Adventures in Interoperability: The SML.NET Experience

Nick Benton
nick@microsoft.com

Andrew Kennedy
akenn@microsoft.com

Claudio V. Russo
crusso@microsoft.com

Microsoft Research Ltd
7JJ Thomson Ave
Cambridge, United Kingdom

ABSTRACT

SML.NET is a compiler for Standard ML that targets the Common Language Runtime and is integrated into the Visual Studio development environment. It supports easy interoperability with other .NET languages via a number of language extensions, which go considerably beyond those of our earlier compiler, MLj.

This paper describes the new language extensions and the features of the Visual Studio plugin, including syntax highlighting, Intellisense, continuous type inference and debugger support. We discuss our experiences using SML.NET to write SML programs that interoperate with other .NET languages, libraries and frameworks. Examples include the Visual Studio plugin itself (written in SML.NET, using .NET's COM interop features to integrate in a C++ application) and writing ASP.NET and Pocket PC applications in SML.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments*; D.3.2 [**Programming Languages**]: Language Classifications—*Standard ML, applicative (functional) languages, multiparadigm languages, object-oriented languages*; D.3.4 [**Programming Languages**]: Processors—*compilers*; D.2.12 [**Software Engineering**]: Interoperability—*data mapping*

General Terms

Languages

Keywords

Functional programming, applications of declarative programming, integration of paradigms, programming environments

1. INTRODUCTION

The .NET Common Language Runtime [8] presents a exciting opportunity for programming language researchers. High-level runtime infrastucture can reduce the burden of producing a compiler, and, more importantly, rich standard libraries and very good support for interoperability between languages can make niche languages viable alternatives to more mainstream commercial ones for writing all or part of useful applications. The CLR was designed from the ground up to support multiple languages, whilst Microsoft's development environment, Visual Studio .NET, is able to host multiple compilers and can manage solutions consisting of multiple projects, implemented in different languages.

SML.NET[2, 5] is a compiler for Standard ML [16] that targets the Common Language Infrastructure (CLI), producing Common Intermediate Language (CIL) as its object code [9, 15]. SML.NET is a further development of the MLj compiler [6], which compiled for the Java Virtual Machine. The basic compilation strategy of SML.NET is essentially the same as that of its predecessor: it is a whole-program, type-directed optimizing compiler. Polymorphic code is specialized at representation (not source) types, minimising code blowup whilst allowing efficient non-uniform data representations, such as natural-sized integral types and unwrapped datatypes. For example, *ty* `option` will map to the same CLI type as *ty* does, provided that is a reference type and the representation of *ty* does not use the null value. Basic type, effect and flow information is used to perform transformations such as eliding unit values and types, removing dead code, minimising closure allocation, flattening tuples and varying calling conventions. Simple recursive functional loops are compiled using jumps and we also use the CLR's support for tail call optimization, which was unavailable on the JVM. (Although the CLR *tailcall* is quite expensive, it is crucial in allowing some programs, such as the bootstrapped compiler, to run at all.)

This rest of this paper describes SML.NET from the point of view of the user, rather than that of the compiler implementer. We first give an overview of the compiler and its integration within Visual Studio and then give a more detailed account of the extensions we have made to SML to support interoperability, concentrating particularly on those features which differ from, or extend, the MLj design [4]. We then discuss three interesting examples of interop. One is the use of SML.NET to write its own COM plugin for Visual Studio .NET, exploiting the CLR's COM interface and SML.NET's new-found ability to bootstrap (a noted shortcoming of MLj). The others are the use of SML.NET to script ASP.NET pages, and an experiment running SML.NET code on mobile devices.

PPDP'04, August 24–26, 2004, Verona, Italy.

2. THE SML.NET USER EXPERIENCE

2.1 Language and Compilation Model

SML.NET compiles *all* of Standard ML '97 [16] (MLj omitted functors), and provides an almost complete SML Basis Library [10]; omissions and discrepancies are documented in the manual [5]. SML.NET extends the SML language to support safe, convenient use of the .NET Framework libraries and code written in other languages for the CLR. SML.NET can consume and produce .NET classes, interfaces, delegates, etc. (see Section 3).

The output of the SML.NET compiler is verifiable CIL for the CLI, either a library (`dll`) or a standalone executable (`exe`). Like other whole-program ML compilers (MLj and MLton [1]), SML.NET does not have an interactive read-eval-print loop. The exposed interface of an SML.NET application or library may (roughly speaking) only refer to CLI types (classes, interfaces, delegates, etc.). It may not reveal Standard ML-specific types (functions, datatypes, records, etc.).

Despite the whole-program model, SML.NET does implement a limited form of separate compilation. A precompilation phase takes place at top-level module boundaries; this includes parsing, type checking, translation to intermediate form, and some optimization. The results of precompilation are cached, but most optimization, and all code generation, is deferred until after linking. SML.NET has a convenient build system, which requires only the names of a root module and any imported CLI libraries together with a source path. A dependency analysis then determines which individual files are required and which need re-(pre)-compilation. The command-line compiler can be run in either batch or interactive mode. The latter lets the user set and query options incrementally and see the signatures of compiled and imported modules; it also speeds up re-compilation by caching precompiled modules in memory rather than on disk.

2.2 Performance

Compilation times using SML.NET are, as one might expect from a whole-program compiler, high. Using SML.NET compiled with SML/NJ on a modern machine, a complete recompile of SML.NET itself (~80,000 lines) takes around 20 minutes. Although this can be cut by a factor of nearly 3 by using MLton instead of SML/NJ, it is still slow. On the other hand, SML.NET typically produces small executables with respectable runtime performance.

Looking at the performance of SML.NET on the usual ML benchmark suite, it is immediately apparent that programs fall into three categories:

1. Mostly-first-order code that manipulates numeric values performs between 20% faster and 20% slower than the most recent release of SML/NJ. Typical benchmarks in this category are `fft`, `mandelbrot`, `nfib`, `raytrace` and `simple`.

2. Higher-order code or datatype-heavy code performs between 2 and 4 times slower than SML/NJ. Typical benchmarks in this category are `life`, `logic`, `primes`, and `sort`.

3. Code that making heavy use of exceptions for control flow can perform between 20 and 40 times slower than SML/NJ. Typical benchmarks in this category are `boyermoore` and `knuthbendix`.

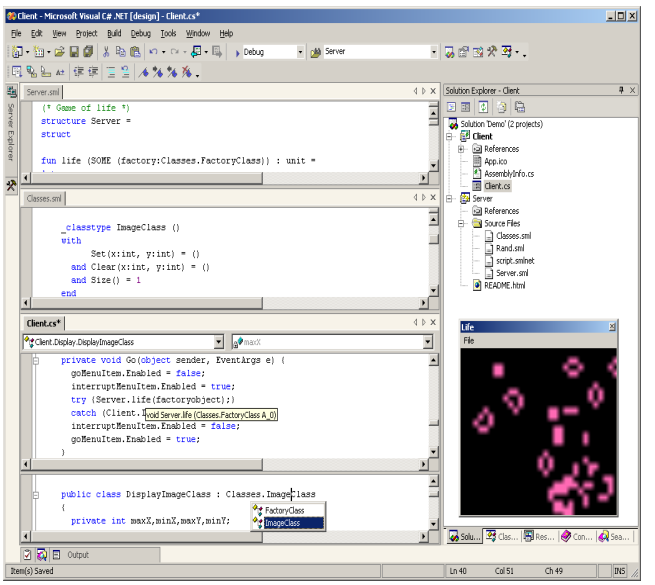

Figure 1: A mixed-language application combining SML.NET and C♯ projects.

