# Automatic Generation of C Library Bindings

Inferring Nullability through Structure Fields

Jaro S. Reinders, 1008847
Supervised by Jurgen J. Vinju and Tom Verhoeff

October 10, 2021

**Abstract**

Programming languages have been and are still evolving rapidly. Like natural languages, it is not clear which language is better. However, over time, common patterns from old languages have appeared as more specific abstractions in new languages.

A problem that many new languages face is a lack of existing libraries. To overcome that problem, many new languages have a mechanism for calling functions from other languages. However, this still means that we either lose the advantage of programming in a higher-level language, or it requires manual labour to write a higher-level interface to external functions.

In this work, we build upon earlier research which introduces a suite of program analysis tools to reduce the labour required for making such a higher-level interface. The existing suite infers information about error messages, pointer semantics, and memory management mechanisms, from programs written in the popular C programming language.

A limitation of the existing suite is that it cannot infer information about pointers which are stored in structures. We discuss the challenges of removing this limitation and propose an improvement to the inference algorithm, such that it can make use of that information. We evaluate our improvements and show that it can improve the inference results on a real world library.

# Contents

# Preface

In the fifth year of my studies at Eindhoven University of Technology and the second year of my master programme, it is now finally time to conclude with this master thesis. Since before starting my bachelor, even, I have been interested in functional programming. During my master that interest has expanded to programming languages in general. With this master thesis I hope to take one of my first steps in contributing to the literature.

While working on hobby projects written in the functional programming language Haskell, I have run into issues with missing or unmaintained bindings for C libraries, especially for graphical user interfaces. This has been a source of personal motivation to start researching solutions to this problem.

This master project has been preceded by an exploratory project for the seminar of the Software Engineering and Technology group. The most promising existing technology in this area is the analysis suite developed by Tristan Ravitch in his PhD dissertation. It was sad to see that his tools have not seen much use in practice and required extensive maintenance.

From the reader I expect a basic knowledge of C or a C-like programming language, especially experience with pointers is useful. For the technical details, it is also very useful to have some experience with data-flow analysis. The main audience of the analysis suite are software developers who create and maintain software in higher-level languages which needs to interact with software written in C. The theoretical side of these analyses might also be interesting to programming language researchers.

Finally, I would like to thank my mother, father, and brother, for accommodating me while working on this project, their non-technical advice, and relaxation in my free time. And I would like to thank my supervisors Jurgen Vinju and Tom Verhoeff for their advice and guidance on the program analysis techniques, on the evaluation, on writing this thesis, and on navigating the bureaucratic landscape of the university, even in the face of troubling personal circumstances and busy schedules.

# Chapter 1

# Introduction

This is my master thesis about interface glue inference through structure fields. The main purpose of this inference is to guide the automatic generation of library bindings from low-level C libraries to high-level libraries in languages such as Python and Haskell. During my master project we continue the work of Tristan Ravitch on Inferred Interface Glue [8]. First, we will motivate this area of research.

## 1.1 Motivation and Context

Since the inception of the first higher-level programming languages in the 1950s there have always been many different programming languages. There are plenty of valid reasons for this: they were improvements of previous languages, simply have different purposes, or are suited to preferences of certain groups of developers. There is no reason to believe that this proliferation of languages will converge to a single "optimal" language.

However, there has already been an enormous amount of work spent on programs written in current programming languages. As time progresses and even more work is spent on programs written in these languages, it becomes harder and harder for new languages to compete in terms of the amount of readily available libraries.

The most common solution to this problem is to allow programs in new languages to use functionality from programs written in old languages. Instead of writing a new program from scratch, we define a new function in our new language which outsources all its work to functions from the old language. We call the connecting of functions from one language to another *binding* and a set of functions that are connected *bindings*. For that purpose, many programming languages include a way to indicate that a function outsources all its work to another programming language, which is usually called a foreign function interface.

However, only indicating that a function outsources all its work is often not enough, the input and output types of these functions might also need to be converted. Especially if

you as an author bindings want to integrate them seemlessly into the new language, you will need to translate concepts from the other language to your new language.

This process can require significant effort and it is error-prone, because these foreign functions are often not checked by the compiler. Additionally, the functions to which you create bindings might still change or new functions might be added in the future. So, this can require a continuous maintenance effort.

In our personal experience we have encountered missing or poorly maintained bindings while using the programming language Haskell. Haskell is not a mainstream programming language and therefore does not have the privilege of being targeted as a first class language for many cross-language library projects. Most notably are the GUI libraries which often do target more mainstream languages such as C, C++ and Python.

As a solution to this problem, Ravitch [8] proposes to use program analysis techniques to recover much of the necessary information to write seemless bindings to libraries written in the popular C programming language. Binding authors can use Ravitch's tools to hook into the compilation process of a C library and automatically produce annotations, which they can then use with another one of Ravitch's tools automatically produce a Python library containing bindings to the C library which convert concepts like error-handling, pointer semantics, and memory management to their Python equivalents. The tools are not perfect, so authors can inspect the inferred annotations in an interactive interface to identify mistakes or omissions. If the problem has been found, the bindings author can manually overwrite annotations and re-run the tools.

Our work builds upon these tools developed by Ravitch to improve the annotations they produce. We focus on one of the future work suggestions from Ravitch's dissertation: inference through structure fields.

## 1.2 Related Work

In this section, we discuss work related to making it easier for people to write bindings to libraries from other languages.

One of the oldest and most popular tools that aid in the generation of library bindings is called SWIG [1]. SWIG allows users to write language-independent interface descriptions of C and C++ libraries. These descriptions can be used to generate bindings for these libraries for a wide range of languages. Language support can be added to SWIG through an extensible module system.

An extension of SWIG is called AutoWIG [4], which is a tool that can automatically generate bindings for Python from C++ code. They automate the process of writing bindings with SWIG. The generation of wrappers and interfaces is automated even further by parsing the C++ source code and mapping semantic elements from C++ to Python. This results in relatively idiomatic bindings in many cases. This approach works better when the C++ library uses high-level C++ features such as smart pointers and the standard

library. AutoWIG is also able to convert Doxygen documentation to Sphynx documentation. And another important feature is interactivity. AutoWIG itself is a Python module which can be used interactively and it is also very customizable.

The GTK GUI project [11] has developed several C libraries for writing programs with graphical interfaces. They make it easier to write bindings for their libraries by providing additional information, called GI [5], about their API that can be used by library writers in other languages to automate the generation of bindings to their libraries. The "haskell-gi" [12] project, for example, uses GI information to automatically generates bindings to the GTK libraries in Haskell.

This approach still has three areas that can potentially be improved: it is specific to the GTK project, the GI information itself still needs to be written by hand, and the generated bindings are relatively low-level and not idiomatic. An example of the first two limitations already comes up inside the GTK project itself. A sub-library of GTK, called Cairo, does not have GI information that can be used to automatically generate bindings. The third limitation is exemplified by the "gi-gtk-hs" and "gi-gtk-declarative" libraries which both offer a higher-level and more idiomatic wrapper around the low-level "gi-gtk" bindings that were generated with "haskell-gi".

## 1.3 Inferred Interface Glue

This section introduces the main related work: the dissertation of Ravitch called "Inferred Interface Glue" [8]. This thesis builds heavily on that work, so it is important to know more about it.

Here, we explain the structure of the existing program analysis suite using three different views. The first view, Section 1.3.1, shows the process of going from source code to generated bindings. The second view, Section 1.3.2, considers the packages that comprise the analysis suite and strongly related packages. The third view, Section 1.3.3, shows which individual analyses exist in the suite and how they depend on each other.

### 1.3.1 The Binding Generation Process

We start by describing a high-level overview of the steps required to generate bindings from C source code, as shown in Figure 1.1. The analysis suite starts from the source code of a program. It hooks into a normal make workflow for compiling C libraries using a wrapper around the LLVM-based C compiler clang called WLLVM. The WLLVM tool generates a shared library file which contains LLVM bitcode, which can be extracted using the extract-bc tool. The extract-bc tool generates one big file containing all LLVM bitcode of the entire library. The iiglue tool reads in this bitcode and runs program analyses on the LLVM IR. When the analyses are finished, the iiglue tool produces a JSON file containing inferred annotations for the functions and arguments in the library. Optionally, the iiglue tool can also produce an HTML report as a more human-readable presentation of the

inferred annotations. It also includes and interactive source code viewer which can show where the inferred information originated from. Finally, these JSON annotations can be used by other tools to generate bindings in a higher-level language. The included iigen tool generates bindings for the programming language Python.

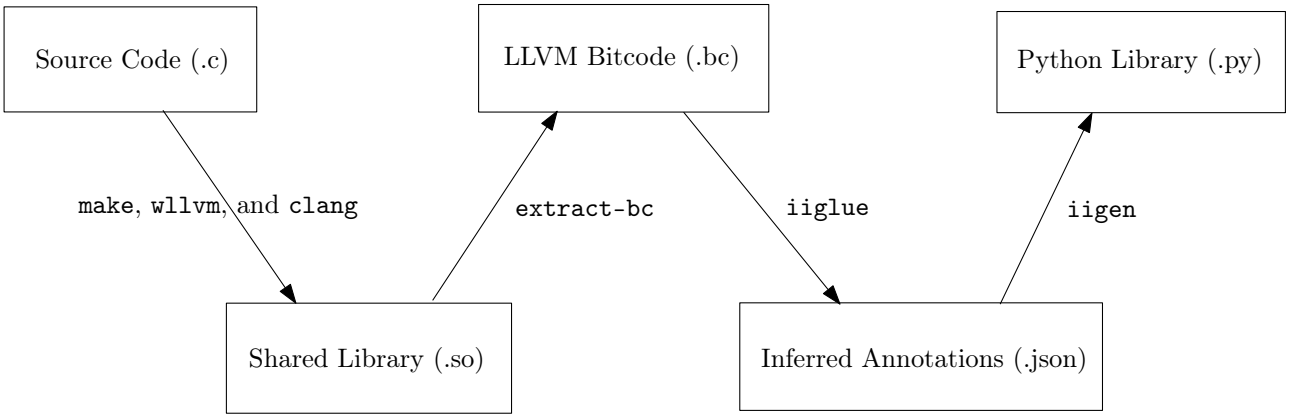


Figure 1.1: The binding generation process.

### 1.3.2 The Packages Involved

We continue with a slightly more detailed view that shows the Haskell packages involved in the analysis suite. There are three main packages, two packages for interacting with the LLVM IR, and three utility packages.

The main executable can be found in the iiglue package, which wraps all the analyses up into one convenient executable. The iiglue package also contains several other executables including iigen for generating Python libraries from the inferred information, but we will not go into detail about those executables. The iiglue package depends on the 'foreign-inference' package which implements the main analyses. These analyses include the error code analysis, pointer semantics analyses, and memory usage analyses which we will discuss in more detail later. The 'foreign-inference' package, in turn, depends on the 'llvm-analysis' package which implements basic analysis on the LLVM IR. Examples of these basic analyses are: constructing a control flow graph, constructing a call graph, a framework for data flow analyses, a points-to analysis, and an access path analysis.

All of these packages depend on a way to interact with the LLVM IR. This capability is split over two packages. The general interface is provided by the 'llvm-base-types' package, which defines Haskell data types that describe the LLVM IR. In these Haskell data types, the indirect references present in the LLVM IR are resolved into direct references, so no further lookups are necessary when dealing with these types. The other package, 'llvm-data-interop', is a binding to the original LLVM C++ library. It is used to read out the LLVM bitcode files and convert it into the types defined in 'llvm-base-types'.

Finally, there are three notable packages which provide more general utility functionality,

but were still developed in the context of this analysis suite. A package called 'hbgl-experimental' is used to represent and interact with graphs. Graphs are used in the 'llvm-analysis' package, for example for the call graph. The second utility package is 'ifscs', which is a set inclusion constraint solver. This constraint solver is used in problems which can be expressed as graph reachability. The third utility package is 'archive-inspection' which provides a uniform interface for interacting with compressed archives.

### 1.3.3 The Analysis Pipeline

We zoom further in to the 'foreign-inference' package and consider the analysis pipeline that it contains, which is shown in Figure 1.2. The input is the LLVM IR defined in the 'llvm-base-types' package and information derived from that using the 'llvm-analysis' package such as the control flow graph. The result of the analysis pipeline is a collection of annotations that can be used to generate higher-level bindings. The first step in the pipeline is a separate analysis to uncover returned error codes and error handling information. After that there are two groups of analyses, in order they are: analyses which infer information about the semantics of pointers, and analyses which infer information about memory usage. The inferred information from all of these analyses is collected into the output of the pipeline.

