MSc Thesis Computing Science
Faculty of Science

An operational semantics for the C99 restrict type qualifier

Ties Klappe
s1030293

May 8, 2024

Supervisor:
dr. Robbert Krebbers

Second assessor:
dr. Freek Wiedijk

Radboud University

Abstract

Even after more than two decades since the first introduction of the *restrict* type qualifier in the C programming language its definition in the ISO/IEC standard raises questions and confusion. For example, recently several proposals for a new definition have been submitted to the international standardization working group WG14, who maintain the standard. Restrict effectively serves as a *promise* from the programmer to the compiler that a restrict qualified pointer will not *alias* with specific other pointers under certain conditions. If the promise is broken the program is said to have *undefined behavior*, which may lead to tricky bugs which are difficult to detect.

In this work we take a different direction than (re)defining restrict in natural language. We present *Crestrict*, an *operational semantics* which refines the restrict fragment of the C-in-$\mathbb{K}$ semantics for undefined behavior by Hathhorn *et al.* Our work redevelops the original C-in-$\mathbb{K}$ semantics for restrict in a functional style. We find six programs for which we argue the semantics gives either too much or too little undefined behavior with respect to the ISO/IEC standard and/or existing compiler optimizations. From these programs we propose several refinements and integrate the new Crestrict semantics in a small C-like language. We also implement the semantics in an interpreter to be able to test a given C program for undefined behavior induced by restrict. We evaluate Crestrict under an extensive test suite and show that it gives more undefined behavior for test programs for which the C-in-$\mathbb{K}$ semantics gives too little undefined behavior and vice versa.

With this work we give an alternative resource for restrict, which we argue to be more complete than existing work. In particular, we provide a way for C programmers to systematically test whether a given program utilizing restrict induces undefined behavior. In the future the semantics could be used to prove restrict related compiler optimizations to be correct.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Robbert Krebbers. Firstly, for suggesting the research topic which turned out to be an interesting avenue to learn more about the subtleties of the C programming language, its formalizations, and compiler optimizations. Secondly, for the weekly meetings and discussions about the restrict semantics. These have been crucial for achieving the results presented in this thesis. Thank you also to Dr. Freek Wiedijk for agreeing to be the second reader and giving some insight into the ISO standard and WG14.

I would also like to thank Dr. Chris Hathhorn for sharing a more elaborate paper on the C-in-$\mathbb{K}$ semantics for undefined behavior. I am also grateful to Dr. Richard Biener, for answering several restrict-related GCC questions.

Then, I would like to thank my student colleagues in the so-called thesis factory. In particular Orpheas van Rooij, for proofreading a large portion of the thesis and providing me with useful feedback. But also all the others with whom I have shared countless coffee breaks, lunch walks, and performances as a vocal ensemble.

Lastly, I would also like to thank my family and friends for their support throughout the last eight months.

Contents

Chapter 1

Introduction

Despite being over five decades old C remains a widely used general purpose programming language, recently ranking fourth in the IEEE Spectrum 2023 index [14] and second in the TIOBE January 2024 index [43]. An important reason for this is *performance*: the language provides fine-grained control over memory and other hardware, which makes it especially suitable for the development of system software.

This control comes with responsibilities because a compiler does not ensure that a program which successfully compiles also complies with all the obligations imposed by the ISO/IEC standard. A program which violates these obligations is said to have *undefined behavior* [15, 3.4.3], "for which the International Standard imposes no requirements", *i.e.* a compiler may do *anything* for such a program. Undefined behavior (UB) effectively serves as a *contract* between the programmer and the compiler. The programmer is responsible for ensuring a program is free of undefined behavior, *e.g.* by intuitive reasoning or a static analysis/verification tool. The compiler may in turn assume a program does not contain undefined behavior, which allows it to generate more efficient code because it has to perform fewer checks. For example, dereferencing out-of-bounds array pointers is considered undefined behavior, hence the compiler does not insert bound checking instructions.

If undefined behavior leads to a run-time error, *e.g.* a termination signal from the Linux Kernel such as SIGSEGV, this can be considered a good thing as it could be detected during testing. A more tricky situation arises when a program is seemingly running correctly, but produces an "incorrect" result due to a bug inducing undefined behavior. Wang *et al.* [45] find that such bugs are "tricky to identify, hard to detect and understand, leading to programmers brushing them off incorrectly as GCC bugs".

One specific feature of C with a subtle semantics which may induce undefined behavior is the *restrict* type qualifier, first introduced in the ISO/IEC C99 standard [15, 6.7.3.1]. This type qualifier can only be applied to pointer types. The intended use is for a programmer to hint a compiler that only the restrict pointer will be used to access the object it points to. This allows a compiler to assume that under certain conditions specific pointers do not *alias*, *i.e.* they point to different objects. This information is used by the compiler to justify program optimizations.