The performance of typical SML.NET programs is thus usually quite acceptable provided that exception-based control-flow is avoided.

2.3 Visual Studio Integration

A major recent addition to SML.NET is a plugin for Microsoft Visual Studio .NET that allows the user to create, edit, build and debug SML.NET projects from within the .NET development environment. Figure 1 shows a simple multi-language graphical application, a version of Conway's Game of Life, within VS. The `Solution Explorer` window shows the two projects that make up the application: the `Server` project is an SML.NET library for computing the generations of the game using list processing. The `Client` project is a C♯ front-end, which produces an an animated display of the evolving generations (shown running in the `Life` window). The `Server.sml` window shows the pure SML definition of the function `Server.life`, which is called by the C♯ code in the top half of the `Client.cs` window. The bottom half shows a C♯ class extending an SML.NET (interop) class defined in file `Classes.sml`.

A custom language service for SML.NET supports intelligent editing of SML.NET code, shown in Figure 2. The language service provides the usual basic features, colouring keywords and automatically highlighting matching delimiters. It also supports more advanced features like *Intellisense*, which suggests member completions against the inferred types of objects and other bindings, displaying members and type information in drop down boxes and method tips. Whether completing against a CLI or SML.NET library, the types are always rendered under the SML.NET translation, aiding the programmer. Here, all the members of the object factory are shown along with the type of the selected member. Hovering over a locally declared identifier

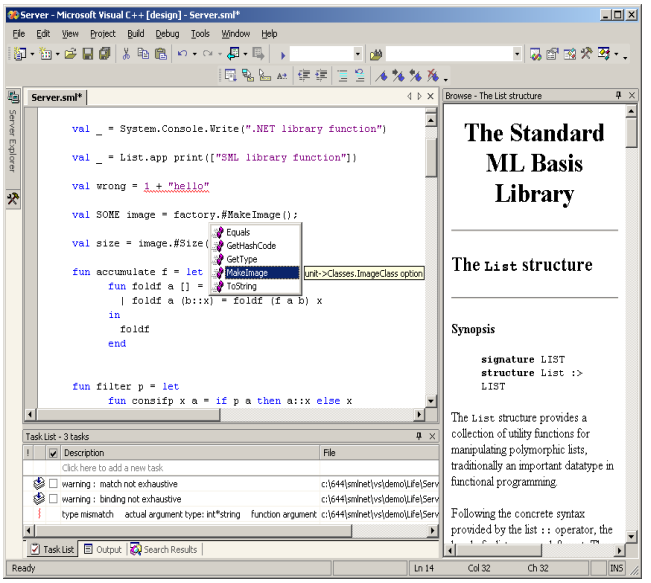

Figure 2: The SML.NET editor showing Intellisense and interactive type inference at work.

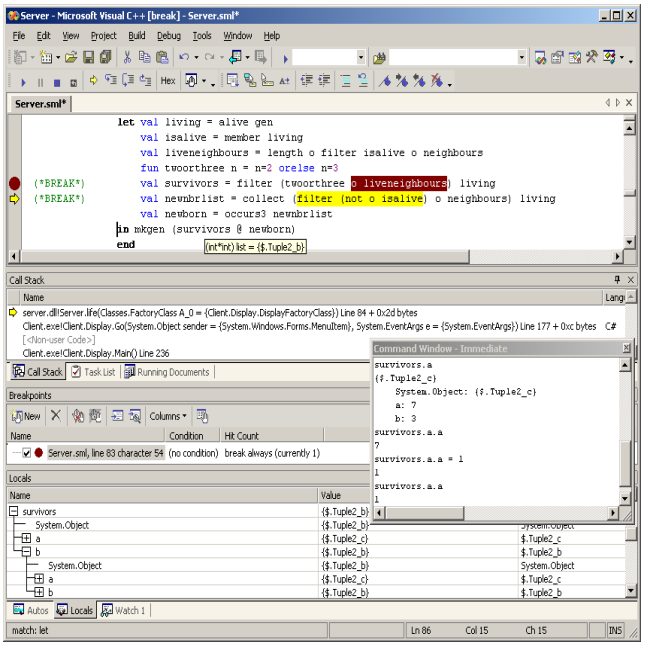

Figure 3: Debugging an SML.NET application.

shows its inferred type. Notice the red-squiggly underlining the type error in the definition of `wrong`. The language service actually performs continuous type inference, reporting type errors early in the `Task` window, without requiring a full re-build of the project. Mundane syntax errors are reported early in this way too. The `Task` window also shows two build warnings, which were harvested automatically by VS from the output of the last build.

SML.NET projects can be compiled in debug configurations, allowing them to be run under the Visual Studio source code debugger. Figure 3 shows the debugger attached to a paused instance of the `Life` application. The `Call Stack` window contains a mixture of C♯ and SML.NET stack frames. A breakpoint is set in the margin of the `Server.sml` source code (the SML expression is highlighted in brown). The current execution line, to which we have advanced the debugger by stepping from the breakpoint, is marked with an arrow with the current expression highlighted in yellow. The `Locals` window shows the current values of local bindings, including the list of integer pairs `survivors` (notice the type reported by Intellisense). The `Command` window shows the result of inspecting the first tuple (`survivors.a`) of the survivors list and interactively modifying one of its components, for exploratory debugging. The figure also reveals some limitations of the debug experience. In the locals window, the types of values are shown with their CLI representations (e.g. the cryptic `$.Tuple2_c` for the type of `survivors`). In the `Command` Window, values are inspected and modified using C♯ expression syntax, not SML.NET syntax. It should be possible to write our own expression evaluator for SML.NET but we have not tried this yet. Other losses of fidelity include stack frames and bindings being removed due to inlining and other optimizations by the compiler, and spurious temporary locations (unrelated to source bindings) appearing in the `Locals` window. Producing precise symbolic debug information is a difficult problem for any optimizing compiler, but we could easily do a little better. Fortunately, source stepping is precise, and the debugger support, limited though it is, has proved useful on a number of occasions.

As well as being useful, the VS package is itself a demonstration of our interoperability features. VS is a legacy C++ COM application, but our VS package is implemented entirely in SML.NET! Section 4 describes the architecture.

3. LANGUAGE EXTENSIONS FOR .NET

An important feature of SML.NET is its support for smooth interoperation with other .NET languages and libraries. We continued with the MLj model [4], mapping .NET features into equivalent SML features, where possible, but, where there is no obvious equivalent, introducing extensions. An additional guideline was the CLI's Common Language Subset (CLS) [9], which specifies which features of the CLI every language should support for interoperability.

3.1 Namespaces, classes and nesting

If one ignores the class hierarchy and instance fields and methods (i.e. a non-object-oriented fragment of .NET), then .NET classes can be seen (and are used) as a minimal module system, providing a way of packaging together static fields, methods, and nested classes that are logically related. Namespaces in .NET provide a further level of structuring, grouping related classes. We model both using the SML module system.

SML.NET reflects a top-level namespace (e.g. `System`) as a top-level structure (e.g. structure `System`); a nested namespace (e.g. `System.Drawing`) becomes a substructure.

SML.NET reflects a class as three eponymous bindings:

- as a type identifier (see Section 3.2),
- as a (possibly overloaded) function used to construct instances of the class (see Section 3.3.1),

- as a substructure containing value bindings, reflecting static fields and methods (see Sections 3.4.1 and 3.4.2), and any further bindings from nested classes.