In the C language there is no built-in exception mechanism, so error codes have to be managed manually. A common way to do this is by returning a dedicated error value from functions that may fail. Usually, an integer is used where 0 indicates success and -1 indicates failure, other values can be used to indicate different types of failure. However, this technique can only be used on functions that would otherwise not return anything. Another common case are functions that normally return non-negative integers, they can return negative integers to indicate failure. If this behaviour is not properly documented, then it can only be recovered by looking at the implementation of the library.

Another powerful feature of C are its pointers. Pointers are references to memory locations that you can read from and write to. They are very powerful and flexible, but it is also very easy to misuse them. For this reason, many high-level languages abstract over common use cases of pointers. Ravitch identifies three more nuanced semantic properties that pointers can have: pointers can represent arrays, pointers can be used as output variables and pointers can be nullable or non-nullable. The analysis suite contains analyses which infer information about the semantic properties of pointers from how they are used in a library.

The final set of analyses concern how memory is used. The C language has no built-in automatic memory management system. The programmer has to allocate and deallocate memory manually. If a function deallocates one of its arguments then it needs to be the owner of that allocated value. An allocated value cannot be passed to an owning function and, after that, to another function because that can cause use-after-free or double-free bugs. It is also important that an allocated value always gets passed to at least one owning function to ensure that it is deallocated eventually and does not cause a memory

leak. Most higher-level languages have automatic memory management, so care must be take to make sure that the automatic memory management of the language in which the bindings are written does not conflict with the manual management of the C functions. The analysis suite includes analyses to recover this information.

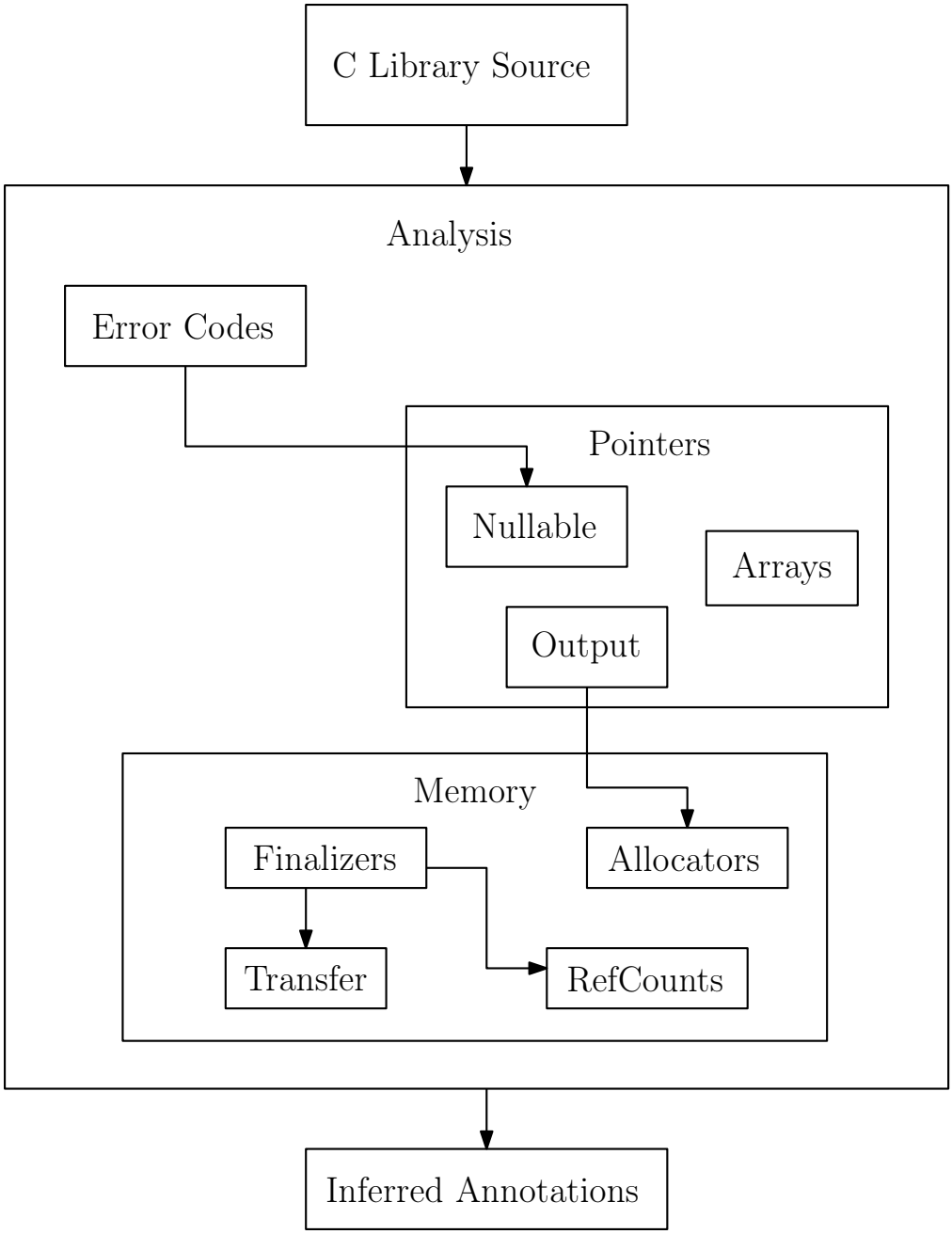


Figure 1.2: Ravitch's analysis pipeline. This is a slightly modified version of Figure 1.1 in Ravitch's dissertation.

## 1.4 Research Question

In this thesis we will consider the following research question:

> How can the analysis suite be improved to infer more annotations?

## 1.5 Research Method

In this section we will look at the methods that we use to find an answer to our research questions. We consider the general approach to research in this thesis and the method for evaluating the results.

Our research question is open-ended, we do not have a clear existing method for answering this question. We must first consider the context and identify when the existing algorithm fails to produce annotations. This calls for a bottom-up or inductive research approach where we first observe patterns in practice, then connect them to the existing theory, and finally use that theory to improve the analysis algorithm. In practice, this process is also iterative, so after one such cycle it might turn out that the result is not yet as desired, so we restart the cycle with our new information. Of course, in this thesis we will present a cherry-picked version of events.

This inductive approach already involves observing patterns in real-world examples, so it is straightforward to combine it with a qualitative analysis of the results of our improvements on those same examples. So, we use qualitative analysis to evaluate our answer to our research question. That means that we manually inspect the results and describe their qualities subjectively. Such a subjective description has the disadvantage that it is less precise and it often involves a larger amount of manual labour when compared with a quantitative analysis, which means that our sample size is small.

This is as opposed to quantitative analysis which would mean to run many examples through an often automated process which produces objective numbers in the end. We believe it is important to perform a quantitative analysis to know exactly how useful our work is, however, a fully automated approach still requires some more work and we first want to focus on exploring the possible options and determining if this improvement is feasible at all. So, we leave a full quantitative analysis as future work.

## 1.6 Contributions and Roadmap

In this section, we list our contributions and deliverables and we present a roadmap of the structure of this thesis.

Our contributions are the following:

- We have updated the Ravitch's analysis suite to work with more recent versions of GHC and LLVM.

- We have designed improvements to the nullability inference to account for pointers in structure fields.
- We have implemented experimental support for nullability inference through structure fields.
- We have evaluated the new inference algorithm on GNU Libmicrohttpd 0.9.73.

We have produced the following deliverables:

- The updated software bundled up in 'iiglue-bundle' repository [9], including the packages: 'llvm-analysis', 'llvm-pretty-referential' (new), 'foreign-inference', and 'iiglue'.
- Upstream fixes and additions: a fix for a small critical issue in 'ifscs', changes for visibility and linkage for 'llvm-pretty' and 'llvm-pretty-bc-parser', and a fix for a compilation error in 'hgbl-experimental'.
- This thesis, where we describe the existing tools, present our improvements, and evaluate them on a practical library.
- Our experimental input: 'libmicrohttpd' version 0.9.73, included in the 'iiglue-bundle' repository.

This thesis is structured as follows:

- First, we have focused on inference through structure fields, specifically nullability inference. On this subject, we explain what it means for a pointer to be nullable and the existing nullability analysis of Ravitch in Chapter 2.1. Additionally, we describe background knowledge on structure fields and their semantics in Chapter 2.2.
- Then, we propose an improvement of the inference algorithm to account for structures in Chapter 3.
- Afterwards, we evaluate this improvement by using them to infer properties of a real C library and we discuss the improvements in Chapter 4.
- Finally, we conclude in Chapter 5
- In Appendix A, we discuss our experience renovating Ravitch's tool suite.

# Chapter 2

# Background Knowledge and Problem Analysis

In this chapter we present required background knowledge and we analyse the problem that we address in this thesis. We explain the Ravitch's existing nullability inference algorithm in Section 2.1. We describe and show examples of structures in C and LLVM in Section 2.2. Finally, we analyse the problem in Section 2.3.

## 2.1 Nullability Inference

In this section, we describe nullability analysis, which is an analysis that recovers higher-level semantics from low-level pointers in C. We first specify what it means for a pointer to be nullable in C. Then we describe the nullability inference algorithm developed by Ravitch in his dissertation.

### 2.1.1 Nullability in C

In general pointers in C are just numbers with extra operations for accessing memory at the location indicated by that number. However, not all numbers are valid memory locations. The most well known and most used invalid memory location is called the null pointer, which is a reserved constant, usually zero, that is guaranteed never to be valid. The main use of null pointers is to indicate the absence of a value. For example a function might accept a null pointer for optional arguments, which are checked and replaced by a default value if the argument is the null pointer.

Unfortunately, every pointer type in C contains a null pointer constant, which means that all pointer arguments are theoretically optional and should be checked before they are used. Practically, it is really easy to omit the checks and it is common practice to implicitly or through documentation add the assumption that certain pointer arguments are not nullable. Some more modern languages like Haskell, Rust and ML-derived languages

allow the programmer to specify whether a certain value is nullable or not explicitly, with `Many` and `Option` respectively Even languages that do not have such a feature can still benefit from run time checks to ensure that no null pointers are passed to functions that expect arguments that are not nullable. So, to be able to produce high-quality bindings we should recover the implicit nullability information from C functions.

### 2.1.2 The Inference Algorithm

The work of Ravitch, which we build upon in this thesis, includes a nullability inference algorithm, which we will describe here.

#### Nullability, Specifically

In general, whether a pointer is nullable or not is decided by the intent of the author of the library. If the nullability is not specified explicitly, such as in C, then we need a more specific definition of nullability, such that we can consistently infer the nullability of a pointer from the behaviour of the library. However, any more specific definition is likely to be subjective and conflict with the intent of at least some library authors.

Therefore, Ravitch instead considers what users of the generated bindings want, which is to be able to access all useful functionality of a library. He proposes that a pointer argument is not nullable if an *undesirable event* occurs when that argument is a null pointer. The undesirable events are:

- dereferencing a null pointer, which is usually the argument in question
- never returning, e.g. calling the `abort` or `exit` functions, which stops the execution of the program
- returning an error code, which has been inferred during an earlier analysis

If one of these undesirable events occur, then the function does not do useful work and therefore the argument can be inferred to be non-nullable.

An important restriction is that an argument is only considered non-nullable if the undesired event *must* happen. If there is even a tiny chance that other arguments or side-effects cause the undesired event not to happen then the argument is still considered nullable. The reason behind this is that we do not want to make this potentially useful functionality inaccessible in the bindings that we eventually generate from our inference results.

#### Input

As we have seen in Section 1.3.1, Ravitch's actual algorithm does not run directly on C source code. Instead, the source code is first compiled into LLVM IR. The algorithm is written in Haskell, so this IR is converted to Haskell data types. Finally, these data types are transformed to algebraic data types with cycles and sharing in which all explicit

references of the LLVM representation are resolved and each instruction is labeled with a unique number, we will call this the *referential representation*.

The algorithm also requires the call graph, which shows the dependencies between functions, and the control-flow graph, which shows the dependencies between intstructions within a function. These graphs are constructed using auxiliary analyses.

### Data-Flow Analysis

To determine which pointer arguments cause undesirable events when they are null, Ravitch uses a forward data-flow analysis [6], over the referential representation. This analysis traverses strongly connected components of the call graph in topological order and analyses all functions that they contain, remembering results of functions that have been analysed. This analysis tracks data-flow facts before, *pre*, and after, *post*, each instruction of a function. Over the course of this algorithm, these data-flow facts become better and better (over-)approximations of the correct solution.

In general, the data-flow facts should be a lattice with a top element $\top$ and a meet operation $\sqcap$. In this case, the lattice is the powerset of the set of all pointer arguments of the function under analysis. The top element of that lattice is the complete set, and the meet operator is set intersection.

