Type and Effect Systems

Asumu Takikawa

March 30, 2012

Imagine that you are provided a nullary function f that has the type $\rightarrow$ `Void`. What can you say about what this function does if and when you call it? Can you memoize this function safely? Can you run this on another core without any synchronization? Do you even know if control flow will return to you when you call this function?

The answer to all of these questions in most languages is "I don't know" unless you have access to the source code. The reason is that the `Void` type means the function might have some unspecified side effects. This could be anything from writing to disk, transferring control flow to an entirely different part of the program, or launching some ballistic missiles.

The motivation behind this talk is to allow automated reasoning about the *effects* of a program—without using monads or a uniqueness type system—by utilizing *type and effect systems*.

1 Types and Effects in FX

The idea originated at the MIT AI Lab where several researchers worked on a typed variant of Scheme called FX. The "Report on the FX Programming Language" has the following definition of an effect:

> An *effect* is a static description of the side-effects an expression may perform when it is evaluated. Just as a type describes *what* an expression computes, an effect describes *how* an expression computes.

To build up a formal system to talk about effects, let's start with System F and gradually enrich it. Below is a standard grammar for System F, with an abstract set of base terms b and base types $\mathcal{B}$. The type rules are also shown below. The operational semantics is omitted since it is not important for now.

$$e ::= b \mid \lambda x{:}\tau.\, e \mid e\ e \mid \Lambda\alpha{::}k.\, e \mid e\,[\tau] \qquad \text{(expressions)}$$

$$\tau ::= \mathcal{B} \mid \tau \rightarrow \tau \mid \forall\alpha{::}k.\tau \mid \alpha \qquad \text{(types)}$$

$$k ::= * \qquad \text{(kinds)}$$

The core calculus of FX is just a small addition to System F. We add a new syntactic category of effects and then introduce effects where appropriate into the expressions and types. Most types stay unchanged, but we introduce an effect annotation over the arrow of the function type. These are called the *latent effects* of the functions, which we will discuss in detail in a second.

Effects also introduce a new kind, excuse the pun, of polymorphism and thus we allow instantiation of polymorphic lambdas with effects. We also add a new kind $\square$ for effects.[1]

[1] This is non-standard notation. The FX report uses a verbose `effect` kind and the ATTAPL chapter elides a kind system entirely.

$e ::= \ldots \mid e\,[d]$ (expressions)

$d ::= \tau \mid \phi$ (description)

$\tau ::= \mathcal{B} \mid \tau \xrightarrow{\phi} \tau \mid \forall\alpha{::}k.\tau \mid \alpha$ (types)

$\phi ::= \mathcal{E} \mid (\cup\, \phi_1 ... \phi_n) \mid \beta$ (effects)

$k ::= * \mid \Box$ (kinds)

The effects are also parameterized by a set of base effects $\mathcal{E}$. The actual FX language includes many base types and effects. The base types are not that interesting to talk about (the FX report describes standard types such as numbers, strings, etc.) but the base effects are interesting.

The FX report includes the following effects in $\mathcal{E}$: `pure`, `read`, `write`, `init`. All of these have the kind $\Box$. A variant of FX that we will talk about called *FX/R* will modify these base effects slightly.

In FX, terms can have both a type τ and effect ϕ so instead of a type judgment $\Gamma; \Delta \vdash e : \tau$ we use a type and effect judgment $\Gamma; \Delta \vdash e : \tau \,!\, \phi$. Effects can be composed with a union operator $\cup$. Ordering for effects is not significant in FX, so you can think of the operator as being commutative and associative. This is not the case in all type effect systems. For example, the Nielsen and Nielsen type and effect system for CML uses a flow-sensitive system.

Before we get too deep into the formalism, let's think about how these types are actually used. Bare System F has no I/O primitives, reference cells, or anything else that can cause a side effect. To actually demonstrate anything interesting, we need to provide some base operations. So imagine that we extend the core calculus with I/O. Consider what type we should give the following code:

(**displayln** (**read-line**))

In just System F with some base operations and the `Void` type, this expression would just get the `Void` type. However, in FX this also has the effect ($\cup$ `read write`). Now let's consider a more interesting scenario. What type and effect do we give the following code?

(λ ()
  (**displayln** (**read-line**)))

Clearly the type of the lambda expression is $\rightarrow$ `Void`. Does the lambda expression have an effect though? That's actually a trick question: both answers could be valid, depending on your point of view. FX takes the view that a lambda abstraction itself has no effect (i.e, its effect is `pure`) when created. However, we will see that Tofte-Talpin region type system uses a different view.

