Leveraging dynamic typing through static typing

Paola Giannini*[2] and Daniele Mantovani[1] and Albert Shaqiri[1 2]

[1] AlgorithMedia, Alessandria, Italy
[2] CSI, DISIT, Univ. del Piemonte Orientale, Italy

Introduction Implementing more than a trivial 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 rarely used for more than just 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. 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. 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.
The paper is organized as follows. We first introduce the syntax of the core of the intermediate language. Then, we present the translation from `F#` to this intermediate language, and from the intermediate language to both JavaScript and Python. We do this via some examples that highlight the features of the intermediate language and the differences between the two target languages. Then, we briefly discuss correctness, and implementation. Finally, we compare our approach with related work, and discuss plans for future work.

Intermediate language The intermediate language is higher level than most intermediate languages. The syntax of the language is presented in Fig.1. There are three syntactic categories: *expressions*, e, *statements*, st, and *sequence statements*, s, which are sequences of statements returning a value. A *program* is a sequence statement. Although in `F#` everything is an expression, we introduce a 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 [3] for a similar choice. The construct of the language which is most useful in the translation is `stm2exp`, which is a sequence statement, s, whose free mutable variables are a subset of $\{u_1, \ldots, u_n\}$; `stm2exp` is a value (like the lambda abstraction), and therefore, may be passed around. Its type, $\langle u_1{:}t_1, \ldots, u_n{:}t_n\rangle t$ says that in an environment in which the mutable variables u_i have type t_i $(1 \leq i \leq n)$, then s has type t. The construct `exc` e evaluates the expression e, which is supposed to be a `stm2exp` in the current store, dynamically binding its free mutable variables in the execution environment. The use of the construct will be explained when presenting the translation.

s	$::=$ `return` $e \mid st; s$	sequence statements
st	$::=$ $u{:=}e \mid$ `let` $x{:}t{=}e \mid$ `let!` $u{:}t{=}e \mid$ `return` $e \mid$ `if` e `then` s_1 `else` s_2	statements
e	$::= x \mid n \mid$ `tr` $\mid$ `fls` $\mid e_1$`+`$e_2 \mid$ `fun` $x{:}t$`->`$s \mid e_1\ e_2 \mid$ `stm2exp`$(s, \{u_1{:}t_1, \ldots, u_n{:}t_n\}) \mid$ `(int)`$e \mid$ `(bool)`$e \mid$ `exc` e	expressions
t	$::=$ `int` $\mid$ `bool` $\mid t_1 \to t_2 \mid \langle u_1{:}t_1, \ldots, u_n{:}t_n\rangle t$	types
v	$::= n \mid$ `tr` $\mid$ `fls` $\mid$ `fun` $x{:}t$`->`$s \mid$ `stm2exp`$(s, \{u_1{:}t_1, \ldots, u_n{:}t_n\})$	values

Fig. 1. Syntax of core intermediate language

* The work of this author was partly funded by the project MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

Translation by examples 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. On the left of Fig. 2 is the F# code. As we can see, on the right-hand-side side of `let x=` we have a sequence of expressions: the

```
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 7
    even <- (temp % 2 = 0)
    temp
x
```

```
let y = stm2exp(
    let fib = fun x:int ->
          if x < 3 then return 1
          else return (fib (x-1) + fib (x-2));
    let temp = fib 7;
    even := temp % 2 = 0;
    return temp;,
  {even:bool});
let! even = false;
let x = exc y
return x;
```

Fig. 2. Translation of F# sequence of expressions in the intermediate language

definition of the function `fib` followed by the definition of `temp`, etc. This sequence is, in F#, an expression. On the right side of Fig. 2 is the translation into the intermediate code. The sequence of statements is translated in a `stm2exp` expression whose first component is the sequence of statements, and the second the set of free mutable variables occurring in such statements with their type: in this case the variable `even` of type `bool`, and bound to the variable `y`. The variable `x` is then bound to the `exc` expression applied to `y` (to obtain the result that we would have by evaluating the sequence of statements in the current environment). Assume that, the F# code was mapped to JavaScript literally, we would obtain the program on the left side of Fig. 3. This program is syntactically wrong, since on the right-hand-side of an assignment we must have

```
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(7);
    even = (temp % 2) == 0;
    temp;
return x;
```

```
(function() {
  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(7);
      even = (temp % 2) == 0;
      return temp;
  })();
  return x;
})();
```

Fig. 3. Wrong and Correct JavaScript translations

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 closures and application. Also, the whole program is wrapped into an entry point function. In this way, the code on the right side of Fig. 3 is correct.
Unfortunately, the same cannot be done in Python as its support for 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

```
def temp1(even):
    def temp2(even, fib, x):
        if (x < 3):
            return 1
        else:
            return fib(x - 1) + fib(x - 2)

    fib = lambda x: temp2(even, fib, x)
    temp = fib(7)
    even.value = ((temp % 2) == 0)
    return temp
```

```
def __main__():
  even = false;
  wrapper1 = ByRef(even)
  x = temp1(wrapper1)
  even = wrapper1.value
  return x

__main__();
```

Fig. 4. Translation in Python

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`. Going back to our intermediate language, we use the construct `stm2exp` to provide the information needed to produce both translations, recording the information on the free mutable variables, needed for any language not supporting closures. We do not record free immutable variables, as they can be substituted with their values.