The *pre* of the first instruction in the function is initialised to the empty set because, at that point, no arguments are known to cause undesirable events. The rest of the data-flow facts are initialised to the top element of the lattice, so they initially contain all arguments to the function under analysis.

For each instruction, labeled $\ell$, in the function, this analysis applies the corresponding transfer function to $pre_\ell$ and stores the result in its $post_\ell$. Then, for each successor instruction $k$ according to the control-flow graph, we set $pre_k \leftarrow post_\ell \sqcap pre_k$. This process is repeated until the (maximal) fixed point is reached.

When the fixed point is reached we will know that our solution satisfies the following equations:

$$\langle \forall \ell :: post_\ell = f_\ell(pre_\ell) \rangle$$
$$\langle \forall \ell, k : k \text{ is a control-flow successor of } \ell : pre_k \subseteq post_\ell \rangle$$

Where $f_\ell$ is the transfer function for instruction $\ell$.

For each function, the result of this analysis is the *post* of the last instruction in that function. After the fixed point has been reached, that data-flow fact will be taken as the set of all non-nullable arguments of that function.

### The Transfer Functions

The most important part of this analysis are the transfer functions. A transfer function specifies which information can be inferred for each instruction in the program.

For this nullability analysis, there are three relevant instructions: **load**, **store**, and **call**. The transfer function, called simply $f_\ell$, which is parameterised by the unique number of each instruction, is shown in Figure 2.1. There are four groups of transfer functions, the three that are associated with the **load**, **store**, and **call** instructions are shown in the Figure and the transfer functions in the remaining group for all other instructions are simply identity functions.

$$
\begin{aligned}
&[\textbf{load } ptr]_\ell : && f_\ell(x) = x \cup \text{deref}(ptr) \\
&[\textbf{store } val\ ptr]_\ell : && f_\ell(x) = x \cup \text{deref}(ptr) \\
&[\textbf{call } calledFunc(x_1, x_2, \ldots, x_n)]_\ell : \\
&\quad f_\ell(x) = x \\
&\qquad \cup \begin{cases} \top, & \text{if noRet}(calledFunc). \\ \bigcup\{\text{deref}(x_i) \mid i \in \{1, \ldots, n\}, \text{notNullable}(calledFunc, i)\}, & \text{otherwise.} \end{cases} \\
&\qquad \cup \begin{cases} \text{deref}(calledFunc), & \text{if indirect}(calledFunc). \\ \emptyset, & \text{otherwise.} \end{cases}
\end{aligned}
$$

Figure 2.1: Transfer function for nullability analysis.

In this transfer function, we are using a few functions that we have not yet defined. The 'noRet' predicate determines whether the called function does not return based on results from a previous analysis. Examples of functions that do not return are `exit` and `abort`, which stop the current program from running, or functions like `longjmp` which replace the normal program flow. Encountering such a function is considered undesirable behaviour and therefore the transfer function returns the top element of the lattice, which means that all arguments to the current function are considered to be non-nullable at that instruction.

The 'notNullable$(f, i)$' predicate determines if the $i$th argument of function $f$ is nullable based on earlier results from this nullability analysis. Usually, this analysis follows the call-graph, so this information should be available, but in the case of mutual recursion it might be necessary to iterate the analysis a few times to reach a fixed point.

The 'indirect' predicate determines if the called function is an indirect call. A call is indirect if the callee is a function pointer and if the function that it points to cannot be determined at compile-time. This function pointer could itself be an argument to the function that we are analysing, so in this case we consider that pointer to be dereferenced.

Finally, we need to define what it means for a value to be dereferenced. The input of 'deref' is a value of type pointer, and the output is a possibly empty set of non-nullable arguments. The 'deref' function inspect its argument following the resolved references in our referential representation, if it reaches an argument of the current function then it

returns the singleton set of that argument, otherwise it returns an empty set. Along the way there can be a few different obstacles.

When we have established that a value is dereferenced, the first thing we do is try to find the base location that is accessed. We search backwards through the resolved references and ignore bitcasts. At this point there are several possibilities:

- If we run into a global variables or local fresh allocation then we do nothing, because both of these are always safe to dereference.
- If we run into a `load` instruction then that means we are dereferencing a pointer which itself was stored in memory, so we have to give up and do nothing. It should be noted, however, that we do run optimisations beforehand which try to avoid memory accesses if they are statically known, so if these are still left in the program then it must mean they are difficult or impossible to resolve statically.
- If we run into a `getelementptr` instruction then that means the program dereferences a base pointer with a certain offset. We do not care about the specific offset and continue searching for the base location of this base pointer.
- Otherwise we strip any remaining bitcasts off and continue with the resulting value.

The next step is to deal with possible control dependencies. The first possible control dependency is introduced by the `select` instruction. This instruction represents an inline choice between two values based on a predicate, similar to the ternary operator in C: `cond ? iftrue : iffalse`. If we encounter such a `select` instruction then we have to give up and do nothing. Again, we assume that previous optimisation would eliminate this instruction if it was possible to know statically which value is selected.

The other instruction which introduces a control dependency is the `phi` instruction. This instruction chooses a value from multiple choices based which control flow is taken to reach this instruction. In general it is also not possible to know which value will actually be chosen at run-time, but there is one situation in which we can that one value will be chosen at least once. That is, if all but one of the predecessors require the control flow to first pass at least once through the phi node and if there is only one unconditional jump into this loop. Then we know that the value that is associated with that unconditional jump will be taken at least once.

```
int f(int *x) {
  int n = 0;
  do {
    n += *x;
  } while (n < 10);
  return n;
}
```

Figure 2.2: Do-while loop example in C.

A common example of this is a do-while loop in C as shown in Figure 2.2. This loop gets compiled to the LLVM IR shown in Figure 2.3.

```
define i32 @f(i32* %0) {
  %2 = load i32, i32* %0, align 4
  br label %3

; <label>:3:                                      ; preds = %3, %1
  %4 = phi i32 [ 0, %1 ], [ %5, %3 ]
  %5 = add i32 %2, %4
  %6 = icmp slt i32 %5, 10
  br i1 %6, label %3, label %7

; <label>:7:                                      ; preds = %3
  ret i32 %5
}
```

Figure 2.3: Do-while loop example in LLVM IR.

As you can see there is a phi node under label 3 with two predecessors, but one of the predecessors is a branch from label 3 itself and the other predecessor is an unconditional branch. So, in every execution of the function `f` we know that the identifier `%4` takes on the value `0` at least once.

#### Witnesses

In Section 1.3.1 we mentioned that the iiglue tool, which contains this analysis, is able to produce an interactive code viewer which shows where the annotations originate from. This knowledge about the provenance of the annotations is tracked along with the data-flow facts. When the 'deref' function identifies a new argument as non-nullable, then it inserts the argument as key and a singleton set containing the current instruction, called a *witness*, along with a string simply containing "deref" as value into a map to keep track of the provenance of this information. The transfer functions could also add arguments in the case of a **call** instruction which does not return, then it adds a witness with the text "arg/noret". The meet operation is an intersection of the keys in the maps, but if a key occurs in both maps then the sets of witnesses are combined using a set union operation.

These witnesses are only used in the interactive code viewer. They are not stored in the JSON file with annotation, which is used to generate bindings. Because of this and to keep things simple, we also will not address witnesses in the rest of this thesis.

### 2.1.3 An Example

To make the steps described above more concrete, we now consider an artificial example. The example that we will discuss is shown in Figure 2.4. This has been constructed to

include: a function call, control flow, and multiple nullable and non-nullable arguments. We will now show how nullability information is inferred from this code.

```
long * f(long *x) {
  return x + *x;
}

void g(long *x, long *y, long *z) {
  if (x == NULL) {
    x = f(y);
  } else {
    *y = *x;
    *z += 1;
  }
}
```

Figure 2.4: The example that we will use to show how nullability inference works.

The first step of the analysis suite is to convert the C source code into LLVM IR. For this example, the resulting IR is shown in Figure 2.5. This IR is modified slightly to make it easier to read and to make it more suitable for showing the analysis process.

Before starting the nullability inference algorithm, the analysis suite will first perform some other program analyses. At this point, the most important is the call graph analysis. This analysis determines which functions depend on which other functions. In this case it is clear that `f` has no dependencies and `g` depends only on `f`. From this graph a linear order is constructed using a topological sort, in this case the result is simply: `f` then `g`.

The nullability inference algorithm is run separately on all functions according to this topological ordering, so we start by analysing the function `f`. The function `f` consists of three instructions. We start with the first instruction and our data-flow fact is the empty set, so at this instruction none of the arguments are inferred to be non-nullable.

The first instruction, `load i64, i64* %0`, immediately matches the first case of the transfer function as shown in Figure 2.1. So, the resulting data-flow fact after the first instruction is $\emptyset \cup \text{deref}(\%0)$. In this case, $\text{deref}(\%0)$ is equal to `%0` because `%0` is an argument of the current function. So, our data-flow fact is now $\{\%0\}$ indicating that the first argument of `f` is inferred to be non-nullable at this instruction.

The remaining two instructions are not relevant to the nullability analysis and do not have special control flow apart from that the control flow of `f` ends at the last instruction. This means that the data-flow fact is simply copied and the result of the analysis of `f` is $\{\%0\}$, which means that the first, and only, argument of `f` is non-nullable.

```
define i64* @f(i64* 0%) {
  %2 = load i64, i64* %0
  %3 = getelementptr i64, i64* %0, i64 %2
  ret i64* %3
}

define void @g(i64* %0, i64* %1, i64* %2) {
  %4 = icmp eq i64* %0, null
  br i1 %4, label %5, label %7

; <label>:5:                                              ; preds = %3
  %6 = call i64* @f(i64* %1)
  br label %11

; <label>:7:                                              ; preds = %3
  %8 = load i64, i64* %0
  store i64 %8, i64* %1
  %9 = load i64, i64* %2
  %10 = add i64 %9, 1
  store i64 %10, i64* %2
  br label %11

; <label>:11:                                             ; preds = %7, %5
  ret void
}
```

Figure 2.5: The LLVM IR produced from the code in Figure 2.4.

We continue along the call-graph with the only remaining function: `g`. This function is more complicated: it consists of three basic blocks which are blocks of contiguous instructions that are not interrupted by control flow. The first block is a comparison and a conditional branch based on that comparison. Then there are two possible blocks which join up at the return instruction at the end. A graph representation of this is shown in Figure 2.6.

Again the inference starts with an empty set. The comparison `icmp` and the branch `br` intructions are not relevant, so in the first block nothing of interest happens. For the branching, we propagate the information gathered up to now, in this case the empty set, to both branches, however, now we need to decide in which order we traverse block 5 and block 7. An important property of data-flow analyses like this is that it does not matter which order you choose to traverse the control-flow graph. As long as you keep propagating information to all successor blocks when you infer new information about a block, then you will usually reach the unique solution. We will choose a depth-first traversal which starts with basic block 5, then continues with block 11, then returns to

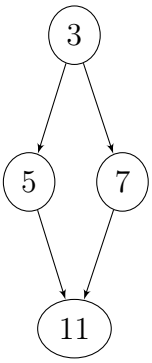


Figure 2.6: Control-flow graph of the function `g` from Figure 2.5. The numbers are the labels of the basic blocks.

block 7, and finally visits block 11 one more time.

The basic block with label 5 starts with a `call` instruction, calling the function `f`. This instruction is relevant: we can see from our transfer function in Figure 2.1 that we union our current set, in this case the empty set, with two other sets. The function `f` does return and it has one argument which in this case has the value `%1`, so we take the set $\{\text{deref}(\texttt{\%1}) \mid \text{notNullable}(f, 1)\}$. From our previous analysis of the function `f` we know that the first argument is indeed non-nullable, so we can evaluate this set further to $\{\text{deref}(\texttt{\%1})\}$. And in this case `%1` is one of the arguments of `g`, so we end up with the set $\{\texttt{\%1}\}$.

We propagate this information to basic block 11. This block only contains an irrelevant return instruction, so now the result of the inference for the function `g` is the set $\{\texttt{\%1}\}$, but we still need to visit block 7.

In block 7 we start again from the empty set that we got from block 3. Here, there are two `load` instructions and two `store` instructions which are relevant. The first load instruction dereferences `%0` which is added to the current set because it is an argument. Next is a store instruction, which dereferences `%1` and that is also an argument, so it is added to the set. Then we encounter another load instruction, which dereferences `%2` and that is also an argument, so at this point in the program all the arguments of `g` are in the set of non-nullable arguments. The next instruction is an irrelevant `add` instruction, which we skip. Following is a relevant store instruction, which dereferences the argument `%2`, but that is already in the set, so nothing changes. The last instruction is an unconditional branch to block 11.

