An Intermediate Language for Compilation to Scripting Languages*

Paola Giannini and Albert Shaqiri
Computer Science Institute, DiSIT, Università del Piemonte Orientale
Via Teresa Michel 11, 15121 Alessandria, Italy

Keywords: Scripting Languages, Functional Languages, Intermediate Language, Translation.

Abstract: In this paper we introduce an intermediate language for translation of F#, a functional language polymorphically typed relying on the .Net platform, to different scripting languages, such as Python and JavaScript. This intermediate language (IL for short) is an imperative language, with constructs that make possible to move a code fragment outside its definition environment, during the translation. Definition of names (variables and functions) are done in blocks, like in Python (and JavaScript) and do not have to statically precede their use. We present a translation of a core F# (including mutable variables) into IL.

1 INTRODUCTION

Implementing an application in JavaScript (or any other dynamically typed language) can cause problems due to the absence of type checking. Such problems can lead to unexpected application behaviour followed by onerous debugging. Although dynamic type checking and automatic type casting shorten the programming time, they introduce serious difficulties in the maintenance of medium to large applications. This is the reason why dynamically typed languages are used mostly for prototyping and quick scripting.

We propose to deal with these problems using dynamically typed languages as "assembly languages" to which we translate the source code from F# which is statically typed. In this way, we take advantage of the F# type checker and type inference system, as well as other F# constructs and paradigms such as pattern matching, classes, discriminated unions, namespaces, etc., and we may use the safe imperative features introduced via F# mutable variables. There are also the advantages of using an IDE such as Microsoft Visual Studio (code organization, debugging tools, IntelliSense, etc.).

To provide translation to different target languages we introduce an intermediate language, IL for short. This is useful, for instance, for translating to Python that does not have complete support for functions as first class concept, or for translating to JavaScript, using or not libraries such as jQuery.

Our aim is to prove the correctness of the compilers produced. To do that we formalize IL, and the translation from the source language to IL. The language IL is imperative, and has some of the characteristics of the scripting languages that makes them flexible, but difficult to check, such as blocks in which definition and use of variables may be interleaved, and in which use of a variable may precede its definition. (IL is partly inspired by IntegerPython, see (Ranson et al., 2008).) Therefore, the proof of correctness of the translation from the source language F# to IL already covers most of the gap from F# to the target scripting languages. In IL we also have some construct that may be used to manipulate safely fragments of open code.

The paper is organized as follows. In Section 2, we introduce the challenges of the translation from F# to Python and JavaScript via some examples, that led us to introduce our intermediate language. We also outline the translation from IL to both JavaScript and Python. In Section 3 we define the fragment of F# used as source language, and in Section 4 we formalize IL. The formal translation from F# to IL is defined in Section 5, where it is stated to preserve the dynamic semantics of F#. In Section 6 we compare our work with the work of others, and finally in Section 7 we summarize our work, discussing briefly the implementation issues and highlighting our plans for future work.

*This work has been partially supported by MIUR CINA-Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

2 TRANSLATION BY EXAMPLES: DESIGN CHOICES

In the fragment of F# we consider as source of our translation we have the typical functional language constructs: function definition and application, integers, booleans, addition and the conditional expression, and an imperative fragment including mutable variables, assignment, and sequences of expressions. On the left-hand-side of an assignment there must be a variable that was introduced with the `mutable` modifier.

2.1 Sequences of Expressions

Many F# constructs can be directly mapped to JavaScript (or Python), but when this is not the case we obtain a semantically equivalent behaviour by using the primitives offered by the target language. E.g., in F# a sequence of expressions is itself an expression, while in JavaScript and Python it is a statement. Suppose we want to translate a piece of code that calculates a fibonacci number, binds the result to a name and also stores the information if the result is even or odd. In Fig. 1 we have one possible F# implementation.

```
let z=7
let mutable even = false
let x =
   let rec fib x =
      if x < 3 then 1
      else fib(x - 1) + fib(x - 2)
   let temp = fib z
   even <- (temp % 2 = 0)
   temp
x
```

Figure 1: F# program containing sequence of expressions.

As we can see, on the right-hand-side of "`let x=`" we have a sequence of expressions: the definition of the function `fib` followed by the definition of `temp`, etc. This sequence is, in F#, an expression. If we directly map this code into JavaScript we obtain the syntactically incorrect code of Fig. 2. This program is syntactially wrong, since on the right-hand-side of an assignment we must have an expression, while a sequence of expressions is, in JavaScript, a statement. To transform a sequence of statements in an expression, in JavaScript, we wrap the sequence into a function, and to execute it we call the function, i.e., we use a JavaScript closure and application. Also, the whole program is wrapped into an entry point function. In this way, the code of Fig. 3 is correct. Unfortunately, the same cannot be done in Python as its support for

```
var z = 7;
var even = false;
var x =
   var fib = function (x) {
      if (x < 3) return 1;
      else return fib(x-1)+fib(x-2)};
   var temp = fib(z);
   even = (temp % 2) == 0;
   temp;
return x;
```

Figure 2: Naive translation into JavaScript of sequence of expressions.

```
(function() {
   var z = 7;
   var even = false;
   var x = (function () {
      var fib = function (x) {
         if (x < 3) return 1;
         else return fib(x-1)+fib(x-2)};
       var temp = fib(z);
       even = (temp % 2) == 0;
       return temp })();
  return x })();
```

Figure 3: Correct JavaScript translation.

closures is partial. So we have to define a temporary function, say `temp1`, in the global scope and to execute it we have to call `temp1` in the place where the original sequence should be. However, variables such as `even` will be out of the scope of their definition, and this would make the translation wrong. To obtain a behaviour semantically equivalent, we have to pass to `temp1` the variable `even`, by reference, since it may be modified in the body of `temp`. Note that, this problem is not present in JavaScript where the closure is defined and called in the scope of `even`. Another problem in Python is related to lambdas, whose body must be an expression (not a sequence). So we define the function `temp2` whose body contains the statements that should be placed where an expression is expected. In Fig. 4 we can see the translation of the F# code into Python. The class `ByRef` is used to wrap the mutable variable `even` to obtain a parameter called by reference. The Python code generator inserts the needed wrapping and unwrapping before and after the call of `temp1`, and in the body of `temp1`.

The problem we illustrated above occurs whenever in the target language we get a statement where an expression is expected. Since the target languages handle the situation differently, we abstract from this specific problem, and consider the more general problem of moving "open code" from its context, replacing it with an expression having the same behaviour. Taking inspiration from work on dynamic binding,

```
def temp1(w, z):
   def temp2(w, fib, x):
        if (x < 3):  return 1
        else: return fib(x-1)+fib(x-2)
   fib = lambda x: temp2(w, fib, x)
   temp = fib(z)
   w.value = ((temp % 2) == 0)
   return temp
def __main__():
   z = 7
   even = False
   wrapper1 = ByRef(even)
   x = temp1(wrapper1, z)
   even = wrapper1.value
   return x
__main__();
```

Figure 4: Correct Python translation.

see (Nanevski, 2003) and recent work by the authors, see (Ancona et al., 2013), we define a pair of boxing/unboxing contructs, that we call: `stm2exp`, and `exc`. The construct `stm2exp` wraps “open code” (in this case a sequence of expressions) providing the information on the environment needed for its execution, that is the mutable and immutable variables occurring in it. This construct defines a value, similar to a function closure. The construct `exc` is used to execute the code contained in `stm2exp`. To do this it must provide values for the immutable variables, in our example the variable `z`, and bindings for the mutable variables to variables in the current environment, since when executing the code we have to modify the variable `even`.

