_0 \rightsquigarrow v_0$. Finally we can derive the evaluation of $S(e\texttt{.tyapp<}A^\star\texttt{>()}) = Se\texttt{.tyapp<}(\rho A)^\star\texttt{>()}$ by rule (e-meth):

$$\frac{\begin{array}{l}Se \Downarrow^{\mathcal{D}} \texttt{new}\ C\texttt{<}\overline{B}^\star\texttt{>(}\overline{v}\texttt{)} \\ mbody_{\mathcal{D}}(C\texttt{<}\overline{B}^\star\texttt{>.tyapp<}(\rho A)^\star\texttt{>}) = \langle (), [(\rho A)^\star/X, \overline{B}^\star/\overline{X}]e' \rangle \\ [(\rho A)^\star/X, \overline{B}^\star/\overline{X}, \texttt{new}\ C\texttt{<}\overline{B}^\star\texttt{>(}\overline{v}\texttt{)/this}]e' \Downarrow^{\mathcal{D}} v_0\end{array}}{Se\texttt{.tyapp<}(\rho A)^\star\texttt{>()} \Downarrow^{\mathcal{D}} v_0}$$

as required.

Case (e-app)

Let $M_0 = M\ N$. The evaluation derivation must end with the rule

$$\frac{\begin{array}{ll}\rho \vdash M \Downarrow_e \langle \rho', \texttt{rec}\ y(x{:}A){:}B.M' \rangle & (1)\\ \rho \vdash N \Downarrow_e \gamma & (2)\\ x{:}A \mapsto \gamma, y{:}A \to B \mapsto \langle \rho', \texttt{rec}\ y(x{:}A){:}B.M' \rangle, \rho' \vdash M' \Downarrow_e \gamma_0 & (3)\end{array}}{\rho \vdash M\ N \Downarrow_e \gamma_0}$$

and the translation derivation must end with

$$\frac{\Gamma; \psi \vdash M : A_1 \to A_0 \rightsquigarrow e \text{ in } \mathcal{D}_1 \qquad \Gamma; \psi \vdash N : A_1 \rightsquigarrow e' \text{ in } \mathcal{D}_2}{\Gamma; \psi \vdash M\ N : A_0 \rightsquigarrow e\texttt{.app(}e'\texttt{)} \text{ in } \mathcal{D}_1 \cup \mathcal{D}_2}$$

Applying the induction hypothesis to the derivation ending with (1), there exists some v such that $Se \Downarrow^{\mathcal{D}} v$ with $\vdash^{\mathcal{D}} \langle \rho', \texttt{rec}\ y(x{:}A){:}B.M' \rangle \rightsquigarrow v$. Applying the induction hypothesis to the derivation ending with (2), there exists some v' such that $Se' \Downarrow^{\mathcal{D}} v'$ with $\vdash^{\mathcal{D}} \gamma \rightsquigarrow v'$. Suppose $\rho' = \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}$. The closure translation must have been derived using rule (tr-absclos):

$$\frac{\vdash^{\mathcal{D}} \overline{\gamma} \rightsquigarrow \overline{v} \quad \overline{X}, \overline{x}{:}\overline{A}; y(x{:}A){:}B \vdash M' : B \rightsquigarrow e'' \text{ in } \mathcal{D}' \quad \mathcal{D}' \subseteq \mathcal{D}}{\vdash^{\mathcal{D}} \langle \rho', \texttt{rec}\ y(x{:}A){:}B.M' \rangle \rightsquigarrow \texttt{new}\ C\texttt{<}\overline{B}^{\star}\texttt{>(}\overline{v}\texttt{)}}$$

$$\begin{array}{l}\mathcal{D}(C) = \\ \texttt{class}\ C\texttt{<}\overline{X}\texttt{>}\ \texttt{:}\ (A \to B)^{\star}\ \{ \\ \quad \overline{A}^{\star}\ \overline{x}\texttt{;} \\ \quad \texttt{public override}\ B^{\star}\ \texttt{app(}A^{\star}\ x\texttt{)}\ \{\ \texttt{return}\ e''\texttt{;}\ \}\ \}\end{array}$$