For example, within the namespace `System.Threading` (reflected as nested structure `System.Threading`) the class `Mutex` is mapped to an SML type identifier `Mutex`, to a value identifier `Mutex`, and to a structure `Mutex`.

Namespaces and classes, once interpreted as modules, can be manipulated like any other structure in SML: they can be rebound, constrained by a signature, passed to functors, and `open`-ed. Opening a namespace is like C♯'s `using` construct but provides unqualified access to nested namespaces too. Opening a class is more powerful, permitting unqualified access to its static fields and methods and any nested classes.

3.2 Types

Many .NET primitive types map to direct equivalents in SML.NET. Within the `System` namespace, `Boolean`, `Char`, `Double`, `Single`, `IntNN`, `SByte`, `UIntNN`, `Byte`, non-null `String` and `Exception`, map to SML.NET types `bool`, `char`, `real`, `Real32.real`, `IntNN.int`, `Int8.int`, `WordNN.word`, `Word8.word`, `string` and `exn` (where `NN` is `64`,`32` and `16`). In SML.NET the default `int` and `word` types are the natural `Int32.int` and `Word32.word`. Amongst .NET types only the usual SML Basis types admit equality that can be tested using SML's polymorphic equality operator `=` (i.e. all of the above except the `real` types and `exn`).

Any named .NET type (class, value type, enumeration, interface or delegate) can be referred to from within SML.NET using the same syntax as in C♯. This syntax works because of the interpretation of .NET namespaces as nested structures, discussed above. For example:

```
type XMLParser = string -> System.Xml.XmlDocument
```

Single-dimensional .NET arrays behave almost exactly like SML arrays: their size is fixed at time of creation, indexing starts at zero, equality is based on identity not value, and an exception is raised on out-of-bounds access or update. Thus we identify the SML type constructor `array` with the .NET type constructor `[]` (and the .NET exception `System.IndexOutOfRangeException` with the SML exception `Subscript`).

Some, but not all, array types inherit from the class `System.Array`, so one can invoke methods on values of these array type and cast their array values to and from class types. The restriction is that the element type must be an *interop input type* in the sense of Section 3.2.2.

3.2.1 Null values

In .NET, objects of class, interface, array or delegate type (known collectively as *reference* types) are allowed to take on the value `null` in addition to actual instances. CLI member and array access raise `NullReferenceException` if their instance operand is `null`. SML does not have this notion, and values must always be defined (e.g. there is no null in SML's string type). SML operations for assignment, indirection, and array access are inherently safer than the corresponding operations on .NET. We wished to retain this safety in our extensions and so interpret a value of .NET reference type as "non-null instance". Nevertheless, when a .NET field of reference type is accessed from SML or a value of reference type is returned from an external method call, it may have the undefined value `null`. Moreover, SML.NET must be able to pass null values to .NET methods and null .NET fields, when required. To express such values of 'possibly-null' reference types, as in MLj[6], we re-use the SML basis library's `option` type:

```
datatype 'a option = NONE | SOME of 'a
```

The `valOf` function (`: 'a option -> 'a`) can be used to extract the underlying value, raising `Option` on `NONE`(`null`).

We interpret possibly-null values of .NET reference type that cross the border between SML and .NET as values of an appropriate `option` type. For example, the method `Join` in class `System.String` has the C♯ prototype:

```
public static string Join(string separator,
                          string[] val);
```

This maps to an SML function with signature

```
val Join : string option*string option array option
             -> string option
```

3.2.2 Interop types

We will use the term *interop type* for the types described in the previous sections and new .NET types defined inside SML.NET code. Interop types can be used in SML extensions such as casts, overloading, implicit coercions, and (with additional restrictions) in exported structures.

A special case of an interop type is an interop *input* type. These describe values that might be passed *into* SML.NET from external .NET sources, and therefore must include possible null values for reference types.

More precisely, an interop type is one of the following:

1. A .NET value type (primitive or struct). Examples: `int`, `System.DateTime`.
2. A .NET class, interface or delegate type defined externally or from within SML.NET code by `_classtype` or `_interfacetype` (see Sections 3.8). Examples: `string`, `System.IEnumerable`, `System.EventHandler`.
3. An array whose element type is an interop *input* type. Examples: `string option array`, `System.DateTime array`.
4. Possibly-null versions of either of the above. Examples: `string option array option`, `System.EventHandler option array option`.

A type is an interop *input* type if it is one of (1) or (4) above.

3.3 Objects

3.3.1 Creating objects

In C♯, instances of a class are created using the syntax `new` *class*(*arg*$_1$, ..., *arg*$_n$), where *arg*$_i$ are the arguments to one of the constructors defined by the class. We avoid the need for any new syntax in SML.NET by binding the class name itself to the constructor function. If there is more than one constructor, then the binding is overloaded. Constructors always return non-null values so never have an `option` result type.

Constructors may be used as first-class values; implicit argument coercions are applied using the same rules as for methods (see below). This example first-classes the `Font` constructor, applying an implicit coercion from `string` to `string option` on its first argument.

```
map System.Drawing.Font (("Times", 10.0)::pairs)
```

3.3.2 Creating and invoking delegate objects

.NET and C♯ support a notion of first-class method called a *delegate*. A delegate object is an instance of a named delegate type that wraps up a method and, for an instance method, its target object. SML.NET maps delegate types to two SML bindings: the type itself, just as with class types, and a function binding for the delegate constructor, which takes an SML function as its only argument.

An example is the C♯ delegate from `System.Threading`:

```
public delegate void ThreadStart();
```

This is reflected as an SML type `ThreadStart` and the higher-order function

```
val ThreadStart : (unit->unit) -> ThreadStart
```

used to construct delegate objects from ordinary SML functions. For example,

```
open System.Threading
val cookie = "cookie"
val monster =
  ThreadStart(fn () => while true do print cookie)
val _ = Thread(monster).#Start()
```

constructs a delegate from an SML function, capturing the `cookie` binding, and uses it to construct and start a new thread. Delegates are called via their `Invoke` method (e.g. `monster.#Invoke()` prints lots of `cookie`s). Working with delegates in C♯ 1.0 can be clumsy because there is no capture of free variables; programmers often perform tedious manual closure conversion in order, for example, to pass parameters to thread creation. Having proper lexically-scoped closures, SML.NET is much more convenient in this regard.

3.3.3 Casts and cast patterns

SML.NET retains MLj's expression syntax, *exp* `:>` *ty* to express C♯-style casts. One use is to cast an object up to a superclass:

```
open System.Drawing
val c = SolidBrush(System.Color.get_Red()) :> Brush
```

Explicit coercions are sometimes required when passing .NET objects to SML functions and constructors, as implicit coercions are only applied when invoking .NET methods.

The same syntax can also be used to cast an object down to a subclass, with `System.InvalidCastException` thrown if the actual class of the object is not compatible. A safer alternative, combining downcasting with C♯'s `is` construct, is to use the new *vid* `:>` *ty* pattern syntax inside SML patterns. This can be used to provide a construct similar to type-case found in some languages. For example, the following code switches on the type of an `XmlNode`:

```
fun nodetoxmldata (n : XmlNode) =
 case n of
   elem :> XmlElement =>
    let val SOME name = elem.#get_Name()
        val first = elem.#get_FirstChild()
        val children = gather (first, [])
    in SOME (Elem(name,List.mapPartial
                        nodetoxmldata children))
    end
 | data :> XmlCharacterData =>
    let val SOME s = data.#get_Data()
    in SOME (C(stringtoscalar s)) end
 | _ => NONE
```

The pattern *vid* `:>` *ty* matches only when the examined expression has the class type *ty*, in which case the identifier *vid* is bound to the expression casted down to type *ty*.

