MPRI course 2-4
"Functional programming and type systems"
Programming project

Xavier Leroy

December 21, 2015

1 Summary

The purpose of this programming project is to implement a type-checker and a compiler (down to a simple abstract machine) for a small functional language nicknamed SUB, featuring numbers (integers and floating-point), records with named fields, and subtyping. Similar techniques are used in the OCaml and SML/NJ compilers to implement the ML module language (with structures as records and functors as functions).

Subtying is an extension of simple types whereas an expression of type τ can be used in any context expecting a type τ' that is "bigger" than τ. In more technical terms, we say that τ' must be a *supertype* of τ, or equivalently that τ must be a *subtype* of τ'. For example, we treat the type `int` of integers as a subtype of the type `float` of floating-point numbers. Likewise, a record type is subtype of record types having fewer fields.

For background information on subtyping, you can refer to part III of Pierce's textbook *Types and Programming Languages*.

To compile SUB down to the Modern SECD abstract machine, we use *type-directed data representations*: the machine representation of the value of an expression depends on the type with which this expression is viewed. For example, the number 2 is represented as an integer if viewed with type `int` and as a FP number if viewed with type `float`. Likewise, the record $\{x = 1; y = 2\}$ is represented by the pair $(1,2)$ if viewed with type $\{x : \texttt{int}; y : \texttt{int}\}$, but as the pair $(2,1)$ if viewed with type $\{y : \texttt{int}; x : \texttt{int}\}$. To perform this compilation, we use an intermediate language nicknamed IL that features *coercion terms* materializing the changes of data representations required to implement *subsumption steps*, i.e. the points in the execution where an expression is viewed with a supertype of its type.

The project can be implemented in any language of your choice, but we strongly recommend using Caml, as the sources we provide are written in Caml.

2 Required software

To use the sources we provide, you will need:

OCaml Any version ≥ 4.00 should do. You can use the packages available in your Linux distribution, or the OPAM package manager `https://opam.ocaml.org`. or the source distribution from `http://caml.inria.fr/ocaml/release.html`.

Linux, MacOS X, or Windows with the Cygwin environment The sources that we distribute were developed and tested under Linux. They should work under other Unix-like environments.

3 Overview of the provided sources

As a starting point, we provide a number of OCaml source files in the `src/` directory. To build them, you can change to the `src/` directory and type `ocamlbuild Main.byte` (for compilation to bytecode) or `ocamlbuild Main.native` (for compilation to native code).

You will need to study carefully the following interfaces:

AST.mli Abstract syntax for the source language SUB, plus some useful functions.

IL.mli Abstract syntax for the intermediate language, plus a pretty-printer for IL terms.

Mach.mli The instruction set for the Modern SECD, plus an interpreter to execute Modern SECD programs.

The following parts of the implementation are provided and should not need to be modified, unless you add "extra credit" features:

Lexer.mll Lexical analyzer for SUB.

Parser.mly Parser for SUB. The concrete syntax of SUB is in appendix. See examples in the `tests/` directory.

Main.ml The driver for the whole program. After compilation by `ocamlbuild Main.byte`, you can run the program as follows:

`./Main.byte`

- *filename* to typecheck, compile and execute the SUB expressions contained in file *filename*. Each expression is terminated by `;;` (double semi-colon).

`./Main.byte`

- without arguments provides a toplevel interactive loop similar to OCaml's. Just enter expressions terminated by `;;`.

The following templates are to be completed by you:

Typing.ml The type-checker (task 1).

Elab.ml Type-directed elaboration to the intermediate language (task 2).

Expand.ml Expansion of coercion terms (task 3).

Compile.ml Compilation to abstract machine code (task 4).

Optimize.ml Optimizations of redundant coercions (task 5).

In the `tests/` subdirectory you'll find several files containing SUB expressions. Some are well typed and should execute without errors. Others are ill typed and should be rejected.

4 Programming tasks

Task 1: type-checking

Implement a type checker for the language SUB. Given a closed SUB expression, the type checker should compute its most precise type if it is well typed, and raise the `Type_error` exception with an explanatory message if it is ill typed. The notion of "most precise" is with respect to subtyping and is explained below. The file to modify is `Typing.ml`.