Consider the function `foo` on page 4 for a small example of an optimization permitted by restrict. This function takes two restrict qualified integer pointers p and q as arguments, consecutively stores 10 via p and 11 via q, and returns the integer value p points to. Because restrict allows the compiler to assume that p and q do not alias in the context of `foo`, it knows that the store via q could not have modified the object p points to. This means that at line 5, the object where p points to must still contain the value 10. When compiling with optimization flag -O3, both GCC and Clang optimize line 5 to simply return 10 instead of loading from p due to this information. This optimization saves only a single memory load, but some more interesting examples will be presented in chapter 2.

```c
int foo(int* restrict p, int* restrict q)
{
    *p = 10;
    *q = 11;
    return *p; // Desired optimization: replace with return 10;
}
```

The conditions under which a compiler may assume specific pointers do not alias and whether a program is well-defined regard the actual *usage* of a restrict pointer. For a well-defined program, qualifying a pointer with restrict has the following (simplified) meaning: **if** a pointer expression "based on" the restrict qualified pointer is used to *access* the object it points to **and** that object is also *modified* in some way, **then** all accesses of that object must happen via pointer expressions based on the restrict qualified object. A pointer expression is said to be *based on* a restrict qualified pointer if it depends on its value. For example, in the example program above the pointer expression p is based on p and the pointer expression q is based on q. The use of restrict is a *promise* by the programmer and not checked by the compiler. If the requirements are violated, for example if p and q would alias in the example program above, the program has undefined behavior.

Problem statement

The simplified definition we described above is already quite complex, but the complete specification of restrict in the standard is even more tricky to say the least. The text is heavily technical and suffers both from the imprecision of natural language and lack of clarity on the semantics in specific contexts, *e.g.* the meaning of nested restrict pointers.

Recently, several proposals for a new definition in the standard have been submitted. Gustedt [9] gives a list of problems with the specification and points out that due to "a delicate mix up of semantic concepts the semantics of restrict is almost impossible to comprehend from the specification". MacDonald *et al.* [33, 34] identify a problem with the definition of "based on" which prohibits certain programs from being optimized.

The subtleties of the semantics are further substantiated by the limited amount of optimizations performed by compilers. In LLVM, which serves as the compiler backend for *e.g.* Clang, restrict is only supported on function arguments and therefore misses opportunities for optimizations. An ongoing effort for a more complete restrict implementation, including but not limited to restrict qualified struct members, was started in 2019 under LLVM-dev RFC 135672 [5]. GCC maintains a "meta-bug" to track issues related to restrict [2]. Most of the open issues are about missed optimizations, suggesting that GCC also ignores restrict in several places. In GCC 7.3.0 a bug was found which led to an incorrect optimization due to restrict [17] and was later resolved.

Overall, the problem is that the standard fails to adequately describe the semantics of restrict while correct usage is crucial both for avoiding undefined behavior and promoting code optimizations.

Approach

A well-established technique which rigorously describes what programs are well-defined and what program constructs induce undefined behavior is a *formal semantics*. There exists a vast landscape of formal semantics for C. Already in 1998, Norrish formalized a large fragment of C89 in the HOL4 proof assistant [38]. Several years later Leroy *et al.* created a large formalization in the Coq proof assistant for their verified CompCert compiler [25]. They prove for specific compiler optimizations that the program semantics are preserved between compiler optimization passes [26, 29]. In 2015, Krebbers presented the CH_2O formalization for a large fragment of C11 including the *strict aliasing rule*, a feature closely related to restrict [24]. More recently, Memarian presented Cerberus, an executable semantics for a large fragment of the "de facto" C11 standard [36].

Although much work has been done, restrict is omitted by most formalizations. To the best of our knowledge, the only formal semantics project for C which does incorporate restrict is C-in-$\mathbb{K}$, an executable semantics by Ellison and Roşu [6]. More specifically, the extension to their semantics by Hathhorn *et al.* [12] aims to capture the semantics of undefined behavior and also addresses restrict. However, the paper on this extension includes only a single paragraph on restrict which makes it somewhat unclear what the exact behavior in non-trivial programs is. Although their test suite contains multiple tests for restrict, they do not include some common situations in which restrict may induce undefined behavior.

The C-in-$\mathbb{K}$ implementation uses the $\mathbb{K}$-framework [40], a rewrite-based semantics framework, and spans several thousands of lines of rewrite rules and definitions. We redevelop the restrict fragment of this implementation in a functional style in order to better understand it. The redevelopment consists of an *operational semantics*, a description of the interpretation of a program as a sequence of computational steps, and an implementation of the semantics in an interpreter. We evaluate the semantics under a more extensive test set and find six programs for which we argue the semantics is incorrect. We propose refinements to the semantic domains and rules to fix these problems and test them with our interpreter. Then, we incorporate the new semantics in a small sequential C-like language and describe the language semantics as a big-step operational semantics. The language limits the variable types to `I32` (32-bit signed integers), $\texttt{Ptr}(\tau)$ and $\texttt{Array}(\tau, \texttt{n})$. Although simple, we argue this provides enough variety to define a relevant semantics for restrict. Firstly, because the inner type of a composite type is irrelevant for the restrict semantics, we have omitted other integers types, floating points and other base types. Secondly, supporting arrays provides the necessity for reasoning about objects containing multiple values, due to which we have omitted other composite types.