Finally, in block 11, we combine the previous result $\{\texttt{\%1}\}$ with this new result $\{\texttt{\%0}, \texttt{\%1}, \texttt{\%2}\}$ using the meet operator. In this case, the meet operator is just set intersection, so we end up with the set $\{\texttt{\%1}\}$ as the final result of the inference for the function `g`. The fully annotated LLVM IR is shown in Figure 2.7.

```
define void @g(i64* %0, i64* %1, i64* %2) {
  ; {}
  %4 = icmp eq i64* %0, null
  ; {}
  br i1 %4, label %5, label %7
  ; {}

; <label>:5:                                      ; preds = %3
  ; {}
  %6 = call i64* @f(i64* %1)
  ; {%1}
  br label %11
  ; {%1}

; <label>:7:                                      ; preds = %3
  ; {}
  %8 = load i64, i64* %0
  ; {%0}
  store i64 %8, i64* %1
  ; {%0, %1}
  %9 = load i64, i64* %2
  ; {%0, %1, %2}
  %10 = add i64 %9, 1
  ; {%0, %1, %2}
  store i64 %10, i64* %2
  ; {%0, %1, %2}
  br label %11
  ; {%0, %1, %2}

; <label>:11:                                     ; preds = %7, %5
  ; {%1}
  ret void
  ; {%1}
}
```

Figure 2.7: The LLVM IR produced from the code in Figure 2.4.

It was to be expected that the first argument is nullable, because it is explicitly checked for nullability in the code and only dereferenced in one of the branches. Similarly, the second argument is dereferenced in all branches, so it should obviously be non-nullable. Perhaps less obvious is that the third argument is not inferred as non-nullable. It is only safe to pass a null pointer for this third argument if the first argument is also null, because then the branch with block 5 is taken and the third argument is not dereferenced. However, the current inference algorithm is pessimistic and only infers non-nullability for

an argument if an undesirable event must occur when a null pointer is passed for that argument. Otherwise we may end up excluding a valid code-path in the bindings we generate from this information.

## 2.2 Structures

In this thesis we extend the existing nullability inference algorithm to take pointers stored in structure fields into account, so it is important to know more about structures. In this section, we introduce structures as they appear in C source code and also show the translation to LLVM IR.

### 2.2.1 Structs in C

Informally, structures are collections of data in a fixed form or shape. For example, we can define a structure for persons as shown in Figure 2.8 which consists of two double precision floating point numbers that indicate the cartesian coordinates $x$ and $y$. In general a structure can be defined by listing a number of named fields along with their types.

```c
#include <stdlib.h>
#include <stdio.h>

struct person {
  char *first_name;
  char *last_name;
};

struct person * new_person(char *first_name, char *last_name) {
  struct person *p = malloc(sizeof(struct person));
  p->first_name = first_name;
  p->last_name = last_name;
  return p;
}

void greet(struct person *p) {
  printf("Hello %s %s!\n", p->first_name, p->last_name);
}
```

Figure 2.8: Basic structure syntax in C.

Figure 2.8 also shows how you can write values into struct fields and how you can read values out of the struct fields. The main syntax we use for this is the arrow, for example in `p->x`. If this is used on the left hand side of an assignment then the value on the right

hand side is written into the structure 'p' at field 'x'. When this syntax is used in an expression then it means to read out the value from the strucuture 'p' at field 'x'.

There are other ways to access struct fields. Note that the arrow syntax is used to access struct fields of pointers to structs. A similar notation, but with a dot instead: `p.x`, can be used if we have direct access to a struct. Figure 2.9 shows an alternative implementation of the `greet` function which uses direct struct access. However, especially for larger structs, it is uncommon to see that kind of access due to performance considerations which we will not go further into.

```
void greet_direct(struct person *p) {
  struct person q = *p;
  printf("Hello %s %s!\n", q.first_name, q.last_name);
}
```

Figure 2.9: Direct struct field access.

A third way to access struct fields is to copy over entire structs by assigning one struct to another. Figure 2.10 shows an example where all the fields of an existing person struct are copied into a newly allocated person struct. In the end, this has the same effect as copying all the fields individually, but it is often implemented more efficiently.

```
struct person * copy_person(struct person *p) {
  struct person *q = malloc(sizeof(struct person));
  *q = *p;
  return q;
}
```

Figure 2.10: Struct copy.

Finally, no feature of C would be complete without a way to subvert it. In this case, the memory that stores the struct fields can be accessed directly if you can guess the right memory address or more realistically by pointer arithmetic on pointers to nearby data. For example, as shown in Figure 2.11, you could convert a pointer to the struct into a void pointer, add 8 to it, and finally we convert it back into a pointer to a string, which we can dereference to get the '`last_name`' field. Of course this behavior is implementation defined.

```
char * subvert(struct person *p) {
    return *(char **) ((void *)p + 8);
}
```

Figure 2.11: Subverting struct field access.

### 2.2.2 Structs in LLVM

In this section we look back on the forms of struct access and show what they look like after they are compiled to the LLVM IR. The inference algorithm works on this LLVM representation, so it is important to have a basic understanding of what it looks like. Along the way we explain the most important aspects such that even people with little to no knowledge about LLVM should be able to follow it, although a basic understanding of programming in a C-like language is still assumed. For clarity and to make everything fit on the page we have left out some irrelevant details such as metadata annotations and modifiers.

```
%struct.person = type { i8*, i8* }

@.str = private constant [14 x i8] c"Hello %s %s!\0A\00", align 1

define %struct.person* @new_person(i8* %0, i8* %1) {
  call void @llvm.dbg.value(metadata i8* %0, metadata !19)
  call void @llvm.dbg.value(metadata i8* %1, metadata !20)
  %3 = call i8* @malloc(i64 16)
  %4 = bitcast i8* %3 to %struct.person*
  call void @llvm.dbg.value(metadata %struct.person* %4, metadata !21)
  %5 = bitcast i8* %3 to i8**
  store i8* %0, i8** %5, align 8
  %6 = getelementptr i8, i8* %3, i64 8
  %7 = bitcast i8* %6 to i8**
  store i8* %1, i8** %7, align 8
  ret %struct.person* %4
}

define void @greet(%struct.person* %0) {
  call void @llvm.dbg.value(metadata %struct.person* %0, metadata !41)
  %2 = getelementptr %struct.person, %struct.person* %0, i64 0, i32 0
  %3 = load i8*, i8** %2, align 8
  %4 = getelementptr %struct.person, %struct.person* %0, i64 0, i32 1
  %5 = load i8*, i8** %4, align 8
  %6 = call i32 (i32, i8*, ...) @__printf_chk(
      i32 1,
      i8* getelementptr ([14 x i8], [14 x i8]* @.str, i64 0, i64 0),
      i8* %3,
      i8* %5
    )
  ret void
}
```

Figure 2.12: Basic structure syntax in LLVM IR.

In Figure 2.12 we show what the basic struct example from Figure 2.8 looks like in the LLVM representation. There are three top-level declarations: the `person` struct with two fields (the field names are gone), a string constant containing the template for the greeting, a `new_person` function, and a `greet` function.

Perhaps the first thing that attracts your attention is that LLVM is a typed language, so you see types almost on every line. One of the most basic types are the integers; these have names starting with an 'i' and then a number indicating how many bits wide that integer is. In the struct fields we see two pointers, denoted by `*`, to eight-bit integers, which are used to store the characters of the first name and last name. The string constant as a fixed-length array type denoted by `[14 x i8]`, meaning that the array contains fourteen eight-bit integers. Furthermore, we see that the `%struct.person` can itself be used as a type, we see different width integers: `i32` and `i64`, and finally we see a special `metadata` type for metadata.

In the body of the functions we see many straightforward instructions: `call` for calling functions, `load` for loading values from memory, `store` for storing values in memory, `bitcast` for changing the type of a value without changing the actual information, and finally `ret` for returning a value from a function. You will also see special function calls to the `@llvm.dbg.value` function. That function attaches some metadata information from its second argument to the value given as its first argument.

The last instruction that we need to explain is `getelementptr`. The `getelementptr` instruction is critical for accessing structure fields, but it does not do any memory access by itself; that needs to be done separately with the `load` instruction. Rather, given a pointer to a structure, or otherwise indexable value, it calculates an offset based on given indices. Structures can be nested, so sometimes multiple indices are necessary.

```
define void @greet_direct(%struct.person* %0) {
  call void @llvm.dbg.value(metadata %struct.person* %0, metadata !50)
  %2 = getelementptr %struct.person, %struct.person* %0, i64 0, i32 0
  %3 = load i8*, i8** %2, align 8
  call void @llvm.dbg.value(metadata i8* %3, metadata !51)
  %4 = getelementptr %struct.person, %struct.person* %0, i64 0, i32 1
  %5 = load i8*, i8** %4, align 8
  call void @llvm.dbg.value(metadata i8* %5, metadata !51)
  %6 = call i32 (i32, i8*, ...) @__printf_chk(
      i32 1,
      i8* getelementptr ([14 x i8], [14 x i8]* @.str, i64 0, i64 0),
      i8* %3,
      i8* %5
    )
  ret void
}
```

Figure 2.13: Direct struct field access in LLVM IR.

Figure 2.13 shows that the LLVM representation of the `greet_direct` function from Figure 2.9. The function is almost identical except for two extra metadata calls. So, the LLVM representation does not distinguish between these two ways to access structure fields.

```
define %struct.person* @copy_person(%struct.person* %0) {
  call void @llvm.dbg.value(metadata %struct.person* %0, metadata !61)
  %2 = call i8* @malloc(i64 16)
  %3 = bitcast i8* %2 to %struct.person*
  call void @llvm.dbg.value(metadata %struct.person* %3, metadata !62)
  %4 = bitcast %struct.person* %0 to i8*
  call void @llvm.memcpy(i8* align 8 %2, i8* align 8 %4, i64 16, i1 false)
  ret %struct.person* %3
}
```

Figure 2.14: Struct copy in LLVM IR.

The LLVM representation of the `copy_person` struct from Figure 2.10 is shown in Figure 2.14. Here we see a call to the `@llvm.memcpy` function which copies an area of memory from one location to another, in this case it copies all fields from the given `person` struct to a newly allocated piece of memory.

In Figure 2.15 you can see what happens to the C source code from Figure 2.11 where we cast a pointer struct and manually dereference an offset of that pointer. Notice that LLVM has completely recovered normal struct field access. So, if we base our inference on struct field access, then we can even get information in some situations where the C source code directly manipulates memory.

```
define i8* @subvert(%struct.person* %0) {
  call void @llvm.dbg.value(metadata %struct.person* %0, metadata !74)
  %2 = getelementptr %struct.person, %struct.person* %0, i64 0, i32 1
  %3 = load i8*, i8** %2, align 8
  ret i8* %3
}
```

Figure 2.15: Subverting struct field access in LLVM IR.

## 2.3 Problem Analysis

In the previous sections, we have discussed nullability inference and structure fields, so now we have the required background knowledge to refine our research question and discuss approaches answering it.

We have seen in Section 2.1.2 that the inference algorithm will trace back the origin of a value that is dereferenced. If the origin is a function argument, then we can annotate that

argument as being non-nullable. However, we might instead run into a `load` instruction, which means that the dereferenced pointer is stored in memory, so we cannot easily determine the real source of that pointer. In some of these cases the memory that stores the pointer is actually part of a struct. If we can identify those cases then we can remove part of this blind spot of the nullability inference algorithm.

So, our refined research question is:

> How can we improve the existing nullability inference algorithm to infer the nullability of pointers stored in structure fields?

An example where such an extension would be useful is the `greet` function that we have seen in Figure 2.8; this function extracts the first name and last name fields from the person struct and prints them, so these pointers are dereferenced. If we would construct a new person using the `new_person` function that contains null pointers for the first name or last name fields, then we would encounter undesirable behaviour. Hence, our analysis should enable us to annotate the `first_name` and `last_name` arguments of the `new_person` structs as non-nullable.

If we can achieve that, then our analysis results will be more accurate for programs that store pointers in structure fields, which is common in practice.

# Chapter 3

# Nullability Inference through Structure Fields

In this chapter we look back on the existing nullability analysis discussed in Section 2.1. The existing analysis can not infer any information about pointers which are stored in, and extracted from, structs. We consider how we can apply the knowledge about structs discussed in Section 2.2 to improve the nullability analysis, such that it can take structs into account.

Informally, the existing nullability inference algorithm works as follows: if we have a function that takes a pointer argument and dereferences it then we can infer that the pointer argument is non-nullable, because a null pointer would cause an undesirable event when dereferenced. We can adapt this to structure fields as follows: if we have a struct field which stores a pointer, the pointer is retrieved from that field and finally the pointer is dereferenced, then we can infer that the pointer field is non-nullable.