Cast patterns can be used like any other pattern. They can appear in `val` bindings, as in

```
val x :> System.Windows.Forms.Window = y
```

to give an effect similar to downcasting in expressions but raising SML's `Bind` exception when the match fails. They can also be used in exception handlers, as in

```
open System.IO System.Environment
fun chDir (s:string) =
  set_CurrentDirectory(s) handle
    e :> DirectoryNotFoundException => raise e
  | e :> IOException =>
        raise (Fail(valOf(e.#ToString())))
```

in order to catch (and possibly deconstruct) .NET exceptions. The *order* in which handlers appear is important. In this example, type `DirectyNotFoundException` is a subclass of `IOException` so if the handlers were switched the second handler would never be reached.

3.4 Fields, methods and properties

Static (per-class) fields and methods are mapped to value and function bindings in SML located in the structure corresponding to their class. For example, the `PI` static field in the `System.Math` class, accessed from C♯ using `System.Math.PI`, maps to a value binding for `PI` in the SML structure `System.Math` accessed using the same syntax. Likewise, the `Cos` static method in the same class is mapped to a value binding of `Cos` in the structure `System.Math`.

Non-static (instance) fields and methods are handled using MLj's syntax *exp*`.#`*name* for accessing members. Here *exp* is an SML expression with a .NET object type, and *name* is the name of an instance field or method.

Properties are really just C♯ syntactic sugar formalizing the commonplace "get/set" design pattern. No special support is provided in SML.NET, so they must be accessed through their underlying methods which have the names `get_`P and `set_`P for a property called P.

3.4.1 Fields

Immutable .NET fields (`readonly` and `const` in C♯) are given types as explained in Section 3.2, using `option` to denote the possibility of null values for objects or arrays. For example, a field declared in C♯ using

```
public static readonly string language_name = "C#";
```

is interpreted as having type `string option`.

For the most part, mutable fields can be treated as if they had SML `ref` types: they can be dereferenced using `!` and assigned to using `:=`. So fields declared by

```
class C {
  public static int counter;
  public int size;
}
```

can be used as if they were SML reference cells of type `int`:

```
C.counter := !C.counter + 1
fun size (x:C) = !(x.#size)
```

In fact, mutable fields are given special types (Section 3.7).

3.4.2 Methods

.NET method types are interpreted as follows. `void` methods are seen as having `unit` result type and methods with no arguments in .NET map to ones taking a `unit` in SML.NET. .NET methods with multiple arguments map to ones taking a tuple in SML.NET. External methods with arguments or results of reference types are mapped using the `option`-type translation described in Section 3.2.1.

Consider the following method from `System.String`:

```
public static string[] Split(char[] sep,int count);
```

Its type is interpreted as

```
val Split : char array option * int
              -> string option array option
```

and can be called using an ordinary function application:

```
fun split(c:char array,i:int) =
       valOf(System.String.Split(SOME c, i))
```

Here is an example of instance method invocation:

```
(* Create an object *)
val xmldoc = XmlDocument()
(* Invoke an instance method on it *)
val _ = xmldoc.#Load(filename)
```

.NET allows method *overloading*, i.e. the definition of multiple methods with the same name within a single class. The methods must be distinguished by their argument types. Our approach to inference, overloading and coercions is, with minor extensions, unchanged from MLj [4]. Briefly, we do not allow method and constructor ambiguity to be resolved by C♯-style most specific method rules, as these interact unpleasantly with SML's type inference: our inference algorithm accepts a program only if there is an unambiguous resolution of all method invocations. SML.NET does allow implicit coercions on method and constructor invocation using C♯'s reference widening coercions together with an additional coercion from *ty* to *ty* `option` for any .NET reference type *ty*.

3.5 Value Types

.NET provides support for an extensible set of unboxed, structured values called value types (C♯'s *structs*). Consider the C♯ declaration of lightweight, integer pairs:

```
public struct Pair {
  public int x,y;  /* both mutable! */
  /* constructor */
  public Pair(int i,int j){x = i; y = j;}
  /* functional swap, returns a new pair */
  public Pair swap(){return new Pair(y,x);}
  /* destructive swap, modifies 'this' pair */
  public void invert(){int t; t = x; x = y; y = t;}
}
```

Value types, like ordinary classes, can have fields and instance methods. However, because value types are sealed (cannot be subclassed), they do not need to be boxed on the heap (otherwise used to provide uniform representations); nor do they need to carry run-time type descriptors (used to support checked downcasts and virtual method invocation). For the most part, one can view instances of value types as structured primitive types. Indeed, primitive type such as `int` *are* value types. Values of value types do not have identity and are passed by copying, not by reference.

In SML.NET, one can use the same syntax for accessing fields and invoking methods on a value of value type as for objects of reference types. So for a pair `p`, `p.#x, !(p.#x), p.#swap()` are all legal. Unlike in MLj, method invocation works at primitive types too, e.g. `1.#ToString()`.

In SML.NET, invoking on a value type first copies the value, takes the address of the copy and then passes this address as the "this" argument of the method. Similarly, accessing a mutable field of some value first copies that value. This ensures that bindings are effectively *immutable* in SML.NET, preserving the fundamental invariant of functional languages. Thus `p.#invert()` has no effect on the value of `p`, nor does `let val r = p.#x in r := 1 end`. The expressions do have side-effects, but they cannot alter `p`.

Some value types do have mutable semantics, perhaps employing mutable instance fields that are updated by instance methods. Our copying semantics implies that these updates cannot be observed (since they mutate a temporary copy of the value, not the original value itself). To cater for such types, SML.NET supports an alternative invocation semantics. In addition to invoking directly on a value, SML.NET lets one invoke on any kind of SML.NET *reference* to a value type. Thus, `(ref p).#invert()` and, more usefully,

```
let val r = ref (Pair(1,2)) in r.#invert();!r end,
```

which returns the modified value `Pair(2,1)`, are both legal. This works as expected for fields too:

```
let val r = ref (Pair(1,2))  in (r.#x) := 3;!r end
```

returns the modified value `Pair(3,2)`.

This mechanism applies not only to SML `ref` types, but to all SML.NET `reference` types, regardless of *kind* (see Section 3.7). By accessing instance fields or methods of these storage types one can observably and selectively update their contents, either directly through a field update, or as a side effect of a method call.

3.5.1 Boxing and unboxing

In the CLI every value type, whether primitive or user-defined, also has an implicit boxed form, which is a reference type (subclassing `System.ValueType`). Values of all types may thus be stored in `object`-based collections, etc. providing a more uniform object model. (Less pleasingly, boxed values also have identity and may be mutated.)

Boxing a value allocates a new, appropriately type tagged, object on the heap and copies that value into the object. In SML.NET, the boxed form of a value type is obtained by an *upcast*, e.g. `p:>System.Object`, to some suitable CLI reference type. The supertype is typically `System.Object`, but may be `System.ValueType`. Boxing casts work for primitive types too, e.g. `1:>System.Object`.

Unboxing an object extracts a value from a heap allocated object (and requires a dynamic type check). In SML.NET this is achieved by a *downcast* from some object type to a value type. For instance, for `obj : System.Object`, the expressions `obj:>Pair` and `obj:> int` unbox the underlying

value, returning a value. Note that, like any other downcast, these involve a dynamic test that can fail by raising `System.InvalidCastException`.

