A Unified Library of Primitives for QML-Compilers and Interpreters

The Bypass Standard Library

Mathieu Barbin Mehdi Bouaziz

École Normale Supérieure, Paris
mathieu.barbin@ens.fr/mehdi.bouaziz@ens.fr

Abstract

We present in this document a new approach to the management of primitives of the QML language, in all kind of applications, including compilers, interpreters, as well as static analysers, ... QML being the internal translation of the main language OPA designed by MLstate to build commercial web-applications, QML must support several kinds of compilations, targetting the client-side language JavaScript, and server-side languages OCaml, and more recently LLVM. By *primitives*, we denote here the part of the implementation used as initial library for QML (and so, OPA) applications, including functions to handle web protocols and markup languages, mathematics, low level database management, etc. This can be seen as the core part of QML which has to be implemented in each target language, unlike the rest of the language constructions which are automatically and efficiently compiled.

We expose in this document the main ideas leading to a standardized and shared interface between the implementation of these primitives, and the MLstate applications, offering a possible yet static typing verification of the initial code, dynamic introspection features, and a shared implementation of the primitives between interpreters or compilers manipulating codes with different runtime algebras, thanks to regenerated specializations of the initial library, using a type projection system.

General Terms Languages, Software Engineering, Design

Keywords Primitives, Bypass, Compilers, Type Checking, Type Projection

MLstate'09 March 1-September 3, 2009, Edinburgh, Scotland, UK.

1. Introduction

We assume in this document that the reader is familiar with the QML language [Binsztok, 2007]. A general overview of the language types and expressions algebras can be found in appendix.

One of the particularities of QML on which we will focus here is that the language has no initial environment, which means that no type or primitive is defined at starting point of QML programs. However, it would not be possible to write complex applications without any primitive. That's why the language provides a special primitive construction system named "**bypass**" (fig. 1). Historically, QML used to support only a compilation to OCaml. The text present between the delimiter characters "%%" was read as a literal OCaml code portion, and directly inserted into the generated OCaml code without any transformation. This construction was a

```
val print_endline =
    %% print_endline %% : string →unit
```

Figure 1: Syntax example: a bypass definition.

way to "bypass" the processing done by the QML-to-OCaml compiler on the rest of the language constructions, using the OCaml language as the primitive language, where the rest of the QML code was translated into OCaml by the compiler. This design decision used to bring some advantages:

- avoid the *ad hoc* definition of an initial environment
- use all OCaml primitives in QML for free

Unfortunatly, this system started to show some gaps with the introduction of new goals for QML:

1. Compilation to LLVM
2. Compilation to OCaml using a code generator using custom runtime representations for values
3. Simplified compilation to JavaScript

4. Several code analysers needing the understanding of the semantics of these bypasses, or needing to know what primitives were available on what side (e.g. the slicer)
5. A toy-interpreter for QML

A first look at goals 1 and 2 may lead to a solution based on code-duplication, replacing all written OCaml-bypasses by a code written in the target language of a specific compiler. Actually, a look at goals 3 and 4 reveals the essence of the problem: the same QML code must support several compilations with different target languages. The slicer [Bouaziz, 2009] decides which part of the code must be compiled to the client-side language (that is JavaScript) and which part to the current language used as the server-side target language (LLVM or OCaml). Finally for goal 5, writing an interpreter for a language including OCaml code portions would mean to write at least an interpreter for OCaml, moving the problem of primitives declaration to an other part of the code.

For further developments (such as [Barbin, 2009] and [Bouaziz, 2009]), QML needed a more generic way to replace the bypass system described above. We planned then to design and implement a new one: this document presents the solution we have proposed to MLstate, which is today fully integrated in the framework.

2. The Benefits of a Lookup System

2.1 Original Idea

The main idea is to use keys instead of a row code between the delimiter "%%". This change involves changes in applications inspecting QML codes. When inspecting the bypass nodes of the QML abstract syntax tree (AST), a lookup in a dictionary (the bypass map) at the given key will be now necessary to find the needed information. Although simple, this idea affects a huge part of the source code: at the beginning, this solution looked dangerous and regressive, because of the numerous updates needed in the framework, and because with it, we lost the property to have injection of OCaml code into QML for free, without any additional work. About this last point:

1. This was possibly used only in the context of a particular OCaml code generator
2. However, we planned to provide a simple mechanism to still be able to use the OCaml library without too much work (reduced to the registration procedure, see Section 3).

To assure a temporary compatibility with the former system during a short transitional period, the choice of keys for primitives has been based on OCaml function names, which meant that, waiting for updates, the code generator could still use the keys as it was the pointed data itself.

```
val + = %%+%% : int →int →int
(* In the Bypass Map :
  (key "+" ) →{ ocaml : "+" ; ... } *)
```

Figure 2: Using a bypass map.