The typing rules for SUB are those of the simply-typed λ-calculus extended with records and subtyping. For records, we use types of the form $\{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\}$ denoting records having n fields labeled $\ell_1 \ldots \ell_n$ and associating values of type τ_i to label ℓ_i. The typing rules for record construction and for access to a labeled field are:

$$\frac{\Gamma \vdash e_i : \tau_i \text{ for } i = 1, \ldots, n \qquad l_i \neq l_j \text{ if } i \neq j}{\Gamma \vdash \{\ell_1 = e_1; \ldots; \ell_n = e_n\} : \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\}} \text{ (RECORD)}$$

$$\frac{\Gamma \vdash e : \{\ell_1 : \tau_1; \ldots; \ell_n : \tau_n\} \qquad \ell = \ell_i}{\Gamma \vdash e.\ell : \tau_i} \text{ (PROJ)}$$

For subtyping, we add the subsumption rule that allows us to consider an expression with any supertype of its type:

$$\frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'} \text{ (SUB)}$$

The subtyping relation $<:$ is defined by the following rules:

$$\tau <: \tau \text{ (SUBREFL)} \qquad \tau <: \top \text{ (SUBTOP)} \qquad \texttt{int} <: \texttt{float} \text{ (SUBNUM)}$$

$$\frac{\tau_1 <: \sigma_1 \qquad \sigma_2 <: \tau_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2} \text{ (SUBFUN)} \qquad \frac{\forall i \in \{1, \ldots, n\}, \exists j \in \{1, \ldots, m\}, \;\; l_i = l'_j \wedge \sigma_j <: \tau_i}{\{\ell'_1 : \sigma_1 \ldots \ell'_m : \sigma_m\} <: \{l_1 : \tau_1 \ldots \ell_n : \tau_n\}} \text{ (SUBREC)}$$

Note that rule SUBREC supports both *width subtyping* (the subtype has more labels than the supertype) and *depth subtyping* (the types of a label ℓ present in both record types are in the subtyping relation).

In this type system, an expression can have several types: not just τ, but also all supertypes of τ, owing to the subsumption rule SUB. However, it is the case that every well-typed expression has a type that is minimal with respect to the $<:$ subtype ordering. (See Pierce's textbook for a proof.) The goal of your type-checker is to compute such a minimal type if it exists.

Task 2: type-directed compilation to the intermediate language

Implement a translation from SUB expressions to expressions of the intermediate language IL. The file to modify is `Elab.ml`.

The intermediate language is described in file `IL.mli`: it is an untyped λ-calculus with tuples (instead of records with named fields), these tuples being accessed by position ("extract the n-th field") instead of by label names. Hence, your translation must transform record accesses (by label names) into tuple projections (by position), using the type of the record being accessed to determine the position. For example, $e.x$

becomes an access to the second component of a pair if e is viewed with type $\{y : \texttt{int}; x\texttt{int}\}$. If e is viewed with type $\{x : \texttt{int}; y : \texttt{int}\}$, an access to the first component of a pair should be generated instead.

Moreover, IL features a construct $\texttt{Lcoerce}(a, c)$ representing the application of a *coercion* c to an IL term a. The algebra of coercions is shown in file `IL.mli`. Coercions materialize the changes in data representation that need to be performed when an expression is viewed with a supertype τ' of its original type τ. For example:

- When an expression of type `int` is viewed with type `float`, the coercion is `Cint2float`, materializing the conversion of an integer to a FP number.
- When an expression of type τ is viewed with type $\top$, no change of representation is needed and the coercion is simply `Cid` (identity coercion).
- When a record of type $\{x : \texttt{int}; y : \texttt{int}\}$ is viewed with supertype $\{y : \texttt{float}\}$, the coercion is $\texttt{Crecord}[(2, \texttt{Cint2float})]$, meaning "build a 1-tuple by taking the second field of the original record and applying it the `Cint2float` coercion".

As suggested by the description above, the translation depends strongly on the types assigned to the various subexpressions. The easiest way to keep these types straight during translation is to recompute them on the fly, like the type checker of Task 1 does. Hence, the translation to be written here is an extension of the type checking function of Task 1, returning a pair (minimal type, IL term) instead of just the minimal type.

Task 3: expansion of coercions

Implement an IL-to-IL translation that removes `Lcoerce` coercion applications, replacing them by actual computations of the IL language. The file to modify is `Expand.ml`.

As an example of expansion, $\texttt{Lcoerce}(a, \texttt{Cint2float})$ should become $\texttt{Lunop}(\texttt{Ofloatofint}, a)$, materializing the `Cint2float` coercion from integers to FP numbers by an explicit application of the operator `floatofint`.

Pay attention to efficiency of the expanded expressions. The expansion of $\texttt{Lcoerce}(a, c)$ should evaluate a exactly once (in call-by-value), no matter how complex the c coercion is.

(You may wonder why Task 2 generates coercions that we eliminate in this task, instead of directly producing the corresponding IL computations during Task 2. The reason is that, by representing coercions specially, we can perform optimizations over coercions more easily: this is the topic of Task 5 below.)

Task 4: compilation to the Modern SECD

Implement a compiler from IL expressions (not containing `Lcoerce` constructs) to machine code for the abstract machine described in file `Mach.mli`. The file to modify is `Compile.ml`.

The abstract machine is a simple extension of the Modern SECD from Leroy's lecture 2. The compilation scheme to implement is similar to that for the Modern SECD. Try to perform tail-call optimization.