The operational semantics of restrict we present in this thesis uses an inherently different style than the *axiomatic style* of the standard section on restrict. In such an axiomatic description one needs to verify whether certain conditions are met in order to determine if a program is well-defined or has undefined behavior. An operational semantics is considered more intuitive and better suitable for implementation [39]. The interpreter allows one to systematically test a program for undefined behavior, which is a great advantage over both the current standard and the aforementioned proposals for a new definition in natural language. Another advantage of a formal semantics over a definition in natural language is that it provides a way to construct formal proofs which can justify optimizations permitted by restrict, although we will not do this in this thesis.

Contributions

- We redevelop the restrict fragment of the C-in-$\mathbb{K}$ semantics in a functional style. We evaluate this fragment of their semantics for C and give arguments for its incorrectness for six programs, in relation to the ISO/IEC standard and/or existing compiler optimizations (chapter 3).

- We propose refinements to fix the identified problems with the restrict semantics and integrate the new *Crestrict* semantics into a small C-like language whose meaning is described as a big-step operational semantics (chapter 4 and 5).

- We implement the new semantics in the eponymous CRESTRICT interpreter[1], which is able to detect undefined behavior induced by programs violating the restrict semantics (chapter 6).

[1] Available as Zenodo artifact [21]

Non-goals
We will not fix the definition of the standard in natural language.

Notations and conventions
The type $Opt(\tau)$ denotes the optional type. It is either *some* value of type τ, or *none* denoted by ϵ. The ? operator has the same meaning as in Rust and denotes the monadic bind for the optional type: if the result of the preceding expression had a value, retrieve that value from the *some* constructor. Otherwise, the function returns immediately, propagating the ϵ value.

Lists are denoted as cons lists, in which [] denotes the empty list, and $x : xs$ the list xs with the element x in front of it. We use lists to denote stacks as well. For example, $(x : ... : [y])$ denotes the stack with the item x on top and y at the bottom.

Maps from K to V are denoted as partial functions $K \rightharpoonup V$. The notation $M\{k \leftarrow v\}$ denotes the map M with the value $v \in V$ at key $k \in K$ (which overwrites the previous value at k if there was one). The notation $\{k \mapsto v\}$ denotes the singleton map with the value v at key k. Element retrieval is denoted by $M(k)$, which gives the value v at key k or ϵ if the map had no value at k. The notation $M_1 \setminus M_2$ denotes the set M_1 with all elements from M_2 removed (if they were in M_1).

Records use the same notation as in Stacked Borrows [19]. That is, a record is denoted as $\{\ A : \mathbb{Z}, B : \mathbb{B}\ \}$ (*e.g.* a record with members A of type $\mathbb{Z}$ and B of type $\mathbb{B}$). Updates of a record $Y := [A, B]$ are denoted as $Y' := Y$ **with** $[A := A']$ (*i.e.* Y' is the record Y with member A updated to A').

Finally, for simple tuple types such as $sl \in \mathit{SimpleLoc} := \mathit{Block} \times \mathit{Offset}$ we will assume functions which return a specific component exist with an eponymous name to the component type. For example, `block`(sl) denotes the first component of the *SimpleLoc* *sl*.

Chapter 2

Restrict by example

In chapter 1 we have already seen a simple example of an optimization permitted by restrict. To get a better intuition for the applications of restrict, we will look at some more example programs in this chapter. In section 2.1 and section 2.2 we show how programs which perform redundant memory operations (loads and stores) can be optimized using restrict. In section 2.3 we show how a potential implementation of C's standard library function `memcpy` specialized for integers can benefit from using restrict. Finally, section 2.4 discusses the official definition of restrict in the ISO/IEC C11 standard.

Sections 2.1 and 2.2 use x86 assembly code whereas section 2.3 uses RISC-V (specifically, RV32IV) assembly code. This distinction is made to simplify explaining the effect of restrict on a program, but not important for permitting optimizations (*i.e.* all compiled programs could also have been shown in the same assembly language).

2.1 Redundant load

Consider the function `foo`, taken from Fasselt's report on C keywords [7], defined on the left in figure 2.1. This function takes three integer pointers a, b and c as arguments, of which only pointer c is restrict qualified. The body of `foo` consecutively adds the value of c's pointee (the object referred to by c) to the pointees of a and b.

The first observation we make is that the pointer expression c is based on the restrict qualified object c and used to load (*i.e.* an *access*) from its pointee, which we call X. Because X is accessed through c we know that if X gets modified then all accesses must happen through pointer expressions also based on c. Secondly, the pointer expressions a and b both store to their pointees (*i.e.* they *modify* the objects they point to), and are **not** based on c. A compiler may therefore conclude that both a and b do not alias with c.