On the other hand, that wasn't entirely a trick question. What happens when a function is applied rather than created? If you think about a lambda as a delayed computation, then it would make sense for the delayed computation—in this case the I/O calls—to have a delayed effect. That's why we have a notion of latent effect.

Latent effects are bookkeeping on function types so that the type system can retrieve the correct delayed effect of the computation later. When you use higher-order features of your language, latent effects have other uses as well.

Now let's proceed with the formalism again. Since our system has kinds, we need a kinding judgment. There is nothing too surprising about the kinding rules. The main difference from System F is the addition of the effect joining operation, which has the obvious kinding judgment.

K-Var
$$\frac{\alpha :: k \in \Delta}{\alpha :: k}$$

K-ForAll
$$\frac{\Delta, \alpha :: k \vdash \tau :: *}{\Delta \vdash \forall\alpha{::}k.\tau :: *}$$

K-Max
$$\frac{\Delta \vdash \phi_i :: \Box \quad (1 \leq i \leq n)}{\Delta \vdash (\cup\, \phi_1 \ldots \phi_n) :: \Box}$$

The type judgments for FX are more interesting. Let's start out with the variable rule:

$$\textsc{T-Var}\ \frac{\Gamma(x) = \tau}{\Gamma; \Delta \vdash x : \tau \,!\, \texttt{pure}}$$

The rule is virtually unchanged except that we give it a pure effect since variable reference does not affect the store (in our calculus anyway) or do I/O in any sense.

Next let's consider lambda abstraction and application:

$$\textsc{T-Abs}\ \frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,!\, \phi}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \xrightarrow{\phi} \tau_2 \,!\, \texttt{pure}} \qquad \textsc{T-App}\ \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2 \,!\, \phi_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \,!\, \phi_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau \,!\, (\cup\ \phi\ \ \phi_1\ \phi_2)}$$

We discussed earlier that lambda abstractions would be pure. On the other hand, the computation inside may have an effect. The rule just takes the effect for the inner expression and adds it as the latent effect of the function type.

Function application is slightly more complicated. Which expressions in the application can have an effect? Well, both of them can. Is that it? That's not quite enough yet. With the application rule, we have to remember to propagate the latent effect without dropping it on the floor. The effect of the entire expression is just the composition of the three effects we have.

Finally, we have type abstraction and application. Before we look at the stock FX rules, let's go back to the System F rule and think about how we would modify it. Here's the System F rule with $\boxed{?}$ as a placeholder for effects:

$$\textsc{T-TAbs}\ \frac{\Delta, \alpha :: k; \Gamma \vdash e : \tau \,!\, \boxed{?}}{\Delta; \Gamma \vdash \Lambda\alpha{::}k.\, e : \forall\alpha{::}k.\tau \,!\, \boxed{?}}$$

What should we put in those boxes? Let's think about our choices. In general, the expression e might have any effect, but type abstractions also delay the computation inside. We solved this for plain lambdas by introducing a latent type. We could do the same thing here, but we would then be forced to add an annotation on the $\forall$ type.

Another option is to require that the expression inside of a type abstraction is pure. That sounds limiting, but would that actually be a problem? If you think about it, type abstractions usually surround a plain application, which is pure itself. It is unlikely that there are many useful expressions that have an effect between the plain lambda and the outer big lambda. The advantage of making type applications pure is that they can be entirely erased from the program without any overhead.[2]

$$\textsc{T-TAbs}\ \frac{\Delta, \alpha :: k; \Gamma \vdash e : \tau \,!\, \texttt{pure}}{\Gamma; \Delta \vdash \Lambda\alpha{::}k.\, e : \forall\alpha{::}k.\tau \,!\, \texttt{pure}} \qquad \textsc{T-TApp}\ \frac{\Delta; \Gamma \vdash e_1 : \forall\alpha{::}k.\tau \,!\, \phi \qquad \Delta \vdash d :: k}{\Delta; \Gamma \vdash e_1\,[d] : \tau[\alpha/d] \,!\, \phi}$$

[2] Contrast this with System F, where erasure requires that you replace big lambdas with small lambdas if you have effects.