Let $\rho'' = x{:}A \mapsto \gamma, y{:}A \to B \mapsto \langle \rho', \texttt{rec}\ y(x{:}A){:}B.M' \rangle, \rho'$. It is easy to see that

$$\rho'' \rhd^{\mathcal{D}}_{\overline{X}, \overline{x}{:}\overline{A}; y(x{:}A){:}B} [v'/x, \texttt{new}\ C\texttt{<}\overline{B}^{\star}\texttt{>(}\overline{v}\texttt{)/this}, \overline{B}^{\star}/\overline{X}]$$

so we can apply the induction hypothesis to the subderivation ending with (3) to show that there exists some v_0 such that

$$[v'/x, \texttt{new}\ C\texttt{<}\overline{B}^{\star}\texttt{>(}\overline{v}\texttt{)/this}, \overline{B}^{\star}/\overline{X}]e'' \Downarrow^{\mathcal{D}} v_0$$

Finally consider the evaluation of $S(e\texttt{.app(}e'\texttt{)}) = Se\texttt{.app(}Se'\texttt{)}$. Using rule (e-meth) we can derive

$$\frac{\begin{array}{l}Se \Downarrow^{\mathcal{D}} \texttt{new}\ C\texttt{<}\overline{B}^{\star}\texttt{>(}\overline{v}\texttt{)} \\ Se' \Downarrow^{\mathcal{D}} v' \\ mbody_{\mathcal{D}}(C\texttt{<}\overline{B}^{\star}\texttt{>.app<>}) = \langle x, [\overline{B}^{\star}/\overline{X}]e'' \rangle \\ [v'/x, \texttt{new}\ C\texttt{<}\overline{B}^{\star}\texttt{>(}\overline{v}\texttt{)/this}, \overline{B}^{\star}/\overline{X}]e'' \Downarrow^{\mathcal{D}} v_0\end{array}}{Se\texttt{.app(}Se'\texttt{)} \Downarrow^{\mathcal{D}} v_0}$$

as required.

Part (2). By induction over the structure of the derivation of $Se_0 \Downarrow^{\mathcal{D}} v_0$.

Case (e-fld)

From the translation, we must have used rule (tr-var) on $M_0 = x$ to get $e_0 = \texttt{this}.x$. By evaluation rule (e-var) we have $\rho \vdash x \Downarrow_e \rho(x)$. Then by Definition 2 we obtain have $\vdash^{\mathcal{D}} \rho(x) \rightsquigarrow v$ and $S(\texttt{this}.x) \Downarrow^{\mathcal{D}} v$ as required.

Case (e-new)

There are three subcases.

- Suppose $M_0 = x$ and translation rule (tr-argvar) or (tr-funvar) was used. By evaluation rule (e-var) we have $\rho \vdash x \Downarrow_e \rho(x)$ and from Definition 2 we have $\vdash^{\mathcal{D}} \rho(x) \rightsquigarrow v_0$ as required.
- Suppose $M_0 = \Lambda X.M$ and $\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}$. The translation derivation ends with (tr-tyabs):

$$\frac{\begin{array}{ll} X, \Gamma \uplus \psi; \bullet \vdash M : A \rightsquigarrow e \text{ in } \mathcal{D}' & (1) \\ \Gamma; \psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \text{ in } \{\} & (2) \end{array}}{\begin{array}{l} \Gamma; \psi \vdash \Lambda X.M : \forall X.A \rightsquigarrow \texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \text{ in } \{C \mapsto cd\} \uplus \mathcal{D}' \\ \quad C \notin \mathrm{dom}(\mathcal{D}') \text{ and } cd = \\ \quad \texttt{class } C\texttt{<}\overline{X}\texttt{>} : (\forall X.A)^\star \ \{ \\ \quad\quad \overline{A}^\star\ \overline{x}\texttt{;} \\ \quad\quad \texttt{public override } A^\star \texttt{ tyapp<}X\texttt{>() \{ return } e\texttt{;} \} \ \} \end{array}} \quad (3)$$

  Now $S(\texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)}) = \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}S\overline{e}\texttt{)}$. Hence the evaluation derivation ends with the following instance of (e-new):

$$\frac{S(\overline{e}) \Downarrow^{\mathcal{D}} \overline{w}}{\texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}S\overline{e}\texttt{)} \Downarrow^{\mathcal{D}} \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}\overline{w}\texttt{)}}$$