2.2 Type Checking

As shown in Figures 1 and 2, bypass nodes are coerced with a type annotation. Initially, this coercion was strictly needed by the QML typer, because there is no simple way to infer the type of a row code between the delimiters "%%". (would need an OCaml, a C, and a JavaScript type checker). So processing this way and trusting user type annotation does not let easy possibility to detect potential errors soon enough (at compile time of the QML code) and could result in unwanted behaviour (usually runtime error, *segmentation fault* or security issues in the worst case). Even if type coercions are carefully written, this is not satisfying and a real type checking *is needed*.

```
val +. = %%+%% : float →float →float
val x = 2. +. 1.14
```

Figure 3: Bypass type error

The Figure 3 illustrates a wrong definition of the QML function +. (the addition operator on **float**), using a bypass binding. The type given in coercion is coherent with the name defined for this function in QML. The question we had in mind was : "At what step of the program's life (even runtime) does it fail ? and how (what kind of error exactly) ?". With the former raw code solution, since the coercion in Figure 3 give the type **float** →**float** →**float**, and as long as the QML value +. is used as the addition operator on floats, no QML type error is detected. The raw code + would be injected in the generated OCaml code, and then, if we are "lucky enough" [1] a static typing error will be detected by the OCaml typer, at compile time of the generated server code, which is very late, and makes the error report process hard (because of other preprocesses performed on the QML code, like alpha-conversion, rewritings, ...).

Thanks to our key system, there is no limitation on the number of attributes attached to each primitive. So we decided to add a field containing the type of the bypass in the map. In a case of a type-annoted bypass definition, we can then compare the type provided in the coercion with the type of the primitive found in the map from the key.

Thus we now do not trust the user, but we still trust the creator of the library. Actually, this represents only half of the work of bypass types verification, because nothing guarantees *a priori* that the registered type in the map is the

[1] The generated OCaml code contains a lot of Obj.magic needed to be more flexible with a low level management of runtime values.

```
##format bind−module "#m␣=␣#m"
##format bind "#n␣=␣(#n)" "␣;␣"
##format in−module "let␣#n␣=␣%%#k%%␣:␣#t␣in"
##format sub−module "let␣#m␣=␣\n#{#rec,␣in−module}\n␣{␣#{bind−module␣bind}␣}␣in"
##format module "val␣#m␣=␣\n#{␣sub−module,␣in−module␣}\n␣␣{␣#{bind−module␣bind}␣}"
```

Figure 4: Recursive Formats.

real type of the finally called primitive. This verification is performed during the map building process with the register mechanism (see next section below).

2.3 Applications Using the Bypass-Map

We can sum up the new actions done by the applications working at some point with primitives:

1. Find a bypass in the AST,
2. Do a lookup in the bypass-map,
3. Use the part of the information provided corresponding to the nature of the application:
   - QML typer: type of the primitive;
   - OCaml, JavaScript, LLVM compilers: specializes raw code of the primitive in the target language;
   - Slicer: client-server repartition of the available implementations;
   - Interpreter: a dynamic function pointer.

2.4 Production of the Binding Code

Another idea developed in the library came from the observation that most of the bypass definitions are just simple bindings between new QML values and bypass-keys (Figure 5). With a map, this kind of code can easily be automatically

```
val + = %%+%% : int →int →int
val − = %%−%% : int →int →int
val ∗ = %%∗%% : int →int →int
...
```

Figure 5: Some Bypass definitions.

produced (a simple iteration on tuples (key, val)), and bring safety with the guarantee that the coercion provided really reflects the type present in the map. Moreover it helps the maintenance in case of changes by avoiding manual updates.

We added a preprocess directives system to produce this kind of code (Figure 5), without using a particular and fixed syntax, using custom format definitions. We completed it also with a support for hierarchical bypass definitions (modules), with iterations on module contents, and recursive formats (Figure 4).

```
##format function "val␣#n␣=␣%%#k%%␣:␣#t"
(∗ all bypasses from mod. Pervasives ∗)
##include function Pervasives
```

Figure 6: Bypass definition preprocessor.

We have seen in this part the point of view of the applications using directly the bypass map, i.e. the introspection of the bypass library.

3. The Registering Process

We see here how the bypass map is built.