In this section we make this informal description more formal. First we consider two important nuances: how we identify structures in Section 3.1 and the scope and control flow considerations in Section 3.2. Afterwards, we propose changes to the transfer function to implement this new understanding in Section 3.3. Finally, we will go through some artificial examples to show how this extension can improve the inference results in Section 3.4, real-world examples will be discussed in Chapter 4.

## 3.1 Identification of Structures

The first important nuance in the semantics of structs is how we identify structs. In Figure 2.8 we define a struct called ‘person’, we could use this name as identification. That means that all structs with this name are seen as equal and inferred properties are shared between all occurrences of structs with this name. We call this perspective the *global* identification of structs.

For an example of how this can improve inference we can look at Figure 2.8 again. We see that the `greet` function uses the `first_name` and `last_name` fields in the `printf` function, which requires that its arguments are non-nullable. In the global perspective we infer from this that the `first_name` and `last_name` fields of the `person` struct are non-nullable. Furthermore, in the `new_person` function we write the `first_name` and `last_name` arguments into their respective fields of the `person` struct, so these arguments are non-nullable. In this way, the `greet` function is linked to the `new_person` function even though they are not connected in the call graph.

Alternatively, we can view the identity of struct fields from a *local* perspective. In this view, every struct gets a new identity whenever it is initialised or modified. In Figure 2.8, the struct in the argument of the `greet` function is not necessarily the same struct as the struct produced by the `new_person` function. So, we cannot infer that the arguments of the `new_person` function are non-nullable.

For an example where this local view does infer nullability of function arguments we can consider a function that combines `new_person` and `greet` as in Figure 3.1. Reasoning backwards, the `greet` function requires that the fields of `p` are non-nullable and the `new_person` function writes the `first_name` and `last_name` arguments into that struct, therefore those arguments must also be non-nullable.

```
struct person * greet_new_person(char *first_name, char *last_name) {
  struct person *p = new_person(first_name, last_name);
  greet(p);
  return p;
}
```

Figure 3.1: Example where local inference works.

## 3.2 Control Flow

In the previous section, we have shown how the global perspective on the identification of structures combines information about the access of structure fields even across different functions. Here, we consider how the inference algorithm should work inside functions with more complex control flow.

An option is to consider any pointer structure field that is dereferenced anywhere in a function to be non-nullable. However, this contradicts a very intuitive property, namely that if there is a check that the pointer structure field is not null before dereferencing it, then it should not be inferred to be non-nullable.

An example where the pointer is checked before dereferencing, is shown in Figure 3.2. We have a function `f` which takes as input a person struct, which we defined in Figure 2.8. The function simply reads the `first_name` field from the record, checks if it is not null, and returns the first character if that is true and the

0 byte otherwise. If the `person` struct `x` contains a null pointer for the `first_name` field then nothing goes wrong: it just returns a default value. So, here we should *not* infer that the `first_name` field is non-nullable.

```
char f(struct person *x) {
  char c = '\0';
  char *y = x->first_name;
  if (y != NULL) {
    c = *y;
  }
  return c;
}
```

Figure 3.2: Example of structure access with nullability check.

On the other end of the spectrum, we could stay closer to the approach from Section 2.1.2 where we only infer non-nullability if running a function must cause the structure field to be dereferenced. This might seem like an appropriate conservative solution, but it is perhaps too conservative.

Consider an example where there is control flow but it is not related to the nullability of a structure field, such an example is shown in Figure 3.3. The function `g` takes as input a pointer to a person struct `x` and an integer `b`. The integer `b` is used to decide whether the `first_name` field should be dereferenced, if it is 0 then the field is not dereferenced otherwise it is dereferenced. If we stick to the conservative approach, then we cannot infer that the `first_name` is dereferenced.

```
char g(int b, struct person *x) {
  char c = '\0';
  if (b) {
    c = *x->first_name;
  }
  return c;
}
```

Figure 3.3: Example of structure access with unrelated check.

Now, let us consider a slight alteration to this example. We move the dereferencing of the `first_name` field into its own function. The result of this simple transformation is shown in Figure 3.4. Because of the global identification of structure fields, running the inference on this new function `h` will let us infer that the `first_name` field is non-nullable. This suggests that a more lenient approach might be feasible.

```
char h(struct person *x) {
  return *x->first_name;
}

char g(int b, struct person *x) {
  char c = '\0';
  if (b) {
    c = h(x);
  }
  return c;
}
```

Figure 3.4: The same as Figure 3.3, but now with dereferencing in separate function.

Let us also quickly revisit the first example where we do check for nullability from Figure 3.2. There, the `first_name` field is accessed in two steps: first the `char` pointer is extracted from the `first_name` field of the person struct, that pointer is only dereferenced later in the if-statement. If we would want to extract that usage into a separate function we would have to include the whole if-statement, so the inference results would not change. This provides more evidence that there is perhaps a suitable middle ground.

## 3.3 The Improved Inference Algorithm

In this section we describe how to integrate inference through structure fields with the observations from Section 3.1 and Section 3.2 into the existing inference algorithm from Section 2.1.2.

The core of the data-flow analysis algorithm stays the same: our input is the referential representation, we apply transfer functions to data-flow facts before each instruction, and propage information along the control flow graph until a fixed point is reached. We describe how to extend the lattice and transfer functions for nullability inference through structure fields.

Remember from Section 2.1.2 that the lattice, the information that we are inferring at each instruction in the program, for the normal nullability analysis is the set of non-nullable arguments of the current function. To track information about the nullability of structure fields we need to have some additional container which stores the struct and field names of non-nullable fields.

Simply storing the struct and field names in a set with set intersection as meet would give us the conservative behaviour and with set union as meet would give us the behaviour that is too lenient. Instead, we store the non-nullable fields in a map or dictionary, so our data-flow facts are now tuples of sets of non-nullable arguments and maps of struct names and sets of their non-nullable fields. In our formulas we will show maps as sets of tuples,

where the struct names are the first element of the tuple and the sets of non-nullable fields are the second element of the tuple.

First, we discuss when new elements are inserted. In Section 2.2.2, we have seen that dereferencing a pointer that is stored in a struct happens in three steps. Given a pointer to a struct, the first step is to get a pointer to the field that we want to access using the `getelementptr` LLVM instruction, this yields a pointer to the pointer that we want to access. The second step is to load the field from the pointer to the field using the `load` instruction, which results in the pointer which was stored in the structure. Finally, the third step is to access this pointer, which could mean directly dereferencing it, passing it to another function, or perhaps first checking for nullability and then doing other things. It is only after the second step that there could be a nullability check, so at the second step we insert an entry into the map with the struct name as field and an empty set as value. If the pointer that was stored at the field is finally dereferenced in the third step, then we insert that field in the set which is stored in the map at the key of the struct that the field belongs to.

By inserting the field and struct separately we allow for the possibility of control flow in between. If the pointer stored in the struct is checked for nullability before it is dereferenced, then the control flow will split into two parts. In one branch the struct will be associated with an empty set and in the other branch the struct will be associated with a singleton set containing the field that was dereferenced. If we merge these two maps by taking the intersection of sets at the same key, then we end up with a map where the struct is associated with an empty set as we would expect because the pointer is checked for nullability before it is dereferenced.

It remains to describe what happens when a key is present in one list but not the other. This can happen when there is a condition where in one branch it will dereference a field $f_1$ from one struct $s_1$ and in the other branch it dereferences a field $f_2$ from another struct $s_2$. When these two control flow paths join up, one map will contain $s_1$ associated with $\{f_1\}$ and another map contains $s_2$ associated with $\{f_2\}$. In this case, we choose to be lenient and assume both accesses are independent, so we combine both into a map which now has two keys.

$$
\begin{aligned}
&(\mathit{args}_1, \mathit{fields}_1) \sqcap (\mathit{args}_2, \mathit{fields}_2) = \\
&\quad (\mathit{args}_1 \cap \mathit{args}_2 \\
&\quad , \{t \mapsto x_1 \cap x_2 \mid t \mapsto x_1 \in \mathit{fields}_1, t \mapsto x_2 \in \mathit{fields}_2\} \\
&\qquad \cup \{t \mapsto x_1 \mid t \mapsto x_1 \in \mathit{fields}_1, \forall x_2 : t \mapsto x_2 \notin \mathit{fields}_2\} \\
&\qquad \cup \{t \mapsto x_2 \mid \forall x_1 : t \mapsto x_1 \notin \mathit{fields}_1, t \mapsto x_2 \in \mathit{fields}_2\} \\
&\quad )
\end{aligned}
$$

Figure 3.5: The meet of our improved algorithm.

To recap: our lattice is a map where the keys are structs and the values are sets of non-nullable fields. The meet operation, shown in Figure 3.5, takes the union of the keys in both maps, but the intersection of the associated sets if a key is present in both maps.

Changing the transfer functions is relatively simple, the improved transfer functions are shown in Figure 3.6; again, all the tranfer functions for other instructions are identity functions. All the previous behaviour on the set of non-nullable arguments is retained, but now there are some extra cases for the map of non-nullable structure fields.

$$
\begin{aligned}
&[\textbf{load } ptr]_\ell : && f_\ell(x) = x \oplus \text{deref}(ptr) \\
&[\textbf{store } val\ ptr]_\ell : && f_\ell(x) = x \oplus \text{deref}(ptr) \oplus \text{store}(val, ptr) \\
&[\textbf{call } calledFunc(x_1, x_2, \ldots, x_n)]_\ell : \\
&\quad f_\ell(x) = x \\
&\qquad \oplus \begin{cases} \top, & \text{if noRet}(calledFunc). \\ \bigoplus\{\text{deref}(x_i) \mid i \in \{1, \ldots, n\}, \text{notNullable}(calledFunc, i)\}, & \text{otherwise.} \end{cases} \\
&\qquad \oplus \begin{cases} \text{deref}(calledFunc), & \text{if indirect}(calledFunc). \\ \emptyset, & \text{otherwise.} \end{cases}
\end{aligned}
$$

Where $\oplus$ performs a union of the sets of non-nullable arguments and a union of the keys in the map and a union of the sets of non-nullable fields in the map if a key occurs in both maps, or as a formula:

$$
\begin{aligned}
&(args_1, fields_1) \oplus (args_2, fields_2) = \\
&\quad (args_1 \cup args_2 \\
&\quad , \{t \mapsto x_1 \cup x_2 \mid t \mapsto x_1 \in fields_1, t \mapsto x_2 \in fields_2\} \\
&\qquad \cup \{t \mapsto x_1 \mid t \mapsto x_1 \in fields_1, \forall x_2 : t \mapsto x_2 \notin fields_2\} \\
&\qquad \cup \{t \mapsto x_2 \mid \forall x_1 : t \mapsto x_1 \notin fields_1, t \mapsto x_2 \in fields_2\} \\
&\quad )
\end{aligned}
$$

Figure 3.6: The improved transfer function for nullability analysis through structure fields.

In the 'deref' function, where previously we stopped if we ran into a `load` instruction, we can now continue and check if the load instruction is actually loading a structure field. Specifically, loading a structure field means that after the `load` instruction we encounter a `getelementptr` instruction for which the base type is a pointer to a struct type and for which there are two indices: the first one dereferencing the base pointer and the second one is the index of the structure field. In that case we add the index of the dereferenced field to the set of non-nullable fields for the struct in the map.

For the `store` instruction, in addition to dereferencing the base pointer we now also need to check if a pointer is stored in a structure. For that, we introduce a 'store' function, whose first argument is a value *val* and the second argument is a pointer *ptr* to the location where the value should be stored. If the *val* itself is also a pointer and the *ptr* is a pointer to a non-nullable structure field, then the first pointer should be non-nullable too. To detect if *ptr* points to a structure field, we, again, check if we encounter a `getelementptr` instruction where the base is a pointer to a struct, and there are two indices, of which the second is the field that *val* is stored into. If that field number of that struct is a non-nullable field, then we apply the 'deref' function to *val*. Otherwise, 'store' simply returns an empty set and empty map.

For each function, the output of this improved algorithm is now the set of non-nullable arguments, as before, and also a map of structs along with their non-nullable fields.

## 3.4 Examples

In this Section we present a few artificial examples to make it clearer how the inference algorithm works and to show that it produces the expected results. We revisit the examples we have used to justify our design, namely the example with a check for nullability before dereferencing from Figure 3.2 and the example with unrelated control flow from Figure 3.3.