  Putting this together with (2) we can use Definition 2 to show $\vdash^{\mathcal{D}} \rho(\overline{x}) \rightsquigarrow \overline{w}$. We can then use (1) and (3) to apply translation rule (tr-tyabsclos):

$$\frac{\vdash^{\mathcal{D}} \rho(\overline{x}) \rightsquigarrow \overline{w} \quad X, \Gamma \uplus \psi; \bullet \vdash M : A \rightsquigarrow e \text{ in } \mathcal{D}'}{\vdash^{\mathcal{D}} \langle \rho, \Lambda X.M \rangle \rightsquigarrow \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}\overline{w}\texttt{)}}$$

  Finally by rule (e-val) we also have $\rho \vdash \Lambda X.M \Downarrow_e \langle \rho, \Lambda X.M \rangle$ as required.
- Suppose $M_0 = \texttt{rec } y(x{:}A){:}B.M$ and $\Gamma \uplus \psi = \overline{X}, \overline{x}{:}\overline{A}$. The translation derivation ends with (tr-abs):

$$\frac{\begin{array}{ll} \Gamma \uplus \psi; y(x{:}A){:}B \vdash M : B \rightsquigarrow e \text{ in } \mathcal{D}' & (1) \\ \Gamma; \psi \vdash \overline{x} : \overline{A} \rightsquigarrow \overline{e} \text{ in } \{\} & (2) \end{array}}{\begin{array}{l} \Gamma; \psi \vdash \texttt{rec } y(x{:}A){:}B.M : A \to B \rightsquigarrow \texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)} \text{ in } \{C \mapsto cd\} \uplus \mathcal{D}' \\ \quad C \notin \mathrm{dom}(\mathcal{D}') \text{ and } cd = \\ \quad \texttt{class } C\texttt{<}\overline{X}\texttt{>} : (A \to B)^\star \ \{ \\ \quad\quad \overline{A}^\star\ \overline{x}\texttt{;} \\ \quad\quad \texttt{public override } B^\star \texttt{ app(}A^\star\ x\texttt{) \{ return } e\texttt{; } \} \ \} \end{array}} \quad (3)$$

Now $S(\texttt{new } C\texttt{<}\overline{X}\texttt{>(}\overline{e}\texttt{)}) = \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}S\overline{e}\texttt{)}$. Hence the evaluation derivation must have ended with the following instance of (e-new):

$$\frac{S(\overline{e}) \Downarrow^{\mathcal{D}} \overline{w}}{\texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}S\overline{e}\texttt{)} \Downarrow^{\mathcal{D}} \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}\overline{w}\texttt{)}}$$

Putting this together with (2) we can use Definition 2 to show $\vdash^{\mathcal{D}} \rho(\overline{x}) \rightsquigarrow \overline{w}$. We can then use (1) and (3) to apply translation rule (tr-absclos):

$$\frac{\vdash^{\mathcal{D}} \rho(\overline{x}) \rightsquigarrow \overline{w} \quad \Gamma \uplus \psi; y(x{:}A){:}B \vdash M : A \rightsquigarrow e \text{ in } \mathcal{D}'}{\vdash^{\mathcal{D}} \langle \rho, \texttt{rec } y(x{:}A){:}B.M\rangle \rightsquigarrow \texttt{new } C\texttt{<}\rho(\overline{X})^\star\texttt{>(}\overline{w}\texttt{)}}$$

Finally by rule (e-val) we also have $\rho \vdash \texttt{rec } y(x{:}A){:}B.M \Downarrow_e \langle \rho, \texttt{rec } y(x{:}A){:}B.M\rangle$ as required.

