Towards decoupling nullability semantics from indirect access in pointer use *

Richárd Szalay[a]

[a]Eötvös Loránd University, Faculty of Informatics,
Department of Programming Languages and Compilers,
Pázmány Péter stny. 1/C., 1117 Budapest, Hungary,
szalayrichard@inf.elte.hu

Abstract

The special "null-pointer" (C, C++, Ada) or "null-reference" (Java, C#, Python, ...) value for a pointer-like type is often used to indicate the lack of a meaningful result/data. Accessing a non-existing value is an undefined operation, resulting in either unpredictable behaviour of the program or the raising of an exception. The usage of pointers often leads to a defensive design: it is expected of the programmer to preemptively guard against the nullness of a pointer, or handle the resulting exception. Together with a code organisation principle to prefer "early returns", this defensive mechanism may result in variables in the local scope polluting the list of available symbols. These variables' existence do not pose a performance overhead at run-time as virtually all compilers optimise the variable away by caching the loaded value. However, during code comprehension, these symbols remain visible, suggested by code completion tools which hinder understanding. Some programming languages offer "conditional dereference" operations: in C#, the `?.` operator propagates a null reference; in Haskell, the `Maybe` monad allows expressing such semantics. Modern C++ versions support expressing Maybe-like values with the `optional<T>` class template, but it encapsulates the values, not the indirect access. Adaptation of new language features or changing user-facing API is often met with business or technical challenges and is thus a slow process. In this paper, we discuss our investigation of the usage of pointer-like types (including iterators) for nullability semantics, not only for indirect access. We devised an automated analysis tool that marks potentially redundant pointer variables, lowering the number of visible local symbols. A post-refactoring view can show the landscape of the program where descent in complex data structures (such as configuration maps) can be expressed in a more concise manner.

Keywords: C++ programming language, encapsulation, memory model, pointers, software design

*This work presented in this paper was supported by the European Union, co-financed by the European Social Fund in project `EFOP-3.6.3-VEKOP-16-2017-00002`.