MPRI course 2-4-2
"Programmation fonctionnelle et systèmes de types"
Programming project

Giuseppe Castagna François Pottier Didier Rémy

2009–2010

An up-to-date version of this document can be found at http://gallium.inria.fr/~remy/mpri/.

1 Summary

The purpose of this programming project is to implement the mid-course "partiel" exam, more precisely to implement :

1. a type-checker for System F^η, with existential types and record types ;
2. a type-preserving translation from System F^η into System F.

The type-checker will be used to check the program before and after the translation. The following components of the system are provided : definitions of the syntaxes of terms, coercions, and types ; a lexer, parser, a pretty-printer, and support code for dealing with binders and reporting errors.

2 Required software

To use the sources that we provide, you will need :

Objective Caml Any version ≥ 3.0 should do, but in doubt install version 3.11 from http://caml.inria.fr/ocaml/release.en.html or from the packages available in your Linux distribution.

The Menhir parser generator Available at http://gallium.inria.fr/~fpottier/menhir/. This tool is required in order to produce `parser.mli` and `parser.ml` out of `parser.mly`.

Linux, FreeBSD, MacOSX, or some other Unix-like system The `Makefile` that we distribute has not been tested under Microsoft Windows. You are on your own if you insist on using Windows.

3 Overview of the provided sources

In order to make this a reasonably nice type-checker (with good error messages, etc.), we have provided a lot of support code. This code is briefly described in the list below. The modules at the beginning of the list, up to `pprint` are generic, and could be used as such in type-checkers for other languages. The rest of the modules, beginning with `syntax`, are specific to our little language.

The files that you should study first are probably `types.mli`, `terms.mli`, and `coercions.mli`. These files define the internal representations of types, terms, and coercions. You will work with these representations.

In the `src/` directory, you will find the following files :

identifierChop.mll, identifier.{ml, mli} Identifiers. An identifier is essentially a pair of a sort and a string. Each sort defines a disjoint namespace.

atom.{ml, mli} Atoms. An atom is the internal object used to represent a name.

error.{ml, mli} Generic error reporting facilities.

import.{ml, mli} Generic facilities for converting identifiers to atoms and detecting unbound identifiers.

export.{ml, mli} Generic facilities for converting atoms back to identifiers, while avoiding unintentional capture.

lexerUtil.{ml, mli} Generic utilities for lexical analysis.

pprint.{ml, mli} Generic pretty-printing facilities.

syntax.ml Abstract syntax for the types, coercions, and terms of System F^η. This syntax is produced by the parser. In this version of the syntax, names are represented as identifiers. Here, there are three sorts of identifiers, for term variables, type variables, and record labels.

parser.mly, lexer.mll Together, the lexer and parser define the concrete syntax for the language.

types.{ml, mli} Internal representation for the types of System F, up to α-conversion. This representation is used directly by the type-checker and translator. Informally, the abstract syntax of types is :

$$T ::= X \mid T \to T \mid \{\vec{\ell} : \vec{T}\} \mid \forall X.T \mid \exists X.T$$

terms.{ml, mli} Internal representation for the terms of System F^η. This representation is used directly by the type-checker. It also serves as both the source and target languages of the translation. Informally, the abstract syntax of terms is :

$$\begin{array}{rcl} t & ::= & x \\ & \mid & \texttt{fun}\ f\ (x : T) : T = t \\ & \mid & \texttt{fun}\ (x : T) : T = t \\ & \mid & t\ t \\ & \mid & \texttt{let}\ x = t\ \texttt{in}\ t \\ & \mid & \texttt{fun}\ X \to t \\ & \mid & t\,[T\,] \\ & \mid & \{\vec{\ell} : \vec{t}\} \\ & \mid & t.\ell \\ & \mid & \texttt{pack}\, t : \exists X.T \\ & \mid & \texttt{unpack}\, X, x = t\ \texttt{in}\ t \\ & \mid & \langle c \rangle\ t \end{array}$$

In this representation, all atoms are unique – that is, a term variable or a type variable is never bound twice in different places. This assumption can be exploited in the type-checker. Hence, this invariant must be carefully preserved when generating new terms. By comparison with the exam and the course, parameters of functions are explicitly annotated, so that typechecking is decidable. The return type of recursive functions is also required, but it is optional for non recursive functions $\texttt{fun}\ (x : T) : T = t$.

coercions.{ml, mli} Internal representation for the coercions. This representation is used by the typeche-

cker and translator. The abstract syntax of coercions is :

$$
\begin{array}{rcl}
\pi & ::= & \triangleright \mid \triangleleft \\
c & ::= & \# \\
& \mid & c; c \\
& \mid & c \to c \\
& \mid & \texttt{distrib} \\
& \mid & \forall X.c \\
& \mid & +\forall \\
& \mid & -\forall \pi\, T \\
& \mid & +\exists \pi\, T \\
& \mid & -\exists \\
& \mid & \exists X.c \\
& \mid & \{\vec{\ell} : \vec{c}\} \\
& \mid & \texttt{distrib}.\ell
\end{array}
$$

As for terms, all atoms are unique in this representation.