3.5.2 Default values

In SML.NET, every non-primitive value type has a defined default value, bound to the identifier `null` in the structure corresponding to the type (e.g. `Pair.null` is equivalent to `Pair(0,0)`). The default is provided in case the value type has no associated .NET constructor: it can be used to initialise a reference, then set up its state. For example:

```
let val r = ref Pair.null
in  (r.#x) := 1; (r.#y) := 2; !r
end
```

returns the value `Pair(1,2)`, without calling a constructor.

3.6 Enumeration Types

In .NET, an enumeration type is a distinct value type, declared with a set of named constants of that type. Every enumeration, in its boxed form, derives from class type `System.Enum`, a subclass of `System.ValueType`. Each enumeration type has an underlying (signed or unsigned) integral type (one of the SML.NET `int` or `word` types).

In SML.NET, a .NET enumeration type is imported as a pseudo datatype of the same name. The datatype has a single, unary constructor, named after the type, that constructs an enum from a value of the underlying type. The named constants of the enumeration are imported as equivalently named constant constructors, abbreviating particular applications of the proper constructor. The derived constructors are bound in a separate structure, named after the enumeration type. The proper constructor and its derived constant constructors can be used in patterns as well as expressions.

For example, values of the C♯ enumeration type:

```
public enum MyEnum { A , B , C = A }
```

may be manipulated in SML.NET as follows:

```
type enum = MyEnum
fun toString MyEnum.A = "A"
  | toString MyEnum.B = "B"
  | toString MyEnum.C = "C" (*redundant case*)
  | toString (MyEnum i) = Int.toString i
fun fromString "A" = MyEnum.A
  | fromString "B" = MyEnum.B
  | fromString "C" = MyEnum.C
  | fromString s = MyEnum(valOf(Int.fromString s))
```

Whether in a pattern or expression, writing `MyEnum.A` is completely equivalent to writing `MyEnum 0` instead.

Note that, in .NET, the constants defined for an enumeration type are rarely exhaustive, and often share the same value (e.g. `4` also has type `MyEnum` and `C=A=0` above). This feature distinguishes enumeration types from the familiar SML pattern of defining a finite type as a datatype with n constant constructors. Enums are often used as bit vectors, using constants as bit masks: for completeness, we must be able to map between all values of the underlying type.

3.7 Storage Types

SML.NET compiles values of type *ty* `ref` to an instance of a private class with a single mutable instance field of type *ty*. However, the CLI provides a much wider range of mutable locations: static fields of classes, instance fields of heap allocated objects and instance fields of stack allocated value types, all multiple and indexed by name. The CLI even provides a general address type to support call-by-reference parameter passing (C♯'s `ref` and `out` parameters). For interop, SML.NET has to support addresses as well.

In SML.NET, all of these types are described as particular instantiations of a more general type constructor, (*ty*,*kind*) `reference`. Unlike SML's `ref` type constructor, this type constructor takes *two* type parameters. The first simply describes the type of the value stored in the cell. The second parameter is a pseudo (or phantom) type that identifies the particular *kind* of storage cell (others have used related tricks for interop, for example Blume's FFI for SML/NJ [7]). The kind of a storage cell describes its physical representation and thus the precise runtime instructions needed to implement reads and writes.

The advantage of introducing a parametric notion of storage cell, indexed by kind, is that it allows SML.NET to treat dereferencing (`!`) and assignment (`:=`) as generic operations, polymorphic, as in ML, in the contents of a storage cell but also polymorphic in the physical representation of that cell. To achieve this, we generalise the Standard ML types of these operations to the following *kind*-polymorphic types in SML.NET:

```
val !   :  (ty,kind) reference -> ty
val :=  :  ((ty,kind) reference * ty) -> unit
```

The programmer gets to use the same notation to manipulate all kinds of storage cells, but it compiles to kind-specific instructions. As kind instantiations are fully specialized at compile-time they have no run-time representation or cost.

3.7.1 Storage kinds

The *kind* parameter of a storage cell of type (*ty*,*kind*) `reference` can take one of the following forms:

`heap:` used for ordinary Standard ML references, this kind describes a single-field, ML allocated object. SML's primitive *ty* `ref` type constructor merely abbreviates the SML.NET reference type:

```
type ty ref  =  (ty,heap) reference
```

so SML's allocating `ref` *function* actually has type:

```
val ref  :  ty -> (ty,heap) reference
```

(*classty*,*fieldname*)`static:` used for a static field, this kind describes a static field by the name of the class and the name of the field.

(*classty*,*fieldname*)`field:` used for an instance field, this kind describes the class or value type in which the field *fieldname* is declared.

`address:` used to describe the address of a storage cell. An address can point to the interior of an ML ref, to a static field, a stack-allocated local variable (e.g. from C♯), an instance field of a heap allocated object or an inlined value type. Addresses are an abstraction of all of the above storage types.

Address kinds are used to describe the type of C♯ call-by-reference `ref` and `out` parameters, and typically occur in the (argument) types of imported methods. Storage cells of different kinds have distinct types and are thus *not* type compatible, e.g. it is a static type error to create a list containing both an integer heap reference and an integer static field reference (because their kinds are not unifiable).

3.7.2 Address operators (&)

One can take the *address* of any kind of storage cell (including another address) using the SML.NET primitives:

```
type ty &  =  (ty,address) reference
val &      :  (ty,kind) reference -> ty &
           =  (ty,kind) reference ->
              (ty,address) reference
```

The `&` function allows one to pass any kind of SML.NET storage cell (including an SML `ref`) to a C♯ method expecting a call-by-reference parameter (marked as `ref` or `out` in C♯) by taking, and then passing, the address of that cell.

For instance, given the C♯ swap function:

```
public static void swap(ref int i, ref int j){
  int temp = i; i = j; j = temp; return;
}
```

which is imported with SML.NET type:

```
val swap : (int &) * (int &) -> unit
```

Then we can `swap` two (ML) references as follows:

```
val (ra,rb) = (ref 2,ref 3)
val _ = swap(& ra,& rb)
val (ref 3,ref 2) = (ra,rb)
```

Operationally, taking the address of a storage cell returns the address of the particular field storing its contents; taking the address of an address is simply the identity.

3.7.3 Byref types

The various kinds of storage in SML.NET fall into two categories: storage cells whose representations are first-class values of the CLI and storage cells whose compiled representations are second-class CLI *addresses*. This second category includes references of kind `address` and, less obviously, `(`*`classty`*`,`*`fieldname`*`)field`, where *classty* is a value type (not a reference type). In SML.NET, these are collectively called *byref* types.

In the CLI, an address can, amongst other things, be the address of a value allocated on the call-stack (e.g. an imported address of C♯ local variable), or the address of a field of a value on the stack (e.g. an imported address of a field of a C♯ local variable). Such an address is only valid for the lifetime of the stack frame from which it was taken. To prevent such ephemeral addresses from being read or written outside their lifetime (when they are unsafe dangling pointers), the CLI imposes (and the verifier checks) restrictions on values of address type: they cannot be stored in static fields or instance fields of CLI values or objects.