With these constructs, the `F#` code of Fig. 1 would be translated into the IL code in Fig. 5. All the `let` constructs are translated to variable definitions. The sequence of statements on the right-hand-side of “`let x=`” is packed into a `stm2exp` expression. Its first component is the translation of the sequence of statements, the second `w->EV` says that in the execution environment there should be a rebinding of the global name `EV` to a variable. Such variable may (in this case will) be modified by the execution of the code through assignment to the local variable `w`. The third component says that a value for `u` must be provided. The variable `u` is not modified by the execution of the code. We choose to use global names to unbind/rebind mutable variables, `w` in our example, so that the local variables can be consistently renamed without affecting the semantics of the construct as formal parameters of functions. Instead names such as `EV` are global to the whole program.

```
def y = stm2exp(
   def fib =
      fun x ->
         if x < 3 then 1
         else (fib (x-1) + fib (x-2));
   def temp = fib u;
   w <- temp % 2 = 0;
   temp,
   w->EV,  u);
def z = 7;
def even = false;
def x = exc(y, EV->even, z);
x
```

Figure 5: Translation of `F#` sequence of expressions in the intermediate language.

To obtain the result that we would have by evaluating the sequence of statements in the current environment, to the variable `x` it is assigned the `exc` expression applied to `y`, which is bound to `stm2exp(···)`. The name `EV` is bound to the (mutable) variable `even` and the variable `u` will be assigned the value of the variable `z`. Regarding the different treatment of mutable and immutable variables, notice that, even though our intermediate language is imperative, we know, since we are translating `F#` code that some variables are immutable, so we have to provide just an initial value.

The constructs `stm2exp` and `exc` have a different translation into the target languages JavaScript and Python, in particular for JavaScript we can take advantage from the fact that the closure wrapping the code can be inlined in the position where we have `exc`, so we can substitute both the mutable and immutable variables, instead the translation to Python treats the two kind of variables differently.

2.2 Dynamic Type Checking

JavaScript, and many dynamically typed languages, lack a rigorous type system. On the contrary, in `F#` if we write a function that adds two integers, say:

```
let add x y = x + y
```

we get

```
val add : int -> int -> int
```

because, even though we do not specify type information, the interpreter infers the type shown after the function definition. Therefore, there is no way of calling `add` with arguments that are not of type integer. However, if our translation in the intermediate code would produce a function whose body was simply `x+y`, which in turn could be translated in the corresponding expression in both JavaScript and Python, the target JavaScript function could be called, e.g., `add("foo")(1)` and obtain the string `"foo1"` which is not what we wanted. In Python the situation would be better, in the sense that we cannot call `add` on a string and an integer, however, due to overloading we can call it on two floating points obtaining a floating point. To prevent this, the translation in the intermediate language, which follows, insert dynamic checks on parameters of functions.

```
def add = fun x ->
  def x1 = check(int, x);
  fun y ->
    def y1= check(int, y);
    x1 + y1;
```

These checks are translated into dynamic type checking in JavaScript and Python. In JavaScript we use the function `checkInt` (that we defined) that returns its argument if it is an integer, and fails, raising an exception, if the parameter is not an integer:

```
var add = function (x) {
   var x1 = checkInt(x);
   return function(y) {
      var y1 = checkInt(y);
      return x1 + y1 } }
```

Similarly for Python:

```
def temp__1(y, x):
   y1 = checkInt(y)
   return  (x + y1)

def temp__2(x):
   x1 = checkInt(x)
   return lambda y: temp__1(y, x1)

add = lambda x: temp__2(x)
```

3 CORE F#

The syntax for the core F# language is presented in Fig.6. We sacrificed minimality to clarity, including constructs, such as `let`, `let mutable`, and `let rec` that are used in the practice of programming and that raise challenges in the translation to dynamic languages. We also did not introduce imperative features through reference types, but through mutable variables, since this is closer to the imperative style of programming. Moreover, we present a typed version of F# without type inference, since this is performed by the F# compiler. In the type system we omit type variables, as they do not add complexity to the translation.

$$\begin{array}{lll} e & ::= & x \mid n \mid \texttt{tr} \mid \texttt{fls} \mid e\texttt{+}e \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \\ & & \mid \mathbf{fun}\ x{:}T\texttt{->}e \mid \texttt{let}\ [\texttt{mutable}]\ x{=}e\ \mathbf{in}\ e \\ & & \mid e\ e \mid \mathbf{let\ rec}\ \overline{x}{:}\overline{T}{=}\overline{v}\ \mathbf{in}\ e \mid x\texttt{<-}e \mid e,e \\ T & ::= & \texttt{int} \mid \texttt{bool} \mid T \to T \\ v & ::= & n \mid \texttt{tr} \mid \texttt{fls} \mid \mathbf{fun}\ x{:}T\texttt{->}e \end{array}$$

Figure 6: Syntax of F#.

In the grammar for expressions, in Fig.6, the square brackets "[...]" delimit an optional part of the syntax, we use x, y, z for variable names, and the overbar sequence notation is used according to (Igarashi et al., 2001). For instance: "$\overline{x}{:}\overline{T}{=}\overline{v}$" stands for "$x_1{:}T_1{=}v_1 \cdots x_n{:}T_n{=}v_n$". The empty sequence is denoted by "$\emptyset$". For an F# expressions e the *free variables of* e, $FV(e)$ are defined in the standard way. An expression e *is closed* if $FV(e) = \emptyset$.

The `let rec` construct introduces mutually recursive variables. Variable names, in this constructs are meant to be bound to functions (as seen for `fib` in the example of Fig. 1). The `let` construct (followed by an optional `mutable` modifier) binds the variable `x` to the value resulting from the evaluation of the expression on the right-hand-side of $=$ in the evaluation of the body of the construct. As usual the notation `let` $f\ x{=}e_1$ `in` e_2 is a short hand for `let` $f{=}$**fun** $x{:}T\texttt{->}e_1$ **in** e_2 where T is the type of e_1. Similarly for `let rec`. In the (concrete syntax) of the examples, as in F#, "," and `in` are substituted by a return without indentation.