Case (e-meth)

From translation, we must have used rule (tr-app) or rule (tr-tyapp). First consider (tr-app), with $M_0 = M\ N$:

$$\frac{\Gamma; \psi \vdash M : A_1 \to A_0 \rightsquigarrow e \text{ in } \mathcal{D}_1 \qquad \Gamma; \psi \vdash N : A_1 \rightsquigarrow e' \text{ in } \mathcal{D}_2}{\Gamma; \psi \vdash M\ N : A_0 \rightsquigarrow e\texttt{.app(}e'\texttt{)} \text{ in } \mathcal{D}_1 \cup \mathcal{D}_2}$$

The evaluation derivation is as follows:

$$\frac{\begin{array}{ll} Se \Downarrow^{\mathcal{D}} \texttt{new } I\texttt{(}\overline{w}\texttt{)} & (1)\\ mbody_{\mathcal{D}}(I\texttt{.app<>}) = \langle x, e''\rangle & (2)\\ Se' \Downarrow^{\mathcal{D}} v & (3)\\ [v/x, \texttt{new } I\texttt{(}\overline{w}\texttt{)/this}]e'' \Downarrow^{\mathcal{D}} v_0 & (4)\end{array}}{Se\texttt{.app(}Se'\texttt{)} \Downarrow^{\mathcal{D}} v_0}$$

Applying the induction hypothesis to (1), there exists γ such that $\rho \vdash M \Downarrow_e \gamma$ and $\vdash^{\mathcal{D}} \gamma \rightsquigarrow \texttt{new } I\texttt{(}\overline{w}\texttt{)}$. Applying the induction hypothesis to (3), there exists γ' such that $\rho \vdash N \Downarrow_e \gamma'$ and $\vdash^{\mathcal{D}} \gamma' \rightsquigarrow v$. Now by type preservation for System F, the closure value γ must have function type and so have the form $\gamma = \langle \rho', \texttt{rec } y(x{:}A){:}B.M'\rangle$ with $\rho' = \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}$. The closure translation must have been derived using with rule (tr-absclos):

$$\frac{\vdash^{\mathcal{D}} \overline{\gamma} \rightsquigarrow \overline{w} \quad \overline{X}, \overline{x}{:}\overline{A}; y(x{:}A){:}B \vdash M' : B \rightsquigarrow e''' \text{ in } \mathcal{D}' \quad \mathcal{D}' \subseteq \mathcal{D}}{\vdash^{\mathcal{D}} \langle \rho', \texttt{rec } y(x{:}A){:}B.M\rangle \rightsquigarrow \texttt{new } C\texttt{<}\overline{B}^\star\texttt{>(}\overline{w}\texttt{)}}$$

$\mathcal{D}(C) =$
```
class C<X̄> : (A → B)* {
  Ā* x̄;
  public override B* app(A* x) { return e'''; } }
```

So $I = C\texttt{<}\overline{B}^\star\texttt{>}$ and to fit premise (2) above $e'' = [\overline{B}^\star/\overline{X}]e'''$. Now let $\Gamma' = \overline{X}, \overline{x}{:}\overline{A}$ and $\psi' = y(x{:}A){:}B$. Then we have

$$\rho', x{:}A \mapsto \gamma', y{:}A \to B \mapsto \gamma \rhd^{\mathcal{D}}_{\Gamma';\psi'} [v/x, \texttt{new } I\texttt{(}\overline{w}\texttt{)/this}, \overline{B}^\star/\overline{X}]$$

Using this to apply the induction hypothesis to derivation (4) there must exist γ_0 such that $\rho', x{:}A \mapsto \gamma', y{:}A \to B \mapsto \gamma \vdash M' \Downarrow_e \gamma_0$ and $\vdash^{\mathcal{D}} \gamma_0 \rightsquigarrow v_0$. Then we can derive the required result using evaluation rule (e-app):

$$\frac{\rho \vdash M \Downarrow_e \gamma \quad \rho \vdash N \Downarrow_e \gamma' \quad x{:}A \mapsto \gamma', y{:}A \to B \mapsto \gamma \vdash M' \Downarrow_e \gamma_0}{\rho \vdash M\ N \Downarrow_e \gamma_0}$$

Now the second possibility for case (e-meth) is that we used the translation rule (tr-tyapp) with $M_0 = M\ A$:

$$\frac{\Gamma; \psi \vdash M : \forall X.B \rightsquigarrow e \text{ in } \mathcal{D}}{\Gamma; \psi \vdash M\ A : [A/X]B \rightsquigarrow e\texttt{.tyapp<}A^\star\texttt{>()} \text{ in } \mathcal{D}}$$

