# WHAT WE NEED

In order to well-understand which level and type of expressiveness is required by an interesting analysis of the unsafe code, I studied deeply the non-trivial cases obtained when analysing the use of the memory allocated in the stack. In particular I focused on the cases in which a localloc instruction appears but there is no stind and ldind instruction in the code of the same method, as the pointer is passed to other methods that read from and write on it. All the following examples refer to the mscorlib.dll library.

## Case study

First of all, a quite common approach is to give to a method a pointer and the dimension of the memory allocated on it (for instance Buffer.memcpy – used by methods #4652 Mutex.#ctor –, or MetadataImport.GetAssociates and Associates.AssignAssociates – used both by method #8207 CustomAttributeData.getCustomAttributeRecords). Another simpler approach is when the size of the allocated memory is known “a priori” and so a constant (like method #3010 System. Number.ParseDecimal when invoking methods StringToNumber and PackForNative): in this case we just need to trace the constant upper bound, instead of relating it to a parameter of the method.  
In many other cases the method of mscorlib.dll invokes an external method passing the pointer and the size of the allocated space (like methods #3447 MemberInfoCache.PopulateRtFields – invoking declaringTypeHandle.GetFields and PopulateRtFields – and #7420 CustomAttributeData.getCustomAttributeRecords – invoking MetadataImport.EnumCustomAttribute -> Enum -> \_Enum that is external). Finally another case is when a method stores in an object a pointer and the size of the allocated memory, and then invokes on this object a method that supposes it (for instance method #9023 DateTimeParse.TryParse call the method DateTimeRawInfo.Init that stores it and utilizes the allocated space through other methods).

## Pointers “in the middle” and arrays

Moreover note that sometimes we can provide to a method a pointer that is “in the middle” of the allocated memory (for instance method #3010 System.Number.ParseDecimal gives to NumberBuffer a pointer like ptr+12, so that is allocated for instance on \*(ptr-1)). Another well-known example is the one provided by <http://msdn2.microsoft.com/en-us/library/cx9s2sy4(VS.80).aspx>. We may rewrite it just providing a function that returns p[-1]+p[-2]. In this case we would check that the pointer passed to the function is allocated on \*(ptr-1) and \*(ptr-2).

Note that this situation is quite different from the one of the arrays, as the memory allocated on them is between the index 0 and the array.length-1; it is not possible to move the array inside the memory with the arithmetical operator, like array+5. Indeed unsafe pointers require sometimes a check on the lower bound.

## Heap and stack

Even if all these ideas are come analyzing the accesses to the stack memory, it appears clear that such a language may be used also for the analysis of the heap accesses.

First of all, note that when a function receives a pointer it does not distinguish between pointer to the stack or to the heap. Moreover all the considerations just proposed may be applied to the heap: a pointer may be “in the middle”, we know the dimension of the allocated memory (and its allocation is quite similar to the one of the stack, just using an external method provided by the operating system and not the localloc instruction), etc etc...

The main difference is that the heap memory once is allocated it may be preserved (just using the fixed keyword) and used by different objects, and outside the analyzed method (while the allocated stack once the method is executed it is unallocated).

The stack analysis may be just a first step in order to successively extend the analysis and the language to the heap analysis. Moreover another interesting analysis may be to check which area of the memory are read and written, in order to check that all the reads through a pointer are performed on an area of the memory previously written.

# A FIRST SKETCH OF THE LANGUAGE

The constraints on the pre and post conditions of the methods may be written in the following way:

Ptr.spaceAllocated = [espr\_left .. espr\_right]

espr :== ArithmeticalExpression  
 | Constant  
 | Variable

where variable may be a parameter of the method or a field of the object containing the method.  
Note that it is just a first sketch of the language, intended to explain what we need. Another approach to define the same things may be something like: ptr.spaceAllocated < espr\_right && ptr.spaceAllocated>espr\_left , splitting the constraints on the upper and the lower bound of the allocated space (note that the < and > operator does not mean the usual numerical comparison!).