As before the first step is to convert the C code to the optimised LLVM representation. A simplified version of this LLVM representation is shown in Figure 3.7.

```
define i8 @f(%struct.person* %0) {
  %2 = getelementptr %struct.person, %struct.person* %0, i64 0, i32 0
  %3 = load i8*, i8** %2, align 8
  %4 = icmp eq i8* %3, null
  br i1 %4, label %7, label %5

; <label>:5:                                        ; preds = %1
  %6 = load i8, i8* %3, align 1
  br label %7

; <label>:7:                                        ; preds = %1, %5
  %8 = phi i8 [ %6, %5 ], [ 0, %1 ]
  ret i8 %8
}
```

Figure 3.7: Example of structure access with nullability check in LLVM.

As usual, the inference starts with an empty state, but in this case that means an empty set of non-nullable arguments and an empty map of structs with their non-nullable fields.

The first instruction we encounter is `getelementptr` which is not relevant for now, so we skip it. The second instruction loads the pointer generated by that `getelementptr` instruction, so we consider the base pointer to be dereferenced, which is the first argument `%0` in this case. With the new rules for dereferencing structure fields, and because the `getelementptr` instruction is applied to the "person" structure, we add a new entry into the map for the person struct with an empty set as value. Then, there is a comparison and a branch, which are both irrelevant.

We continue at label 5 which starts with a `load` instruction which actually loads the field, so we add the field number, in this case 0, into the map at the "person" key. Then, there is an irrelevant branch again.

```
define i8 @f(%struct.person* %0) {
  ; {}, []
  %2 = getelementptr %struct.person, %struct.person* %0, i64 0, i32 0
  ; {}, []
  %3 = load i8*, i8** %2, align 8
  ; {%0}, [person: {}]
  %4 = icmp eq i8* %3, null
  ; {%0}, [person: {}]
  br i1 %4, label %7, label %5
  ; {%0}, [person: {}]

; <label>:5:                                     ; preds = %1
  ; {%0}, [person: {}]
  %6 = load i8, i8* %3, align 1
  ; {%0}, [person: {0}]
  br label %7
  ; {%0}, [person: {0}]

; <label>:7:                                     ; preds = %1, %5
  ; {%0}, [person: {}]
  %8 = phi i8 [ %6, %5 ], [ 0, %1 ]
  ; {%0}, [person: {}]
  ret i8 %8
  ; {%0}, [person: {}]
}
```

Figure 3.8: Inference results of structure access example with nullability check.

Finally, two branches merge at label 7, so we apply the meet operation to the sets in both branches. In both branches the set of non-nullable arguments contains the only argument `%0`, since these are the same the result of applying the meet operation is also this same set. The maps of structure names and non-nullable fields are different. Both branches

have the "person" key, but only one branch has an element in the associated set. So, we take the union of the keys, which is just "person" in this case, and the intersection of the associated sets, which is the empty set in this case. All in all, we end up with the set $\{\%0\}$ and the map $[person \mapsto \emptyset]$. The actual instructions in this last block, a `phi` and a `ret` instruction, are both irrelevant, except for their influence on the control flow. So, the result of the analysis for this function $f$ is $\{\%0\}, [person \mapsto \emptyset]$. That means that the first and only argument of $f$ is non-nullable, but for the rest no information is inferred about any structure fields, which is exactly what we expected because of the check for nullability.

The final result at each instruction is shown in Figure 3.8. There we use the notation `[key: value]` to denote a map.

The second example for which we will show the new inference algorithm in detail is the example of structure access with unrelated control flow from Figure 3.3. The optimised and simplified LLVM representation of this example is shown in Figure 3.9.

```
define i8 @g(i32 %0, %struct.person* %1) {
  %3 = icmp eq i32 %0, 0
  br i1 %3, label %8, label %4

; <label>:4:                                        ; preds = %2
  %5 = getelementptr %struct.person, %struct.person* %1, i64 0, i32 0
  %6 = load i8*, i8** %5, align 8
  %7 = load i8, i8* %6, align 1
  br label %8

; <label>:8:                                        ; preds = %2, %4
  %9 = phi i8 [ %7, %4 ], [ 0, %2 ]
  ret i8 %9
}
```

Figure 3.9: Example of structure access with unrelated control flow in LLVM.

We start, again, with an empty initial state. This time the first block contains no relevant instructions, so we continue to the second block with this empty initial state.

Now, the `getelementptr` and the two `load` instructions are in the same block. We first skip the `getelementptr`. Then we add argument `%1` to the set of non-nullable arguments and we add the key "person" to the map with an empty set as value for the first `load` instruction. For the second `load` instruction, we add the field 0 to the set that is associated to the "person" key in the map. With the last branch we go to the final block.

In the final block we have again that two branches merge. In this case, however, we see that there are two different sets of non-nullable arguments, in one branch the set is empty and in the other branch the set has one element. We take the intersection, so we end

up with an empty set of non-nullable arguments. Similarly, in one branch there is an empty map and in the other branch there is a map with a "person" key and its first field is marked as non-nullable. The key is missing in one map and present in the other so, taking the union, in the result the key will be present with its set from the branch that it came from. We skip the two irrelevant instructions in this final block, so the end result is an empty set of non-nullable arguments and the map $[person \mapsto \{0\}]$ which means that the first field of the "person" struct is marked as non-nullable.

The final result at each instruction are shown in Figure 3.10.

```
define i8 @g(i32 %0, %struct.person* %1) {
  ; {}, []
  %3 = icmp eq i32 %0, 0
  ; {}, []
  br i1 %3, label %8, label %4
  ; {}, []

; <label>:4:                                        ; preds = %2
  ; {}, []
  %5 = getelementptr %struct.person, %struct.person* %1, i64 0, i32 0
  ; {}, []
  %6 = load i8*, i8** %5, align 8
  ; {%1}, [person: {}]
  %7 = load i8, i8* %6, align 1
  ; {%1}, [person: {0}]
  br label %8
  ; {%1}, [person: {0}]

; <label>:8:                                        ; preds = %2, %4
  ; {}, [person: {0}]
  %9 = phi i8 [ %7, %4 ], [ 0, %2 ]
  ; {}, [person: {0}]
  ret i8 %9
  ; {}, [person: {0}]
}
```

Figure 3.10: Inference results of structure access example with unrelated control flow.

These two examples have shown that this algorithm works and how it works for single functions. It should be noted that this analysis no longer simply follows the call-graph. The information we infer about the nullability of structure fields can influence the results of any function that uses these structure fields. To work around this we have chosen the simple solution of considering every function dependent on every other function. This can cause performance problems with larger applications, so ideally steps should be taken to

determine more fine-grained dependencies between functions or even between individual basic blocks.

# Chapter 4

# Evaluation

In this chapter we substantiate our claim that our work improves the nullability inference by showing the impact on libraries in practice. We pick the GNU Libmicrohttpd library for a qualitative analysis in Section 4.1. To provide more conclusive evidence we wanted to do an extensive quantitative study, but practical roadblocks prevented large scale automated application of our inference algorithm. We discuss these practical challenges, but defer a proper quantitative study to future work. Furthermore, we mentioned that locally identifying structures is more conservative than globally identifying structures. We make this formal by considering the soundness of our approach in Section 4.2. Finally, we reflect briefly on the use of LLVM rather than C source code for inference algorithms like ours in Section 4.3.

## 4.1 Case Study: GNU Libmicrohttpd

The first library we use for testing is GNU Libmicrohttpd. It is a small C library which makes it easy to embed a simple HTTP server into another application. We chose it as a first test subject, because it is small and easy to understand.

Inspecting the library, we quickly found a situation where our improved analysis indeed gives better results.

In this case, the struct that contains the pointers is called `MHD_HTTP_Header`, which is shown in Figure 4.1. This struct is chained in a linked list with the `next` field. It stores the name of the header as a string in the field `header` and its length in `header_size`. The value of the header is stored as a string in the field `value` with its length in `value_size`. Finally, it stores the kind of header in the field `kind`.

```c
/**
 * Header or cookie in HTTP request or response.
 */
struct MHD_HTTP_Header
{
  /**
   * Headers are kept in a linked list.
   */
  struct MHD_HTTP_Header *next;

  /**
   * The name of the header (key), without the colon.
   */
  char *header;

  /**
   * Number of bytes in @a header.
   */
  size_t header_size;

  /**
   * The value of the header.
   */
  char *value;

  /**
   * Number of bytes in @a value.
   */
  size_t value_size;

  /**
   * Type of the header (where in the HTTP protocol is this header
   * from).
   */
  enum MHD_ValueKind kind;

};
```

Figure 4.1: The `MHD_HTTP_Header` struct in GNU Libmicrohttpd

Usually, linked lists are terminated by null pointers, so the `next` field is probably nullable. The two remaining pointers are the `header` and `value` strings. You could say that storing null pointers in these fields would be redundant, because you can always also store an empty string. Additionally, you would expect a header to have a name and probably

a value too. However, this struct definition and documentation do not give conclusive evidence whether these are nullable or not.

There are many clues about the nullability of the `header` and `value` fields scattered throughout the code. For instance, the `MHD_lookup_header_token_ci` function, shown in Figure 4.2, traverses the linked list of received headers and checks if a certain header name and value pair is present. For every header it checks the header kind, the length of the header name, and if the header name pointer is equal to the pointer we are searching for, but the important checks are at the end, namely the checks that use the `MHD_str_equal_caseless_bin_n_` and `MHD_str_has_token_caseless_` functions. Both of these functions exhibit dubious behaviour when applied to null pointers.

```
static bool
MHD_lookup_header_token_ci (const struct MHD_Connection *connection,
                            const char *header,
                            size_t header_len,
                            const char *token,
                            size_t token_len)
{
  struct MHD_HTTP_Header *pos;

  if ((NULL == connection) || (NULL == header) || (0 == header[0])
      || (NULL == token) || (0 == token[0]))
    return false;

  for (pos = connection->headers_received; NULL != pos; pos = pos->next)
  {
    if ((0 != (pos->kind & MHD_HEADER_KIND)) &&
        (header_len == pos->header_size) &&
        ( (header == pos->header) ||
          (MHD_str_equal_caseless_bin_n_ (header,
                                          pos->header,
                                          header_len)) ) &&
        (MHD_str_has_token_caseless_ (pos->value, token, token_len)))
      return true;
  }
  return false;
}
```

Figure 4.2: The `MHD_lookup_header_token_ci` function.

The `MHD_str_equal_caseless_bin_n_` function takes two strings and a length and checks if the strings are equal up to the given length ignoring upper-case and lower-case differences. The source code of this function is shown in Figure 4.3. You can see that the strings are unconditionally dereferenced inside the loop, so if they are null point-

ers, then the loop body must never be executed. The body of the loop is executed until the variable `i` is smaller than the `len` argument and `i` is initialised to zero, so the loop body is always executed unless `len` is zero too. Is this enough reason to conclude that the string arguments should really be non-nullable? According to our algorithm it is not enough, because the loop body might not be executed. However, Ravitch writes in a comment in the source code of his inference algorithm shown in Figure 4.5 that, in cases like this, the strings could be inferred non-nullable, because the function cannot do any useful work if the arguments are null.

```
bool
MHD_str_equal_caseless_bin_n_ (const char *const str1,
                               const char *const str2,
                               size_t len)
{
  size_t i;

  for (i = 0; i < len; ++i)
  {
    const char c1 = str1[i];
    const char c2 = str2[i];
    if ( (c1 != c2) &&
         (toasciilower (c1) != toasciilower (c2)) )
      return 0;
  }
  return ! 0;
}
```

Figure 4.3: Source code of the `MHD_str_equal_caseless_bin_n_` function.

The `MHD_str_has_token_caseless_` shown in Figure 4.4 behaves similarly: when the `token_len` argument is zero then the function immediately returns `false`, so technically the string arguments are nullable, but it is never useful.

We can relate our observations back to how these functions are used in the `MHD_lookup_header_token_ci` function. For the `value` field it is straightforward that a null pointer can be dereferenced if it is stored in the header that is looked up and if the token that is looked up has a non-zero length.

On the other hand, the situation with the `header` field is more complicated. The `header_len` argument is assumed to be equal to the length of the `header` argument and the `pos->header_size` field is assumed to be equal to the length of the `pos->header` string. Before the `pos->header` field is checked for equality with the `header`, the code will first check that their length is equal. If a `header` field is null, then the only reasonable length is zero, which means that the equality will only be checked if the `header_len` argument is also zero, but then the equality check function is harmless. So, for the `header` field, this situation is not conclusive about nullability.

```
bool
MHD_str_has_token_caseless_ (const char *str,
                             const char *const token,
                             size_t token_len)
{
  if (0 == token_len)
    return false;

  while (0 != *str)
  {
    ...
```

Figure 4.4: Excerpt of the `MHD_str_has_token_caseless_` function.

```
-- TODO: If a function can return without having *any* side effects
-- while a parameter is NULL, that parameter is not nullable.
--
-- > void bzero(char* p, int n) {
-- >   for(int i = 0; i < n; ++i ) {
-- >     p[i] = 0;
-- >   }
-- > }
--
-- In fact, passing NULL for p and 0 for n allows this function to be
-- called safely.  However, doing so has no value and it would be fair
-- to at least warn about seeing NULL passed for p here.
```

Figure 4.5: Comment by Ravitch about nullability when functions perform no side-effects.