When the `let` construct is followed by `mutable` the variable introduced is mutable. Only mutable variables may be used on the left-hand-side of an assignment. This restriction is enforced by the type system of the language. The type system enforces also the restriction that the body of a function cannot contain free mutable variables, even though it may contain bound mutable variables. So, the function `f` in Fig. 7 is not correct, whereas the definition of `g` that follows is correct. A type environment Γ is defined by:

```
let mutable z = 0

let f x  =
   if (x > 0) then z <- x
   else z <- -x
   z

let g x  =
   let mutable w = 0
   if (x > 0) then w <- x
   else w <- -x
   w

 z
```

Figure 7: Typing functions in F#.

$$\Gamma ::= x{:}T, \Gamma \mid x{:}T!, \Gamma \mid \emptyset$$

that is Γ associates variables with types, possibly followed by ! . If the type is followed by ! this means that the variable was introduced with the `mutable` modifier. Let $\dagger$ denote either ! or the empty string, and let $dom(\Gamma) = \{x \mid x{:}T\dagger \in \Gamma\}$. We assume that for any variable x, in Γ there is at most an associated type. We say that the *expression e has type T in the environment* Γ if the judgement

$$\Gamma \vdash e : T$$

is derivable from the rules of Fig. 8. In the rules of Fig. 8, with $\Gamma[\Gamma']$ we denote the type environment such that $dom(\Gamma[\Gamma']) = dom(\Gamma) \cup dom(\Gamma')$ and:

- if $x{:}T\dagger \in \Gamma'$ then $x{:}T\dagger \in \Gamma[\Gamma']$, and
- if $x{:}T\dagger \in \Gamma$ and $x \notin dom(\Gamma')$, then $x{:}T\dagger \in \Gamma[\Gamma']$.

In the following we describe the most interesting rules.
Consider rule (TyAbs): to type the body of a function we need assumptions on its free variables and formal parameter. From the definition of $\Gamma[\Gamma']$ we have that the assumptions on its free variables must coincide with the one present in the environment of the definition of the function. Moreover, none of them may have been declared as mutable. However, in the environment in which the function is defined, $\Gamma[\Gamma']$, there can be mutable variables, as long as they are not needed to type the body of the function. In the example of Fig. 7, if the definition of the function `f` were typable, it should have been typed from the environment $\Gamma[\Gamma'] = z{:}\texttt{int!}$, therefore, to type its body we would have used the environment $z{:}\texttt{int!}, x{:}\texttt{int}$, i.e., $\Gamma' = z{:}\texttt{int!}$. However, this is not possible. Instead, the definition of `g`, which is again typed in $\Gamma[\Gamma'] = z{:}\texttt{int!}$, not having z free in its body, can be typed from $x{:}\texttt{int}$, by defining $\Gamma' = \emptyset$.
The rules (TyLet) and (TyLetMut) bind a variable, x, to the expression e_1 in the expression e_2. So the expression e_2 is typed in a type environment in which x is associated with the type of e_1.
In the rule (TyLetMut) the type is followed by ! so that inside e_2 the variable x may be used on the left-hand-side of an assignment, see rule (TyAssign).

Our core `F#` language has imperative features, so for the definition of the operational semantics we use a store. The *runtime configurations* are pairs “expression, store”, $e \mid \rho$, where a *store* ρ is a mapping between locations and values:

$$l_1 \mapsto v_1, \ldots l_n \mapsto v_n$$

In Fig. 9 we define:

- *runtime expressions*, which are expressions including locations (generated by the evaluation of mutable variables definitions);
- *evaluation contexts* defining, in conjunction with rule (Ctx-F), the reduction strategy of the language, which is call-by-value, with evaluation left-to-right, and
- the *rules for the evaluation relation*, $\longrightarrow$.

In the rules, with $e[x := e']$ we denote the result of *substituting x with e' in e* with renaming if needed. Moreover, $\rho[x \mapsto v]$ is defined by: $\rho[x \mapsto v](x) = v$, and $\rho[x \mapsto v](y) = \rho(y)$, when $x \neq y$.

The evaluation of the sum expression assumes that the operand be integers, and returns n, which is the numeral corresponding to the sum of the values of n_1 and n_2. For the conditional statements we have two rules corresponding to the (boolean) value of the condition. Both the evaluations of the application, rule (App-F), and `let`, rule (Let-F), substitute x with its the value in the body of the construct. This is in accord with the fact that x is immutable. Instead, for a variable defined `mutable`, rule (LetMut-F) , a new location l is generated, added to the store with the initial value v, and the variable x is substituted with l. Therefore, during evaluation, expressions may contain locations. Indeed, since variables on the left-hand-side of assignments where always introduced by `let mutable`, when an assignment is evaluated, rule (Assign-F), we have a configuration: l`<-`$v \mid \rho$ which is evaluated by changing the value of the location l to be v. The evaluation of `let rec`, rule (Let-F), produces the body e in which each variable x_i is substituted with a `let rec` expression with body v_i, so that if x_i is evaluated all the variables $\overline{x}$ will be substituted with their definitions $\overline{v}$. Evaluation of a location, rule (Loc-F), produces the value associated in the store. Finally in rule (Ctx-F) the context $\mathcal{E}$ selects the first sub-expression to be evaluated. We can show that *evaluation is deterministic*.

The typing rules in Fig.8 are for the (source) expression language, so they do not include a rule for locations. To type run-time expressions we need a store environment Σ assigning types to locations. The type judgement should therefore be:

$$\Gamma \mid \Sigma \vdash e : T$$

and the typing rule for locations

$$\Gamma \mid \Sigma \vdash l : \Sigma(l) \quad \text{(TyLocF)}$$

All the other rules are obtained by putting $\Gamma \mid \Sigma$ on the left-hand-side of $\vdash$ in the typing rules of Fig.8.

Definition 1. *A store ρ is* well-typed *with respect to a type environment Γ, and a store environment Σ, written $\Gamma \mid \Sigma \vdash \rho$, if $dom(\rho) = dom(\Sigma)$, and for all $l \in \rho$, we have that $\Gamma \mid \Sigma \vdash \rho(l) : \Sigma(l)$.*

Types are preserved by reduction, and progress holds, as the following two theorems state.

Theorem 2 (Preservation). *Let $\Gamma \mid \Sigma \vdash e : T$, and ρ be such that $\Gamma \mid \Sigma \vdash \rho$. If $e \mid \rho \longrightarrow e' \mid \rho'$, then $\Gamma \mid \Sigma' \vdash e' : T$, for some $\Sigma' \supseteq \Sigma$ such that $\Gamma \mid \Sigma' \vdash \rho'$.*

Theorem 3 (Progress). *Let $\emptyset \mid \Sigma \vdash e : T$, then either e is a value or for any store ρ such that $\emptyset \mid \Sigma \vdash \rho$ there are, e', and ρ' such that $e \mid \rho \longrightarrow e' \mid \rho'$.*

$$\Gamma \vdash n : \texttt{int} \quad \text{(TyNum)} \qquad\qquad \Gamma \vdash \texttt{tr},\texttt{fls} : \texttt{bool} \quad \text{(TyBool)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash_e e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{+}e_2 : \texttt{int}}\ \text{(TySum)} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : T}\ \text{(TyIf)}$$

$$\frac{\Gamma'[x{:}T] \vdash e : T' \quad \forall y, T''\ y{:}T''! \notin \Gamma'}{\Gamma[\Gamma'] \vdash \texttt{fun}\ x{:}T\ \texttt{->}e : T \to T'}\ \text{(TyAbs)} \qquad \frac{\Gamma \vdash e_1 : T \to T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T}\ \text{(TyApp)}$$