There are several changes in notation by comparison with the exam and new coercion forms for existentials and products :

- The identity coercion `id` is written $\#$.
- The `intro` and `elim` T forms for the $\forall$-quantifier are written $+\forall$ and $-\forall\pi\, T$. The $+$ and $-$ signs replace the `intro` and `elim` keywords for brevity. We added quantifiers in the notation to use a similar notation for existentials. The π flag in the forall elimination form is there to distinguish the direct $-\forall\triangleright T$ and indirect $-\forall\triangleleft T$ forms and emphasize the different meaning of T in these two forms. This distinction is necessary for type inference and explained in Section **??**.
- In coercion forms for existential types, the role of introduction and elimination are inverted by comparison with universal types : there is one introduction form $+\exists\pi\, T$ with explicit type information and one elimination form $-\exists$ with no type information.
- For records, we introduced the congruence form $\{\vec{\ell} : \vec{c}\}$ where each coercion applies to the corresponding record field and the `distrib`.ℓ coercion that pushes a toplevel universal quantifier inside field ℓ, while retaining the toplevel quantifier for other fields.
- Although not apparent in the syntax, we choose the distrib-right version of rule distrib that pushes the universal quantifier on the right-hand side of an arrow type provided the universal variable does not appear free on the left-hand side.

See the appendix for the typing and translation rules for coercions.

internalize.{ml, mli} Checks that all identifiers are bound, and replaces identifiers with unique atoms, so as to produce an internal representation of types and terms. This code comes after the parser.

print.{ml, mli} Pretty-printers for the types and terms of System F.

typerr.ml Some functions to report type errors during typechecking or elaboration. You can add more functions there.

typematch.{ml, mli} Functions to deconstruct types of some expected shape.

translate.{ml, mli} A type-checker and translator for coercions. This file is incomplete.

typecheck.{ml, mli} A type-checker for System F^η. This file is incomplete.

main.ml This driver interprets the command line and invokes the above modules as required.

Makefile, Makefile.auto, Makefile.shared, ocamldep.wrapper Build instructions. Issue the command "`make`" in order to generate the executable.

joujou The executable file for the program. Type "`./joujou` *filename*" to process the program stored in *filename*. This typechecks and translates the expression and retypechecks the translated expression to

verify that both types are equal. By default, this does not print anything to stdout and only reports type errors to stderr. Use the options "`-pty`" to print the inferred type or "`-pterm`" to print the translated program.

In the `test/` directory are small programs written in our functional language, which you can give as arguments to `joujou`. The programs in the `test/good` subdirectory are well-typed and should pass type-checking. The programs in the `test/bad` subdirectory contain type errors and should fail type-checking. Just run "`make test`" to make sure that these files are properly processed or rejected, as appropriate.

4 Tasks

We recommend completing task 1 before doing task 2. Only examples prefixed with "co_" require task 2 to be implemented.

Task 1 Implement a type-checker for System F. The file to modify is `typecheck.ml`. Follow Pottier's second lecture for directions. To prepare for task 2, the typechecker will return both a type and a term of that type, which may be a copy of the original term for the moment. The typechecker should fail if coercions are used and Task 2 is not yet implemented.

Task 2 Extend the type-checker to support coercions and to return a translation of terms with coercions into terms of System F. The source and target languages are identical, although translated programs should not contain coercions any more. The file to modify is `translate.ml`.

For extra credit If you wish to go further, you can extend the coercion language, the typechecker and translator (for instance) one or several of the following features :

- an option to allow the coercion of a record with more fields into a record with fewer fields.
- an option to allow the definition of coercions by a well-typed term of System F whose type-erasure is, after let-reduction, an η-expansion of the identity.

Advice The implementation of both tasks 1 and 2 could be split into 3 successive subtasks by

1. restricting to core System F (without existential types and records) ;
2. adding support for existential types ;
3. treating the full language with records.

We strongly recommend that you take checkpoints after completion of each subtasks so that you can later easily roll back to a previous consistent state in case you fail implementing the next task. Using a versioning tool such as `cvs` or `svn` is recommended.

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. Be sure to run "`make test`" !
- Reading your source code, for correctness and elegance.

