CS4215—Programming Language Implementation

Martin Henz and Chin Wei Ngan

Sunday 8th January, 2017

Chapter 10

Adding Polymorphism and Exception

The language simPL makes extensive use of functions for writing code and has a big advantage in that it is *strongly-typed* with each of its accepted expression being given a valid type. However, each of its functions is currently specified with a *monomorphic* type which denotes a particular type, in the sense that it is equal to itself and to no others. To support more reusable programming that remains well-typed, we shall look at how to extend the type system with *polymorphism*, and also investigate a way to automatically infer types for simPL programs. Such polymorphic types are used to possibly cover an infinite number of distinct type instances.

Therefore, we extend simPL's type system to a polymorphic version in Section 10.1 which permit the programmer to directly write more generic codes that remains well-typed. This will be especially important later when we support more complex data structures, and it is good to show how this useful feature can be achieved for a no-frills simPL language. In Section **??**, we look at how type inference may be carried out, using a decidable algorithm that was invented by Hindley and Milner, to minimise on the need for writing type annotations.

In Section 10.3, we take a fresh look at how to handle exceptional situations that arise from partial functions (that are undefined for certain inputs), and how the programmers can control such situations. We shall also see how to support exceptions in a type-safe manner. Finally, we extend the virtual machine for simPL by instructions that allow us to support the exception raising and exception handling in an efficient manner. As for polymorphism, we do not need to so any special things for the virtual machine. The only thing that we need to ensure is that each polymorphic function takes parameters and return results that are of the same size. We are already doing that for our virtual machine through the use of values (or pointers) that are of a word size each. Hence, polymorphic functions are readily supported.

10.1 Polymorphism for simPL

It is good thing that simPL is a strongly-typed, since many programming errors can be caught earlier at the program development stage with the help of a such a type system.

One flip side on the current simPL is that it is based on monomorphic typing, whereby its parameters and result are each given only a particular type. Let us look at a deficiency of monomorphic type system, with the help of a simple function.

```
fun {..} x -> x end
```

What type should you give to this identity function? Should you write its type as `int->int` or should it be `bool->bool`? You may retort by saying that it would depend on how you intend to use this function?

But what if you wanted to use this identity function in a number of contexts that requires different types, such as the simple example below:

Example 10.1 *Consider:*

```
let id = fun {..} x -> x end
 in {int}
 if (id false) then 3
 else (id 4) end
end
```

The first `id` *call,* `id false`*, would require it to have the type* `bool->bool`*, but the second* `id` *call,* `id 4`*, would require it to have the type* `int->int`*. Each of these types are* monomorphic *and would apply to only a single context of use. This example cannot be typed safely using only a monomorphic type system.*

To help us support reusable codes, we can either abandon types altogether and use a dynamically-typed language, or introduce a stronger type system with support for type polymorphism. The latter can be achieved by enriching the type system domain to capture *type variables* of the form `'a`, `'b` etc that can be used to denote some arbitrary types.

Thus, our simple identity function could be given the following polymorphic type, which makes it reusable in an infinite range of scenarios. This included also the possibility of passing functions as its parameter.

```
fun {'a->'a} x -> x end
```

Two other simple examples of polymorphic functions in simPL are:

```
fun {('a->'b)->'b} f x -> f x end
fun {('a->'a)->'a} f x -> f (f x) end
```

The first example applies a function-type parameter `f` to its second parameter `x`. The next example apply `f` twice to the second parameter `x`. Can you think of more examples of functions with polymorphic types for simPL? For example, what would be the type of a constant function, like `fun x -> 3 end` be?

To be more precise, our polymorphic types should have been formally written as

```
forall 'a. 'a->'a
forall 'a,'b. ('a->'b)->'b
forall 'a. ('a->'a)->'a
```

which says the type variables, `'a,'b` are universally quantified. Such types with universal quantified type variables are referred to as *polymorphic types* or *poly-type*, in short.