With this information, the compiler knows that the store at line 3 does not change the value of X. When the value of X is required again at line 4, it can simply reuse the previously fetched value instead of performing a new memory load. The x86 assembly produced by GCC and Clang when compiling this program without optimizations is shown on the right. Line 4 of the assembly corresponds to the second load of X. When compiled with optimization flag -O3 this line is omitted by both compilers. This optimization does not happen without restrict, which shows that it actually permits the desired optimization.

```
1 void foo (int* a, int* b, int* restrict c)
2 {
3     *a += *c; // First load of c
4     *b += *c; // Redundant reload from c
5 }
```

```
1 foo:
2 mov     eax, DWORD PTR [rdx]
3 add     DWORD PTR [rdi], eax
4 mov     eax, DWORD PTR [rdx] # Redundant
5 add     DWORD PTR [rsi], eax
6 ret
```

Listing 2.1: Redundant memory load

2.2 Redundant store

Another optimization permitted by restrict which a compiler may perform is the removal of redundant memory stores. Consider the function `foo`, inspired by Jung *et al.*'s `example3_down` function in Stacked Borrows [19], on the left in figure 2.2. This function takes a restrict qualified pointer p as argument. At line 5 the value 10 is stored into p's pointee, which we call X. Because X gets modified, all pointer expressions which also access X must also be based on p. The call to the externally defined function `bar` does not get any arguments passed, which means that it is not allowed to access X! This means that when executing line 7 the first store becomes redundant: it is overwritten without its value having been loaded in between.

The x86 assembly produced by GCC and Clang when compiling this program without optimizations is shown on the right. Line 5 of the assembly corresponds to the first store to X. When compiled with optimization flag -O3 this line is omitted by Clang, which does not happen when restrict is not present. GCC does not perform this optimization, which is fine: an implementor *may* use restrict to perform more optimizations, but performing an optimization is by no means required. In fact, the ISO/IEC definition explicitly mentions that an implementor may ignore all uses of restrict.

```
1  extern void bar();
2
3  void foo(int* restrict p)
4  {
5      *p = 10; // Redundant store
6      bar();
7      *p = 11; // Overwritten
8  }
```

```
1  foo:
2  push    rbx
3  xor     eax, eax
4  mov     rbx, rdi
5  mov     DWORD PTR [rdi], 10 # Redundant
6  call    bar
7  mov     DWORD PTR [rbx], 11
8  pop     rbx
9  ret
```

Listing 2.2: Redundant memory store

2.3 Memcpy

In Listing 2.3 a possible implementation for a function which copies n integers from `src` to `dst` is given. For architectures which have vector operations at their disposal, a modern compiler typically wants to utilize these operations to efficiently load the `src` array and store the retrieved data in the `dst` array.

LLVM (the compiler backend of Clang among others) is one such compiler, which has an "auto vectorizer" looking to vectorize parts of the code where possible. When we compile the copy function using Clang 14.0.0, the generated RV32IV assembly code indeed shows that vector instructions are utilized.

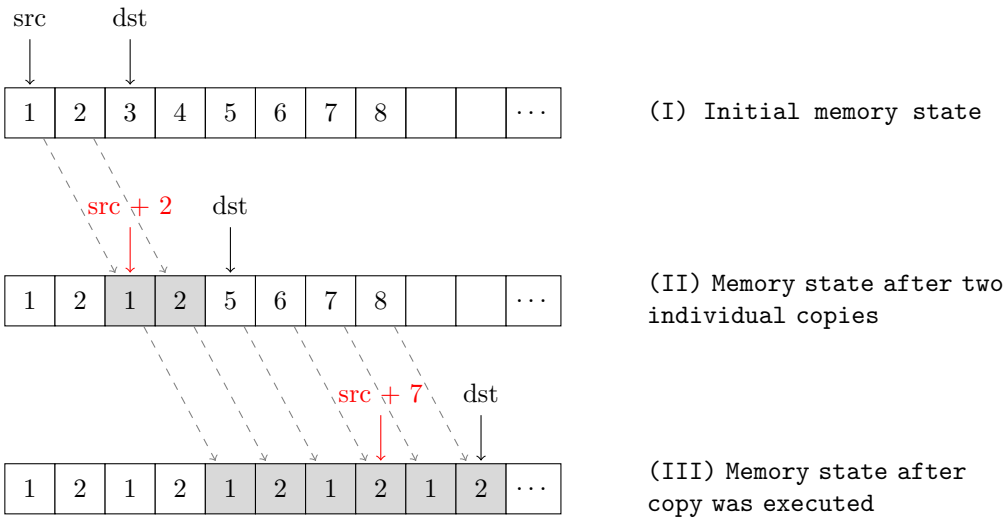
However, if `src` and `dst` point to overlapping regions in memory, such optimizations may not always be performed. An example invocation (with $n = 8$) which could potentially produce an "incorrect" output is depicted in figure 2.1: here the destination array has overlap with the source array because the `dst` pointer has the value `src` + 2.

Figure 2.1a depicts an element-wise copy: after two copies the incremented `src` pointer lies within previously changed data, and thus the next copy will now refer to this "new" data. After eight copies, the destination array has become `[1,2,1,2,1,2,1,2]`.