$$\frac{x{:}T\dagger \in \Gamma}{\Gamma \vdash x : T}\ \text{(TyVar)} \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma[x{:}T] \vdash e : T'}{\Gamma \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : T'}\ \text{(TyLet)}$$

$$\frac{\begin{array}{l}\Gamma[\overline{x}{:}\overline{T}] \vdash v_i : T_i\ (1 \leq i \leq n)\\ \Gamma[\overline{x}{:}\overline{T}] \vdash e : T\end{array}}{\Gamma \vdash \texttt{let rec}\ \overline{x}{:}\overline{T}\texttt{=}\overline{v}\ \texttt{in}\ e : T}\ \text{(TyRec)} \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma[x{:}T!] \vdash e : T'}{\Gamma \vdash \texttt{let mutable}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : T'}\ \text{(TyLetMut)}$$

$$\frac{\Gamma \vdash e : T \quad x{:}T! \in \Gamma}{\Gamma \vdash x\texttt{<-}e : T}\ \text{(TyAssign)} \qquad \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T'}{\Gamma \vdash e_1, e_2 : T'}\ \text{(TySeq)}$$

Figure 8: Typing rules of core F#.

$$\begin{array}{lll} e & ::= & \cdots \mid l \\ \mathcal{E} & ::= & [\,] \mid \mathcal{E}\texttt{+}e \mid n\texttt{+}\mathcal{E} \mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ e\ \texttt{else}\ e \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{let}\ [\texttt{mutable}]\ x\texttt{=}\mathcal{E}\ \texttt{in}\ e \\ & & \mid u\texttt{<-}\mathcal{E} \mid \mathcal{E}, e \end{array} \qquad \begin{array}{l}\text{runtime expression}\\ \text{evaluation contexts}\end{array}$$

$$\begin{array}{lll}
n_1\texttt{+}n_2 \mid \rho \longrightarrow n \mid \rho & \text{if} \quad \tilde{n} = \tilde{n}_1 +^{\texttt{int}} \tilde{n}_2 & \text{(Sum-F)}\\
\texttt{if tr then}\ e_1\ \texttt{else}\ e_2 \mid \rho \longrightarrow e_1 \mid \rho & & \text{(IfTrue-F)}\\
\texttt{if fls then}\ e_1\ \texttt{else}\ e_2 \mid \rho \longrightarrow e_2 \mid \rho & & \text{(IfFalse-F)}\\
(\texttt{fun}\ x{:}T\ \texttt{->}e)\ v \mid \rho \longrightarrow e[x := v] \mid \rho & & \text{(App-F)}\\
\texttt{let}\ x\texttt{=}v\ \texttt{in}\ e \mid \rho \longrightarrow e[x := v] \mid \rho & & \text{(Let-F)}\\
\texttt{let rec}\ \overline{x}{:}\overline{T}\texttt{=}\overline{v}\ \texttt{in}\ e \mid \rho \longrightarrow & & \\
\quad e[x_i := (\texttt{let rec}\ \overline{x}{:}\overline{T}\texttt{=}\overline{v}\ \texttt{in}\ v_i) \mid 1 \leq i \leq n] \mid \rho & & \text{(Rec-F)}\\
\texttt{let mutable}\ x\texttt{=}v\ \texttt{in}\ e \mid \rho \longrightarrow e[x := l] \mid \rho[l \mapsto v] & l \notin dom(\rho)\ \text{new} & \text{(LetMut-F)}\\
l\texttt{<-}v \mid \rho \longrightarrow v \mid \rho[l \mapsto v] & l \in dom(\rho) & \text{(Assign-F)}\\
v, e \mid \rho \longrightarrow e \mid \rho & & \text{(Seq-F)}\\
l \mid \rho \longrightarrow v \mid \rho & \text{if}\ \rho(l) = v & \text{(Var-F)}
\end{array}$$

$$\frac{e \mid \rho \longrightarrow e' \mid \rho' \quad \mathcal{E} \neq [\,]}{\mathcal{E}[e] \mid \rho \longrightarrow \mathcal{E}[e'] \mid \rho'}\ \text{(Ctx-F)}$$

Figure 9: Operational semantics of core F#.

4 INTERMEDIATE LANGUAGE

The intermediate language, IL, is an imperative language with three syntactic categories: expressions, statements and blocks. We introduce the construct that wraps code that need to be moved from its definition environment, and the one that executes such code in the runtime environment.

The syntax of IL is presented in Fig.10.

There are three syntactic categories: *blocks*, *statements*, and *expressions*. We introduce the distinction between expressions and statements as many target languages do. This facilitates the translation process and prevents some errors while building the intermediate abstract syntax tree, see (Appel, 1998) for a similar choice. Blocks are sequences of statements or expressions ended by an expression. In our translation we flatten the nested structure of `let` constructs so we need blocks in which definitions and expressions/statements may be intermixed. Moreover, since we do not have a specific `let rec` construct use of a variable may precede its definition, e.g., when defining mutually recursive (or simply recursive) functions. Statements may be either assignments or variable definitions. Our compiler handles many more statements, but these are enough to show the ideas

$$
\begin{array}{lll}
bl & ::= & st;bl \mid e;bl \mid e \\
st & ::= & x\texttt{<-}e \mid \texttt{def}\ x\texttt{=}e \\
e & ::= & x \mid n \mid \texttt{tr} \mid \texttt{fls} \mid e\texttt{+}e \mid \texttt{fun}\ x\texttt{->}\{bl\} \mid e\ e \\
 & & \mid \texttt{if}\ e\ \texttt{then}\ \{bl\}\ \texttt{else}\ \{bl\} \mid \texttt{check}(T_p, e) \\
 & & \mid \texttt{stm2exp}(\{bl\}, \overline{y} \mapsto \overline{Y}, \overline{x}) \\
 & & \mid \texttt{exc}(e, \overline{Y} \mapsto \overline{y}, \overline{e}) \\
T_p & ::= & \texttt{int} \mid \texttt{bool} \\
v & ::= & n \mid \texttt{tr} \mid \texttt{fls} \mid \texttt{fun}\ x{:}T\texttt{->}\{bl\} \\
 & & \mid \texttt{stm2exp}(\{bl\}, \overline{y} \mapsto \overline{Y}, \overline{x})
\end{array}
$$

Figure 10: Syntax of IL.

behind the design of IL. Our intermediate language is inspired (especially for the block structure) to IntegerPython, see (Ranson et al., 2008). Variables are statically scoped, in the sense that, if there is a definition of the variable x in a block, all the free occurrences of x in the block refer to this definition. However, we can have occurrences of x preceding its definition. E.g.,

```
def f = fun y -> { x };
def x = 5;
f 2
```

correctly returns 5, whereas the following code would produce a run-time error:

```
def x =7;
if (x > 3) then {
   def f = fun y -> { x };
   f 2
   def x = 5;
   3 }
else { 4 }
```

since when f is called the variable x, defined in the inner block, has not yet been assigned a value. Instead, if x was not defined in the inner block, like in the following

```
def x =7;
if (x > 3) then {
   def f = fun y -> { x };
   f 2 }
else { 4 }
```

the block would return 7, since x is bound in the enclosing block. This is also the behaviour in JavaScript and Python.