Because SML.NET's byref storage types compile to CLI addresses, which must obey the CLI's rules, byref values and types can only be used in limited ways. SML.NET imposes these restrictions (and some more, see [5]):

- A value of byref type must not occur as a free variable of any function or class declared within its scope. Otherwise the byref value might be stored on the heap in the closure's environment.
- A byref type cannot be used as the argument type of a datatype or exception constructor or component type of a first-class tuple (since these are heap allocated).
- When matching a structure to a signature, the implementation of any opaque type in that signature may not be a byref type.
- Finally, to ensure the above properties are preserved by SML's type instantiation of polymorphic values and parameterised type constructors, type arguments (explicit or inferred) cannot be byref types, nor may they be *kinds* describing byref types (with the obvious exceptions for `!`, `:=` and `&`).

All of the restrictions are designed to prevent a byref value being stored on the heap.[1] The restrictions explicitly allow addresses to be used as method parameters and arguments, supporting Pascal-style call-by-reference passing of L-values.

3.8 Defining new .NET reference types

The mechanisms described so far give the SML programmer access to .NET libraries, but they do not support the creation of new class libraries, nor do they allow for the specialisation of existing .NET classes with new methods coded in SML. For this, we adapt some MLj syntax:

dec	::=	`_classtype` ⟨*attribs*$_1$⟩⟨`[`*cmod*`]` ⟨*attribs*$_2$⟩⟩ *class-name pat* ⟨`:` *superdecs*⟩ `with` ⟨`local` *dec* `in`⟩ ⟨ *methoddecs* ⟩ `end`
superdec	::=	*ty* | *ty exp*
superdecs	::=	*superdec* | *superdec* `,` *superdecs*
methoddec	::=	⟨*attribs*⟩⟨`[`*mmod*`]`⟩*method-name pat* `=` *exp*
	|	⟨*attribs*⟩⟨`[`*mmod*`]`⟩*method-name* `:` *ty*
methoddecs	::=	*methoddec* | *methoddec* `and` *methoddecs*

This declares a new class type *class-name* as follows:

- The optional, typically absent, *attribs*$_1$ and *attribs*$_2$ list any class or constructor attributes (see Section 3.9).
- The optional class modifier in *cmod* can be `abstract` or `sealed` and has the same meaning as in C♯.
- The expression *class-name pat* acts as a 'constructor header', with *pat* specifying the formal argument (or tuple of arguments) to the constructor. Any variables bound in *pat* are available throughout the remainder of the class type construct. Multiple constructors are unnecessary and *not* supported.
- The optional *superdecs* specifies the superclass that *class-name* extends and any interfaces that it implements. The superclass clause contains an argument (or tuple of arguments) *exp* to pass to the superclass (*ty*) constructor. The remaining types in a *superdecs* clause must be interfaces.

[1] We believe SML.NET's optimizing rewrites preserve this property, but have not attempted a formal proof.

```
open System.Web.Services
_classtype WebFac() : WebService() with
local val SOME clsname = this.##ToString()
      fun fact(0,acc) = acc
        | fact(n,acc)= fact(n-1,n*acc)
in
    Fac(n,acc) = fact(n,acc)
and {WebMethodAttribute()
     where CacheDuration(60) end}
    Fac(n:int) = this.#Fac(n,1)
and ToString() = SOME("WebService:"^clsname)
end
```

Figure 4: A simple WebService in SML.NET

- *dec* is a set of SML declarations that are local to a single instance of the class.
- The optional *methoddecs* is a simultaneous binding of instance method declarations, defined using a syntax similar to that of ordinary functions, but with optional qualifiers `final` and `protected` preceding the method identifiers. An abstract method is declared by omitting is implementation, but declaring its (function) type. Methods may be overloaded and may have attributes.

The class body may refer to both to the class itself and to the current instance (via `this`). Allowing `this` to be bound with the local declarations as well as the method declarations does open up a type loophole: we dicuss the rationale for allowing this in [4].

Figure 4 shows a (contrived) implementation of a web service, defined by deriving a new class from `WebService` and attributing its exported factorial method. The local definition of string `clsname` calls the `ToString` method in the base class, using MLj's *exp*`.##`*method-name* syntax to access methods from the base class. The `ToString` method is an override and `Fac` is overloaded.

Very similar syntax is used to declare new interface types:

dec ::= `_interfacetype` ⟨*attribs*⟩
interface-name⟨`:` *superdecs*⟩
`with` ⟨ *methoddecs* ⟩ `end`

An interface has a name, but no constructor argument. The types in superdecs must all denote interfaces (not classes): these are interfaces extended by this declaration. The methods in *methoddecs* must all be abstract.

SML.NET users may also declare their own delegate types using a concise form of classtype declaration:

dec ::= `_classtype` ⟨*attribs*⟩ *delegate-name* `of` *ty*

The type argument *ty* of a delegate must be an SML function type. The declaration introduces the type *delegate-name*; the delegate constructor *delegate-name* of (higher-order) type *ty* `->` *delegate-name* and an implicit `Invoke` instance method of type *ty*.

For example, the `ThreadStart` delegate class of Section 3.3.2 could be declared within SML.NET as:

```
_classtype ThreadStart of unit -> unit
```

Unlike ordinary SML function types, delegates can be exported in the sense of Section 3.10, providing a rudimentary way of inter-operating at the level of first-class functions.

3.9 Custom Attributes

SML.NET, like C♯, enables programmers to use and declare new forms of custom meta-data using *attribute classes*. Programmers can annotate SML.NET code with *instances* of attribute classes, whether these classes were imported or declared within SML.NET itself. These attribute values may be retrieved at run-time using reflection or can be used statically by tools. Figure 4 already illustrates attaching an instance of the `WebMethodAttribute` to mark a method for publication in a web service.

In SML.NET, as in C♯, a new attribute class is defined by declaring a class that extends `System.Attribute`. Attributes can be attached to many declarations in an SML.NET program using *attribute expressions*. An attribute expression is an essentially static description of the data required to construct the corresponding instance of the attribute class. Syntactically, an attribute expression is simply an application of an instance constructor of some attribute class to a tuple of constant arguments. Here is another example of a `WebMethodAttribute`, this time calling a constructor that takes some arguments, including an enum value:

```
WebMethodAttribute(true,Disabled,60)
```

An attribute expression may be further modified by a sequence of (mutable) field and property initialisers, again supplied with constant values. The initialisers are executed in order, just after the instance is constructed by reflection (an example is the setting of `CacheDuration` in Figure 4).

Ignoring any type annotations used to resolve overloading, constant values must be literals or values immediately constructed from literals, of the following SML.NET types (required by the CLS): `string` (`option`), `char`, `bool`, `Word8.word`, `Int16.int`, `int`, `Int64.int`, `Real32.real`, `real`, any imported enumeration type, or `System.Type`.

This is the precise grammar of attribute expressions:

attribs ::= `{`⟨*attexpseq*⟩`}`
attexp ::= *exp* ⟨ `where` *namedargs* `end` ⟩
attexpseq ::= *attexp* | *attexp* `,` *attexpseq*
namedarg ::= *fieldname* `=` *exp* | *propertyname exp*
namedargs ::= *namedarg* | *namedarg* `,` *namedargs*

3.10 Exporting structures

By default, all SML.NET declarations are private to the generated executable or DLL. In order to make declarations available outside – even simply to expose an entry point – it is necessary to *export* selected top-level structures using the compiler's `export` command.

In the current version of the compiler it is only possible to export structures whose signature can be mapped directly back to .NET types. The following rules apply:

- Top-level SML structures export as sealed .NET classes.
- Value bindings with function type are exported as static methods. A function type is exportable if
  - Its result type is an exportable type.

  - Its argument type is either `unit`, an exportable *input* type, or a tuple of exportable input types. An argument type, or tuple component, may also be the address of an exportable input type.