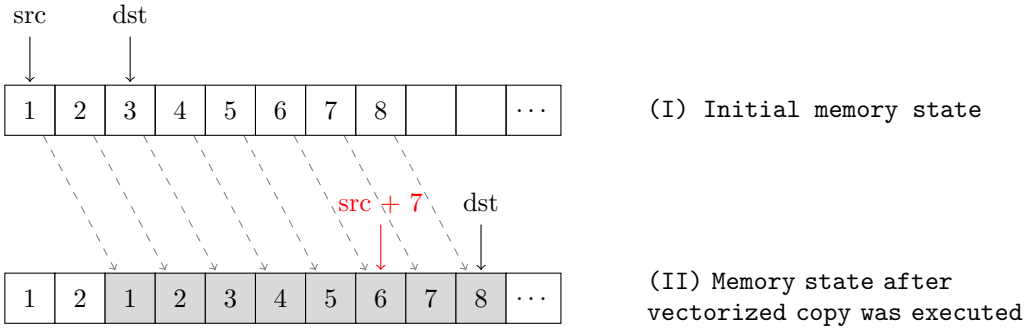
Figure 2.1b depicts a vectorized copy, with the vector size = $n = 8$. Here, the generated code will first load all 8 integer values from `src`, and then store them with a single instruction in `dst`. The destination array has become `[1,2,3,4,5,6,7,8]`.

```
1    void copy(int* /*restrict*/ src, int* /*restrict*/ dst, int n) {
2        for (int i = 0; i < n; ++i) {
3            dst[i] = src[i];
4        }
5    }
```

Listing 2.3: A possible implementation of `copy`



(a) Memory progression of element-wise copy $n = 8$



(b) Memory progression of vector copy $n = 8$

Figure 2.1: Memory progressions of element and vectorized copies

As the optimized code produces a different result than the element-wise copy, the compiler cannot just insert vector operations as it would make this specific invocation incorrect. LLVM deals with this problem by adding *runtime pointer checks*[1].

An excerpt of the generated RV32IV assembly code which includes this check for our `copy` function is given in listing 2.4. In lines 3 and 4 the end address of the `src` (`a3`) and `dst` (`a4`) arrays are computed. Lines 5 and 6 determine whether there is overlap between them, by looking at the start and end addresses of both arrays. If we assume the array from example 2.1b starts at addresses 0, the overlap is detected because $2 < 3$ (the `dst` array begins before `src` ends) and $0 < 10$ (the `src` array begins before `dst` ends). Upon detecting overlap, the function immediately jumps to `.LBB0_7`, after which it will perform the element-wise copy. Otherwise, the function does not jump and continues, utilizing the vector operations where possible.

If the programmer annotates the `src` and `dst` pointer declarations with restrict, such runtime checks are omitted by the compiler. In fact, the only difference in the generated code when the restrict type qualifiers are present, is the omission of this specific code snippet (besides some renaming of labels).

```
1     ...
2     slli    a3, a2, 2           # a3 := (n * 4)
3     add     a4, a1, a3          # a4 := dst + a3
4     add     a3, a3, a0          # a3 := a3 + src
5     sltu    a3, a1, a3          # a3 := (a1 < a3) ? 1:0
6     sltu    a4, a0, a4          # a4 := (a0 < a4) ? 1:0
7     and     a3, a3, a4          # a3 := a3 & a4
8     li      a6, 0               # a6 := 0
9     bnez    a3, .LBB0_7         # (a3 != 0) ? goto .LBB0_7
10    ...
```

Listing 2.4: Runtime overlapping check in RV32IV assembly, emitted by Clang 14.0.0

2.4 ISO/IEC definition

Section 6.7.3.1 of the ISO/IEC C11 standard [15] describes the "formal definition" of restrict. As we indicated in the introductory text, this definition can be considered quite complicated and unclear with both implementors and users of the programming language having trouble interpreting its exact meaning. Several proposals for a new or updated definition [17, 33, 34, 9] even two decades after its original publication (the definition has remained unchanged since) further substantiate this claim.

In this thesis, we stated that fixing the "formal definition" of restrict in the standard or analyzing the proposals for new definitions in natural language in detail are explicit non-goals. However, we still want the operational semantics we develop in this thesis to relate closely to the *intended* semantics of restrict in the standard (especially for contexts in which its meaning is clear). Therefore, we will explain our interpretation without reciting the entire definition. This interpretation will be used in chapter 3 to argue what the expected semantics of a program utilizing restrict with respect to the standard should be.

The standard definition starts by outlining the context of a restrict type qualifier by defining several variables. D is a declaration of an ordinary identifier. P is an object which can "be designated" as a restrict qualified pointer to type T through D. It is not explicitly defined what it means to be able to designate an object as a restrict qualified pointer through a declaration. We assume here that either some part of the type of D has type `T* restrict`, or some part of a member type of a composite type has

[1] `https://llvm.org/docs/Vectorizers.html#runtime-checks-of-pointers`

type `T* restrict`. For example, the declaration $D1$ in `int* restrict *D1 = ...;` allows to designate such a restrict qualified object through `*D1 = ...;` and the declaration $D2$ in `struct s {int* restrict p;} D2;` allows to designate such an object through `D2.p = ...;`. Finally, B denotes the block scope in which D appears (which is `main` if D has storage class extern or does not occur in a block). We refer to this block as the *restrict block*.

