Type inference (Informal outline)

Wednesday, July 30, 2014 9:10 PM

This note presents a simplified, informal, overview of the type inference algorithm. More formal and specific details can be found in the other page Region Type Inference.

Overview

The inference algorithm processes a set of class-definitions. The class-definitions are processed in an order that respects the dependences between classes:

1. If class X is a derived class of Y, then X depends on Y.
2. If class X has a field of type Y, then X depends on Y.
3. If a parametric class X<T> has constraints of the form "T extends S", then X depends on S.
4. If method of class X makes use of a variable of type Y, then X depends on Y.

In the presence of mutually recursive dependences (e.g., X depends on Y and Y depends on X), the inference algorithm needs to process these classes together. To begin with, let us ignore this complication and assume that there is no mutually recursive dependence. If X depends on Y, then the type inference algorithm is first applied to Y and then applied to X. Let us now address the problem of type inference for a single class X, assuming that all classes that X depends on have already been processed.

Challenges: Interfaces and abstract methods [and, more generally, method overriding] pose a challenge to the inference. Specifically, we end up with a mutual dependence between an interface and its implementation, or a overridden method and the overriding method [since the region-constraints of the overriding method must be incorporated in the region-constraints of the overridden method]. This can be handled if we do inference for a whole program, but is difficult if we do inference for a library or incomplete program. We may have to exploit some heuristics to infer a type that is not the most-general type: e.g., restricting an interface or an abstract method (signature) to have a single region-variable. E.g., in the general case, a *List<T>* can be parameterized over two region-variables, one indicating where the list itself is allocated, and one indicating where the elements of the list are allocated. For the interface *IList<T>*, we may have to heuristically force these two regions to be the same, or analyze the interface together with an implementation to produce the more general typing.

The inference consists of two aspects:

1. Identify the set of region-variables to be introduced (and express the types of various entities in terms of these region-variables).
2. Identify constraints among these region-variables, which may cause some of these variables to be unified, and to incorporate these constraints in the inferred types.

Part 1: Introducing region-variables and elaborated-types

Let us consider problem (1).

a) The set RV(X) of region-variables for the class X is the union of the
   i) The set RV(Y) of region-variables associated with the base-class Y of X (if any), and
   ii) The set RV(f) of region-variables for every field f in class X, with the exception of any recursive field (that is, any field of type X), and
   iii) The set RV(tp) of region-variables for every type-parameter of class X (if X is a parametric class), and
   iv) One distinguished region-variable (representing the allocation-region).
b) The set RV(f) of region-variables of a (non-recursive) field f of X is obtained by introducing a new

region-variable for every region-variable in the type of the field f. Thus, if f is of type Z, then RV(f) consists of a new region-variable for every region-variable in RV(Z). Note that if the type of f is of the form Z<W>, where Z is a parametric class, then, we will need to introduce new region-variables corresponding to those in RV(W) union RV(Z). (Note that this applies when the classes Z and W are real classes, not when Z or W is a type parameter of the parametric class X. For type parameters, we don't need to introduce any new region-variables.)

c) The set RV(tp) of a bounded type-parameter tp of class X with a constraint "tp extends Z" is obtained by introducing new region-variables for every region-variable in the type Z.
d) The set RV(f) for a recursive field f of X is the same as RV(X).
e) For any method m, RV(m) is defined as the union of (fresh copies) of the
    i) region-variables RV(type(p)) of every parameter p, and
    ii) region-variables RV(type(v)) of the return value v
f) For any variable v of type T, RV(v) is obtained from the type T (just like in the case of fields).

This can be extended to deal with mutually-recursive classes. If classes X and Y are mutually recursive, then the set RV(X) and RV(Y) are defined to be the same and this set is computed by analyzing the definitions of class X and Y together as above.

We order the set of region-variables associated with any entity (class, field, variable, etc.) into a sequence, and this produces the region-parameterized types such as C<R1,R2>. As usual, the distinguished allocation-region is specified as the first element in this sequence (where appropriate). This is mostly straightforward, with the exception of (higher-order) functions, as detailed below.