The construct stm2exp is used to move a block, bl, outside its definition context. To produce a closed term, the *mutable variables* free in bl, $\overline{y}$, are unbound by associating them to *global names* $\overline{Y}$ not subject to renaming. The variables $\overline{x}$, instead, are *immutable variables* free in bl, i.e., they are not modified by the execution of bl. The metavariables, X, Y, Z are used to denote names.

The operational semantics of IL, see Fig. 11, is given, by defining a reduction relation for blocks. So our configurations will be pairs: "block, store". In order to specify the order of reduction we define evaluation contexts for blocks, containing evaluation contexts for expressions. As for F# we have to add to the syntax of expressions locations, l, as they are generated during the evaluation of blocks. Moreover, we add two constructs wrapping blocks: $\{bl\}$ and $\texttt{eval}(bl)$. The first will be used to do the initial allocation of variables needed to reproduce the previously described semantics, and the second to execute a block in a position where an expression would be required. Note that these expressions are not in IL but are just introduced to describe its semantics.

As for F#, the evaluation contexts of Fig. 11 specify a call-by-value, left-to-right reduction strategy.

The first rule is used before the evaluation of a block to allocate the variables defined in a block. The function def mapping a block to the set of variables defined in it is defined by:

- $def(e) = \emptyset$,
- $def(e;bl) = def(x\texttt{<-}e;bl) = def(bl)$, and
- $def(\texttt{def}\ x\texttt{=}e;bl) = \{x\} \cup def(bl)$.

The initial value of the locations is set to undefined, ?, so if an access to a variable is done before the evaluation of an assignment or a definition for this variable *undErr* is returned. Note that, *this will never happen for* IL *programs which are translation of* F# *programs*. After this initial allocation a block will not contain free variables (but locations).
Rules (Assign) and (Def) continue the execution of the expressions/statements in a block in a store in which the value of location l is v. So after this the value of l is not undefined. Rule (Exp) throws away the value of an expression and continues the execution of the block. The rules for $+$, and if are trivial. Rule (App) allocates a location in the memory, assigning the value of the actual parameter to it, then the location is substituted for the formal parameter in the body of the function. Note that, being in an imperative language, the formal parameter could be modified in the body of the function, however, this change would not be visible in the calling environment, since the location is new. After this allocation the execution continues with the evaluation of the body $\{bl\}$, i.e., applying rule (Alloc). The rules (TypeYes), and (TypeNo) check whether a value is of the right primitive type. The function *typeof* from values to types is defined by: $typeof(\texttt{tr}) = typeof(\texttt{fls}) = \texttt{bool}$, $typeof(n) = \texttt{int}$, and undefined for the other values. The evaluation of the exc construct, rule (StToExp), expects the first argument to be a stm2exp, such that the names of its unbindings are a subset of the one of the rebindings provided by exc. If this is the case, it allocates new loca-

$$\begin{array}{lcll}
e & ::= & \cdots \mid l \mid \{bl\} \mid \texttt{eval}(bl) & \text{runtime expression}\\
\mathcal{S} & ::= & l\texttt{<-}\mathcal{E};bl \mid \texttt{def }l\texttt{=}\mathcal{E};bl \mid \mathcal{E};bl \mid \mathcal{E} & \text{block evaluation context}\\
\mathcal{E} & ::= & [\,] \mid \mathcal{E}\texttt{+}e \mid n\texttt{+}\mathcal{E} \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \texttt{if }\mathcal{E}\texttt{ then }\{bl\}\texttt{ else }\{bl\} \mid \texttt{check}(T_p,\mathcal{E}) & \text{expression evaluation context}\\
 & & \mid \texttt{exc}(\mathcal{E},\overline{Z}\mapsto\overline{l},\overline{e}) \mid \texttt{exc}(v,\overline{Z}\mapsto\overline{l},\overline{v}\,\mathcal{E}\,\overline{e}) \mid \texttt{eval}(\mathcal{S}) &
\end{array}$$

$$\begin{array}{lll}
\{bl\} \mid \rho \longrightarrow bl[\overline{x}:=\overline{l}] \mid \rho[\overline{l}\mapsto\overline{?}] & \text{if } \overline{x}=\mathit{def}(bl) & (\text{ALLOC})\\
 & \quad \overline{l}\notin \mathit{dom}(\rho)\text{ new} & \\
l\texttt{<-}v;bl \mid \rho \longrightarrow bl \mid \rho[l\mapsto v] & & (\text{ASSIGN})\\
\texttt{def }l\texttt{=}v;bl \mid \rho \longrightarrow bl \mid \rho[l\mapsto v] & & (\text{DEF})\\
v;bl \mid \rho \longrightarrow bl \mid \rho & & (\text{EXP})\\
n_1\texttt{+}n_2 \mid \rho \longrightarrow n \mid \rho & \text{if}\quad \tilde{n}=\tilde{n}_1 +^{\texttt{int}} \tilde{n}_2 & (\text{SUM})\\
(\texttt{fun }x\texttt{->}\{bl\})\ v \mid \rho \longrightarrow \{bl[x:=l]\} \mid \rho[l\mapsto v] & \quad l\notin \mathit{dom}(\rho)\text{ new} & (\text{APP})\\
\texttt{if tr then }bl_1\texttt{ else }bl_2 \mid \rho \longrightarrow \{bl_1\} \mid \rho & & (\text{IFTRUE})\\
\texttt{if fls then }bl_1\texttt{ else }bl_2 \mid \rho \longrightarrow \{bl_2\} \mid \rho & & (\text{IFFALSE})\\
\texttt{check}(T_p,v) \mid \rho \longrightarrow v \mid \rho & \text{if}\quad \mathit{typeof}(v)=T_p & (\text{TYPEYES})\\
\texttt{check}(T_p,v) \mid \rho \longrightarrow \mathit{typeErr} & \text{if}\quad \mathit{typeof}(v)\neq T_p & (\text{TYPENO})\\
\texttt{exc}(\texttt{stm2exp}(\{bl\},\overline{y}\mapsto\overline{Y},\overline{x}),\overline{Z}\mapsto\overline{l}',\overline{v}) \mid \rho \longrightarrow & \text{if}\quad \overline{Y}\subseteq\overline{Z} & (\text{STTOEXP})\\
\quad \texttt{eval}(\{(bl[\overline{x}:=\overline{l}])[y_i:=l'_j \mid Y_i=Z_j\ \ 1\leq i\leq n]\}) \mid \rho[\overline{l}\mapsto\overline{v}] & \quad \overline{l}\notin \mathit{dom}(\rho)\text{ new} & \\
\texttt{eval}(v) \mid \rho \longrightarrow v \mid \rho & & (\text{EVAL})\\
l \mid \rho \longrightarrow v \mid \rho & \text{if } \rho(l)=v & (\text{LOCDEF})\\
l \mid \rho \longrightarrow \mathit{undErr} \mid \rho & \text{if } \rho(l)=? & (\text{LOCUND})
\end{array}$$

$$\frac{e \mid \rho \longrightarrow e' \mid \rho' \quad \mathcal{S}\neq[\,]}{\mathcal{S}[e] \mid \rho \longrightarrow \mathcal{S}[e'] \mid \rho'}\,(\text{CTX}) \qquad \frac{e \mid \rho \longrightarrow \mathit{err} \quad \mathit{err}=\mathit{typeErr}\vee\mathit{undErr} \quad \mathcal{S}\neq[\,]}{\mathcal{S}[e] \mid \rho \longrightarrow \mathit{err}}\,(\text{CTXERROR})$$

