Translating a Subset of OCaml into System F

Jonathan Protzenko

June 3, 2010

Contents

Part I
Some background

Type-checking functional languages à la ML is a long-time topic that has been well-studied throughout the past 30 years ([Hue75, GMM+78, DM82]). More recently, there has been a surge of interest regarding the translation of rich, complex, higher languages into simpler, core languages.

These core languages are well-typed, and the combination of simplicity and rich type information makes them well-suited for program analysis, program transformation and compilation.

Indeed, Haskell now features "System FC" [SCJD07], a core language that is both minimalistic and powerful. We have successfully applied the same design principles to a translation from OCaml [LDG+10] to System F_η, that is, System F with coercions [Mit88].

Our contribution is twofold: firstly, a strategy that builds upon previous work by François Pottier *et. al.* [PR05] and leverages constraint solving to build an explicitely typed representation of the original program; secondly, a translation process that explicits all the subtyping relations and transforms the original, fully-featured program into a System F_η term, where all coercions have been made explicit, all redundant constructs eliminated, and well-typedness preserved. This final representation can be type-checked before going any further, to ensure the consistency of the process.

This report is structured as follows. In the next paragraphs, we briefly describe how our translation process works, introduce the main concepts and our original motivation. The following part is devoted to our first contribution: adapting constraint generation to build a decorated term that corresponds to the original program with full type annotations. Part 3 is all about our translation process from the decorated AST to a simpler, core, fully-explicit System F term. Finally, we reflect on our work and possible extensions.

1 Motivations

OCaml is an old, feature-rich language. It features many extensions to the original ML language, such as

- a full module system, including first-class modules, recursive modules, functors,
- structural objects, with polymorphic methods, classes, class types,
- polymorphic variants, with private types,
- and a few more "dark corners".

Even if we consider a small subset of the language, some features are somehow unusual: for instance, **let**-patterns in OCaml are always generalizing, unlike Haskell.

The current trend is to prove as much as possible of the compilation process [L+04]. Indeed, it is difficult to fully

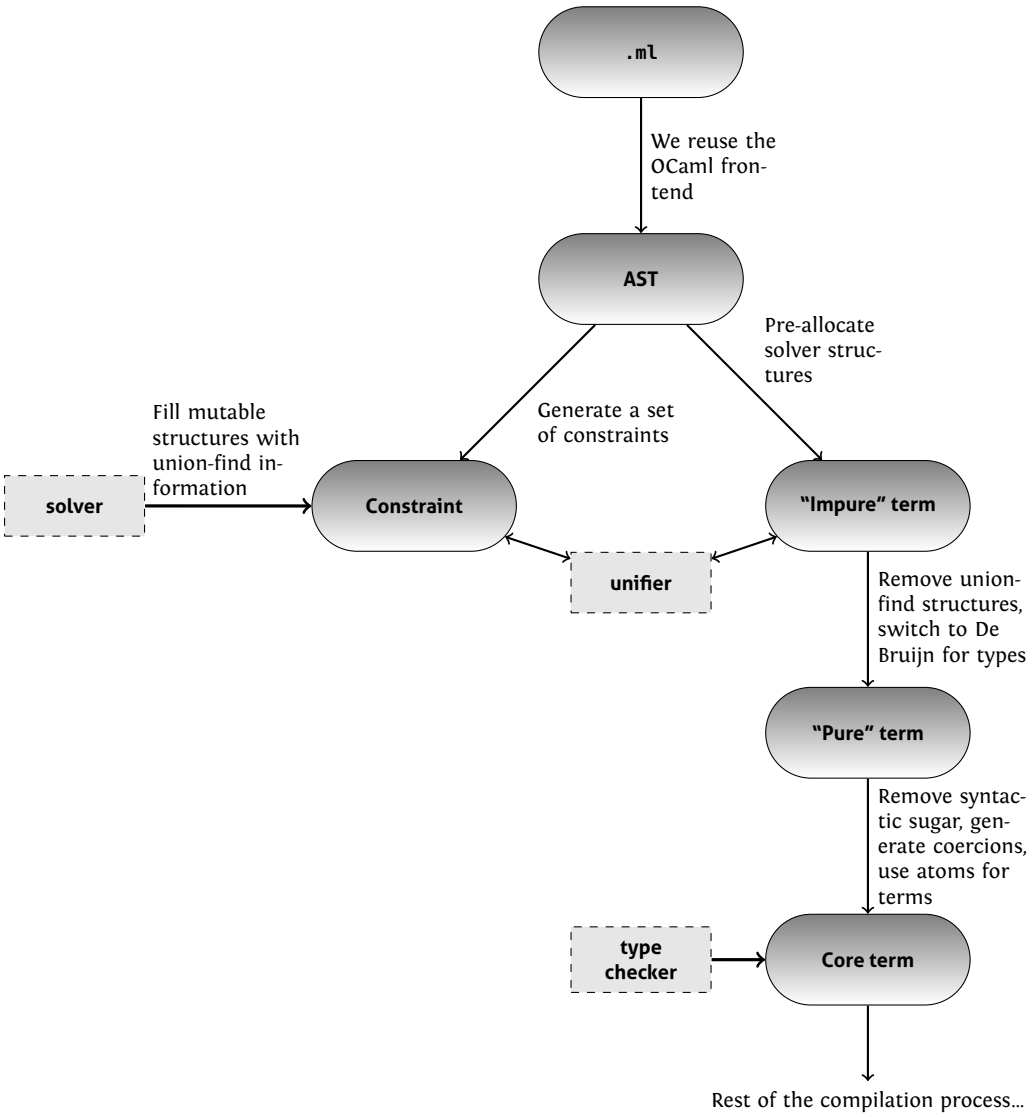

Figure 1: The overall structure of our translator

trust OCaml, because of its many features and possibly complex interactions between all the extra features.