In the above examples, we have placed all the quantifiers at the outermost level, and they can be instantiated to some arbitrary types depending on the contexts of their uses. The position where the quantifiers are placed is significant. As a simple example, consider a monomorphic type without any quantifiers, namely:

```
('a->'a)->int
```

There are two ways of making this type into a polymorphic type by placing a quantifier for the type variable `'a`. They involve placing its quantifier at two different positions, as illustrated below:

```
(i)  forall a.('a->'a)->int
(ii) (forall a.'a->'a)->int
```

The first version places the quantifier at the outermost position, while the second version places the quantifier for just the function-type parameter. By default, when no quantifiers are being written, we typically assume that the type variables are quantified at the outermost level, and would pick the first version as the above two polymorphic types. However, the second version is actually better since it allows the function-type parameter to be used in a polymorphic manner.

Example 10.2 *As an example, we can re-write Example 10.1 to the following:*

```
 (fun {..} id ->
   if id false then 3 else id 4 end end)
  (fun {'a->'a} x -> x end)
```

Technically, we must give `(forall a.'a->'a)->int)` *as the missing type of the first function to allow* `id` *function to be used generically within the function definition.*

How do we classify such polymorphic types? Rank-0 types do not use any quantifier, while Rank-1 types uses only the outermost quantifiers. Types where the quantifiers appear at some inner locations are typically referred to as higher-ranks types. One problem with higher-rank types is that the inference process for them is, in general, undecidable.

Fortunately, there is a simple way to achieve inference for programs using only rank-1 types. This is done by assuming that let construct always introduces universal quantification for each of its let bound variables, where possible. Thus, for Example 10.1, we can achieve its intended generic behaviour by using only rank-1 types, as shown below:

```
let id = fun {'a->'a} x -> x end
 in {int}
 if (id false) then 3
 else (id 4) end
end
```

When this get transformed to

```
 (fun {..} id ->
   if id false then 3 else id 4 end end)
  (fun {'a->'a} x -> x end)
```

The missing type `{..}` that will be obtained from transforming away let construct would be a rank-2 type of the form `(forall a.'a->'a)->int`, but this transformation is carried out only after type inference, so it will not affect our attempt to support type inference for a polymorphic variant of simPL. Thus, let construct is more than just a syntactic sugar as it allows language implementors to support decidable inference for rank-1 polymorphic types. We shall next look at a polymorphic type system for simPL, and show how we may support type inference using it.

10.2 Type Inference for simPL

We shall now add type variables, denoted by `'a,'b,..` into our type system. This simple extension would result in types being captured by the following syntax rules:

$$\frac{}{'a \in Type} \qquad \frac{}{int \in Type} \qquad \frac{}{bool \in Type}$$

$$\frac{t_1 \in Type \qquad t_2 \in Type}{t_1\texttt{->}t_2 \in Type}$$

Without any quantifiers, the above type remains a *mono-type* with whereby each type variable can be used to denote only a particular but arbitrary type instance. To support polymorphic types, we may introduce universal quantifier to it at the outermost level using the following extra category, name as a *poly-type*.

$$\frac{t \in \mathit{Type}}{\texttt{forall}\ \texttt{'a}_1 \ldots \texttt{'a}_n \cdot t \in \mathit{PType}}$$

Such a polymorphic type is typically referred to as a rank-1 polymorphic type, which we shall be inferring for our simPL language.

If we wish to support higher-rank types, we would need to use the following more general construction for polymorphic types.

$$\frac{t \in \mathit{Type}}{\texttt{forall}\ \texttt{'a} \cdot t \in \mathit{Type}}$$

Here, the universal quantifiers for types can appear also in some inner parameter locations. Though type-checking is still possible for such types, the type inference process for higher-ranked types (beyond rank-2) have been shown to be problematic.