3.1 The Directive ##register

There was a few possible ways to do it. Whereas the implementation files are written in different languages (OCaml, C, JavaScript), we wanted to write a unified parser for them. We decided to use preprocessor directives as code annotations, to register the primitives. During parsing the different files given, we also parse the type and check that there is no duplication of primitives and no clash between the same function in different languages.

```
(∗ in the ML implementation of the library ∗)
##register add_int : int →int →int
let add_int = ( + )

/∗ in the C implementation of the library ∗/
##register add_int : int →int →int
int add_int (int a, int b) { return(a+b); }

/∗ in the JS implementation of the library ∗/
##register add_int : int →int →int
function add_int(a, b) { return a+b };

# example : run application bslbrowser
# as a sample of an application
# with introspection features
bslbrowser:bypervasives/$ ls | grep add_int
   add_int : int →int →int {c, js, ml}
```

Figure 7: The registering process.

The rest of the code is not parsed, but collected in a file where the directives have been removed. This file can

then be checked and compiled using external applications (ocamlopt, gcc, JavaScript validators, ...).

3.2 Type Checking in OCaml and JavaScript

4. Implementation

The implementation of our library, called libBSL, and all the needed command line applications used to parse and register implementation files, build bypass libraries and plugins, using meta-programming technics, ... represents a total amount of about 4800 loc. The library is today included in the MLstate framework, and used by the QML interpreter, all the compilers (client and server side), and the new version of the slicer.

A bunch of tests of all the main parts of the library are available to assure the non regression of the code itself, but also its interface with the rest of the code (especially the type inference and code generators).

The implementation contains some documented mli files. We have also provided an internal manual for developers. It contains in particular more details about technical issues, which do not appear in this document.

5. Conclusion and Future Work

In this document we have reported the design and the implementation of a library used by QML compilers and interpreters, to get dynamically introspection information about primitives, based on a simple registration mechanism combined with a meta-programming system. This new library is now included in the MLstate's framework, and brings in particular the properties that the primitives declaration types are statically checked at compile-time of the initial library, and the implementations can be shared between interpreters and compilers manipulating different runtime algebras. This last feature has been really improved, and is one of the key of the semantic agreement between the QML interpreter and compilers available in the framework [Barbin, 2009].

As a future work we would like to further explore the applicability of the types projection system used with OCaml primitives, for JavaScript and LLVM®. We would like to think about a way to extend the type verification done with OCaml and C at compile-time to the initial library for JavaScript. We would also like to introduce a strong mechanism to refer to serialization/unserialization functions of extern types, to help the slicer to distinguish between the abstract values which can be transfered on the network unlike the side-specific values (e.g. database path for server, or DOM objects for client).

Acknowledgments

We want to thank Henri Binsztok for his intership proposal and most generally the QML team as well as the OPA team for the quality of the work done at MLstate. We have really enjoyed the discussions around all the subjects aborded by the libbsl, especially the contribution of Louis Gesbert, Rudy Sicard, and Nicolas Pelletier for both their help in the architecture and the design of the lib, and the first feedbacks by testing and including it in the framework. Thanks to Mikolaj Konarski and Geoffroy Chollon for the introduction of the library into the LLVM compiler and the optimized OCaml code generator.

References

[Barbin, 2009] Mathieu Barbin. Semantics & Reference Implementations for a Functional Language with Overloads, Extensible Records and Parallel Evaluation. Technical report, MLstate.

[Binsztok, 2007] Henri Binsztok. QML, a Persistent Functional Language. Technical report, MLstate, France. Internal document.

[Bouaziz, 2009] Mehdi Bouaziz. Optimized Client-Server Distribution of Ajax Web Applications. Technical report, MLstate.

[Leroy et al., 2008] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, documentation and user's manual — release 3.11*. INRIA. http://caml.inria.fr/ocaml/.

[Rémy, 1992] Didier Rémy. Efficient Representation of Extensible Records. In *Proceedings of the 1992 workshop on ML and its Applications*, page 12, San Francisco, USA.

A. The QML Language

A.1 General Overview

QML is a purely functional language, with a syntax not far from ML (e.g. OCaml [Leroy et al., 2008]), including let-binding and pattern matching. One of the particularities of the language is that it contains its own database management, using first class language constructions. This is the only imperative part of the language : data access (read / write), named simply **database** in the rest of this appendix.

A.2 Data-Types

A.2.1 Basic Types

These types are types of constants.

- Void **unit**
- Int **int**
- Float numbers **float**
- Char **char**
- String **string**

Notice that the type boolean is not a primitive type, as we will see in the next part.

A.2.2 Records Types

The records types used in QML are extensible records *à la* Rémy [Rémy, 1992].