Garrigue has been working in this direction ([Gar]) and has successfully extracted a program from a Coq proof that runs a small subset of OCaml, but it seems hard to prove a full, real-world type inferencer. Imperative structures used by classical union-find algorithms are not well-suited to formal proofs, and extracting a program from a Coq proof would most certainly result in a functional-style, less efficient program.

To strike a middle ground, **we propose to translate any OCaml program into an equivalent program written in a System F-like core language**.

The idea is to decompose complex, "hard-to-trust" features[1] into simpler, core building blocks. That way, instead of reasoning on the full OCaml language, we type-check a simple, core language. And this core language, we trust. Thus, we are *sure* that the program we are about to compile is, if not semantically correct, at least type-sound and won't crash.

The motto is, to quote Milner [Mil78], "Well-typed programs can't go wrong", and our aim is to enforce that.

2 A high-level description of our translator

Naturally, the tradeoff is that as expressions become simpler, and as we decompose complex features into a less expressive language, types become possibly more complex. However, this is not considered to be a problem, and although we end up with complex types, the type-checker for F_η remains simple and elegant.

Our process is made of the following steps (in chronological order) :

- we lex and parse the original source code, in order to obtain a parse tree[2];
- we generate constraints, and build an AST decorated

[1] We're specifically thinking about generalizing patterns and the value restriction here.

[2] This is done using the parser and lexer from the OCaml compiler, as we didn't want to waste time on this part.

with empty union-find structures;

- we solve the constraints; as a side-effect, this fills the union-find structures in the decorated AST with relevant information;
- we translate the "impure", decorated AST into a pure AST where union-find structures have disappeared and types are now using De Bruijn indices;
- we desugar this complex AST into a core one, and generate coercions to justify some steps on-the-fly;
- we type-check the resulting CoreML term and ensure well-typedness before going further.

Diagram 1 on the preceding page gives a graphical overview of the whole process. We will detail these steps in more detail.

We decided to reuse the OCaml parser and lexer for our tool. Because parsing OCaml is painful[3], and because we needed to compare the output of our tool with OCaml's, we decided to just use OCaml's **`parsing/`** subdirectory. All the other pieces of code in our tool are original work.

[3]We're thinking about the numerous parsing subtleties: **`let x, y =`** instead of **`let (x, y) =`**, and many more.

Part II

Constraint solving and decorated ASTs

The first requirement for our translation process to work properly was to build a full AST of the original program, *with explicit type annotations*. The type annotations were indeed required: given that type inference in System F is undecidable, the only opportunity we have to perform type inference is when we're still in ML. Once we have that explicit type annotation, we can send our program into System F, Curry-style, and then perform type-checking only.

We did *not* reuse the original OCaml further. Indeed, one of the side goals of this work was to show that constraint-based type-inferencing could be applied to OCaml, and possibly build a minimalistic replacement for the type-checker of OCaml. We are currently far from that goal, but fundamentally, the aim was to try something new, which is why we did not hack the OCaml typer[4].

Another reason for not reusing OCaml's type-inferencer is that the process of generating a fully annotated AST is closely linked to the way the type-inferencer works. We had no guarantee that OCaml's type-inferencer was well-suited for this task, which is why it seemed better to us to start from scratch for the type-inferencer.

In our tool, the part that performs type inferencing corresponds roughly to the left half of figure 1. We perform type inferencing using constraints [PR05]: the constraint generator walks the AST to generate a set of constraints, and another component, called the *solver*, runs a second pass to solve the constraints. The final result is an environment with all the top-level bindings and their types, or an error if the program cannot be type-checked.

This is far from a fully annotated AST, and some work was needed in order to transform this process into one suitable for generating a fully annotated AST. This is the first part of our work: adapting constraint-solving to our needs.

1 Constraint-based type inference

The main advantage of constraint-based type inference is the clean, elegant separation between constraint generation and constraint solving. Whereas traditional unification algorithms generate constraints and solve them on-the-fly, constraint-based type inference first generates constraints and then solves them in a separate pass.

This allows one to change the semantics of the original program (for instance, make some constructs more generalizing) without ever touching the solver. This clean separation is a feature we have tried to keep in our tool.

The reference paper for constraint-based type inference is [PR05]. The whole process is described in great

[4]Moreover, the OCaml type-inferencer is known to be hard to tackle, so as our goal was to build a prototype, it seemed reasonable to us *not* to try to reuse it.