We have $S(e\texttt{.tyapp<}A^\star\texttt{>()}) = Se\texttt{.tyapp<}(\rho A)^\star\texttt{>()}$. The evaluation derivation is as follows:

$$\frac{\begin{array}{ll} Se \Downarrow^{\mathcal{D}} \texttt{new } I\texttt{(}\overline{w}\texttt{)} & (1) \\ mbody_{\mathcal{D}}(I\texttt{.tyapp<}(\rho A)^\star\texttt{>}) = \langle (), e' \rangle & (2) \\ [\texttt{new } I\texttt{(}\overline{w}\texttt{)/this}]e' \Downarrow^{\mathcal{D}} v_0 & (3) \end{array}}{Se\texttt{.tyapp<}(\rho A)^\star\texttt{>()} \Downarrow^{\mathcal{D}} v_0}$$

Applying the induction hypothesis to (1) there exists γ such that $\rho \vdash M \Downarrow_e \gamma$ and $\vdash^{\mathcal{D}} \gamma \rightsquigarrow \texttt{new } I\texttt{(}\overline{w}\texttt{)}$. By type preservation for System F evaluation we know that γ must be a type abstraction closure with the form $\gamma = \langle \rho', \Lambda X.M' \rangle$ where $\rho' = \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}$. The closure translation must have been derived using with rule (tr-tyabsclos):

$$\frac{\vdash^{\mathcal{D}} \overline{\gamma} \rightsquigarrow \overline{v} \quad X, \overline{X}, \overline{x}{:}\overline{A}; \bullet \vdash M' : A \rightsquigarrow e'' \text{ in } \mathcal{D}' \quad \mathcal{D}' \subseteq \mathcal{D}}{\vdash^{\mathcal{D}} \langle \{\overline{X} \mapsto \overline{B}, \overline{x}{:}\overline{A} \mapsto \overline{\gamma}\}, \Lambda X.M' \rangle \rightsquigarrow \texttt{new } C\texttt{<}\overline{B}^\star\texttt{>(}\overline{v}\texttt{)}}$$

$$\begin{array}{l} \mathcal{D}(C) = \\ \texttt{class } C\texttt{<}\overline{X}\texttt{>} : (\forall X.A)^\star \ \{ \\ \quad \overline{A}^\star\ \overline{x}\texttt{;} \\ \quad \texttt{public override } A^\star\ \texttt{tyapp<}X\texttt{>()}\ \{\ \texttt{return } e''\texttt{;}\}\ \} \end{array}$$

In premise (2) above we must have $e' = [(\rho A)^\star/X, \overline{B}^\star/\overline{X}]e''$. Now let $\Gamma' = X, \overline{X}, \overline{x}{:}\overline{A}$. Hence

$$X \mapsto \rho A, \rho' \triangleright^{\mathcal{D}}_{\Gamma';\bullet} [\texttt{new } I\texttt{(}\overline{w}\texttt{)/this}, \overline{B}^\star/\overline{X}]$$

so we can apply the induction hypothesis to (3) to show that there exists γ_0 such that $X \mapsto \rho A, \rho' \vdash M' \Downarrow_e \gamma_0$ and $\vdash^{\mathcal{D}} \gamma_0 \rightsquigarrow v_0$. Then we can derive the required result using evaluation rule (e-tyapp):

$$\frac{\rho \vdash M \Downarrow_e \langle \rho', \Lambda X.M' \rangle \quad X \mapsto \rho A, \rho' \vdash M' \Downarrow_e \gamma_0}{\rho \vdash M\ A \Downarrow_e \gamma_0}$$

□

Specializing to convergence behaviour of closed terms produces the following corollary.

Corollary 2. *If* $\vdash M : A \rightsquigarrow e$ in $\mathcal{D}$ *then* $M \Downarrow_e$ *iff* $e \Downarrow^{\mathcal{D} \cup \mathcal{G}}$.

Putting this together with Lemma 3 applied to closed terms gives us semantic soundness.

Theorem 6 (Semantic soundness) *If* $\vdash M : A \rightsquigarrow e$ in $\mathcal{D}$ *then* $M \Downarrow$ *iff* $e \Downarrow^{\mathcal{D} \cup \mathcal{G}}$.