6 What to turn in

When you are done, please e-mail `Francois.Pottier@inria.fr` and `Didier.Remy@inria.fr` and `Giuseppe.Castagna@pps.jussieu.fr` a `.tar.gz` archive containing :

- All your source files.

– Additional test files written in the small programming language, if you wrote any.
– If you implemented "extra credit" features, a `README` file (written in French or English) describing these additional features, how you implemented them, and where we should look in the source code to see how they are implemented.

7 Deadline

Please turn in your assignment on or before **Sunday, 21 February 2010**.

8 Typing and translation of coercions

The typechecking and translation of coercions can be defined simultaneously in the same judgment $c \triangleright T_1 \rightsquigarrow T_2, \varphi$, which specifies that coercion c of type $T_1 \to T_2$ translates to φ where φ is a function that maps a term t of type T_1 to an η-equivalent term of type T_2 [1]. We use the mathematical notation $\tau \mapsto t$ to define functions φ where the meta-variable τ ranges over terms t and reuse juxtaposition $\varphi\, t$ for mathematical

1. For record coercions, we use a let-binding `let` $x = \tau$ `in` $\{\vec{\ell} = x.\vec{\ell}\}$ to avoid duplication of the record expression to be coerced. So the resulting term is not *per se* an η-expansion of the coerced term. However, this is a restricted form of let-binding that could be written as a record pattern matching `let` $\{\vec{\ell} = \vec{x}\} = t$ `in` $\{\vec{\ell} = \vec{x}\}$ and arguably seen as an η-expansion form for records.

application. The translation rules are given below :

$$\textsc{Refl}\ \frac{}{\# : T \Rightarrow T \rightsquigarrow \tau \mapsto \tau} \qquad \textsc{Trans}\ \frac{c_1 : T_1 \Rightarrow T_2 \rightsquigarrow \varphi_1 \qquad c_2 : T_2 \Rightarrow T_3 \rightsquigarrow \varphi_2}{(c_1; c_2) : T_1 \Rightarrow T_3 \rightsquigarrow \tau \mapsto \varphi_2\ (\varphi_1\ \tau)}$$

$$\textsc{Congr-Arrow}\ \frac{c_1 : T_1' \Rightarrow T_1 \rightsquigarrow \varphi_1 \qquad c_2 : T_2 \Rightarrow T_2' \rightsquigarrow \varphi_2}{c_1 \to c_2 : T_1 \to T_2 \Rightarrow T_1' \to T_2' \rightsquigarrow \tau \mapsto \mathtt{fun}\ x : T_1' = \varphi_2(\tau\ (\varphi_1\ x))}$$

$$\textsc{Congr-Forall}\ \frac{c : T_1 \Rightarrow T_2 \rightsquigarrow \varphi}{\forall X.c : \forall X.T_1 \Rightarrow \forall X.T_2 \rightsquigarrow \tau \mapsto \mathtt{fun}\ X \to \varphi\ (\tau\ X)}$$

$$\textsc{Congr-Exists}\ \frac{c : T_1 \Rightarrow T_2 \rightsquigarrow \varphi}{\exists X.c : \exists X.T_1 \Rightarrow \exists X.T_2 \rightsquigarrow \tau \mapsto \mathtt{unpack}\, x, X = \tau\ \mathtt{in}\ \mathtt{pack}\, \varphi\, x : \exists X.T_2}$$

$$\textsc{Intro-Forall}\ \frac{X \# T}{+\forall : T \Rightarrow \forall X.T \rightsquigarrow \tau \mapsto \mathtt{fun}\ X \to \tau} \qquad \textsc{Elim-Exists}\ \frac{X \# T}{-\exists : \exists X.T \Rightarrow T \rightsquigarrow \tau \mapsto \mathtt{unpack}\, x, X = \tau\ \mathtt{in}\ x}$$

$$\textsc{Elim-Forall-Pos}\ \frac{}{-\forall_{\triangleright} T_1 : \forall X.T \Rightarrow [X \mapsto T_1]T \rightsquigarrow \tau \mapsto \tau\ T_1} \qquad \textsc{Intro-Exists-Neg}\ \frac{}{+\exists_{\triangleleft} T_1 : [X \mapsto T_1]T \Rightarrow \exists X.T \rightsquigarrow \tau \mapsto \mathtt{pack}\, \tau : \exists X.T}$$

$$\textsc{Elim-Forall-Neg}\ \frac{}{-\forall_{\triangleleft} \forall X.T : \forall X.T \Rightarrow [X \mapsto T_1]T \rightsquigarrow \tau \mapsto \tau\ T_1} \qquad \textsc{Intro-Exists-Pos}\ \frac{}{+\exists_{\triangleright} \exists X.T : [X \mapsto T_1]T \Rightarrow \exists X.T \rightsquigarrow \tau \mapsto \mathtt{pack}\, \tau : \exists X.T}$$