σ	::=	*type scheme:*	C, D	::=	*Syntactic sugar for constraints:*
	$\forall \bar{\mathbf{X}}[C].\mathbf{T}$			...	*As before*
C, D	::=	*constraint:*		$\sigma \preceq \mathbf{T}$	$\mathbf{T}$ *is an instance of* σ
	true	*truth*		let $\mathbf{x} : \sigma$ in C	*def +* $\mathbf{x}$ *has an instance*
	false	*falsity*		$\exists\sigma$	σ *has an instance*
	$P\,\mathbf{T}_1\ \ldots\ \mathbf{T}_n$	*predicate application*		def Γ in C	*as before*
	$C \wedge C$	*conjunction*		let Γ in C	*as before*
	$\exists \bar{\mathbf{X}}.C$	*existential quantification*		$\exists\Gamma$	*as before*
	def $\mathbf{x} : \sigma$ in C	*type scheme introduction*			
	$\mathbf{x} \preceq \mathbf{T}$	*type scheme instantiation*			

Figure 2: Syntax of type schemes and constraints

$$
\begin{aligned}
[\![\mathbf{x} : \mathbf{T}]\!] &= \mathbf{x} \preceq \mathbf{T} \\
[\![\lambda \mathbf{z}.\mathbf{t} : \mathbf{T}]\!] &= \exists \mathbf{X}_1 \mathbf{X}_2.(\text{let } \mathbf{z} : \mathbf{X}_1 \text{ in } [\![\mathbf{t} : \mathbf{X}_2]\!] \wedge \mathbf{X}_1 \to \mathbf{X}_2 \leq \mathbf{T}) \\
[\![\mathbf{t}_1\ \mathbf{t}_2 : \mathbf{T}]\!] &= \exists \mathbf{X}_2.([\![\mathbf{t}_1 : \mathbf{X}_2 \to \mathbf{T}]\!] \wedge [\![\mathbf{t}_2 : \mathbf{X}_2]\!]) \\
[\![\mathbf{let}\ \mathbf{z} = \mathbf{t}_1\ \mathbf{in}\ \mathbf{t}_2 : \mathbf{T}]\!] &= \text{let } \mathbf{z} : \forall \mathbf{X}[[\![\mathbf{t}_1 : \mathbf{X}]\!].\mathbf{X} \text{ in } [\![\mathbf{t}_2 : \mathbf{T}]\!]
\end{aligned}
$$

Figure 3: Constraint generation

detail, and this is what our implementation used as a reference. Without giving too much detail, let us just show the syntax of constraints (figure 2) and the rules for constraint generation (figure 3).

2 Generating an explicitly-typed term

Because the original syntax of constraints does not mirror that of the expressions, we have no way to recover the original term from a constraint. For instance, a **`fun x ->`** expression in the AST, a **`match`** in the AST, a **`let`** in the AST all result in a **`let`**-constraint (see figure 3). How can we rebuild the expression from the constraint?

a. Encode the term in the syntax of constraints

A first idea we explored was to extend the syntax of constraints so that they convey enough information to allow us to recover the expression.

As one can see from figure 4 on the following page, this process was quite tedious: there were on the one hand constraints that do not originate from any expression in particular, and constraints that correspond to a specific expression in the AST. In this figure, underlined constraints are labeled with the corresponding expression.

In this scenario, it was up to the solver to rebuild the fully typed AST when solving the constraints. Because the solver walks down the constraints tree, the idea was to make it rebuild the term on-the-fly: for instance, in the case of the application, the solver would run two recursive calls, solve both constraints, obtain two terms, and apply the first to the second one, hence building a new term that can be returned.

With regular constraint solving, the solver is defined as an automaton, with a set of rewriting / solving rules. The state of the solver is a triple, which consists in a *stack*, a *unification environment* and a *constraint*. The final state is as follows:

$$\exists \bar{\mathbf{X}}.[\ldots]; \mathcal{U}; \mathsf{true}$$

The first component is the environment with all the top-level bindings, $\mathcal{U}$ is the unification knowledge obtained so far, and true is the empty constraint, meaning there is nothing left to solve.

In this new scenario, the final state would be:

$$\exists \bar{\mathbf{X}}.[\ldots]; \mathcal{U}; CT$$

Where CT is the final constraint-as-a-term that corresponds to our annotated AST.

The solution was rather attractive for a number of reasons:

- although tedious, the formalization was clear and more in the spirit of previous work made on constraints,
- the whole process of translating the original language was more linear: constraint generator, then solver, which returns an explicitly typed AST, then translation into System F,
- we would get rid of the imperative, union-find structures used by the solver as soon as the solver was done with solving the constraints.

However, this solution was discarded. The main reason was that it boiled down to making the cartesian product of constraints with expressions, that is, for each possible constraint, create a number of variants for each expression it could correspond to. The number of constraints was steadily growing, and we gave up because this solution was not so elegant anymore.

Moreover, some parts were quite unclear. Many implementations details were only roughly specified, and when trying to implement this in practice, it was not easy to follow closely the specification.

C ::=	*constraint:*
$C \wedge C$	*conjunction*
$\exists \bar{\mathbf{X}}.C$	*existential quantification*
$\mathbf{x} \preceq \mathbf{T}$	*type scheme instantiation*
$\mathbf{T} = \mathbf{T}$	*type equality*
...	*as before*

CT ::=	*constraint with term:*
$\exists \bar{\mathbf{X}}.CT$	*existential quantification*
$CT \wedge C$	*mix with a regular constraint*
$C \wedge CT$	*mix with a regular constraint*
$\underline{\mathbf{x}}$	*identifier*
$\underline{\mathbf{x} \preceq \mathbf{T}}$	*instance*
$\underline{\lambda \mathbf{z}.\mathbf{T}.\ CT}$	*function*
$\underline{CT\ CT}$	*application*
$\underline{\text{let } \mathbf{z} : \forall \mathbf{X}[CT].\mathbf{X} \text{ in } CT}$	***let**-binding*

Figure 4: Alternative encoding of expressions in constraints

b. Digging holes in terms

This led us to choose another solution, maybe more down-to-earth, but also much more tractable when it comes to implementing it.