10.2.1 Type Rules

For simplicity, let us restrict the type system for simPL to only rank-1 polymorphic types. We begin with a set of type rules for simPL that is now based on rank-1 polymorphic types. For simplicity, we omit rules relating to primitives, and focus on only functions with one parameter since functions with multiple parameters can be viewed as syntactic sugar of functions with only a single parameter, and the type systems for primitives is essentially similar to that for function applications

$$\frac{}{\Gamma \vdash n : \mathit{int}}[\mathit{Int}] \qquad \frac{}{\Gamma \vdash \texttt{true} : \mathit{bool}}[\mathit{Bool}_1] \qquad \frac{}{\Gamma \vdash \texttt{false} : \mathit{bool}}[\mathit{Bool}_2]$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)}[\mathit{Var}] \qquad \frac{\Gamma[x \mapsto t_1] \vdash e : t_2}{\Gamma \vdash \texttt{fun}\ \{\cdot\}\ x\ \texttt{->}\ e\ \texttt{end} : t_1 \texttt{->} t_2}[\mathit{Fun}]$$

$$\frac{\Gamma[x{\mapsto}t_1][f{\mapsto}t_1\texttt{->}t_2] \vdash e : t_2}{\Gamma \vdash \texttt{recfun}\ \{\cdot\}\ f\ x\ \texttt{->}\ e\ \texttt{end} : t_1\texttt{->}t_2}\,[RecFun]$$

$$\frac{\Gamma \vdash e_1 : t_1\texttt{->}t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1\ e_2 : t_2}\,[Appln] \qquad \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma[x{\mapsto}t_1] \vdash e_2 : t_2}{\Gamma \vdash \texttt{let x} = e_1\ \texttt{in}\, e_2\ \texttt{end} : t_2}\,[Let]$$

$$\frac{\Gamma \vdash e : \texttt{forall 'a}\cdot t \quad fresh\ \texttt{'b}}{\Gamma \vdash e : [\texttt{'a}{\mapsto}\texttt{'b}]t}\,[Inst] \qquad \frac{\Gamma \vdash e : t \quad \texttt{'a} \notin fv(\Gamma)}{\Gamma \vdash e : \texttt{forall 'a}\cdot t}\,[Gen]$$

If you look at the rules, all of them are syntax-driven, except for the last two rules, which are structural in nature. They tell us how to instantiate a universal type and how to universally quantify a given type.

Type inference for polymorphic types relies on three key techniques. Firstly, when should we perform the structural rules. Secondly, if we are unsure about a given type, we can always introduce a fresh type variable to denote it. Lastly, we can rely on type unification to denote and solve for the unknown types. We can use type unification since equational solving is the main algorithm that is needed to decide on suitable types.

10.2.2 Type Unification

Given two types t_1 and t_2, we say that they can be made equal to each other if they can be *unified*. For example, given two types `(int->'b)` and `('a->'c)`, we can make them equal if we have the substitution $[\texttt{'a}{\mapsto}\texttt{int}, \texttt{'b}{\mapsto}\texttt{c}]$. This substitution when applied to the two types would made both of them have the same type, namely `(int->'c)`. Such a substitution can be obtained by applying unification on the two types that are to be made equal.

Formally, we can denote the unification of two types t_1 and t_2 by the process $t_1 \Cup t_2 \Longrightarrow \rho$, which is expected to yield a unifier ρ which when applied to the two types can make them into the same type, as follows $\rho\ t_1 \equiv \rho\ t_2$.

The unification mechanism has been used for executing pattern-matching mechanism for logic (or relational) languages. It is also critical for type inference. We will now illustrate its procedure on a the simple types of our simPL language. But first, we must extend it to the unification of a set of pairs of types, as follows: $t_{11} \Cup t_{12} \wedge t_{21} \Cup t_{22} \wedge \cdots \wedge t_{n1} \Cup t_{n2}$. To focus on one pair at a time, we will be using the notation $\phi \wedge t_1 \Cup t_2$, which focus on the unification of a pair of types t_1 and t_2, while ϕ contains the remaining pairs of types that are yet to be unified.