To summarize, here are all of the type and effect judgments together:

$$\text{T-Var}\quad \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau \,!\, \texttt{pure}} \qquad \text{T-Abs}\quad \frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2 \,!\, \phi}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \xrightarrow{\phi} \tau_2 \,!\, \texttt{pure}}$$

$$\text{T-App}\quad \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \xrightarrow{\phi} \tau_2 \,!\, \phi_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_1 \,!\, \phi_2}{\Delta; \Gamma \vdash e_1\ e_2 : \tau \,!\, (\cup\ \phi\ \phi_1\ \phi_2)} \qquad \text{T-TAbs}\quad \frac{\Delta, \alpha :: k; \Gamma \vdash e : \tau \,!\, \texttt{pure}}{\Delta; \Gamma \vdash \Lambda\alpha{::}k.\, e : \forall\alpha{::}k.\tau \,!\, \texttt{pure}}$$

$$\text{T-TApp}\quad \frac{\Delta; \Gamma \vdash e_1 : \forall\alpha{::}k.\tau \,!\, \phi \qquad \Delta \vdash d :: k}{\Delta; \Gamma \vdash e_1\,[d] : \tau[\alpha/d] \,!\, \phi}$$

One thing we haven't discussed in detail is the subtle change to the rules for polymorphism. Since FX has multiple base kinds, we actually find that polymorphism serves two purposes. To motivate this, consider what the type of the map function should be:

$$\text{map} : \forall\alpha.\forall\beta.(\alpha \to \beta), (\texttt{List}\,\alpha) \xrightarrow{\boxed{?}} (\texttt{List}\,\beta)$$

We can see that we need some effect over the arrow. What effect do we have though? Where FX really shines is that it supports higher-order functions in a natural way even with effects. All we need to do is add another abstraction in the above type:

$$\text{map} : \forall\gamma.\forall\alpha.\forall\beta.(\alpha \to \beta), (\texttt{List}\,\alpha) \xrightarrow{\gamma} (\texttt{List}\,\beta)$$

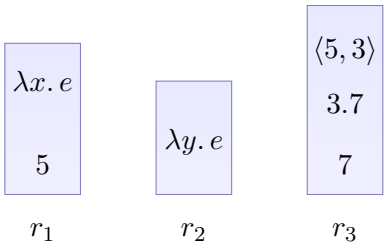
This is assuming that the map function itself has no effects other than its argument functions. In FX, this is technically not the case since all list-manipulating functions have `init` effect, since they allocate objects to memory.

In the full FX system, there are also subtyping and equality relations defined on effects so that some effects can be interchanged. For example, `pure` is the same as a composition of no effects and the composition of a single effect is just the effect itself.

2 Tofte-Talpin and Regions

So far, we have looked at the FX language, which defines a general-purpose type and effect system that is mainly designed to accommodate a dialect of Scheme with a type system similar to System F. Now we are going to switch gears and look at an application of a type and effect system for another aspect of programming languages: memory management.

The type and effect idea is used as a tool to express region-based memory management. A *region* is an abstract chunk of memory that will be allocated and store values from your program. Pictorially, the regions in a program look like a stack with different frame sizes:



Stack-based allocation is a special case where the regions have fixed sizes. Note that the allocations are custom-tailored to the particular program that you write. The tailoring is due either to the explicit annotations in your program—this is how we will formalize regions—or due to the allocations that a region inference algorithm finds. This kind of region-based memory management is actually quite an old idea dating back to the 60s and 70s, but their type-safe implementation in programming languages is a more recent idea.

This kind of memory management is an alternative to manual memory management or garbage collection. Region-based memory management was designed to address shortcomings by providing memory safety guaranteed by the type system, predictability of allocation due to the inference of allocation from your program, and better profiling support.[3]

We're not going to talk about full-blown practical region-based memory management, but we will talk about a calculus called RAL and its typed cousin RTL, which uses the Tofte-Talpin type and effect system.

RAL is an untyped calculus that makes region-based allocation explicit in the program syntax. New regions are created and bound in the body of a `new` $\rho.e$ expression, so that expressions in e can store values in ρ. Regions can either be a variable ρ or the special unallocated region $\bullet$ (i.e., an unallocated/unreachable value). You can imagine this memory management construct as a hybrid between stack allocation and heap allocation. The regions themselves are created in a stack-like fashion, but each region stores multiple values in the heap.

The only values we have in this language are closures and boolean constants. Since booleans are constants, they are not allocated in any regions. However, closures are explicitly allocated in a region. Lambda expressions are annotated with the region they are allocated in, e.g., r is the region that $\lambda x.\, e$ `at` r is allocated in. Similarly, a closure $(\lambda x.\, e)_r$ is the value form allocated at r.