The basic idea is for the constraint generator to pre-allocate solver structures[5]: for instance, when it generates the constraints for a **let**-binding, it pre-allocates a slot σ for the scheme, creates a **let**-constraint with a pointer to this slot, and creates a **let**-expression with a pointer to the *exact same slot*.

When the constraint generation is over, the driver has obtained two trees: one is the constraint tree, and the other one is the decorated AST.

The difference with our previous approach is that now the decorated AST and the constraint tree are separate structures. They are generated in parallel but are distinct entities; they only share pointers to common structures.

At this point, the common structures are empty, because the constraint generator only allocated them. The solver will fill them as it goes. However, as figure 5 shows, the type scheme σ is *shared* by the constraint and the decorated AST.

Constraint	**Decorated AST**
let $\forall \mathbf{X}[C_1] : \underbrace{\mathbf{z} \mapsto \mathbf{X}}_{\sigma}$ in C_2	let $\mathbf{z} : \sigma = E_1$ in E_2
$\exists \mathbf{X}_1 \mathbf{X}_2.(\text{let } \underbrace{\mathbf{z} \mapsto \mathbf{X}_1}_{\sigma} \text{ in } C_1 \wedge \mathbf{X}_1 \to \mathbf{X}_2 \leq \mathbf{T}$	fun $(\mathbf{z} : \sigma) \to E_1$
...	...

(σ allows one to recover the union-find structures for **z**)

Figure 5: Sample output of our modified constraint generator

What happens next is solving: the driver feeds the solver with the constraint it just obtained. In the process of solving it, it encounters the same **let**-constraint we were talking before. It recursively solves C_1 and C_2, and fills the pre-allocated slot σ with all the information gathered for this **let**. Finally, because the union-find structures are inherently mutable, the decorated expression built by the constraint generator now "magically" contains all the needed type information in its slot σ, although the solver never touched it!

c. Formalization

There are two kinds of structures we want to hold on for the rest of the translation process.

- instanciation information, that describes all the variables used to instanciate a type scheme;
- type scheme information: this can be the type scheme of a whole pattern, as in `let (x, y) = f e1 e2`, or only the type scheme associated to **x**[6].

As an optimization, the solver provides to the constraint generator a function for allocating a variable. This means that constraint variables $\mathbf{X}, \mathbf{X}_1, \mathbf{X}_2$ we've used before in the examples are actually unifier variables. This simplifies a lot of work in the solver, although it's not strictly speaking necessary.

Once again, one must think of the explicit System F that we're targeting: type abstraction is explicit, type instanciation is explicit as well, so we're just paving the way for the rest of the translation by collecting all the information we will need.

To summarize, the constraint generator is provided with:

- `new_var` to create a fresh type variable,
- `new_scheme_for_var` to attach a scheme to a given type variable,
- `new_instance_for_var` to attach a list of type variables to a given type variable.

The constraint generator creates variables as it goes, attaches schemes to capture the information needed, puts the scheme in the explicitely-typed term and in **let**-constraints, which is where the solver will find them and fill them with all the required information.

d. About this method

This method has proved to be easy to implement and work with, although it lacks some formal clarity. When used carefully, it provides a very flexible way to forward some information to further parts of the translation process.

[5] In practice, the constraint generator is a functor that is parameterized over the solver types and some solver helpers to create new "slots".

[6] Actually, we will need both

The trick is simply to think of what will be needed for the next steps, pre-allocate it, and then store it somewhere in the explicitely typed term, so that the next passes can find it and use it.

3 Verifying our work

Rewriting a type-inferencer and adapting it to our needs is a rather big task, and implementing it properly requires some care. To make sure our constraint generator and solver were reliable, we successfully developed a series of tests to check both the inferred types and the generated constraints.

The authors of [PR05] developed a prototype implementation of their work, called **mini**. This prototype has a constraint parsing facility, which we took advantage of. One of the first series of test we ran consisted in pretty-printing the constraint tree, and having the prototype implementation solve it. This allowed us to spot a few bugs in **mini**, but also to ensure our results were correct.

Another series of tests took advantage of the original OCaml. After the solver does his job, we obtain an environment with all the top-level **let**-bindings and their types in it. We wrote a quick parser for the output of **ocamlc -i**[7], and we have a series of tests that compares the output of our solver with that of OCaml[8].

A complexity test is also featured. Boris Yakobowski wrote a prototype implementation of MLF [RY08], that was supposedly faster than the prototype implementation for [PR05]. We compared our tool[9] with the two others, and we managed to find evidence for a complexity bug in **mini**[10].

[7] This prints the inferred signature of a compilation unit.

[8] **make tests** in the working directory will run the series of tests.

[9] **make benchs**, assuming you have the relevant packages, should plot some nice data

[10] Which was fixed later on by F. Pottier

Part III
System F and coercions...

System F [Rey74, Gir72] was introduced at the beginning of the '70s. Since then, many extensions have been derived: System $F_{<:}$ ("System F-sub") with subtyping, System F_μ for recursive types, System F_ω with functions from types to types and, finally, System F_η [Mit88].

The previous part was all about running type inference on the original ML program, and building a decorated AST with all the required type information. This part is about translating the decorated AST into a core language, close to System F_η, which is described in figure 6 on the following page.