$$\frac{\phi \Longrightarrow \rho}{\phi \wedge t \Cup t \Longrightarrow \rho}[\textit{U-EQ}] \qquad \frac{\phi \wedge t_1 \Cup t_3 \wedge t_2 \Cup t_4 \Longrightarrow \rho}{\phi \wedge t_1\texttt{->}t_2 \Cup t_3\texttt{->}t_4 \Longrightarrow \rho}[\textit{U-ARR}]$$

$$\frac{\phi \wedge \texttt{'a} \Cup t \Longrightarrow \rho}{\phi \wedge t \Cup \texttt{'a} \Longrightarrow \rho}[\textit{U-COMM}] \qquad \frac{\texttt{'a}\notin t \quad \rho_1 = [\texttt{'a}\mapsto t] \quad \rho_1\phi \Longrightarrow \rho}{\phi \wedge \texttt{'a} \Cup t \Longrightarrow \rho\rho_1}[\textit{U-SUBS}]$$

$$\frac{t_1 \neq t_2}{\phi \wedge t_1 \Cup t_2 \Longrightarrow \bot}[\textit{U-NE}] \qquad \frac{\texttt{'a}\in t \quad \texttt{'a}\neq t}{\phi \wedge \texttt{'a} \Cup t \Longrightarrow \bot}[\textit{U-INF}]$$

The unification algorithm fails under two scenarios, as detected by Rules [*U-NE*] and [*U-INF*]. The failure for [*U-NE*] occurs when we have incompatible types. For example, $\phi \wedge t_1\texttt{->}t_2 \Cup \texttt{int} \Longrightarrow \bot$ leads to a failure due to incompatible types. The failure for [*U-INF*] results from the need for infinite types. An example is $\phi \wedge \texttt{'a->int} \Cup \texttt{'a} \Longrightarrow \bot$ which cannot be finitely captured.

The other rules progressively reduce the equality constraints until substitutions (in the form of most general unifier) are formed.

10.2.3 Hindley-Milner Inference Algorithm

Though types are useful, writing them may take significant programmer efforts, particularly when we use more complex types. It would be nice if our language system can perform inference of types, where possible. In this section, we will look at how to infer polymorphic types for simPL. The algorithm we will show you was first invented by Hindley (in 1969) and independently re-discovered by Milner (in 1972).

The two structural rules, we saw earlier, are now integrated into a set of syntax-directed rules for type inference of the form:

$$\Gamma \vdash e \Longrightarrow t; \phi$$

Using this inference rule, the expression e under type environment Γ can be inferred to have type t that is valid under equational constraint ϕ.

As an example, we may have:

$$\Gamma[\texttt{y}\mapsto\texttt{int}] \vdash \texttt{fun x -> x y end} \Longrightarrow \texttt{'a->'b};\ \texttt{'a}=(\texttt{int->'b})$$

The equational constraint must be used together with the returned type in order to infer the final type of the expression, which can be computed as `(int->'b)->'b` for the above example.

Four key ideas used in this type inference algorithm are:

- We can use a fresh type variable to denote the type of some expression that is initially unknown.
- We can use unification to make two types equivalent to one another.
- We support instantiation of universally quantified type for each use of a variable.
- We generalise the type of each `let`-bound variable, so as to make such variables be of polymorphic type (with universal qunatification).

On the premise of the last idea, the use of `let` construct has moved beyond merely syntactic sugar; since it is now being used to introduce type polymorphism. This is a good thing since `let` construct also help us structure large programs into smaller components.

$$\frac{}{\Gamma \vdash n \Longrightarrow \mathit{int}; \mathit{true}} \qquad \frac{}{\Gamma \vdash \texttt{true} \Longrightarrow \mathit{bool}; \mathit{true}} \qquad \frac{}{\Gamma \vdash \texttt{false} \Longrightarrow \mathit{bool}; \mathit{true}}$$

$$\frac{t{=}\Gamma(x) \quad \mathit{inst}(t) \searrow t_2}{\Gamma \vdash x : t_2} \qquad \frac{\mathit{fresh}\ \texttt{'a} \quad \Gamma[x{\mapsto}\texttt{'a}] \vdash e \Longrightarrow t; \phi}{\Gamma \vdash \texttt{fun}\ \{\cdot\}\ x\ \texttt{->}\ e\ \texttt{end} \Longrightarrow \texttt{'a->}t; \phi}$$

$$\frac{\mathit{fresh}\ \texttt{'a}, \texttt{'f} \quad \Gamma[x{\mapsto}\texttt{'a}][f{\mapsto}\texttt{'f}] \vdash e \Longrightarrow t_2; \phi}{\Gamma \vdash \texttt{recfun}\ \{\cdot\}\ f\ x\ \texttt{->}\ e\ \texttt{end} \Longrightarrow \texttt{'a->}t_2; \phi \wedge (\texttt{'f} \Cup \texttt{'a->}t_2)}$$