And remember that Ravitch's suggestion, about inferring arguments to be non-nullable if functions cannot perform side-effects if those arguments are null, is not yet implemented, so this particular usage pattern does not cause the `header` and `value` fields to be inferred to be non-nullable.

Our algorithm does infer that these fields are non-nullable, because, in the `MHD_del_response_header` function, the `header` and `value` fields are passed as a arguments to the `memcmp` function which is a standard function that requires its arguments to be non-null. Additionally, `value` is used in other circumstances that require it to be non-nullable.

Having established that these fields are non-nullable we continue by searching for places where these fields are written to. That happens in the `MHD_set_connection_value_n_nocheck_` function shown in Figure 4.6. We can see that the function takes a `key` and a `value` string as input and writes them into the `pos->header` and `pos->value` fields respectively. One obstacle is already visible: before these arguments are written into the fields it first checks

if the allocated memory is not a null pointer. This branch of the control flow should be ruled out because the `MHD_NO` constant is used as an error code in this case, but, as it turns out, the current error-return inference is not good enough to recognise this.

```
static enum MHD_Result
MHD_set_connection_value_n_nocheck_ (struct MHD_Connection *connection,
                                     enum MHD_ValueKind kind,
                                     const char *key,
                                     size_t key_size,
                                     const char *value,
                                     size_t value_size)
{
  struct MHD_HTTP_Header *pos;

  pos = MHD_pool_allocate (connection->pool,
                           sizeof (struct MHD_HTTP_Header),
                           true);
  if (NULL == pos)
    return MHD_NO;
  pos->header = (char *) key;
  pos->header_size = key_size;
  pos->value = (char *) value;
  pos->value_size = value_size;
  pos->kind = kind;
  pos->next = NULL;
  /* append 'pos' to the linked list of headers */
  if (NULL == connection->headers_received_tail)
  {
    connection->headers_received = pos;
    connection->headers_received_tail = pos;
  }
  else
  {
    connection->headers_received_tail->next = pos;
    connection->headers_received_tail = pos;
  }
  return MHD_YES;
}
```

Figure 4.6: The source code of the `MHD_set_connection_value_n_nocheck_` function.

### 4.1.1 Roadblock: Inaccurate Error Returns

One major roadblock on the way to using inference through structure fields on the GNU Libmicrohttpd library is the inaccurate inference of error returns. An earlier stage of the inference-suite infers the error codes that some functions return. These error returns are important for the nullability inference, because returning an error is considered an undesirable event. Therefore if the presence of a null pointer in an argument or struct field would force a function to return an error code then that argument or struct field can be considered to be non-nullable. In practice, this means that we can ignore branches in the control flow that lead to returning an error.

```
define i32 @MHD_set_connection_value(%struct.MHD_Connection* %0,
                                     i32 %1,
                                     i8* %2,
                                     i8* %3) {
  ; [...]
  %15 = getelementptr %struct.MHD_Connection,
                      %struct.MHD_Connection* %0,
                      i64 0,
                      i32 10
  %16 = load %struct.MemoryPool*, %struct.MemoryPool** %15, align 8
  %17 = call i8* @MHD_pool_allocate(%struct.MemoryPool* %16, i64 48, i1 true)
  call void @llvm.dbg.value(metadata i8* %17, metadata !758)
  %18 = icmp eq i8* %17, null
  br i1 %18, label %39, label %19

; <label>:19:                                       ; preds = %13
  %20 = getelementptr i8, i8* %17, i64 8
  %21 = bitcast i8* %20 to i8**
  store i8* %2, i8** %21, align 8
  ; [...]
  %24 = getelementptr i8, i8* %17, i64 24
  %25 = bitcast i8* %24 to i8**
  store i8* %3, i8** %25, align 8
  ; [...]
  br label %39

; <label>:39:                                       ; preds = %19, %13
  %40 = phi i32 [ 0, %13 ], [ 1, %19 ]
  ret i32 %40
}
```

Figure 4.7: The LLVM representation of the `MHD_set_connection_value` function from Figure 4.6.

However, these error returns are not always accurately inferred by the existing analysis. This causes the nullability analysis to overlook certain struct fields which can actually be considered non-nullable. In the previous section, we have seen that the `MHD_set_connection_value_n_nocheck_` function from Figure 4.6 stores a header name and value into a `MHD_HTTP_Header` struct. We suggested that the if statement at the beginning of this function obstructs the inference. To see why that happens, we need to inspect the LLVM representation shown in Figure 4.7.

We have elided and simplified some irrelevant parts. It is important to notice that there are three basic blocks: first allocation and a comparison with null, then a block that writes the arguments `%2` and `%3` into the allocated memory, and finally a block that returns `0` or `1` depending on which branch was taken.

The problem with the existing error-return inference is that it only works on the level of blocks. It determines for each block whether it must result in returning an error code. In this case the first block ends in a choice between two branches, of which one is successful, so it is not guaranteed to return an error. The second block is guaranteed to result in a successful return. The final block combines the result of the two branches. As you can see, there is no block that forces an error code to be returned.

To be able to continue with our qualitative analysis, we circumvent this roadblock by manually removing the error returning code paths by commenting out conditional checks. Such drastic measures should not be required during the normal binding generation workflow. To address this situation, we discuss a better long-term solutions in Section 5.1.1 of the Future Work.

### 4.1.2 Roadblock: Bitcasts

The second roadblock we ran into has only become visible now we are inspecting the LLVM representation in Figure 4.7. In the first block, the `MHD_pool_allocate` function returns a pointer to raw bytes. This pointer is used as a pointer to a `MHD_HTTP_Header` struct in the rest of the code, but that is not immediately visible. Indeed, in the second block, this pointer to raw bytes is cast into a pointer to pointers to bytes, before the arguments can be written into it. This casting obfuscates the true meaning of this code and hinders our inference algorithm.

For now, we have circumvented this by making a concrete allocation function for `MHD_HTTP_Header` structs as shown in Figure 4.8. Along with the special "noinline" annotation, this forces the compiler to use a proper struct type as our inference algorithm expects. And, indeed, if we replace original `MHD_pool_allocate` function with this new concrete allocation function, then our algorithm infers the `key` and `value` arguments of the `MHD_set_connection_value` function to be non-nullable.

Again, we do not expect authors of library bindings to perform such manual modifications. In Section 5.1.2 of the Future Work we discuss an automatic solution to this problem.

```
__attribute__((noinline))
struct MHD_HTTP_Header *
MHD_pool_allocate_header (struct MHD_Connection *connection)
{
  return (struct MHD_HTTP_Header *)
    MHD_pool_allocate(connection->pool,
                      sizeof (struct MHD_HTTP_Header),
                      true);
}
```

Figure 4.8: A concrete allocation function for `MHD_HTTP_Header` structs.

## 4.2 Soundness

Ravitch's error-return analysis is already unsound, so some control-flow paths may be erroneously annotated as returning an error. The original nullability analysis ignores control-flow paths that result in errors, so it is also unsound. Our work introduces yet more sources of unsoundness, which we will discuss in this section.

By using the global perspective of identifying structure fields we have abandoned the interpretation of non-nullability to be that a null value *must* cause an undesirable event. Now, a null value *may* cause an undesirable event, but only with respect to pointers stored in structs.

An example situation where our improved inference algorithm is unsound is when there is a check for nullability of a structure field at the beginning of a function which is stored, then there might be some other activities, but in the end the stored result of the nullability check is used to determine whether to access that same structure field or not. Our algorithm will view this second check as unrelated control flow and infer that the structure field is non-nullable.

Another example is a function for which there are preconditions which imply that it may not be used when a certain structure field is null. All other functions could check for nullability of that field, but this one function could be an unsafe exception, where this behaviour is documented in comments. Our inference algorithm does not take this possibility into account and simply labels the field as non-nullable.

A similar situation may occur when there are no unsafe functions, but instead a certain protocol which requires functions to be used in a particular order. If that is the case, then there may be a phase in which a certain structure field can be null, and a second phase in which the field is non-nullable. However, our inference algorithm will infer that field to be non-nullable globally throughout the program.

Generally, if a struct is used in a particular way in one part of a library and if it is used differently in another part of the library, then our current inference algorithm combines the information from both parts. In particular, for nullability inference this might mean

that if a struct field is used in a non-nullable way in a certain function, but it is nullable in another function, then our algorithm infers the field to be non-nullable everywhere, which may prevent users of bindings that are generated based on this information to use certain functionality.

It must be noted that we have not encountered excessive non-nullable annotations during our evaluation. In fact, in the GNU Libmicrohttpd library we have only found two arguments of a single function to have additional non-nullable annotations using our improved algorithm. Possibly, this could be due to the roadblocks we discussed in Section 4.1.1 and Section 4.1.2 that hinder our inference algorithm, or simply due to our small sample-size of just a single library. However, perhaps soundness needs to be sacrificed, to a certain extent, to achieve usable results.

While on this topic, we must also mention the soundiness manifesto [7]. To paraphrase, they claim soundness is not always necessary for extremely disruptive language features, which would require a much more complicated program analysis if it is possible to take them into account at all. However, this soundiness principle does not fit with our work. In our inference algorithm the unsoundness is not (only) due to a select number of disruptive language features. The unsoundness can occur even in benign subsets of the C programming language.

Instead, our justification of the lack of soundness is that the information inferred by our algorithm is intended to assist the writers of library bindings. Our algorithm should not be used for checking safety critical code. In the end, authors of library bindings should decide which arguments should be non-nullable.

## 4.3 Suitability of LLVM

In this section, we discuss the suitability of LLVM IR as the representation used in our inference algorithm.

We have seen in Section 4.1.2 that the LLVM representation can hide structure access in certain situations, so should we avoid LLVM and run our inference algorithm on plain C source code?

We believe that the optimised LLVM representation solves much more problems than it raises. In Section 2.2.2 we have seen that the LLVM representation unifies structure access, and that it can even recover structure access from C source code that uses direct memory access with offsets into the location where a struct is stored. Additionally, optimisations can simplify the code which makes it easier to analyse, for example by removing redundant control flow. Furthermore, essential information about the original source code, like identifiers and types can be recovered from LLVM metadata that is generated when compiling with debug info enabled.

# Chapter 5

# Conclusion

In this chapter we conclude this thesis and discuss opportunities for future work.

In this thesis, we have introduced an improvement to Ravitch's nullability inference algorithm, such that it can infer nullability of pointers that are stored in structs. We achieve this by identifying each structure globally: if a structure field is accessed in a non-nullable manner in one part of the code, and a pointer stored in that field in another part of the code, then that pointer can be considered non-nullable too. We have renovated Ravitch's inference-suite and extended it with this improvement. We have shown that this improvement, aside from two practical roadblocks, is able to infer more nullability annotations in a practical C library called GNU Libmicrohttpd. Our inference algorithm is not sound: if it infers that a pointer is non-nullable then that does not mean that an undesirable event must happen. However, we expect our inference results to be useful in practice. Rather than proving properties of programs we assist writers of bindings in making informed choices.

We believe that the approach of generating library bindings with the help of automatically inferred program properties holds great potential.

## 5.1 Future Work

There are many avenues for future work. Three avenues are immediately important to make our work viable: the error-return inference needs to be improved to give the nullability inference more information to work with, source code information needs to be recovered when LLVM optimises struct access away into flat memory access. Afterwards our improvement can be automatically run and quantitatively evaluated on a larger set of C libraries to confirm that it improves inference results and does not produce too many incorrect results due to the unsoundness.

To reduce the amount of situations in which our inference algorithm is unsound, future work could investigate the local identification of structure fields or a hybrid approach

where structs are grouped into a few larger classes.

Furthermore, we envision future work towards an interactive application for authors of library bindings. It could infer information automatically when there is no, or very little, doubt. When the evidence is not conclusive, it could present evidence for both sides of a certain choice, then the author can make the final decision. This would allow authors to focus on the complicated parts, and reduce the time spent on the obvious parts of writing library bindings.

### 5.1.1 Better Error-Return Inference

In Section 4.1.1, we have seen how an inaccurate error-return analysis causes the nullability analysis to miss inference opportunities as well. We believe a promising approach to recovering information about error-codes is the one developed by Bruntink et al. [2]. We suggest investigating if that can be combined with the existing error-code inference to improve the results. Alternatively, the existing approach could be adapted to the specific roadblock that we encountered.