1 An overview of the translation

a. Our version of System F

We're using our own flavour of System F_η, and the important features are.

- We're using patterns, and these patterns are used only in **match**es.
- We're using coercions *inside patterns* – this will be discussed later on.
- The syntax of types does not include $\forall$: indeed, since we're annotating λ-abstractions only, and since *we're in ML*, the arguments cannot have polymorphic types. Moreover, we're not doing any program transformations that would exhibit such polymorphic arguments.
- $\tau_1 \to \tau_2$ is understood to be the application of the "arrow" type constructor; the same goes for product types. Constant types (**int**, **unit**, ...) are type constructors with arity 0.
- Types are represented using De Bruijn indices, which is why type variables are integers.

b. The steps to System F

If one recalls the previous section, the output of the constraint generator is a term whose type information consists in union-find equivalence classes, with mutable structures and a lot of sharing. We cannot work with such a representation, which is why a first "cleanup" step is needed.

We're using types represented as De Bruijn indices. That is, type variable i refers to the i-th enclosing Λ above the term. This is more convenient and makes typechecking easier.

The first step transforms the union-find structures into those De Bruijn types. We still have a syntax of expressions (not described here) that is more or less equivalent to that of the original OCaml AST. We're not trying to desugar anything. Simply, we're cleaning up the representation of types.

The second step is where all the work is actually performed. A number of OCaml constructs were redundant:

x ::=		*identifier:*
	a	*an atom*
e ::=		*expression:*
	$\Lambda.e$	*type abstraction*
	$e[\tau]$	*type application*
	$e \blacktriangleright c$	*coercion application*
	$\lambda(x:\tau).e$	*λ-abstraction*
	$e_1\ e_2$	*application*
	let $x = e_1$ in e_2	***let**-binding*
	x	*instanciation*
	match e with $p_1 \to e_1 \mid \ldots \mid p_n \to e_n$	***match***
	$(e_1, \ldots, e_n)$	*n-ary tuple*
	...	
p ::=		*pattern:*
	x	*identifier*
	_	*wildcard*
	$p_1 \mid p_2$	*or pattern*
	$(p_1, \ldots, p_n)$	*tuple*
	$p \blacktriangleright c$	*coerce when matching this pattern*

c ::=		*coercion:*
	id	*identity*
	$c_1; c_2$	*composition*
	$\forall$	*∀ introduction*
	$\bullet[\tau]$	*∀ elimination*
	$\forall[c]$	*∀ covariance*
	$\forall\times$	*∀ distributivity*
	$p_i[c]$	*coercion projection on the i-th component*
τ ::=		*type:*
	$F(\tau_1, \ldots, \tau_n)$	*type constructor*
	α	*a type variable (an integer)*

Figure 6: Our syntax for System F

function, **let … and**, **let pattern = …**. We eliminate all those, and sometimes introduce temporary identifiers, or change the original expression a little bit. We also guarantee that identifiers are globally unique in the result of this transformation; they're *atoms*.

Once we've obtained a System F_η term, we can typecheck this term to ensure the consistency before proceeding with the rest of the compilation process.

2 Generating coercions

One feature of OCaml is that, as we said previously, patterns are generalizing. We discard the value restriction for the sake of clarity in this example.

```
let (l, f) = (fun x -> x)([], fun x -> x);;
val l : 'a list = []
val f : 'a -> 'a = <fun>
```

However, the main problem is that the expression on the right-hand side of the equals sign has type $\forall\alpha\beta.\ (\alpha\ \mathsf{list},\ \beta \to \beta)$.

```
# type 'a t = A of ('a -> 'a);;
# let A f = A (fun x -> x);;
val f : 'a -> 'a = <fun>
```

The example above is very similar, as the expression on the right side of the equals sign has type $\forall\alpha.\ \mathsf{A}(\alpha \to \alpha)$.

a. The core issue

If we now consider the expressions on the left-hand side of the equals sign, in the first case, if we think now of *F terms*, we'll be matching the pattern against the following expression:

$$\Lambda\Lambda\big(\Lambda.\ \lambda(x:0).\ x\big)\ \big[(\mathsf{list}[1], 0 \to 0)\big]\ \big(\mathsf{Nil}[1], (\Lambda.\ \lambda(x:0).\ x)[0]\big)$$

Now to assign types to the identifiers in the pattern, we must compute the type of this expression. What comes is[11]:

$$\forall\forall(\mathsf{list}[1], 0 \to 0)$$

The pattern-matching will most likely fail because the pattern on the left-hand side has a type that starts with head symbol "tuple", and the type on the right-hand side starts with an abstraction.

Similarly, the pattern matching in the second example will fail. One might think about changing the original expression, and translating $\Lambda.A(\lambda x.x)$ into $A(\Lambda.\lambda x.x)$ to solve this issue. This sounds like a good idea, but this is not applicable in situations such as the first example. We must operate on types, and we need *coercions*.

b. What is a coercion?

The previous discussion shows there is a need for *subtyping* in our system. We need a subtyping judgement: $\Gamma \vdash \tau \leq \tau'$ that tells us that τ' is a subtype of τ. Moreover, we need to *apply* these subtyping judgements at some specific points in the expressions so that the type of an expression e can be cast from τ to τ'. This is where we use coercions.