Challenge: Consider a higher-order function $f(\ X \rightarrow void\ g)$. Assume that the class *X* has a single region-variable, namely its allocation-region. Thus, the type *X* is elaborated into $X\langle R\rangle$. Now, we have two choices for elaborating the type of f: (a) As type $\forall R.\ ((X\langle R\rangle \rightarrow void) \rightarrow void)$ or as (b) $(\forall R\ (X\langle R\rangle \rightarrow void)) \rightarrow void$. This choice introduces complicates the type inference algorithm. In principle, we could try each choice and see if the program type-checks under this choice, but this can introduce a potential explonential blowup in the number of choices we must consider.

Challenge: In C#, functions and closures are typically represented as objects (of various types such as Action<>, Func<>, etc.). To get the equivalent of polymorphic functions, we will need similar polymorphic classes/objects (e.g. an object of type $\forall R.\ Action\ \langle X\langle R\rangle\rangle$ can be used to represent a procedure that expects an argument of type X, allocated in any region whatsoever. The current type-system does not include such types.

Note: The above process tries to infer as general a type as possible. Hence, it introduces a new region variable wherever possible, forcing different region variables to be the same (via unification) only when necessary. However, this could potentially lead to an explosion in the number of region variables. Some ad-hoc heuristics to limit this may be useful. E.g., consider the following class: "class X { Y y; Z z; }". The general process will utilize different region variables for the container X and the objects that y and z point to. Thus, we could use a "X<R1,R2,R3>" to denote a X allocated in region R1, whose y-field points to an object allocated in region R2, and z-field points to an object allocated in region R3. If class X has hundreds of fields, this could lead to a potential explosion in the number of region-variables.

Example.

Part 2: Constraint generation and simplification

Work in progress: This section is currently incomplete.

In this phase we generate "outlives" constraints on the region variables of the form ($\rho_1 \succcurlyeq \rho_2$). Simplifying these constraints and projecting these constraints onto the relevant variables produces the

types of classes as well as their methods. As an example of the final outcome, we may determine that a region-parametric class "X<R1,R2,R3>" requires "R2 ⩾ R3". Similarly, we may determine that a procedure "f(Y<R1> y, Z<R2> z)" requires "R1 ⩾ R2". Such constraints are incorporated as part of these respective types.

The key sources of constraints are described below:

a) Any field f of any object o1 must point to an object o2 that outlives o1. This is captured as:
    i) For a class C and a field f in the class we have "AllocationRegion(f) ⩾ AllocationRegion(C)".
b) Any stack-variable v points to an object which outlives the variable itself:
    i) "AllocationRegion(v) ⩾ VariableScope(v)"
c) For any assignment statement "lhs = rhs", we have a sub-typing constraint "type(rhs) ⩾ type(lhs)". This sub-typing constraint can be transformed or simplified to a set of outlives constraints.
    i) In the simplest instance, where lhs and rhs are just Objects, for example, this reduces to "AllocationRegion(rhs) ⩾ AllocationRegion(lhs)".
d) For any method call, we have similar typing-constraints between the types of the actual parameters and the formal parameters (and, similarly, for return-values).
    i) To map this to region-variable constraints, we first need to produce a new "instance" of the method's type-signature (by creating new region-variables for every (outer-quantified) region-variable in the method's type). The sub-typing constraints will then generate constraints on these new region-variables.

In addition, there are a number of straightforward ways in which these constraints are propagated or inherited, as follows:

a) For a class C that derives from a class B, we have “AllocationRegion(C) = AllocationRegion(B)”. [The entire object is allocated in one region.]
b) A class C that derives from a class B also inherits "Constraints(B)".
c) A parametric class with a bounded type parameter T (of the form "T extends B") inherits "Constraints(B)".
d) A variable x of type C inherits "Constraints(C)".