Variables in IL are represented by names, while the compilation scheme to the Modern SECD given in Leroy's lecture 2 assumes that variables are represented by de Bruijn indices. You need to convert from variable names to de Bruijn indices, either on the fly during machine code generation, or as a prior pass from IL to a variant of IL that uses de Bruijn indices.

Task 5: optimization of coercions

The IL code produced by the naive type-directed translation from Task 2 can be quite inefficient, owing to useless or redundant or inefficient coercions being generated. (See below for some examples and file `tests/optims` for more examples.) Study these inefficiencies, design optimizations that could remove them, and implement those optimizations as an IL-to-IL compilation pass. The file to modify is `Optimize.ml`.

Some examples of coercions that can be optimized:

- Corresponding to $\{x : \texttt{int}\} <: \{x : \top\}$ is the complex coercion $\texttt{Crecord}[(1, \texttt{Cid})]$ that reconstructs a tuple identical to the tuple argument of the coercion. The coercion could be replaced by the identity coercion, so that the original tuple is reused.
- Two consecutive subsumption steps result in nested coercions $\texttt{Lcoerce}(\texttt{Lcoerce}(a, c_1), c_2)$. Those can be replaced by a single coercion $\texttt{Lcoerce}(a, c)$ for an appropriate c.
- Instead of building a record with many fields then coercing it to a record type with fewer fields, why not build the record with fewer fields directly?

This task is open-ended, as many optimizations can be performed on the IL language. Indeed, since SUB is normalizing, one could execute the IL terms generated from SUB entirely at compile-time, leaving no nontrivial computations to be done at run-time. Please refrain from doing so and focus on optimizations that would remain valid if SUB were extended with recursive functions or other non-normalizing constructs.

For extra credit

If you have time and energy left, this project can be extended in many directions. For example:

Parametric polymorphism (System F) Add support for explicit ploymorphism in the style of System F. Universally-quantified type variables α are subtypes of themselves and of $\top$, but of no other type.

Bounded polymorphism (System $F_{<:}$) Add support for bounded polymorphism $\forall \alpha <: \sigma.\ \tau$ as described in chapters 25, 26 and 27 of Pierce's *Types and Programming Languages*. Warning: this is quite difficult!

Named types and variance inference Add support for parameterized type abbreviations such as

```
type 'a point = {x: int; y: int; info: 'a}
```

To speed up the subtyping check, when checking two types σ `point` and τ `point` for subtyping, we would like to avoid expanding the definition of `point`, and instead just decide based on whether σ and τ are subtypes in one or both directions. To this end, you could analyze type definitions once and for all to determine whether they are *covariant* and/or *contravariant* or *invariant* in their type parameters, then exploit this information during subtyping checks.

5 Evaluation

Assignments will be evaluated by a combination of:

- Testing: your program will be run on the examples provided (in directory `test/`) and on additional examples.
- Reading your source code, for correctness and elegance.

6 What to turn in

When you are done, please e-mail `Xavier.Leroy@inria.fr` a `.tar.gz` archive containing:

- All your source files. (Please remove the `src/_build` directory used by `ocamlbuild`.)
- Additional test files written in the SUB language, if you wrote any.
- A `README` or `LISEZMOI` file explaining briefly how far you went, what kind of optimizations you performed, and any "extra credit" features you added.

Appendix: concrete syntax of SUB

This is the syntax recognized by the provided parser.

Types:

τ	::=	`T`	the "top" universal type, $\top$
	|	`int`	the type of integer numbers
	|	`float`	the type of floating-point numbers
	|	τ_1 `->` τ_2	function types
	|	`{ }`	empty record type
	|	`{`ℓ_1`:` τ_1`;` ... `;` ℓ_n`:` τ_n`}`	record type
	|	`(`τ`)`	

Expressions:

e	::=	x	variable
	|	`fun (`x `:` τ`) ->` e	function abstraction
	|	`fun (`x_1 `:` τ_1`)` ... `(`x_n `:` τ_n`) ->` e	curried function abstraction
	|	e_1 e_2	function application
	|	`let` x `=` e_1 `in` e_2	`let` binding
	|	`let` f `(`x_1 `:` τ_1`)` ... `(`x_n `:` τ_n`) =` e_1 `in` e_2	`let` binding of a function abstraction
	|	`(`e `:` τ`)`	type constraint
	|	`{ }`	empty record
	|	`{`ℓ_1 `=` e_1`;` ... `;` ℓ_n `=` e_n`}`	record construction
	|	e`.`ℓ	access label ℓ in a record
	|	nn	integer constant
	|	$nn.nn$	floating-point constant
	|	`int` e	truncate e to an integer
	|	`float` e	convert e to floating-point
	|	e_1 `+` e_2	integer addition
	|	e_1 `+.` e_2	floating-point addition
	|	`(`e`)`	

Phrases:

p	::=	e `;;`	expression terminated by a double-semicolon

Identifiers:

x, f	variable names, same shape as Caml identifiers
ℓ	record labels, same shape as Caml identifiers