To demonstrate how these variables are constructed for a program, consider the function `foo` below, whose associated function block scope is named $\texttt{si}_{\texttt{foo}}$. The variables for this program are instantiated to $D =$ `int* restrict p` (*i.e.* p is the ordinary identifier), $P = p$, $T =$ `int` and $B = \texttt{si}_{\texttt{foo}}$.

```
1 // Scope si_foo
2 void foo(int* restrict p) {
3 ...
4 }
```

Furthermore, L is any lvalue which has $\&L$ "based on" P and X is the object designated by L. The idea of the "based on" definition was previously explained in the introduction. Basically, a pointer expression E is based on P if it depends directly on the value of p or on another pointer expression which is based on P. We will use the phrase "L is *derived from* from P" to refer to lvalues L which have $\&L$ based on P.

The term *lvalue* is used to refer to an expression which may occur at the left-hand side of the assignment operator and has an actual address in memory.

The official definition of "based on" in the standard is quite obscure. MacDonald *et al.* proposed a new definition for the first time in 2022 [35], which was recently superseded by "a more straightforward fix" [34].

Having set the context, the standard defines two situations which can lead to *undefined behavior* (programs for which anything may happen):

1. During execution of B, L is used to access X and X is modified by any means. Any modification of X is considered to also modify P. T shall not be const-qualified. All other lvalues used to access X must have their address based on P, or the program has undefined behavior.

2. If P is assigned the value of a pointer expression E that is based on another restricted pointer object associated with block $B2$, then either the execution of $B2$ shall begin before the execution of B or the execution of $B2$ shall end prior to the assignment.

In this thesis, we focus on a semantics which is able to detect point 1, *i.e.* the dynamic semantics of memory accesses using lvalues based on a restrict qualified pointer. The quadruple (D, P, X, B) refers to the previously defined variables and will be instantiated for specific programs later on in this thesis. Point 2 is out of scope for this thesis and will be briefly addressed in chapter 8 on future work.

Chapter 3

C-in-$\mathbb{K}$ semantics for restrict

In chapter 2, we have given an intuition for the semantics of restrict and how compilers exploit the information it provides. In this chapter, we will discuss the fragment of the C-in-$\mathbb{K}$ semantics for restrict. Based on an analysis of the K-sources[1], the generated KCC interpreter and an additional unpublished paper (an elaborate version of Hathhorn *et al.* [12]), we redeveloped their semantics in a functional style and found several issues.

In section 3.1 we give some background information on memory models for C and their design considerations. In section 3.2 we present and explain the functional C-in-$\mathbb{K}$ semantics. Finally, section 3.3 argues why some aspects of the semantics are incorrect with respect to the standard. We note that there are several representational discrepancies for either simplification or adaption of the functional style implementation, of which a complete overview can be found in related work section 7.2.

3.1 Memory model

In order to define a semantics for a language such as C, one needs to model the memory to reflect how allocations, stores and other operations affect the "memory state" of the abstract machine. The design of such a model is quite a crucial aspect for a C semantics, because more sophisticated models are able to catch various cases of undefined behavior. As a result there exist many variations of memory models, which can be classified as being concrete, abstract or somewhere in between (hybrid) [28, 37].

A typical *concrete* model, such as the one used by Norrish [38], is defined as $M \in \mathit{Mem}_{\mathit{concrete}} := [0, 2^n) \to [0, 256)$, with n being the size of the address space. This type of model is closest to how an actual computer works, *i.e.* addresses are numerical values which map to integer values between 0 and 256.

A typical *abstract* model was created for the CompCert project by Leroy *et al.* [27]. It is defined as $M \in \mathit{Mem}_{\mathit{abstract}} := \mathit{Loc} \to \mathit{Val}$, thus moving away from the underlying numerical representations. Locations are defined as $l \in \mathit{Loc} := \mathit{Block} \times \mathit{Offset}$, with $b \in \mathit{Block}$ being a reference to the memory block and $\delta \in \mathit{Offset}$ the byte offset into the block. Values are defined as $v \in \mathit{Val} ::= [0, 256) \mid \mathsf{Ptr}(l, m) \mid \mathsf{Undef}$, denoting integer values, pointer fragment m of location l and uninitialized values. The C-in-$\mathbb{K}$ memory model is based on this model. We use the term *provenance*, going back to at least Defect Report #260 [16], to describe the extra information of a pointer value, which is used to distinguish it from another pointer value that would have an equivalent numerical value under a concrete memory model. For example, tracking block references is a form of provenance to determine from which memory block a location is derived.

We stated that an abstract model can capture more undefined behavior than a concrete one. An example of undefined behavior which is captured by the CompCert memory model

[1] Available at `https://github.com/kframework/c-semantics/tree/master`

is out of bounds pointer arithmetic. In C it is illegal to access memory locations via pointer arithmetic, if the arithmetic expression would point into another object than the pointer operand [15, 6.5.6]. Because memory blocks are separated by construction in CompCert, this restriction is directly captured as arithmetic with pointers only changes the location offset (making it impossible to change the block reference). For example, consider the function `problematic` below. It compares the addresses `&x + 1` and `&y`, which could evaluate to true if an implementation allocated `x` and `y` adjacent to each other. Under the CompCert memory model it is straightforward to give undefined behavior, because the pointer arithmetic would result in a location outside the object bounds of `x`.

```
1  void problematic()
2  {
3      int x, y;
4      if (&x+1 == &y) { // Illegal pointer arithmetic
5          ...
6      }
7  }
```

An example of a limitation of the abstract model is that it is not possible to use memory addresses as integers, *e.g.* for pointer to integer casts. A *hybrid* model, such as the PVI and different PNVI models by Memarian *et al.* [37, 41], combines aspects of concrete and abstract models by giving meaning to casts between pointer and integer values while also tracking provenance.