### 5.1.2 Recovering Struct Information using LLVM Metadata

The second roadblock, from Section 4.1.2, concerns struct access operations which are obfuscated in the LLVM representation, because raw memory is converted into a struct too late and the struct access is converted into unstructured memory access. An approach to deal with this is to use LLVM metadata. Immediately after the memory is allocated, you can see in Figure 4.7 that a special `@llvm.dbg.value` function is called, which associates the allocated memory with useful metadata.

```
!237 = !DIDerivedType(tag: @DW_TAG_pointer_type, baseType: !238, size: 64)
!238 = distinct !DICompositeType(tag: @DW_TAG_structure_type,
                                 name: "MHD_HTTP_Header",
                                 file: !56,
                                 line: 313,
                                 size: 384,
                                 elements: !239)
; [...]
!758 = !DILocalVariable(name: "pos",
                        scope: !751,
                        file: !3,
                        line: 345,
                        type: !237)
```

Figure 5.1: Relevant LLVM metadata.

The linked metadata `!758` is shown in Figure 5.1, it is metadata for the local variable "pos" in the original C source code. The metadata links to the metadata `!758`, which is

the C type of the variable, in this case a pointer to the base type described in metadata `!238`. Now, we have reached the `MHD_HTTP_Header` struct, which we were looking for.

If we depend on this metadata, then that does also mean that we depend on the C source code being available. Ravitch suggests that the necessary LLVM representation could perhaps be extracted from compiled binaries using techniques proposed by ElWazeer et al. [3]. Such an approach would require extra care if combined with our inference algorithm.

### 5.1.3 Quantitative evaluation

When the above roadblocks are cleared up, it should be possible to perform a larger-scale quantitative analysis of inference results. Such an analysis is essential for showing how useful our inference algorithm is in practice.

The quantitative analysis should have two goals: showing that there are packages for which our improved inference algorithm produces significantly better results, but also that the unsoundness of our technique does not produce too many false positives in general.

We suggest to evaluate our algorithm on two groups of libraries: one group where each library is selected because they show exceptional results, which means that either our algorithm is very effective or that it produces too many false positives, and one group where the libraries are a representative random sample, which can show how useful our algorithm is in general.

### 5.1.4 Local or Hybrid Identification of Structure Fields

In this thesis, we have only considered the global perspective on struct semantics. It might be useful to explore local or at least hybrid semantics, where the same structure type can be used for different purposes. We believe that an inference algorithm based on the local perspective can be made completely sound, but we expect that it would only improve inference in very rare circumstances.

Between the two extremes of the global and local approaches to identifying structs, there is a possibility of grouping the usages of structs and identifying structs in the same group. These groupings could be based on an inferred protocol of the functions in which the structs occur. Another hybrid approach would be to use a more local identification of structure fields for private functions, because there is a good chance that those have more preconditions.

### 5.1.5 Two-Sided Analysis

Another extension of nullability inference would be to infer both the nullability and the non-nullability of function arguments. For example, if a pointer is assigned null then it must be nullable. In this way, we can show library authors for which arguments we

are certain that they are nullable, and for which arguments we are certain that they are non-nullable, and leave choice for the rest of the arguments up to them.

This idea can be further extended by considering individual pieces of evidence with varying levels of confidence. A pointer being assigned null is very strong evidence that it is nullable and a value being dereferenced is very strong evidence that it is non-nullable. However, there are some signs in between: if a pointer is compared to null then it is probably nullable, but not necessarily. Similarly, if our inference through structure fields with the global identification of structure fields infers that a pointer is non-nullable, then it is probably non-nullable, but not certainly.

All of this information could be summarised and presented to authors of library bindings in an interactive tool with defaults. That enables the authors to make informed decisions on which cases they should manually inspect.

# Appendix A

# Renovating the analysis suite

A significant part of our work had to be spent on renovating the analysis suite. Since the last major developments by Ravitch in 2013, there have been many LLVM and GHC releases. When starting our work it was no longer possible to compile the code in current development environments, let alone that it could be used by authors of library bindings. This renovation is an essential if this work is ever to be used in practice.

## A.1 Glasgow Haskell Compiler

Ravitch wrote his Haskell packages for GHC 7.6. Since then, there have been many new developments in GHC which is now at version 8.10. Here, we give a brief overview of the changes to the analysis suite that were required to update to this new version of GHC.

One change that led to many conflicts is the `Semigroup` superclass of `Monoid` in GHC 8.4. A monoid is any data type that has at least one element, called `mempty`, and a function that combines values of that type, called `mappend`, which must satisfy the laws that `mempty` is the left and right identity of the `mappend` function and that `mappend` is associative. There are some data types that do not have an empty element, but we would still like to provide a general interface for combining values of that type. For those data types you can implement a `Semigroup` instance instead. The `Semigroup` type class existed in a library, which is less widespread than the standard library and having a completely unrelated class leads to code duplication and a higher possibility of inconsistencies. It is clear that this change has advantages, but the main drawback is backwards incompatibility. Since GHC 8.4, every `Monoid` instance now needs an accompanying `Semigroup` instance. Luckily, GHC will quickly point out places where this needs to be done and adding such a `Semigroup` instance is always very easy.

To be able to compile the 'hbgl-experimental' package it was required to enable the `ConstrainedClassMethods` language extension. This extension lifts a restriction that has always been present in Haskell, but GHC only started enforcing it strictly since

version 8.0.1 [1].

The 'sbv' package that can solve satisfiability modulo theories problems has seen many updates, but it only required fixing renamed functions to make the analysis suite compatible with the latest version.

## A.2 LLVM

Updating to GHC 8.10 makes it possible to compile the project, but one big problem still remains: the analysis suite depends on a binding to the libLLVM library which only supports LLVM version 3. LLVM version 3 is now also quite outdated and not available in most package managers of common Linux distributions any more. So, to be able to use the project on a modern system, we also needed to make the analysis suite compatible with newer versions of LLVM. We have chosen to target LLVM 7 because it worked the best in our testing, however, as we will see in this section, the underlying is now much more flexible and it is easy to update it to newer versions of LLVM if that is required.

### A.2.1 libLLVM

First, we review how LLVM was used in the libraries that Ravitch developed in his dissertation. We have seen in Section 1.3.2 that Ravitch uses two packages for this purpose: 'llvm-base-types' and 'llvm-data-interop'. The 'llvm-base-types' package is a mostly Haskell-only implementation of the LLVM data types and the 'llvm-data-interop' package is a binding to the libLLVM C++ library provided by the LLVM project.

Linking to the LLVM C++ library in this way has certain advantages. It is relatively simple because most code is maintained by the LLVM project, and its performance should be good for the same reason. However, combining C++ and Haskell in a single project is very difficult, it requires users to have the correct version of LLVM installed on their system and the bindings themselves require a significant amount of maintenance.

There is a well-maintained package called 'llvm-hs' which provides LLVM support, but it still requires linking to C++ libraries to be able to read in the bitcode files that are produced by WLLVM and `extract-bc`.

### A.2.2 llvm-pretty

Instead, we opted to use the 'llvm-pretty' package, which is a implementation of the LLVM data types, combined with the 'llvm-pretty-bc-parser' package, which can parse LLVM bitcode into the llvm-pretty data types. These packages are written in Haskell without using the C++ library underneath, so it integrates well with the rest of the analysis suite.

[1] https://gitlab.haskell.org/ghc/ghc/-/issues/7854

One thing that these libraries were lacking was support for visibility information about defined functions. We have implemented this support and provided our patches to the upstream GitHub repositories.

This change obviates the 'llvm-base-types' and 'llvm-data-interop' packages and introduces the new libraries: 'llvm-pretty', which contains data types for representing LLVM IR and pretty-printing them, and 'llvm-pretty-bc-parser', which is a pure Haskell implementation of a parser for LLVM bitcode files.

### A.2.3 A referential data type

Instead of working with the basic LLVM IR which contains unresolved references, we want to work on a graph-like representation where all those references are resolved and replaced with direct references.

Luckily, due to Haskell's laziness it is pretty easy to represent this graph-like structure as a lazy tree structure. Using this approach we do need to be careful not to get stuck in an infinite loop traversing this tree.

To resolve the dependencies we have used the attribute grammar compiler that was developed at the Utrecht University called UUAGC [10]. Attribute grammars make it easy to move information from one place in the tree to another, which is exactly what you want to resolve references. Most of the effort was in describing the tree structure.

An important advantage of using UUAGC is that attribute grammars also make it very easy to keep track of additional information during the traversal of a data structure. In our case, we needed to generate unique numbers for each value.

# Glossary

**API** An Application Programming Interface (API) allows programs to interact with each other. 1, 2

**GHC** The Glasgow Haskell Compiler (GHC) is a state-of-the-art, open source, compiler and interactive environment for the functional language Haskell. 7, 50, 51

**GI** GObject introspection (GI) is system of metadata information for C libraries (using object-oriented functionality provided by the GObject library), which facilitates the automatic generation of bindings. 2, 3

**GUI** A graphical user interface (GUI) is a system of interactive visual components for computer software. A GUI displays objects that convey information, and represent actions that can be taken by the user. 2

**HTML** HyperText Markup Language (HTML) is declarative data format for describing the content of document, mainly used for the world wide web. 4

**IR** An Intermediate Representation (IR) is the data structure or code used internally by a compiler or virtual machine to represent source code. An IR is designed to be conducive for further processing, such as optimization and translation. 3, 4, 10, 14–16, 18, 19, 21–24, 45, 52

**JSON** JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. 3, 4, 15

**LLVM** LLVM is a collection of modular compiler technologies. 3, 4, 7, 9–11, 14–16, 18, 19, 21–24, 30–34, 36, 42, 43, 45–48, 50–52

**ML** Meta Language (ML) is a general-purpose functional programming language. It is known for its use of the polymorphic Hindley–Milner type system, which automatically assigns the types of most expressions without requiring explicit type annotations, and ensures type safety – there is a formal proof that a well-typed ML program does not cause runtime type errors. 9

**SWIG** The Simplified Wrapper and Interface Generator (SWIG) is a tool that aids in the development of bindings from higher-level languages to programs written in C and C++. 2

**UUAGC** The University of Utrecht Attribute Grammar Compiler (UUAGC) is a compiler from a custom attribute grammar specificaion language to executable Haskell code. In the attribute grammar specification you can define attributes for nodes in an abstract syntax tree and rules for how to compute them. The primary use-case is defining the semantics of programming languages. 52

**WLLVM** Whole-program LLVM (WLLVM) provides tools for building whole-program (or whole-library) LLVM bitcode files from an unmodified C or C++ source package. 3, 51

# Bibliography


[1] David M Beazley et al. "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++." In: *Tcl/Tk Workshop*. Vol. 43. 1996, p. 74.

[2] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. "Discovering Faults in Idiom-Based Exception Handling". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: Association for Computing Machinery, 2006, pp. 242–251. ISBN: 1595933751. DOI: 10.1145/1134285.1134320.

[3] Khaled ElWazeer et al. "Scalable Variable and Data Type Detection in a Binary Rewriter". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 51–60. ISBN: 9781450320146. DOI: 10.1145/2491956.2462165.

[4] Pierre Fernique and Christophe Pradal. "AutoWIG: automatic generation of python bindings for C++ libraries". In: *PeerJ Computer Science* 4 (2018), e149.

[5] *GObject Introspection*. URL: https://gi.readthedocs.io/en/latest/.

[6] Gary A. Kildall. "A Unified Approach to Global Program Optimization". In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '73. Boston, Massachusetts: Association for Computing Machinery, 1973, pp. 194–206. ISBN: 9781450373494. DOI: 10.1145/512927.512945.

[7] Benjamin Livshits et al. "In Defense of Soundiness: A Manifesto". In: *Commun. ACM* 58.2 (Jan. 2015), pp. 44–46. ISSN: 0001-0782. DOI: 10.1145/2644805.

[8] Tristan Ravitch. "Inferred Interface Glue: Supporting Language Interoperability with Static Analysis". PhD thesis. University of Wisconsin–Madison, 2013.

[9] Jaro S. Reinders. *iiglue-bundle*. https://github.com/noughtmare/iiglue-bundle. 2021.

[10] S Doaitse Swierstra, Pablo R Azero Alcocer, and Joao Saraiva. "Designing and implementing combinator languages". In: *International School on Advanced Functional Programming*. Springer. 1998, pp. 150–206.

[11] The GTK Team. *The GTK Project - A free and open-source cross-platform widget toolkit*. URL: https://www.gtk.org/.

[12] Will Thompson, Iñaki García Etxebarria, and Jonas Platte. *haskell-gi: Generate Haskell bindings for GObject Introspection capable libraries*. URL: https://hackage.haskell.org/package/haskell-gi.