- Value bindings with exportable types are exported as static read-only fields.
- Class, interface and delegate type declarations are exported as .NET (nested) classes, if:
  - The argument of a class constructor is either `unit`, an exportable input type, or a tuple of exportable input types (as for functions).
  - The argument of a delgate constructor is an exportable function type.
  - The methods all have exportable types (as for functions).
- A type is an *exportable type* if it is an interop type (see Section 3.2.2) that does not refer to non-exported class, interface or delegate types. A type is an *exportable input type* if it is an interop input type (see Section 3.2.2) that does not make use of non-exported class, interface or delegate types.
- All other bindings are non-exportable.

In future, SML.NET may support the export of arbitrarily *nested* structures, merging classes and substructures with equivalent names on export. This will provide a convenient way to define classes with both instance and static methods, thus mirroring the semantics of class import.

4. VISUAL STUDIO INTEGRATION

This section describes the use of our interop extensions in implementing the SML.NET plugin for Visual Studio.NET. Microsoft's Visual Studio Industry Partner Programme [3] exposes the internal APIs of the VS IDE to third-party developers. In particular, the VSIP Environment SDK exposes the APIs necessary for implementing a custom *language service*. Language services are used by the VS editor to provide language specific code navigation, syntax colouring, error reporting, and intelligent completion and help on source code identifiers (*Intellisense*).

VSIP provides three ways to build a language service:

1. Implementing one from scratch using the flexible, but difficult to use, low-level APIs exposed by the Environment SDK; these are the same APIs used by Microsoft to integrate its languages.
2. Implementing an instance of a much simpler interface, `IBabelService`, to be used by a pre-supplied generic language service, the `Babel` package. `IBabelService` provides a high-level, but less feature-rich abstraction over the low-level interfaces available in option 1.
3. Providing an instrumented C++-lex and -yacc grammar for your language that completes a boilerplate implementation of the same `IBabelService` interface.

We rejected option (1) as a monumental task and option (3) because it would have meant, at the very least, rewriting a new lexer and parser for SML.NET in C++. All of these APIs, including `IBabelService`, are classic COM interfaces, most suited for use by C++ clients. Fortunately, the CLR provides high-level support for interoperating with COM from managed code, both in client and server roles. We put SML.NET's CLI-interop extensions to the test by implementing, via the CLR's own COM-interop, a managed implementation of the COM `IBabelService` interface.

Even though SML.NET is implemented in SML of New Jersey (for performance reasons), SML.NET is capable of compiling its own sources (unlike MLj). This enabled the following strategy for constructing our language service:

- Import the COM `BabelService.tlb` type library as a CLR library, `BabelServiceLib.dll`, using Microsoft's `tlbimp` tool.
- Write a relatively small amount of SML.NET interop glue code (5 classes and 35 methods) implementing the interfaces in `BabelServiceLib.dll` using `_classtypes` that simply call into appropriate bits of the front-end of the SML.NET compiler.
- Using SML.NET, compile the combined glue code and front-end sources (containing pure SML) into a CLR-class library, `SmlBabelService.dll`.
- Register the `SmlBabelService` CLR-class library with COM to expose its classes as classic COM classes visible to the VS host (we use the `regasm` tool).

This approach allows us to re-use much of the SML.NET front-end: given the additional glue code, all we had to do was instrument the existing SML.NET lexer and parser to perform appropriate callbacks to the glue code. By sharing the front-end sources, we could do far more, with far less effort and duplication, than would have been possible by following the C++ route to language integration. The entire implementation required less than 4,000 lines of new code.

The anatomy of a Babel service is quite simple. When presented with a source document registered for that service, the IDE calls into the service to colour lines and occasionally, on a separate thread, parse the entire document. The `IBabelService` interface requires a `ColorLine` method that, given the cached lexer state at the beginning of a line, scans the line, reporting, via callbacks to a supplied COM class implementing `IColorSink`, the logical color (identifier, keyword, delimiter colour), logical status (identifier, keyword, delimiter status) of each token and any triggers associated with the cursor encountering that location in the line. Finally, the `ColorLine` method returns the new state at the end of the line, which is cached by the caller. Triggers are trip-wires in the source document that fire different kinds of parsing requests. These include requests to match delimiters such as the current brace, providing tool tip documentation for the current identifier or member completion on encountering the separator (such as '`.`') in a qualified identifier. Depending on the context, the job of the `IBabelService::ParseSource` method is either to register the locations of pairs of matching delimiters, report syntax errors or return an instance of the `IScope` COM interface that maps locations to Intellisense information.

Because we have access to the bootstrapped front-end of the compiler itself, we were able to do much more than merely parse files looking for syntax errors. Given a successful parse, the SML.NET language service will go on to

attempt to type-check the current source document using the SML.NET type checker. To the user, a type error appears just like a syntax error: a red-squiggly in the source code with tool tips providing a description. Crucially, type errors are reported early, without requiring a full build of the project. In a further extension, the existing type-checker was slightly modified to construct a map from source locations to inferred SML.NET types. By providing a custom implementation of the `IScope` interface (returned by the `ParseSource` method), we can consult this map and use the results of type inference to provide advanced Intellisense information. The language service determines the environment in which to type check the current document by consulting a well-known location on disk that records the dependency on external CLR libraries and internal SML modules involved in the last successful build of the project. From this it is easy to recover the environment of any imported modules or CLR libraries by consulting the pickled type information produced by the standalone SML.NET compiler. The CLR hosted language service is able to read the binaries produced by the SML of New Jersey hosted SML.NET compiler because the format of our files is implementation-independent [11].

5. ASP.NET

ASP.NET is the CLR-based framework for server-side programming of web applications. ASP.NET has many sophisticated features, but here we will just discuss an interesting interop issue which arose when using it to write simple dynamic web pages in ML.

ASP.NET can separate logic from layout in dynamic web pages. The layout is specified in a file written in an extended form of HTML (e.g. `mypage.aspx`), whilst the corresponding server-side code lives in a separate file (e.g. `mypage.aspx.cs`). The code declares a subclass of `System.Web.UI.Page` defining various event handlers, but how are elements on the `.aspx` page related to objects in the C♯ code?

The answer is that each control in the `.aspx` file has an ID attribute (e.g. `<Button ID="MyButton">`), and the page class declares an appropriately typed field of the same name (e.g. `Button MyButton;`). The framework transparently translates the `.aspx` file into another C♯ class, which *subclasses* the page class defined by the programmer. This automatically generated class initializes the control fields inherited from the parent class.

We did not consider this combination of inheritance and meta-programming in the design of SML.NET. We originally reasoned that no code with which one might wish to interoperate could have an interface which required clients to define classes with particular named fields, and hence omitted the definition of named fields from SML.NET classes. The reasoning is valid for external C♯ (say) libraries which are entirely written 'within the language', as library classes never subclass client ones, but ASP.NET uses code generation to go outside the language. As SML.NET doesn't allow the definition of fields, one cannot simply replace `mypage.aspx.cs` with `mypage.aspx.sml`.