3.2 Restrict semantics

The C-in-$\mathbb{K}$ semantics has two features which are jointly used to support restrict. Pointer values have a provenance called *bases*, which is used to track from which restrict qualified objects they are derived (1). For each location a restrict state is tracked that says which memory accesses are allowed and which are not (2). For feature (2) there is a structure of a stack (called the *restrict stack*) to track the restrict states and of a lattice which defines allowed and disallowed accesses.

By means of the preliminary grammar of figure 3.1 we introduce the domains and types necessary for explaining the restrict features of the C-in-$\mathbb{K}$ semantics in detail. The semantics itself will then be explained by elaborating on some example programs.

$$
\begin{array}{lcl}
b \in \mathit{Block}, \delta \in \mathit{Offset} & := & \mathbb{Z} \\
si \in \mathit{ScopeId} & := & \mathbb{Z} \\
sl \in \mathit{SimpleLoc} & := & \mathit{Block} \times \mathit{Offset} \\
ba \in \mathit{Base} & := & \mathit{Block} \times \mathit{ScopeId} \\
bas \in \mathit{Bases} & := & \mathit{Set}(\mathit{Base}) \\
l \in \mathit{Loc} & := & \mathit{SimpleLoc} \times \mathit{Bases} \\
rs \in \mathit{RestrictState} & ::= & \mathsf{OnlyRead}\ \mathit{bas} \\
 & | & \mathsf{Restricted}\ \mathit{bas} \\
 & | & \mathsf{Unrestricted} \\
 & | & \text{☣} \\
 & | & \bot \\
rm \in \mathit{RestrictMap} & := & \mathit{SimpleLoc} \rightharpoonup \mathit{RestrictState} \\
R \in \mathit{RestrictStack} & := & \mathit{List}(\mathit{ScopeId} \times \mathit{RestrictMap})
\end{array}
$$

Figure 3.1: C-in-$\mathbb{K}$ restrict related domains

Programs in the C-in-$\mathbb{K}$ semantics are evaluated under a restrict stack R, which is modified through scope changes and memory accesses. Locations l are represented by tuples, whose first component sl is the memory location composed of a block reference and offset pair (based on CompCert [27]). The second component of l is a set of *Bases* bas, whose elements ba are tuples (b, si) in which b represents the block reference of the restrict qualified object p from which l is derived and si the unique identifier of the scope in which p was declared. It is a *set* since restrict pointers may be assigned to each other, if the pointer being assigned to is declared in a nested scope[2].

We say that a location (or the location an lvalue exression is stored at) $(sl, \{\ldots, (b, \_), \ldots\})$ is *derived from* a restrict qualified object b if it is obtained through a dereference of a pointer expression which is *based on* b. For example, given the following code snippet `int x; int* restrict p = &x;` we have that the expression $*p$ is derived from the restrict qualified object p because p is based on p, but $\&x$ is not derived from p.

The memory relates *SimpleLoc*s to values and will be formally defined when we present the complete language in section 5.1. Importantly, every variable has an address in memory in order to support the address of (&) operator. We write $M(sl)$ to denote the value stored at sl, at the execution point the annotation is placed at. Pointer values are denoted as $\mathsf{Ptr}\ (sl, bas)$, *i.e.* they contain a *Loc*, with sl being the location pointed towards and bas the provenance of the pointer value. The aim of bas is to track the *based on* definition we previously explained in chapter 1 and section 2.4.

An example of how the set of bases transforms through assignments is given in listing 3.1. In this example, a restrict pointer p is created to the variable x at line 9. The value of p is not just the *SimpleLoc* sl_x representing x's address, but also the base $(b_p, \mathtt{si_{main}})$ to track that p is based on p. The call to `foo` at line 11 then shows how a location can have multiple bases: the value of q preserves the existing provenance but also becomes derived from the restrict pointer q. After adding this new base, the pointer value of q now has the following provenance: $\{(b_p, \mathtt{si_{main}}), (b_q, \mathtt{si_{foo}})\}$.

The restrict stack R contains tuples of scope identifiers si and restrict maps rm from simple memory locations sl to a restrict state rs. This map is used as a total function $\mathit{SimpleLoc} \to \mathit{RestrictState}$ by the following definition: $rm(sl) = rm(sl)$ if sl is a key in rm, and $\bot$ otherwise. A tuple (si, rm) is pushed onto the restrict stack when the program enters a new scope si and popped off the restrict stack when a scope is exited. Consequently, the top item of R always represents the deepest scope that is currently active.