A coercion is a witness for subtyping. We say that a subtyping judgement is witnessed by a coercion:

$$\Gamma \vdash \tau \leq \tau' \rightsquigarrow c$$

For instance, the coercion $(\forall\times)$ witnesses the covariance of the tuple with regard to $\forall$.

$$\Gamma \vdash \forall(\tau_1, \ldots, \tau_n) \leq (\forall\tau_1, \ldots, \forall\tau_n) \rightsquigarrow (\forall\times)$$

The typing rule for a coercion thus becomes:

[11] Please note that we are using De Bruijn indices for types, so there's no identifiers after ∀

$$
\begin{array}{rcl}
[\![\_,\ \tau]\!] & = & \_ \\
[\![\mathbf{z},\ \tau]\!] & = & \mathbf{z} \blacktriangleright \mathsf{elim}(\tau) \\
[\![(p_1 \mid p_2),\ \tau]\!] & = & ([\![p_1,\ \tau]\!] \mid [\![p_2,\ \tau]\!]) \\
[\![(p_1, \ldots, p_n),\ \overline{\forall}(\tau_1, \ldots, \tau_n)]\!] & = & ([\![p_1, \overline{\forall}\tau_1]\!], \ldots, [\![p_n, \overline{\forall}\tau_n]\!]) \blacktriangleright \mathsf{push}(\overline{\forall}) \\
 & & \\
\mathsf{push}(\forall\overline{\forall}) & = & \forall\big[\mathsf{push}(\overline{\forall})\big];\ \forall\times \\
\mathsf{push}(\emptyset) & = & \mathsf{id} \\
 & & \\
\mathsf{elim}(\forall\tau) & = & \forall\big[\mathsf{elim}(\tau)\big];\ \bullet[\bot] \text{ if } 0 \# \tau \\
\mathsf{elim}(\forall\tau) & = & \forall\big[\mathsf{elim}(\tau)\big] \text{ otherwise} \\
\mathsf{elim}(\tau) & = & \mathsf{id} \text{ when } \tau \neq \forall\tau'
\end{array}
$$

Figure 7: Coercion generation

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \leq \tau' \rightsquigarrow c}{\Gamma \vdash (e \blacktriangleright c) : \tau'}$$

Other subtyping rules are pretty standard. Here is the i-th projection for a tuple.

$$\frac{\Gamma \vdash \tau_i \leq \tau_i' \rightsquigarrow c}{\Gamma \vdash (\tau_1, \ldots, \tau_i, \ldots, \tau_n) \leq (\tau_1, \ldots, \tau_i', \ldots, \tau_n) \rightsquigarrow p_i[c]}$$

Or the covariance of $\forall$.

$$\frac{\Gamma \vdash \tau \leq \tau' \rightsquigarrow c}{\Gamma \vdash \forall\tau \leq \forall\tau \rightsquigarrow \forall[c]}$$

c. Coercions in patterns

One feature of OCaml is that *or*-patterns can be nested arbitrarily deep. The following piece of code is valid:

```
type 'a t = A of ('a -> int) | B of ('a -> int) list;;
let (y, (A x | B [x])) = ...;;
```

However, we cannot simply apply a coercion to the expression on the right-hand side of the equals sign: the coercions needed on the left side and the right side of the or-pattern are not the same.

- The left side of the or-pattern will require $\forall\times$, which states that $\forall$ can be distributed inside the data type.
- The right side will require $\forall\times;\ p_A\,[\forall\times]$ which states that $\forall$ can be distributed inside the `t` data type, that its constructor `A` is covariant so we can move the $\forall$ inside the list data type under the `A` constructor.

In the end, we want to assign $\forall.\ 0 \to \mathsf{int}$ to **x**.

A first idea was to add a or-coercion, that is, a coercion $c_1 \mid c_2$ that mirrors the or-pattern and distributes c_1 and c_2 on the left side and the right side of the or-pattern when matching. Unfortunately, this solution was not powerful enough. Indeed, because we decided to implement a generalizing **`match`** (this is the topic of section 3), there were some cases where the coercions needed for each branch of a **`match`** construct were different. If the **`match`** only has one branch, then the or-coercion is enough. But if the **`match`** has different branches, one must choose a coercion for each branch of the **`match`**.

In this scenario, the most reasonable option was to attach coercions to patterns. That way, $e \blacktriangleright c$ becomes syntactic sugar for $\mathsf{let}\ \mathbf{x} \blacktriangleright c = e\ \mathsf{in}\ \mathbf{x}$. Applying a different coercion on each side of the or-pattern boils down to changing the pattern into $p_1 \blacktriangleright c_1 \mid p_2 \blacktriangleright c_2$.

d. A set of rules

What happens now is that, in the process of translating the "pure" term (see figure 1 on page 3), that has De Bruijn types but complex expressions, into System F, we *rewrite* patterns to introduce coercions wherever needed.

Although the subtyping relation in F_η is undecidable [Mit88], we have a deterministic procedure for adding the required coercions. This is due to the fact that we only use very specific coercions that can be determined by examining simultaneously the pattern and the type of the pattern.

Fortunately, the type of the whole pattern is a piece of information we've collected in the decorated AST, and that we've forwarded through the different passes. Thus, we just need to apply the rules in figure 7 to insert coercions in patterns when needed.