A.2.3 Sum Types

Sum types are built from different cases of records types. There is no explicit constructors in QML as we find in ocaml.

typedef	::=	| *typeconst*	(basic type)	*typeconst*	::=	| **unit**
		| *typeident*	(named type)			| **int**
		| { *fieldident* : *typedef*				| **float**
		[; *fieldident* : *typedef* [...]] }	(record)			| **char**
		| *typedef* / *typedef* [/ *typedef* [...]]	(type sum)			| **string**

Figure 8: QML type definitions.

expr	::=	| *const*	(constant value)
		| *ident*	(identifier)
		| (*expr* : *typeident*)	(coercion))
		| **fun** *ident* →*expr*	(lambda)
		| *expr expr*	(apply)
		| **let** *validentifier* = *expr* **in** *expr*	
		| **let rec** *ident* = *expr*	
		[**and** *ident* = *expr* [...]]	
		in *expr*	
		| { *fieldident* = *expr* } / *expr*	(record extending)
		| *expr*.*fieldident*	(field of a record)
		| **match** *expr* **with**	
		| *pattern* →*expr*	(pattern matching)
		[| *pattern* →*expr* [...]]	
		| dbpath	(database path, read)
		| dbpath = *expr*	(database write)
		| %%$bslkey$%%	(function of the library)
		| **assert** *expr* **in** *expr*	(assertion)

const	::=	| unit, a void expression	(type **unit**)
		| an integer	(type **int**)
		| a floating point number	(type **float**)
		| a character	(type **char**)
		| a string of characters	(type **string**)
		| the empty record	(any type record)

typeident ::= a type identifier

pattern	::=	| *const*	(constant value)
		| *ident*	(binding)
		| _	(any value)
		| { *ident* = *pattern* } / *pattern*	(field of a record)

dbpath ::= /*dbident*/$\mathbf{val}_0$/$\mathbf{val}_1$/.../$\mathbf{val}_n$

where $\mathbf{val}_i$	depends on	the type of /*dbident*/$\mathbf{val}_0$/.../$\mathbf{val}_{i-1}$
an expression of type string	in case of a	stringmap
an expression of type int	in case of an	intmap
the name of a field	in case of a	type record

Figure 9: QML expressions.

The language supports parametric, and recursive definitions of new types.

```
type bool = { false } / { true }
type 'a list = { nil } / { hd : 'a ; tl : 'a list }
```

Deconstruction of sum types is done as usual with a **match**:

```
val list_length =
  let rec aux acc x =
    match (x : list) with
    | { nil } →acc
    | { hd ; tl } →aux (succ acc) tl
  in aux 0
```

Conditional statements are possible, using a syntactic sugar: **if** condition **then** $expr_1$ **else** $expr_2$ is rewritten in:

```
match (condition : bool) with
| { true } →expr1
| _ →expr2
```

A.2.4 Usual ML Types

As a language of the ML family, QML contains arrow types, type variables, and named types.

A.2.5 Overloaded Types

In the newest version of QML, operators can be overloaded. The type overload is used to describe that an operator has more than a single type.

```
> val ++. = ( + ) over ( +. )
qmlval ++. : (int →int →int) & (float →float →float)
```

A.3 Expression Algebra

A QML code is composed of:

- Type definitions (Figure 8):

  type *typeident* = *typedef*

- Database definitions:

  val *dbident* : typedecl

- Values (Figure 9):

  val *ident* = *expr*

Operators (like $+$, $-$, $*$, $/$, **not**, &&, ||, ^, ...) are just sugar for call to primitive functions of the library.

B. The MLstate Company

MLstate is a young innovative company that aims to make functional languages more popular. Grown in almost 18 months from 1 to 20 employees including 7 PhDs and 6 Research Engineers, MLstate benefits from the support of Groupe Caisse d'Epargne.

MLstate currently prepares the launch of its revolutionary technology, OPA® (One Pot Application), which is a new programming language for online applications.

OPA, which offers at the same time a new language, a new database and new servers, is a breakthrough in developing Web 2.0 applications. The expected benefit of the single specification exceeds productivity gains by opening a field of research on safety, security and automated analysis of Web applications.

For this, MLstate has a strong will of academic collaborations, although most of its research has been done internally. The preparation of a European project including the University of Edinburgh triggered the opening of a second research office in Edinburgh (after Paris) in the field of languages and functional databases.

OPA has received numerous awards and won the contest of French Ministry of Research 2008. Finalist of Paris Innovation Grand Prix in 2007, OPA has the potential to bring state-of-the-art research to industry and was cited in the 'Best Of' Atelier BNP Paribas in 2008.

MLstate has signed a long term partnership with Epitech, ensuring thereby a presence on the labor market and teaching the programming language to initiate a community of passionate programmers.

MLstate is a member of the competitiveness cluster System@tic, and targets software companies and publishers. In addition, MLstate uses OPA internally to build software such as ERP for SMEs and social networks.