$$\frac{\Gamma \vdash e_1 \Longrightarrow t_1; \phi_1 \quad \Gamma \vdash e_2 \Longrightarrow t_2; \phi_2 \quad \mathit{fresh}\ \texttt{'a}}{\Gamma \vdash e_1\ e_2 \Longrightarrow \texttt{'a}; \phi_1 \wedge \phi_2 \wedge (t_1 \Cup t_2\texttt{->'a})}$$

$$\frac{\Gamma \vdash e_1 \Longrightarrow t_1; \phi_1 \quad \mathit{gen}(\Gamma, t_1) \nearrow t_3 \quad \Gamma[x{\mapsto}t_3] \vdash e_2 \Longrightarrow t_2; \phi_2}{\Gamma \vdash \texttt{let x} = e_1\ \texttt{in}\ e_2\ \texttt{end} \Longrightarrow t_2; \phi_1 \wedge \phi_2}$$

$$\frac{\mathit{fresh}\ \texttt{'b}_1\ldots\texttt{'b}_n}{\mathit{inst}(\texttt{forall}\ \texttt{'a}_1\ldots\texttt{'a}_n\cdot t)\searrow[\texttt{'a}_1\mapsto\texttt{'b}_1\cdots\texttt{'a}_n\mapsto\texttt{'b}_n]t}$$

$$\frac{\{\texttt{'a}_1\ldots\texttt{'a}_n\} = \mathit{fvars}(t)-\mathit{fvars}(\Gamma)}{\mathit{gen}(\Gamma,t)\nearrow\texttt{forall}\ \texttt{'a}_1\ldots\texttt{'a}_n\cdot t}$$

Exercise 10.1 *As an example of type inference, consider the following:*

```
let apply = fun f a -> f a end
in (fun x -> x>0 end 3) & (fun x -> \ x end true)
```

Apply the type inference rule to help determine suitable types for the expression, as well as the types for the intermediate functions used.

10.3 Exceptions

In Section 7.6, we saw that an error value can adequately represent the only exceptional behavior in the interpreter of simPL, namely the partial function division, which is undefined for the argument 0. However, we may also have exceptions when we fail to meet some safety pre-condition of certain functions, or when we encountered some pattern-matching problem with data structures (which will be discussed in the next chapter). For simplicity, we have used just an integer number to represent the different kinds of exceptional situations. We will keep to this integer form for simplicity. However, we may often wish to have a better way for programmer to to recover from exceptional situations.

What we need is a way to wrap an expression E_1 such that when an exception occurs in E_1, its execution is terminated, and another expression E_2 is executed instead, to handle the exception. For example, the following expression tries to evaluate some `input`, which represents the input of the user of a calculator. If an exception occurs, the user may be asked for a new input.

```
try (evaluate input)
catch x
with if x==~1
     then (evaluate (readNewUserInput))
     else ...
     end
end
```

Note that in the expression following `with`, the variable `x` declared by `catch` refers to an integer that describes the exception. The exceptionresulting from a division by zero is being represented by an exceptional value `~1`.

The syntax of `try` expressions is given by the following rule.

$$\frac{E_1 \qquad E_2}{\texttt{try}\ E_1\ \texttt{catch}\ x\ \texttt{with}\ E_2\ \texttt{end}}$$

With the ability of “catching” exceptions in place, it seems natural that the programmer should be able to create her own exceptions, and catch them. The concept is called “throwing” an exception, and is supported as follows.

$$\frac{E}{\texttt{throw}\ E\ \texttt{end}}$$

As we limit ourselves to integer exceptions, the expression E must denote an integer value. By throwing an exception, the programmer can define by herself what situations are considered exceptions in her programs. For example, the following program fragment defines as exceptional any situation where a given `percentage` value exceeds 100, with an integer exception of value `101`.

```
if percentage > 100 then
   throw 101
   end
else ... end
```

This exception can then be caught by a surrounding expression and handled appropriately, as shown below.

```
try ...
catch x
with if x==101
     then ...
     else raise x
     end
end
```

Note how we catch only an exception with value `101`, and re-thrown other exceptions that are not targetted by this current handler. If we wish, we may express such targetted catching of exception by the following short-hand notation:

```
try ...
catch 101
with ...
end
```

where a constant `101` is written in place of an exception variable.