Figure 11: Runtime expressions, evaluation contexts and operational semantics rule for `IL`.

tions for the immutable variables $\overline{x}$ (as in rule (APP) for the formal parameter), instead, for the unbound variables $\overline{y}$ it substitutes the associated locations (via the correspondence of the names in $\overline{Y}$ and $\overline{Z}$). So through assignment to the (local) variables in $\overline{y}$ the execution environment may be modified. The resulting block is wrapped in the `eval` construct. Rule (EVAL) returns its value. (Evaluation inside `eval` is done by the (CTX) rule.) Finally, access to a location may return *undErr* if the location has not been initialized with an assignment of or a definition statement. Rule (CTX) evaluates the first sub-expression selected by the evaluation context. In case the evaluation produces and error rule (CTXERROR) returns the error at the top level. Note that, given a block *bl* if there is $\mathcal{S}$ and *e* such that $bl=\mathcal{S}[e]$, then $\mathcal{S}$ is unique. So evaluation is deterministic.

An `IL` *program* is a closed block, *bl*. The *initial configuration* for a program is $\{bl\} \mid [\,]$.

Let us look at an example of evaluation. Consider the program of Fig. 5. Applying rule (ALLOC) to the block enclosed in brackets we get the configuration $bl \mid \rho$ where *bl* is

```
def lc1 = stm2exp(...);
def lc2 = 7;
def lc3 = fls;
def lc4 = exc(lc1, EV->lc3, lc2);
lc4
```

and $\rho=[\texttt{lc1}\mapsto ?,\texttt{lc2}\mapsto ?,\texttt{lc3}\mapsto ?,\texttt{lc4}\mapsto ?]$.

Applying (DEF) three times we get $bl_1 \mid \rho_1$ where $bl_1\ =\ \texttt{def lc4}\ =\ \texttt{exc}(\texttt{lc1},\texttt{EV}\rightarrow\texttt{lc3},\texttt{lc2});\texttt{lc4}$ and $\rho_1=[\texttt{lc1}\mapsto\texttt{stm2exp}(\ldots),\texttt{lc2}\mapsto 7,\texttt{lc3}\mapsto\texttt{fls},\texttt{lc4}\mapsto ?]$. From rule (CTX) where $\mathcal{S}$ is $\texttt{def lc4}\ =\ \mathcal{E};\texttt{lc4}$ and $\mathcal{E}$ is $\texttt{exc}([\,],\texttt{EV}\rightarrow\texttt{lc3},\texttt{lc2});\texttt{lc4}$, applying rule (LOCDEF) we get $bl_2 \mid \rho_1$ where bl_2 is $\texttt{def lc4}\ =\ \texttt{exc}(\texttt{stm2exp}(\ldots),\texttt{EV}\rightarrow\texttt{lc3},\texttt{lc2});\texttt{lc4}$. From rule (CTX) where $\mathcal{S}_1$ is $\texttt{def lc4}\ =\ \mathcal{E}_1;\texttt{lc4}$ and $\mathcal{E}_1$ is $\texttt{exc}(\texttt{stm2exp}(\ldots),\texttt{EV}\rightarrow\texttt{lc3},[\,]);\texttt{lc4}$, applying rule (LOCDEF) we get $bl_3 \mid \rho_1$ where bl_3 is $\texttt{def lc4}\ =\ \texttt{exc}(\texttt{stm2exp}(\ldots),\texttt{EV}\rightarrow\texttt{lc3},7);\texttt{lc4}$. Again by rule (CTX) where $\mathcal{S}_2$ is $\texttt{def lc4}\ =\ \mathcal{E}_2;\texttt{lc4}$ and $\mathcal{E}_2=[\,]$, and applying rule (STTOEXP), we get $\texttt{def lc4}\ =\ \texttt{eval}(\{bl_4\});\texttt{lc4} \mid \rho_1$, where bl_4 is

```
def fib =
   fun x ->
      if x < 3 then 1
      else (fib (x-1) + fib (x-2));
def temp = fib 7;
lc3 <- temp % 2 = 0;
temp
```

The evaluation proceeds inside the `eval` construct, with rule (CTX) where $\mathcal{S}_3$ is $\texttt{def lc4}\ =\ \mathcal{E}_3;\texttt{lc4}$ and $\mathcal{E}_3$ is $\texttt{eval}([\,])$, applying rule (ALLOC), and producing the configuration $bl_5 \mid \rho_2$ where $\rho_2=[\texttt{lc1}\mapsto\texttt{stm2exp}(\ldots),\texttt{lc2}\mapsto 7,\texttt{lc3}\mapsto\texttt{fls},\texttt{lc4}\mapsto ?,\texttt{lc5}\mapsto ?,\texttt{lc6}\mapsto ?]$, and bl_5 is $\texttt{def lc4}\ =\ \texttt{eval}(\{bl_6\});\texttt{lc4}$ where bl_6 is

```
def lc5 =
   fun x ->
      if x < 3 then 1
      else (lc5 (x-1) + lc5 (x-2));
def lc6 = lc5 7;
lc3 <- lc6 % 2 = 0;
lc6
```

We can see how recursion is handled and how the assignment to `lc3` when evaluated modifies the location of the initial variable `even`.

5 TRANSLATION OF CORE F# INTO IL

In our translation we flatten the `let` constructs transforming them into definitions of the corresponding variables followed by the translation of the expression in their body. Therefore, we have to take into account the fact that in an `IL` block we may have forward binding. E.g., if

```
let y = 3 in
   if ( y = 3) then (
      let f = (fun x -> y)
      let y = 5
      (f 0)  )
   else 4
```

is translated into

```
def y = 3;
   if ( y = 3) then (
      def f = (fun x -> { y });
      def y = 5;
      (f 0)  )
   else 4
```

The translation is incorrect, since in the `IL` code the occurrence of `y` in the body of `f` is bound to the definition of `y` that follows. Therefore the `F#` expression evaluates to 3 whereas its translation in `IL` evaluates to 5. In the translation we use renaming to resolve this problem.

As explained in the Section 2 sequences of expressions will be mapped to sequences of statements, and we use the `stm2exp` and `exc` constructs to simulate the behaviour of the sequence of statements with an expression. So we define two translations of `F#` expressions. The first to `IL` expressions, $[\![\cdot]\!]^{I,M}_{ex}$, and the second to `IL` blocks, $[\![\cdot]\!]^{I,M}_{bl}$. The translations are parametrized by the sets of the immutable variables, I, and mutable variables, M, of the context of the `F#` expression that is translated. The translations produce, in addition to an `IL` expression/block also a sequence of top level variable definition of variables bound to `stm2exp` expressions. In the following we present the translations for function definitions, sequence of expressions, and the `let` construct, which exemplify the technique used.