6. Discussion

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

We have shown that the translation preserves types and convergence behaviour. It is interesting to consider a much stronger property than semantic soundness: *full abstraction*. A translation is fully abstract if source expressions that cannot be distinguished by any source context, that is, are *contextually equivalent*, are mapped to target expressions that are contextually equivalent. It is conjectured that our translation is fully abstract.

Of course, many features of the full C♯ language destroy full abstraction. One such is run-time casts: the `(`T`)`e construct considered in Figure 7. By using casts it is possible to look inside closures. This is easy to fix by introducing a `private` access qualifier on fields. But run-time casts also allow types to be observed after translation, breaking the purely-parametric nature of polymorphism in System F. For example, given distinct types A and B and values $V{:}A$ and $W{:}B$ consider the contextually-equivalent System F terms $\lambda x{:}(\forall X.X \to \mathsf{unit}).x\ A\ V$ and $\lambda x{:}(\forall X.X \to \mathsf{unit}).x\ B\ W$, where $\mathsf{unit} = \forall X.X \to X$. Their translations into C♯ minor with downcasts can be distinguished by passing their `app` methods an object whose `tyapp` method returns a closure object that checks the type of its argument against A before returning.

The distinctiveness of polymorphic virtual methods shows up in implementations. First, observe that such methods cannot be implemented by compile-time monomorphisation or expansion, since it is not possible to determine all instantiations at compile-time. For this reason, C++ template methods are not permitted to be declared `virtual` [18, §13.6.2]. Similarly, the CLR implementation of polymorphic virtual methods [10] involves execution-time code generation, contrasting with the load-time code generation that is sufficient to support non-virtual polymorphic methods.

Much of our translation is concerned with *closure conversion*. Two other works are relevant here. Harper *et al.* have formalized closure conversion for a typed functional language [12]. Their correctness proof uses the method of logical relations, but this technique does not scale well when adding language features such as recursion and impredicative System-F-style polymorphism, both of which their language lacks. On the other hand, logical relations do adapt well to complicated translation schemes. We were able to use direct syntactic techniques in our proof of correctness because the source and target terms match up surprisingly well, despite the "distance" between System F and C♯. The use of environment-style semantics for System F was the crucial component here, as it captured the notion of a 'closure' in the semantics itself and thereby ensured that the translation commuted with evaluation.

Glew has formalized source-to-source closure conversion for an object calculus that supports nested definitions of methods [5]. The language is very different from C♯ in that it does not have classes, and type equivalence is by structure, not name. Glew proves a full abstraction result for the translation with respect to contextual equivalence.

Finally, Igarashi and Pierce have considered Java's *inner classes* feature [7], whereby a classes can be nested and refer to instance fields of enclosing classes. They formalize the translating

away of inner classes, a process akin to closure conversion, and prove correctness by matching reductions in the source to reductions in the target.

ACKNOWLEDGEMENTS

We are grateful to the anonymous reviewers for their comments and to Nick Benton, Gavin Bierman, Luca Cardelli and Claudio Russo for discussions on this work.

REFERENCES

1. G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.
2. G. Bracha, N. Cohen, C. Kemper, S. Marx, M. Odersky, S. Panitz, D. Stoutamire, K. Thorup, and P. Wadler. JSR-000014: Adding Generics to the Java^{TM} Programming Language, April 2001. See `http://jcp.org/aboutJava/communityprocess/review/jsr014/`.
3. 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.
4. 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.
5. N. Glew. Object closure conversion. In *3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, September 1999.
6. A. Hejlsberg. The C$^\sharp$ programming language. Invited talk at Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), 2002.
7. A. Igarashi and B. C. Pierce. On inner classes. *Information and Computation*, 177(1):56–89, August 2002.
8. A. Igarashi, B. C. 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.
9. M. P. Jones. First-class polymorphism with type inference. In *ACM Symposium on Principles of Programming Languages*, pages 483–496. ACM, 1997.
10. 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.
11. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
12. Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.
13. M. Odersky and K. Läufer. Putting type annotations to work. In *ACM Symposium on Principles of Programming Languages*. ACM, 1996.
14. 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.
15. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
16. 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.
17. C. V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4), Winter 2000.
18. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 1997.