10.4 An Interpreter for polyPL

Our interpreter is build on top of simPL with polymorphic types and exception handling. Le us refer to this interpreter as polyPL To support exceptions, we extend our semantic domains as follows.

Sem. domain	Definition	Explanation
Bool	$\{\mathit{true}, \mathit{false}\}$	ring of booleans
Int	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	ring of integers
EV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Exc} + \mathbf{Fun}$	expressible values
DV	$\mathbf{Bool} + \mathbf{Int} + \mathbf{Fun}$	denotable values
Id	alphanumeric string	identifiers
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments
Fun	$\mathbf{DV} \rightsquigarrow \mathbf{EV}$	function values
Exc	$\perp(\mathbf{Int})$	exceptions

Our semantics must now take into account on the possibility of exceptions being raised or otherwise. Note that we can distinguish exceptions from other values because of the disjoint union in the definition of **EV**. In particular, exception values are being represented by the domain **Exc** with values of the form $\perp(\mathbf{Int})$. Note also that **DV** represents a denotable value that excludes exceptions (or non-exceptional).

Any expression whose evaluation encounters an exception evaluates to that exception. As in Section 8.6, it is tedious to write down all possible cases. To show the principle, we look at the case of addition and division.

$$\frac{\Delta \Vdash E_1 \rightarrowtail e}{\Delta \Vdash E_1\texttt{+}E_2 \rightarrowtail e}\ \texttt{if}\ e \in \mathbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v \qquad \Delta \Vdash E_2 \rightarrowtail e}{\Delta \Vdash E_1\texttt{+}E_2 \rightarrowtail e}\ \texttt{if}\ v \notin \mathbf{Exc}\ \texttt{and}\ e \in \mathbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1\texttt{+}E_2 \rightarrowtail v_1 + v_2}\ \texttt{if}\ v_1, v_2 \notin \mathbf{Exc}$$

Note that we need to be a bit more specific here than in Section 8.6, because different exceptions can come from different arguments of the addition.

The first three rules for division are similar.

$$\frac{\Delta \Vdash E_1 \rightarrowtail e}{\Delta \Vdash E_1/E_2 \rightarrowtail e}\,\texttt{if}\ e \in \mathbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v \qquad \Delta \Vdash E_2 \rightarrowtail e}{\Delta \Vdash E_1/E_2 \rightarrowtail e}\,\texttt{if}\ v \notin \mathbf{Exc}\ \texttt{and}\ e \in \mathbf{Exc}$$

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E_1/E_2 \rightarrowtail v_1/v_2}\,\texttt{if}\ v_1, v_2 \notin \mathbf{Exc}\ \texttt{and}\ v_2 \neq 0$$

The last rule for division below covers the case that the value of the second argument of division is 0. We need to "raise the appropriate exception, corresponding to the exceptional situation.

$$\frac{\Delta \Vdash E_1 \rightarrowtail v_1 \qquad \Delta \Vdash E_2 \rightarrowtail 0}{\Delta \Vdash E_1/E_2 \rightarrowtail \bot(-1)}\,\texttt{if}\ v_1 \notin \mathbf{Exc}$$

The rules are carefully designed to be non-overlapping, to avoid ambiguities in the interpreter.

Finally, the meaning of `throw` expressions is defined as follows.

$$\frac{\Delta \Vdash E \rightarrowtail e}{\Delta \Vdash \texttt{throw}\ E \rightarrowtail e}\,\texttt{if}\ \ e \in \mathbf{Exc}$$

$$\frac{\Delta \Vdash E \rightarrowtail n}{\Delta \Vdash \texttt{throw}\ E \rightarrowtail \bot(n)}$$

Note the a subtle point here in the evaluation of E is that exception e may occur. In this case, the exception e will be thrown rather than that of the "intended" integer value of E.