Informally, this procedure does only two things: it pushes the $\forall$s inside the tuples, and once it hits an identifier in the pattern, it removes all the leading $\forall$s in the type that are unused by instanciating them to $\bot$.

e. More on coercions

One of the goals we haven't achieved yet consists in mapping the value restriction *à la* Garrigue into these coercions. Because this just boils down to introducing a few $\forall$s and instanciating some of them to $\bot$, this should be feasible rather easily.

3 Other simplifications

Out of the many constructs that are offered by OCaml, many of them are redundant. For instance, the **`function`** keyword can be replaced by **`fun x -> match x with`**. We might also want stronger guarantees, such as the unicity of identifiers. These many simplifications are all performed in the translation from a "pure" AST to the Core language.

a. Unicity of identifiers

Previously, the identifiers were simply scoped strings. That is, one had to forward an environment with a mapping when walking the tree, in order to map information to identifiers. This is not necessarily an issue, but one construct of the original OCaml caused us some pain through our translations.

```
let _ =
  let i = 2 and j = 3 in
  let i = j + i and j = j - i in
  i, j
;;
- : int * int = (5, 1)
```

Because of these simultaneous definitions, we had to keep not only one scheme in the **let**-constraints, but a list of schemes. Similarly, we had to use lists all the way to the final step of the translation. One could argue that disambiguating this was possible early on. However, the initial steps are not supposed to deal with program transformations, and we leave this to the final step.

In order to recover the simple λ-calculus **let**-binding, we translate all identifiers to *atoms*. Atoms are records with a globally unique identifier and possibly more information, such as the name of the original identifier for errors and pretty-printing.

That way, we desugar the example above into:

```
let _/62 =
  let i/59 = 2 in
  let j/58 = 3 in
  let i/61 = (+)/56 j/58 i/59 in
  let j/60 = (-)/55 j/58 i/59 in
  (i/61, j/60)
in
()
```

Without the unique numbers appended to the original identifiers, this example would be semantically wrong[12]!

b. Desugaring

Here follow some quick descriptions of all the constructs we had to remove to end up with a pure, System F-like language.

⋆ Removing **let**-patterns

Patterns can be used conveniently in many places in OCaml. After removing simultaneous, multiple **let**-bindings, we endeavoured to remove **let**-patterns. Here's a sample program that uses a pattern.

```
let (x, y) = (fun x -> x)(1, 2)
;;
```

We translate it to a program that directly matches the corresponding expression. This is nothing but the standard semantics of **let**-pattern.

```
match
  (λ (x/39: int * int) -> x/39) (1, 2)
with
  | (x/37, y/38) ▸ id; id -> ()
```

⋆ Removing **function**

One feature that was trickier was removing **function**. This keyword allows one to fuse a regular **fun** and a **match** together. Let us consider the sample code below.

```
let fst = function x, y -> x
val fst: ∀ β α. α * β → α
```

The second line is the output from our solver that prints, as an intermediate result, the type of bound identifiers. If one decides to only keep the type schemes for indentifiers (that is, keeping the type scheme of **x** and **y**), then type information is missing to introduce a temporary identifier.

```
let fst/29 =
  ΛΛ. λ (__internal/30: 1 * 0) ->
    match __internal/30 with
      | (x/31, y/32) -> x/31
in
()
```

Fortunately, because we also annotate the decorated AST with the type schemes for the whole patterns, we can introduce an artificial identifier, switch to a regular **fun** and then match on the fake identifier. The resulting program is shown above, as the output of our pretty-printer.

One important thing to remember is that, because we are in ML, we don't need to apply coercions to this identifier. Functions *do not have polymorphic arguments* in ML, so there is really no reason to apply coercions to the **__internal** identifier.

c. Bonus features

We also decided to include a new feature, the *generalizing match*. The following example type-checks with our tool but does not with OCaml.

```
let s, f = match ("", fun x -> x) with
  | _, f ->
      f 2, f 2.
  | _ ->
      42, 42.
```

Because **f** needs to be polymorphic, we must generalize **e** in **match e with…**. However, OCaml does not perform such a generalization. This was originally started as a proof that such trivial changes only require tweaking the constraint generator, but we decided to keep it as the required changes to make it work in System F were minimal.

The important part is that because **match**es now can operate on expressions with polymorphism inside, we need to apply coercions inside **match**es. And because there are possibly different branches to a **match**, we might apply different coercions to each branch. This is what convinced us to attach coercions to patterns.

[12] The **let _** construct is treated as a special case in the original AST. This is treated as a pattern in our tool, and translated to a unique identifier in the final step of our translation. This avoid using a **match**.

```
match
  match
    Λ. ("", (λ (x/69: 0) -> x/69))
  with
    | (_, f/70) ▸ ∀×; id ->
        ((f/70•[int]) 2, (f/70•[float]) 2.)
    | _ ->
        (42, 42.)
with
  | (s/67, f/68) ▸ id; id ->
      ()
```

The sample output above is the pretty-printed F-term that results of this translation. Different coercions are used in the branches of the **match**, which further insists on the need to apply coercions in patterns.

4 An example

We finish this part with a slightly bigger example, and all the intermediate representations.

a. Original OCaml program

This program demonstrates the following features: **let**-patterns, generalization, instanciation, coercions.