A neat new idea that we will use in RAL is the notion of a *region abstraction*. These define new expressions that are abstract over the regions that they allocate in, which is a natural thing to do once you have the idea of regions. This allows, for example, for functions to allocate their results in different regions at different callsites. A region abstraction looks like $(\lambda\rho.\, u)_r$. Unlike lambda abstractions, a region abstraction does not contain a general term but contains an *almost* value—basically a closure or a lambda expression that will become a closure.[4]

$e ::= u \mid x \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid e\ e \mid e\,[r] \mid \texttt{fix}\ x.e \mid \texttt{new}\ \rho.e$ (expressions)

$u ::= v \mid \lambda x.\, e\ \texttt{at}\ r \mid \lambda\rho.\, u\ \texttt{at}\ r$ (almost values)

$v ::= \texttt{true} \mid \texttt{false} \mid (\lambda x.\, e)_r \mid (\lambda\rho.\, u)_r$ (values)

[3] These design motivations are based on the experience of the designers of ML-Kit. For more information, see the notes from the 1997 Summer School on Region-Based Memory Management: http://www.itu.dk/research/mlkit/kit2/summerschool/toc.html

[4] As far as I can tell, there is no need for this limitation except to keep the calculus as conservative as possible. Note that since we have ordinary lambdas, there is no real need to make region abstractions act as a suspended computation as well.

$r ::= \rho \mid \bullet$ (regions)

For the operational semantics, we have an unsurprising set of evaluation contexts as follows:

$E ::= [\cdot] \mid \texttt{if}\ E\ \texttt{then}\ e\ \texttt{else}\ e \mid E\ e \mid v\ E \mid E\,[r] \mid \texttt{fix}\ x.E \mid \texttt{new}\ \rho.E$ (evaluation contexts)

The operational semantics for the language is summarized in the following axioms:

$$\lambda x.\, e\ \texttt{at}\ \rho \longrightarrow (\lambda x.\, e)_\rho \qquad \textsc{Clos}$$
$$(\lambda x.\, e)_\rho\ v \longrightarrow e[x/v] \qquad \textsc{Beta}$$
$$\lambda\rho_1.\, u\ \texttt{at}\ \rho_2 \longrightarrow (\lambda\rho_1.\, u)_{\rho_2} \qquad \textsc{RClos}$$
$$(\lambda\rho_1.\, u)_{\rho_2}\,[r] \longrightarrow u[\rho_1/r] \qquad \textsc{RBeta}$$
$$\texttt{new}\ \rho.v \longrightarrow v[\rho/\bullet] \qquad \textsc{Dealloc}$$

The Beta rule is completely standard. The corresponding Clos rule takes a lambda expression and reduces it to the closure value. This distinction between lambda expressions and closures exists just to make the distinction between the program source and the values that are actually allocated in a region obvious. The corresponding rules for region abstractions are almost exactly the same.

The Dealloc rule is also interesting. The point of the rule is that after the reduction semantics finishes reducing under the allocation, the region is no longer accessible and should be deallocated. This ensures that any values that escape the region cannot be used by other parts of the program.

The other axioms, shown below, are completely standard.

$$\texttt{fix}\ x.v \longrightarrow v[x/\texttt{fix}\ x.v] \qquad \textsc{Fix}$$
$$\texttt{if true then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_1 \qquad \textsc{IfT}$$
$$\texttt{if false then}\ e_1\ \texttt{else}\ e_2 \longrightarrow e_2 \qquad \textsc{IfF}$$

There are some interesting properties to talk about in relation to the RAL calculus. One is that if you have a term e in RAL that contains deallocated values, you can replace these values with anything else and obtain an equivalent term e'. Another property that we'd like to have is that an RAL expression is operationally equivalent to the same term with regions erased (in a simpler language). That is, regions do not affect evaluation aside from the presence of regions.

2.1 Types + RAL = RTL

In the last few pages we have seen a new language construct that lets you do memory management in a new way. So how does this relate to the type and effect systems that we talked about earlier? It turns out that a natural way to ensure that RAL programs do not get stuck is to use a type and effect system. The "effects" in this case are the need for a memory region. The regions that an expression needs to be live to execute are going to be included in the effect. The resulting language is called RTL.

There are two motivations for creating a type system for this language. One is that type safety essentially means we have memory safety, since not getting stuck means you can't access a deallocated region. Another use for the type system is that it provides a hook to actually infer the region annotations that we have in RAL. More on that later.

RTL looks mostly like RAL, but with a grammar for types. Unfortunately, RTL is presented quite differently from FX because its type system uses lambdas without annotations and "guesses" types in many parts of the judgments.