In the formal definition of the translation δ is a metavariable denoting a declaration of a variable "$\texttt{def}\ x{=}e$" and $\overline{\delta}$ a sequence of declarations separated by ";" (semicolon).

The *translations of* `F#` *function definitions to* `IL` *blocks or expressions*:

$$[\![\texttt{fun}\ x{:}T\,\texttt{->}e]\!]^{I,M}_{bl} \qquad [\![\texttt{fun}\ x{:}T\,\texttt{->}e]\!]^{I,M}_{ex}$$

are both equal to:

$$(\,\texttt{fun}\ x\texttt{->}\{\texttt{def}\ y{=}\texttt{check}(T,x);bl[x:=y]\},\overline{\delta}\,)$$

where $[\![e]\!]^{I\cup\{x\},M}_{bl} = (bl,\overline{\delta})$. So the translation of a function produces a function whose body is the translation of the body (to a block) of the original function. In the translation of the body of the function the variable x is added to the set of free immutable variables I. The formal parameter is replaced with a new variable resulting from the type checking of the original parameter. See the discussion about dynamic type checking in Section 2.

In the following, we introduce the definition of the wrapping needed to extrude a block from its definition environment and how the construct `exc` rebinds it in the run-time environment.

Definition 4. *Given an* `IL` *block, and the disjoint sets of variables* $I = \{\overline{x}\}$ *and* $M = \{\overline{y}\}$, *let* $\mathit{blockToExp}(bl,I,M)$ *be*

$$(\texttt{exc}(z,\overline{Y}\mapsto\overline{y},\overline{x}),\delta)$$

where:

- δ *is* $\texttt{def}\ z{:}T''{=}\texttt{stm2exp}(bl,\overline{y}\mapsto\overline{Y},\overline{x})$
- z *is a new variable and* $\overline{Y}$ *are new names.*

Let $\mathit{blockToExp}(bl,I,M) = (e,\delta)$, we can prove that: for all stores ρ we have: $\{\delta;e\}\mid\rho\longrightarrow^{\star} v\mid\rho'$ if and only if $\{bl\}\mid\rho\longrightarrow^{\star} v\mid\rho''$. So the evaluation of the definition δ followed by the generated expression produces the same result as the evaluation of the original block. The difference in the content of the final stores is due to the fact that the evaluation of the definition δ allocates a location and assigns it the `stm2exp` expression, to subsequently substitute this value for the location in the `exc` expression. However, since the variable z is new it does not interfere with the evaluation of the original block/expression.

To give the translation of both sequences of expressions and of the `let` constructs, we introduce the formal definition of the top level variable definition of `F#` expressions, then we define the renaming needed to avoid the capture of forward definitions described at the beginning of this section.

Definition 5. *1. Let* e *be an* `F#` *expression, the function* $def^{\#}(e)$ *returning* the set of variables defined at the top level of e *is defined as follows:*

- *$def^{\#}$*(`let` [`mutable`] x=e_1 `in` e_2) $= \{x\} \cup def^{\#}(e_2)$,
- *$def^{\#}$*(`let rec` $\overline{x}$:$\overline{T}$=$\overline{v}$ `in` e) $= \{\overline{x}\} \cup def^{\#}(e)$,
- $def^{\#}(e_1,e_2) = def^{\#}(e_1) \cup def^{\#}(e_2)$, *and*
- $def^{\#}(e) = \emptyset$ *for all other expresssions e.*

2. *Let* e *be an* F# *expression, and* $\overline{x}$ *a set of variables,* $rn(e,\overline{x})$, renames the top level definitions of the variables $\overline{x}$ in e *as follows:*
   - *if e is* `let` [`mutable`] x=e_1 `in` e_2, *then* $rn(e,\overline{x})$ *is*
     `let` [`mutable`] x=e_1 `in` $rn(e_2,\overline{x})$ *if* $x \notin \overline{x}$
     `let` [`mutable`] z=e_1 `in` $rn(e_2\{x \mapsto z\},\overline{x})$ *if* $x \in \overline{x}$ *and* z *is new*
   - *if e is* `let rec` $\overline{y}$:$\overline{T}$=$\overline{v}$ `in` e, *then* $rn(e,\overline{x})$ *is*
     `let rec` $\overline{y}$:$\overline{T}$=$\overline{v}$ `in` $rn(e,\overline{x})$ *if* $\overline{y} \cap \overline{x} = \emptyset$
     `let rec` $\overline{z}$:$\overline{T}$=($\overline{v}\{\overline{y} \mapsto \overline{z}\}$) `in` $rn(e\{\overline{y} \mapsto \overline{z}\},\overline{x})$ *if* $\overline{y} \cap \overline{x} = \emptyset$ *and* $\overline{z}$ *are new*
   - *if e is* e_1,e_2 *then* $rn(e,\overline{x})$ *is* $rn(e_1,\overline{x}),rn(e_2,\overline{x})$
   - $rn(e,\overline{x})$ *is e for all other expresssions e.*

The *translations of an* F# *sequence of expressions to a* IL *block* is:

$$[\![e_1,e_2]\!]^{I,M}_{bl} = (bl_1;bl_2,\overline{\delta};\overline{\delta}')$$

where:

- $[\![e_1]\!]^{\Gamma}_{bl} = (bl_1,\overline{\delta})$
- $[\![rn(e_2,\overline{z})]\!]^{\Gamma}_{bl} = (bl_2,\overline{\delta}')$ and $\overline{z} = def^{\#}(e_2) \cap FV(e_1)$.

The translation of the sequence is the sequence of blocks which are the translations of the two expressions to blocks. However, before translating the second expression, e_2, we rename all the variables defined in it that are free in e_1, since in e_1 these variables are bound to their definitions in the enclosing environment. In this way we preserve the semantics of the source language F#.

The *translations of an* F# *sequence of expressions to an* IL *expression* is:

$$[\![e_1,e_2]\!]^{I,M}_{ex} = (e,\delta;\overline{\delta})$$

where:

- $[\![e_1,e_2]\!]^{I,M}_{bl} = (bl,\overline{\delta})$ and
- $blockToExp(bl,I,M) = (e,\delta)$.

That is we first translate the sequence to a block, and then return an `exc` expression, and the definition of a new variable bound to an `stm2exp` expression, see Definition 4. Note that the sets of mutable and immutable variable of the environment are needed to generate the correct matching for the expressions `exc` and `stm2exp`.

The *translation of the let construct to an* IL *block*

$$[\![\texttt{let}\ x{=}e_1\ \texttt{in}\ e_2]\!]^{I,M}_{bl} = (\texttt{def}\ x{=}e_1';bl,\overline{\delta};\overline{\delta}')$$

where

- $[\![e_1]\!]^{I,M}_{ex} = (e_1',\overline{\delta})$ and
- $[\![rn(e_2,\overline{z})]\!]^{I\cup\{x\},M}_{bl} = (bl,\overline{\delta}')$ with $\overline{z} = def^{\#}(e_2) \cap FV(e_1)$.