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, see left side of Fig. 5, 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 2 floating points obtaining a floating point. To prevent this, the translation in the

```
let add x y = x + y

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

```
let add = fun x:int ->
    return fun y:int ->
        return (int)x+(int)y;
```

Fig. 5. `F#` code and the corresponding intermediate representation with type casting

intermediate language, see right side of Fig. 5, insert type casting on the occurrences of function parameters. This is translated into dynamic type checking in JavaScript and Python as is shown in Fig. 6, where the function `toInt` tries to convert the argument we pass it to an integer, and if it fails, raises an exception. Our

```
var add = function (x) {
    return function(y) {
      return toInt(x) + toInt(y);
    }
}
```

```
def temp1(x, y):
    return (int(x) + int(y))

def add(x):
    return lambda y: temp1(x, y)
```

Fig. 6. JavaScript and Python version with type casting

intermediate language supports other features such as namespacing, classes, pattern matching, discriminated unions, etc. Some of this features have poor or no support at all in JavaScript or Python although semantically equivalent behaviour can be achieved through other language constructs.

Full abstraction of the translations We have defined an operational semantics for the intermediate language, `IL`, and a type system enforcing the property that well typed programs evaluated in a store that

agrees with their definition environment do not get stuck. Moreover, before the conference we plan to prove the full abstraction of the translations. That is, (1) formalize a fragment of F#, FS^c, a core Javascript, JS^c, and a core Python, PY^c, as we have done for the intermediate language; (2) define the translations from FS^c to IL, and from IL to JS^c and PY^c; (3) prove that the translations preserve the operational semantics (and for the one from FS^c to IL also the typing) of the relative languages.

Implementation The compiler is implemented in F# and is based on two metaprogramming features offered by the .net platform: *quotations* and *reflection*. These mechanisms allow one to extract code and type information during runtime, reason about it and, in our case, are used to build an intermediate language abstract syntax tree from which the target code is generated.

Comparisons and future 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 [4], is an open source F# to JavaScript compiler. It supports many F# features (at the time of this writing it is at version 0.2) and is very well documented. It supports only translation to JavaScript. Websharper, see [5], 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 [8]. 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 [9].
On the theoretical side, a framework integrating of statically and dynamically typed (functional) languages is presented in [6]. In [10] a cast construct wrapping dynamic code is introduced, and it is showed how it can be used to prove the source of run time type errors. Support for dynamic languages is provided with ad hoc constructs in Scala, see [7]. Finally, a construct similar to stm2exp, is studied in [2], where it is shown how to use it to realize dynamic binding and meta-programming, an issue we are planning to address.
Our future work will be on the practical side to use the intermediate language to integrate F# code and JavaScript or Python native code. On the theoretical side, we plan to finish the proof of full abstraction of the translation from F# to the intermediate language, and from this to the target languages. Moreover, we would like to explore meta-programming on the line of [2]. We also plan to explore the extension to polymorphic types of the type system for the intermediate language, which is, as shown in [1] non trivial.

References

1. Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA*, ACM, pages 201–214, 2011.
2. Davide Ancona and Paola Giannini Elena Zucca. Reconciling positional and nominal binding. In *ITRS 2012 (accepted for presentation)*, 2012.
3. Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
4. Mohamed Suhaib Fahad. Pit - F Sharp to JS compiler. http://pitfw.org/, May 2012.
5. Intellifactory. Websharper 2010 platform. http://websharper.com/, May 2012.
6. Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. *ACM Trans. Program. Lang. Syst.*, 31(3), 2009.
7. Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In Oleg Kiselyov and Simon Thompson, editors, *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA*, ACM, pages 117–120, 2012.
8. Tomáš Petříček and Don Syme. AFAX: Rich client/server web applications in F#. http://www.scribd.com/doc/54421045/Web-Apps-in-F-Sharp, May 2012.
9. Jerome Vouillon and Vincent Balat. From bytecode to javascript: the js of ocaml compiler. http://www.pps.univ-paris-diderot.fr/~balat/publi.php.
10. Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *ESOP 2009*, volume 5502 of *LNCS*, pages 1–16, 2009.