The actual types in RTL look pretty standard except for the latent effect on arrows and the new region abstraction type $\Pi\rho.^{\phi}\tau$. Effects are either an empty effect or some sequence of region variables.

$$
\begin{array}{lll}
u & ::= v \mid \lambda x.\, e \;\texttt{at}\; r \mid \lambda\rho.\, u \;\texttt{at}\; r & \text{(almost values)} \\
v & ::= bv \mid (\lambda x.\, e)_r \mid (\lambda\rho.\, u)_r & \text{(values)} \\
bv & ::= \texttt{true} \mid \texttt{false} & \text{(booleans)} \\
\tau & ::= \texttt{Bool} \mid (\tau \xrightarrow{\phi} \tau, r) \mid (\Pi\rho.^{\phi}\tau, r) & \text{(types)} \\
\phi & ::= \emptyset \mid \phi, \rho & \text{(effects)}
\end{array}
$$

The backwards nature of the type judgments can be immediately see in the T-Var rule and rules for base types:

$$\text{T-Var}\ \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau \,!\, \phi} \qquad \text{T-Bool}\ \Delta; \Gamma \vdash bv : \texttt{Bool} \,!\, \phi$$

You can see that a variable can have any effect whatsoever. The idea is that if a typing derivation that is lower in the tree requires any ϕ, this rule can provide it. That is, it does not matter what regions are live in order for a variable to type-check.

Similarly, the rules for abstractions work in this reverse direction. Essentially abstractions just require that the latent effect (i.e., what regions must be live to call this function) is the effect of the expression inside the lambda (i.e., what the function body needs to be live). The effect of an abstraction as a whole just requires that the region of the abstraction itself must be live.

Application is a bit stranger. Both the function and its argument use the same live regions , since there is no way for a binder outside of this application to only provide a live region to one or the other. The latent effect must be a subset of the effect of the application so that all regions needed to run the function are actually live in the application. Region applications work mostly the same, but with a substition of region variables.

$$\text{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \,!\, \phi_\ell \qquad r \in \phi}{\Gamma \vdash \lambda x{:}\tau_1.\, e \;\texttt{at}\; r : (\tau_1 \xrightarrow{\phi_\ell} \tau_2, r) \,!\, \phi} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : (\tau_1 \xrightarrow{\phi_\ell} \tau_2, r) \,!\, \phi \qquad \Gamma \vdash e_2 : \tau_1 \,!\, \phi \qquad r \in \phi \qquad \phi_\ell \subseteq \phi}{\Gamma \vdash e_1\ e_2 : \tau \,!\, \phi}$$

$$\text{T-RAbs}\ \frac{\Gamma \vdash u : \tau \,!\, \phi_\ell \qquad r \in \phi}{\Gamma \vdash \lambda\rho.\, u \;\texttt{at}\; r : (\Pi\rho.^{\phi_\ell}\tau, r) \,!\, \phi} \qquad \text{T-RApp}\ \frac{\Gamma \vdash e : \Pi\rho.^{\phi_\ell}\tau \,!\, \phi \qquad r \in \phi \qquad \phi_\ell[\rho/r] \subseteq \phi}{\Gamma \vdash e\,[r] : \tau[\rho/r] \,!\, \phi}$$

The rule for new regions is important, as it provides evidence that a region is actually live. The idea is that all of the rules push the regions that it needs to be live down, while the T-New rule pushes the actually live regions up. When these unify our program type-checks.

$$\textsc{T-New}\ \frac{\Gamma \vdash e : \tau \,!\, \phi, \rho}{\Gamma \vdash \texttt{new}\ \rho.e : \tau \,!\, \phi}$$

The other type and effect rules are not that interesting, but we include them here for completeness.

$$\textsc{T-If}\ \frac{\Gamma \vdash e_0 : \texttt{bool} \,!\, \phi \qquad \Gamma \vdash e_1 : \tau \,!\, \phi \qquad \Gamma \vdash e_2 : \tau \,!\, \phi}{\Gamma \vdash \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau \,!\, \phi} \qquad \textsc{T-If}\ \frac{\Gamma \vdash e : \tau \,!\, \phi}{\Gamma \vdash \texttt{fix}\ x.e : \tau \,!\, \phi}$$

We mentioned region inference before. The details are too complicated to talk about now, but the essence is that we can build a judgment $\Gamma \vdash e \Rightarrow e' : \tau \,!\, \phi$ that infers a term e' that is equivalent to e except with region annotations. The goal of an algorithm that computes these judgments is to construct e' such that its allocated regions are as small as possible (if they are small enough they could be stack allocated, for example). Region inference is essential for making region-based memory management practical.

3 Summary

Type and effect systems are a natural way to extend type systems to accommodate reasoning about effects. Although the presentation is very different from FX, the idea lives on in programming languages that are still in use today such as Java or Haskell. However, making pervasive use of type and effect systems palatable for practical languages is an open research topic (a recent ECOOP 2012 paper tackles this very issue).