In practice, this is not a problem (if it were, we could just add field definitions to SML.NET). The most convenient way to author ASP.NET pages is to use Visual Studio. This allows graphical editing of the layout (the aspx file) and automatically generates the corresponding skeleton for the C♯ (or indeed VB) code class (aspx.cs). Thus the most straightforward way to write functional server-side

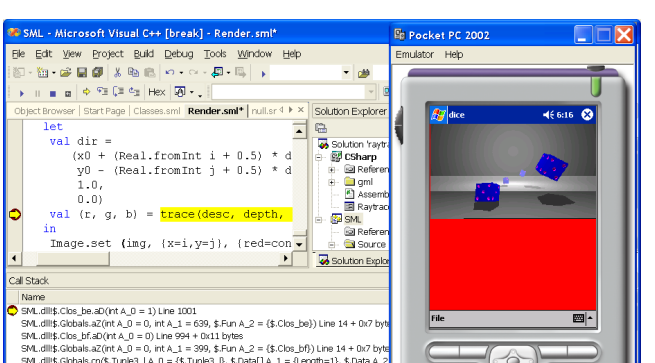

Figure 5: Developing for the Compact Framework.

code is mixed-language: one fills in the autogenerated C♯ stubs with calls to SML classes. We have successfully used this technique to implement, for example, a web-based interface to an interpreter for an XQuery-like language.

The mixed-language visual programming approach also works well for writing SML programs with more conventional GUIs. Although one can write graphical applications entirely within SML.NET, we usually build a mixed C♯/SML solution within Visual Studio, using the visual designers to generate C♯ which interoperates with SML.NET code to perform the interesting parts of the computation.

6. .NET COMPACT FRAMEWORK

The .NET Compact Framework (CF) [14] is an implementation of the CLI, supporting a only a subset of the Framework libraries but running on resource-constrained devices, such as Pocket PCs, with a variety of CPUs.

We have succesfully ported some smallish SML.NET applications to the Pocket PC using the CF. Figure 5 displays an SML.NET raytracer (with C♯ front-end) being debugged with VS while running on a Pocket PC emulator; it runs just the same on the real device. The SML code ($\sim$ 2,500 lines) is adapted from John Reppy's port from OCaml of the winning entry [18] to the 3^{rd} ICFP Programming Contest.

To make this work, we modified the relevant parts of the SML Basis to use only the .NET libraries that are part of the CF. We then ran into two resource problems. The first was stack size: the emulator (like older implementations of the CF for real devices) heap-allocates CLI method frames, but the current implementation on ARM processors uses a fixed-size stack. Worse, the OS gives the main thread of an application only 56KB of stack. Secondary threads, however, can request up to 1MB of stack (via a setting in the binary), so we simply ran the ML code in a new thread. The second limitation we ran into in this case was that current CF implementations reject methods with compiled bodies larger than 64KB. Crudely disabling some inlining options in SML.NET produced runnable binaries without source code modification, though at significant cost in performance; hacking the sources to introduce one or two barriers to inlining would be unpleasant but probably more effective.

A further unfortunate limitation of current CF implementations is that they do not respect the CLI tailcall instruction, so we are unlikely to be able to run SML.NET itself on

a mobile device in the near future. Nevertheless, being able to run SML code on a PDA at all is gratifying, especially as it comes (almost) for free.

7. CONCLUSIONS

It has often been claimed that the CLI (or JVM) is unsuitable as a compilation target for functional languages (or logic languages, or scripting languages, or whatever) because it lacks particular features. SML.NET provides good evidence to the contrary. There are other ML implementations on .NET, including $F^\sharp$ [17] and Moscow ML.NET [13]. These both take a more straighforward approach to compilation than SML.NET. This gives worse runtimes but allows separate compilation (and, in the case of Moscow ML, an interactive loop). Neither $F^\sharp$ nor Moscow ML seriously extends the language for interop, as we have done. However, because they use uniform .NET representations for ML types, it is possible to interoperate with ML programs by writing C$^\sharp$ or VB code which delves into those representations.

The two sweet spots for compiling research languages to the CLI seem to be, firstly, writing a quick and dirty compiler for a new language and, secondly, working harder to write a compiler with good interoperability features and respectable performance. In the first case, leveraging all the CLI infrastructure allows one to write a compiler which will produce code with better performance than a naive interpreter written in a high-level language but with comparable implementation effort. In the second case, one has to perform non-trivial optimizations, make compromises (in our case, separate compilation) and solve tricky language design problems, and still runtime performance will lag that of an optimizing native-code compiler for a single language. But as we have gained experience of the practical benefits of interoperability with industrially mainstream languages and tools, we have only become more convinced that these compromises are worth making. Having immediate and simple access to any .NET library really does make it easier to write functional programs which do practical things than in any other implementation of which we are aware. As more Windows APIs are exposed as managed code, these benefits will become greater (SML.NET code has already been shown running on an early build of Longhorn and accessing the new WinFx APIs).

The benefits of using SML.NET within Visual Studio have surprised us. Despite our all being longtime Emacs devotees, the convenience of Intellisense (especially when working with large object-oriented libraries) interactive type inference and (albeit limited) source-level debugging, have made it our preferred working environment.

The main limitation of SML.NET remains its slow compile-times. Recent work with Sam Lindley on using a mutable graph representation for the rewriting phases of the compiler indicate that an order of magnitude speedup in compilation is possible; we will report on this elsewhere.

Our most pressing future task is to extend SML.NET to support the forthcoming addition of parametric polymorphism (generics) to the CLI[12, 19]. Extending our extensions to interoperate with generic .NET code raises some challenging issues. (For example, our current treatment of null-values is not straightforwardly compatible with allowing class type parameters to range over both ML and .NET types.) We also plan to exploit generics in compiling SML itself, which may lead us back to a more traditional separate compilation model.

8. REFERENCES

[1] The MLton Compiler. `http://www.mlton.org/`.

[2] The SML.NET Compiler. `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/`.

[3] Visual Studio Industry Partner Programme. `http://msdn.microsoft.com/vstudio/extend/`.

[4] N. Benton and A. Kennedy. Interlanguage working without tears: Blending SML with Java. In *4th ACM SIGPLAN International Conference on Functional Programming, Paris, France*, September 1999.

[5] N. Benton, A. Kennedy, and C. Russo. The SML.NET 1.1 user guide. `http://www.cl.cam.ac.uk/Research/TSG/SMLNET/smlnet.pdf`.

[6] N. Benton, A. J. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*, September 1998.

[7] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01: First workshop on multi-language infrastructure and interoperability*, Sept. 2001.

[8] D. Box and C. Sells. *Essential .NET, Volume I: The Common Language Runtime*. Addison Wesley, 2003.

[9] Ecma International. ECMA and ISO C# and Common Language Infrastructure standards. `http://www.ecma-international.org/`.

[10] E. R. Gansner and J. H. Reppy, editors. *The Standard ML Basis Library reference manual*. Cambridge University Press, to appear.

[11] A. J. Kennedy. Functional pearl: Pickler combinators. *Journal of Functional Programming*, 2004.

[12] 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.

[13] N. J. Kokholm and P. Sestoft. Moscow ML.NET owner's manual. `http://www.dina.kvl.dk/~sestoft/mosml/netmanual.pdf`, Nov. 2003.

[14] Microsoft Corporation. The .NET Compact Framework. `http://msdn.microsoft.com/mobile/`.

[15] J. B. Miller and S. Ragsdale. *The Common Language Infrastructure Annotated Standard*. Addison Wesley, 2003.

[16] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Mass., 1997.

[17] D. Syme. The $F^\sharp$ compiler. `http://research.microsoft.com/projects/ilx/fsharp.aspx`.

[18] J. Vouillon, H. Hosoya, E. Sumii, and V. Gapeyev. Team PLClub. `http://www.cis.upenn.edu/~sumii/icfp/`, 2000.

[19] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET Common Language Runtime. In *ACM Symposium on Principles of Programming Languages*, 2004.