$$\textsc{Distrib-Right}\ \frac{X \# T_1}{\mathtt{distrib} : \forall X.(T_1 \to T_2) \Rightarrow T_1 \to (\forall X.T_2) \rightsquigarrow \tau \mapsto \mathtt{fun}\ (x : T_1) = \mathtt{fun}\ X \to (\tau\ X)\ x}$$

$$\textsc{Distrib-Label}\ \frac{}{\begin{array}{c}\mathtt{distrib}\,.\ell_0 : \forall X.\{\ell_0 : T_0; \vec{\ell} : \vec{T}\} \Rightarrow \forall X.\{\ell_0 : \forall X.T_0; \vec{\ell} : \vec{T}\} \rightsquigarrow \\ \tau \mapsto \mathtt{let}\ x = \tau\ \mathtt{in}\ \mathtt{fun}\ X \to \{\ell_0 = (\mathtt{fun}\ Y \to (x\ Y).\ell_0); \vec{\ell} = (x\ X).\vec{\ell}\}\end{array}}$$

$$\textsc{Record}\ \frac{\vec{c} : \vec{T} \Rightarrow \vec{T}' \rightsquigarrow \vec{\varphi}}{\{\vec{\ell} : \vec{c}\} : \{\vec{\ell} : \vec{T}\} \Rightarrow \{\vec{\ell} : \vec{T}'\} \rightsquigarrow \tau \mapsto \mathtt{let}\ x = \tau\ \mathtt{in}\ \{\vec{\ell} = \vec{\varphi}\ (x.\vec{\ell})\}}$$

The rules are deterministic and define an algorithm if c and both the domain and the codomain of c are given and φ is returned. However, these asumptions would imply a lot of redundant type information in the source program. Since a term has a unique type in System F^{η}, the domain of a coercion is always known from the term to which it is applied.

In some cases (such as #) the codomain of the coercion is also fully determined by the coercion and its domain. In other cases, some type information is still needed to reconstruct the codomain of the coercion. For example, the $\forall$ elimination form $-\forall_{\triangleright} T$ must provide the type T to instantiate the domain of the coercion. Moreover, to translate the arrow coercion $c_1 \to c_2$ with domain $T_1 \to T_2$ we must translate the coercion c_1 with codomain T_1 but with unknown domain : the arrow coercion inverts the flow of information. This is why the translation algorithm must be called in one of two modes with either the domain or the codomain of the coercion given. When called with the codomain, the $\forall$ elimination form $-\forall_{\triangleleft} T$ provides the domain

of the coercion (as can be read from the typing rules). The situation is similar, but with opposite cases, for existential coercions.

The translation algorithm can be described as a deterministic judgment $c\,\pi\,T \rightsquigarrow T', \varphi$ where the coercion c, the flag π, the type T that is the domain of the coercion when π is $\triangleright$ and its codomain when π is $\triangleleft$ are inputs and the type T' and φ are outputs. The type T' is the codomain of the coercion when π is $\triangleright$ and its domain when π is $\triangleleft$.

Of course, the algorithm is tightly related to the typing rules above by the following soundness properties : $c \triangleright T \rightsquigarrow T', \varphi$ implies $c : T \Rightarrow T' \rightsquigarrow \varphi$ and $c \triangleleft T \rightsquigarrow T', \varphi$ implies $c : T' \Rightarrow T \rightsquigarrow \varphi$. Conversely, $c : T \Rightarrow T' \rightsquigarrow \varphi$ implies both

- if $c \triangleright \triangleright \rightsquigarrow T, T_0\,\varphi_0$ then T_0 si T' and φ_0 is φ, and
- if $c \triangleleft T \rightsquigarrow \triangleleft, T_0\,\varphi_0$ then T_0 si T' and φ_0 is φ.

However, this is only a partial statement of completeness as it does not specify when both algorithms are defined. To state full completeness, we define positive coercions and negative coercions where $\triangleright$ and $\triangleleft$ appears only at positive and negative occurrences, respectively. Notice that some coercions such as $\#$ are both positive and negative, as they appear at anywhere, and others, such as $+\forall\triangleright T \to +\forall\triangleright T'$, are ill-defined. Then, if $c : T \Rightarrow T' \rightsquigarrow \varphi$, we have $c \triangleright T \rightsquigarrow T', \varphi$ whenever c is positive and $c \triangleleft T' \rightsquigarrow T, \varphi$ whenever c is negative.