We do not show the generated constraints below, as they are quite huge and hard to understand.

```
let a, b =
  let g, h = 2, fun x -> x in
  h 2, h
```

b. The decorated AST

The schemes that are displayed have been converted to De Bruijn. **let**-patterns are annotated with the scheme of the whole pattern, which is necessary to generate proper coercions. It will then be the task of the type-checker to assign schemes to individual identifiers, once it has applied all the required coercions.

```
let (a, b): ∀. [int * (0 → 0)] =
  let (g, h): ∀. [int * (0 → 0)] =
    (2, (fun (x: 0) -> x))
  in
  (h [int] 2, h [0])
in
()
```

Arguments to functions are annotated with their type as well, because we are targeting a Curry-style System F.

c. The core AST

This desugared AST coerces the resulting bindings by eliminating the unused quantification in the first component of the tuple (∀. int becomes just int).

We have chosen to compose the coercions for each branch of a tuple *outside* the tuple, for simplicity reasons. This is strictly equivalent to attaching the coercions inside each branch of the tuple, as we wrote in the rules for generating coercions.

```
match
  Λ. match
       Λ. (2, (λ (x/52: 0) -> x/52))
     with
       | (g/50, h/51) ▸ ∀×; p0[•[bottom]] ->
           (h/51•[int] 2, h/51•[0])
with
  | (a/48, b/49) ▸ ∀×; p0[•[bottom]] ->
      ()
```

The resulting **match** is actually generalizing, and **h** is instanciated twice with different arguments.

Part IV
A reflexion on this work

1 Future improvements

a. Actually writing the type-checker

As time went missing, we did not complete the final type-checker before finishing this report. It should be an easy task, though. Paving the way and finding the correct representation was actually more difficult, and we hope to finish this soon before the final presentation.

b. Enriching the language

Algebraic data types are missing, and we hope to add them soon. This will reveal some interesting challenges: indeed, recursive definitions will require to introduce μ combinators to pack and unpack recursive types, thus transforming our target language into $F_{\mu\eta}$. The coercions shouldn't be changed very much, though, except adding the coercions that witness the covariance of algebraic data types and the distributivity of $\forall$ with regard to them.

Moreover, although the first half of our tool properly supports equirecursive types, and properly prints inferred types that contains equirecursive types, the translation process does not support them at all. If we are to introduce μ-combinators, we also wish to explore the feasibility of introducing them whenever packing/unpacking equirecursive types.

Finally, if we introduce data types, we might as well introduce records. We're also thinking of introducing polymorphic variants to explore how well they translate into System F.

2 Related work

a. Modules

One core feature of ML is modules. Related research [RRD10] has explored a way to translate modules into System F. This matches our goals quite nicely, but recursive modules seem to be unsupported. This is one area that could be explored if we were to further enrich our language.

b. A real compiler

Simon Peyton Jones argues [SCJD07] that such an intermediate representation is well-suited for compilation. Since LLVM [LA04] has been making a lot of progress lately, and already starts to provide hooks for external GCs, it would be interesting to explore the feasibility of a LLVM backend for our intermediate language.

c. More modern features

As the language is still quite "clean", it might be interesting to explore some additions to the regular OCaml: better dealing with effects, possibly adding more modern features such as GADTs [SP04], or type classes. Such work remains a far, distant sight.

3 Conclusion

So far, it seems possible to translate a reasonable subset of OCaml into System F. This is interesting for checking the well-typedness of the intermediate representation, and maybe performing some program analysis, although we have not explored this path yet. Some features that initially seemed overly complicated actually can be translated quite naturally into System F, which justifies their existence. The framework we have built will allow us to type-check *a posteriori* some more complicated features, and we believe this work is already promising.

Some interesting exploration lies ahead of us: we could, for instance, try to translate more "exotic" features, such as polymorphic variants, and see how well they interact with our subset of System F_η. They might translate nicely, which would mean they correspond to some "fundamental" concept. If they do not translate nicely, this might mean they are not the right abstraction; perhaps changing their original semantics might help them express more "basic" ideas.

Finally, this work could be reused as a framework for performing type-checking *à la* ML for other languages. Since we have paid much attention to the general structure of our program, it is perfectly feasible to write a parser and a constraint generator for another language, without touching the rest of the tool. We hope to show with this experiment that a fresh design for type-checking is actually doable, and that it helps us augment our trust in the compilation process.

Part V
Bibliography

References

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[Gar] J. Garrigue. A Certified Implementation of ML with Structural Polymorphism.

[Gir72] J.Y. Girard. Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur. *Thèse d'etat, Université Paris VII*, 1972.

[GMM+78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130. ACM, 1978.

[Hue75] G.P. Huet. A unification algorithm for typed lambda-calculus. *TCS*, 1(1):27–57, 1975.

[L+04] X. Leroy et al. The CompCert verified compiler. *Development available at http://compcert. inria. fr*, 2009, 2004.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[LDG+10] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system release 3.12. *At http://caml. inria. fr*, 2010.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. 76(2–3):211–249, 1988.

[PR05] F. Pottier and D. Rémy. The essence of ML type inference, 2005.

[Rey74] J. Reynolds. Towards a theory of type structure. In *Programming Symposium*, pages 408–425. Springer, 1974.

[RRD10] A. Rossberg, C.V. Russo, and D. Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 89–102. ACM, 2010.

[RY08] D. Rémy and B. Yakobowski. From ML to ML F: graphic type constraints with efficient type inference. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 63–74. ACM, 2008.

[SCJD07] M. Sulzmann, M.M.T. Chakravarty, S.P. Jones, and K. Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, page 66. ACM, 2007.

[SP04] V. Simonet and F. Pottier. Constraint-based type inference for GADTs. 2004.