That is we translate e_1 into an IL expression and the body of the let e_2 into a block. For the translation of e_2 the variable x is added to the immutable variables of the context. Before translating e_2 we rename all the variables defined in e_2 that are free in e_1 (as for the translation of sequences of expressions).
The translation of `let mutable` differs only in the fact that in translattion of e_2, the variable x, being mutable, is added to M.
Note that, this translation produces a block, the definition of x followed by a block. Moreover, the translation of the expression on the right-hand-side of the definition of x, that is e_1, must be an IL expression. Looking at the F# code of Fig. 1 this means that the following F# expression:

```
let rec fib x =
   if x < 3 then 1
   else fib(x - 1) + fib(x - 2)
let temp = fib z
even <- (temp % 2 = 0)
temp
```

which is a sequence of expressions, must be translated to an IL expression.

The *translation of a let expression to an* IL *expression*, is defined as the translation of a sequence of expressions to an IL expression in which $[\![\texttt{let}\ x{=}e_1\ \texttt{in}\ e_2]\!]^{I,M}_{bl}$ substitutes $[\![e_1,e_2]\!]^{I,M}_{bl}$.

Properties of the Translation. The translation preserves the dynamic semantics of the F# expressions, that is let e be an F# program, and $[\![e]\!]^{\emptyset,\emptyset}_{bl} = (bl,\overline{\delta})$. Then $e \mid [\,] \longrightarrow^{\star} v \mid \rho$ if and only if $\{\overline{\delta};bl\} \mid [\,] \longrightarrow^{\star} v \mid \rho'$ for some ρ'. From this result and the fact that F# programs do not get stuck, we can derive that the IL translation of an F# program does not evaluate to an error or gets stuck.

6 COMPARISONS WITH OTHER WORK

Similar projects exist and are based on similar translation techniques, although, as far as we know, we are the first to introduce an intermediate language allowing to translate to many target languages. Pit, see (Fahad, 2012), and FunScript, see (Bray, 2013), are open source F# to JavaScript compilers. They support only translation to JavaScript. FunScript ha support for integration with JavaScript code. Websharper, see (Intellifactory, 2012), is a professional web and mobile development framework. As of version 2.4 an open source license is available. It is a very rich framework offering extensions for ExtJs, jQuery, Google Maps, WebGL and many more. Again it supports only JavaScript. F# Web Tools is an open source tool whose main objective is not the translation to JavaScript, instead, it is trying to solve the difficulties of web programming: “the heterogeneous nature of execution, the discontinuity between client and server parts of execution and the lack of type-checked execution on the client side”, see (Petříček and Syme, 2012). It does so by using meta-programming and monadic syntax. One of it features is translation to JavaScript. Finally, a translation between Ocaml byte code and JavaScript is provided by Ocsigen, and described in (Vouillon and Balat, 2011).

On the theoretical side, a framework integrating statically and dynamically typed (functional) languages is presented in (Matthews and Findler, 2009). Support for dynamic languages is provided with ad hoc constructs in Scala, see (Moors et al., 2012). A construct similar to `stm2exp`, is studied in recent work by one of the authors, see (Ancona et al., 2013), where it is shown how to use it to realize dynamic binding and meta-programming, an issue we are planning to address. The only work to our knowledge that proves the correctness of a translation between a statically typed functional language, with imperative features to a scripting language (namely JavaScript) is (Fournet et al., 2013).

7 CONCLUSIONS AND FUTURE WORK

In this paper we introduced IL an intermediate language for the translation of a significant fragment of F# to scripting languages such as Python and JavaScript. The translation is shown to preserve the dynamic semantics of the original language. A preliminary version of this paper was presented at ICTCS 2012, see (Giannini et al., 2012), which has not published proceedings. We have a prototype implementation of the compiler that can be found at http://www.bluestormproject.org/. The compiler is implemented in F# and is based on two metaprogramming features offered by the .net platform: *quotations* and *reflection*. Our future work will be on the practical side to use the intermediate language to integrate F# code and JavaScript or Python native code. (Some of the features of IL, such as dynamic type checking, were originally introduced for this purpose.) A previous implementation of the translation supported other features such as namespacing, classes, pattern matching, discriminated unions, etc. We are in the process of adding them at the current implementation, since some of this features have poor or no support at all in JavaScript or Python. On the theoretical side, we are planning to complete the proofs of correctness of the translations. We need to formalize our target languages Python and JavaScript, and then prove the correctness of the translation from IL to them. (We anticipate that these proofs will be easier than the one from F# to IL.) Moreover, we want to formalize the integration of native code, and more in general metaprogramming on the line of recent work by the authors, see(Ancona et al., 2013) . We are also considering extending the type system for the intermediate language with polymorphic types, which is, as shown in (Ahmed et al., 2011), non trivial.

ACKNOWLEDGEMENTS

We warmly thank Daniele Mantovani for his support and involvement in the topic of the paper. We also thank the anonymous referees of a previous version of the paper for pointing out some problems which lead to a substantial review of the intermediate language. Any misinterpretation of their suggestions is, of course, our responsibility.

REFERENCES

Ahmed, A., Findler, R. B., Siek, J. G., and Wadler, P. (2011). Blame for all. In *Proceedings of POPL 2011, Austin, TX, USA*, ACM, pages 201–214.

Ancona, D., Giannini, P., and Zucca, E. (2013). Reconciling positional and nominal binding. In *ITRS 2012*, EPTCS.

Appel, A. W. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press.

Bray, Z. (2013). Funscript. http://tomasp.net/files/funscript/tutorial.html.

Fahad, M. S. (2012). Pit - F Sharp to JS compiler. http://pitfw.org/.

Fournet, C., Swamy, N., Chen, J., Dagand, P.-É., Strub, P.-Y., and Livshits, B. (2013). Fully abstract compilation to javascript. In *POPL*, pages 371–384. ACM.

Giannini, P., Mantovani, D., and Shaqiri, A. (2012). Leveraging dynamic typing through static typing. *ICTCS 2012*. http://ictcs.di.unimi.it/papers/paper_4.pdf.

Igarashi, A., Pierce, B., and Wadler, P. (2001). Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450.

Intellifactory (2012). Websharper 2010 platform. http://websharper.com/.

Matthews, J. and Findler, R. B. (2009). Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3).

Moors, A., Rompf, T., Haller, P., and Odersky, M. (2012). Scala-virtualized. In Kiselyov, O. and Thompson, S., editors, *Proceedings of PEPM 2012, Philadelphia, Pennsylvania, USA*, ACM, pages 117–120.

Nanevski, A. (2003). From dynamic binding to state via modal possibility. In *PPDP'03*, pages 207–218. ACM.

Petříček, T. and Syme, D. (2012). AFAX: Rich client/server web applications in F#. http://www.scribd.com/doc/54421045/Web-Apps-in-F-Sharp.

Ranson, J. F., Hamilton, H. J., and Fong, P. W. L. (2008). A semantics of python in isabelle/hol. Technical Report CS-2008-04, CS Department, University of Regina,Saskatchewan.

Vouillon, J. and Balat, V. (2011). From bytecode to javascript: the js of ocaml compiler. http://www.pps.univ-paris-diderot.fr/∼balat/publi.php.