Whenever a program accesses a memory location (sl, bas) the restrict state representing the access type ($\mathsf{OnlyRead}\ bas$ for loads and $\mathsf{Restricted}\ bas$ for stores) is attempted to be *joined* with the current restrict state of sl in the top restrict map of R. The symmetric join ($\sqcup$) operator (defined in figure 3.2) determines what the new restrict state of sl in this map after the join is. If a join resulted in $\mathsf{Unrestricted}$, multiple loads from a location occurred within the same scope but the location provenance (*i.e.* bases) differed. This means the program may now only load from this location (with any pointer) and stores are completely forbidden. If a join resulted in ☣, the accesses have violated the semantics of restrict and the program has undefined behavior. As a result the program execution is aborted and an error is returned.

[2]This corresponds to example 4 of the section on restrict in the standard [15, 6.7.3.1]

```
1  // Scope si_foo
2  void foo(int* restrict q) { // q is stored at sl_q = (b_q, 0)
3      ...                     // M(sl_q) = Ptr (sl_x, {(b_p, si_main), (b_q, si_foo)})
4  }
5
6  // Scope si_main
7  int main() {
8      int x = 0;              // x is stored at sl_x
9      int* restrict p = &x;   // p is stored at sl_p = (b_p, 0)
10                             // M(sl_p) = Ptr (sl_x, {(b_p, si_main)})
11     foo(p);                 // The value of p is passed to q
12     return 0;
13 }
```

Listing 3.1: Designating and transferring bases provenance

$$
\begin{aligned}
\mathsf{OnlyRead}\ bas \sqcup \mathsf{OnlyRead}\ bas &= \mathsf{OnlyRead}\ bas \\
\mathsf{OnlyRead}\ bas \sqcup \mathsf{Restricted}\ bas &= \mathsf{Restricted}\ bas \\
\mathsf{Restricted}\ bas \sqcup \mathsf{Restricted}\ bas &= \mathsf{Restricted}\ bas \\
\mathsf{OnlyRead}\ bas \sqcup \mathsf{OnlyRead}\ bas' &= \mathsf{Unrestricted} \\
\mathsf{OnlyRead}\ bas \sqcup \mathsf{Unrestricted} &= \mathsf{Unrestricted} \\
\mathsf{Unrestricted} \sqcup \mathsf{Unrestricted} &= \mathsf{Unrestricted} \\
\mathsf{OnlyRead}\ bas \sqcup \mathsf{Restricted}\ bas' &= \text{☣} \\
\mathsf{Restricted}\ bas \sqcup \mathsf{Restricted}\ bas' &= \text{☣} \\
\mathsf{Restricted}\ bas \sqcup \mathsf{Unrestricted} &= \text{☣} \\
lhs \sqcup \text{☣} &= \text{☣} \\
\bot \sqcup rhs &= rhs
\end{aligned}
$$

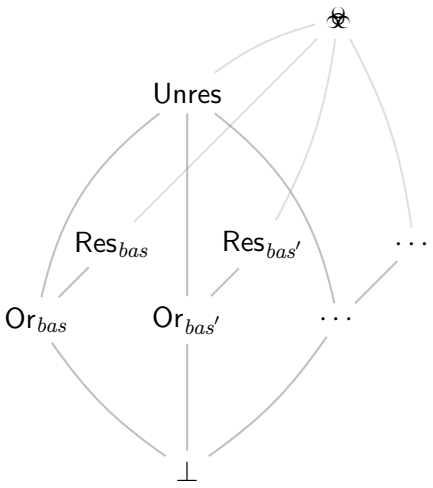


Figure 3.2: Auxiliary symmetric operation $\sqcup$ which joins two restrict states, where $bas \neq bas'$ and the corresponding Hasse diagram of the set $\{\text{☣}, \mathsf{Unres} = \mathsf{Unrestricted}, \mathsf{Res}_{bas} = \mathsf{Restricted}\ bas, \mathsf{Res}_{bas'} = \mathsf{Restricted}\ bas', \mathsf{Or}_{bas} = \mathsf{OnlyRead}\ bas, \mathsf{Or}_{bas'} = \mathsf{OnlyRead}\ bas', \bot\}$

$$
\begin{aligned}
\texttt{filter\_bases} &: \mathit{RestrictState} \times \mathit{ScopeId} \to \mathit{RestrictState} \\
\texttt{filter\_bases}\ (\mathsf{OnlyRead}\ bas, si) &= \mathsf{OnlyRead}\ (\texttt{filter\_bases}'(bas, si)) \\
\texttt{filter\_bases}\ (\mathsf{Restricted}\ bas, si) &= \mathsf{Restricted}\ (\texttt{filter\_bases}'(bas, si)) \\
\texttt{filter\_bases}\ (\mathsf{Unrestricted}, \_) &= \mathsf{Unrestricted}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{filter\_bases}' &: \mathit{Bases} \times \mathit{ScopeId} \to \mathit{Bases} \\
\texttt{filter\_bases}'\ (\emptyset, \_) &= \emptyset \\
\texttt{filter\_bases}'\ ((\_, si) : bas, si) &= \texttt{filter\_bases}'(bas, si) \\
\texttt{filter\_bases}'\ ((b, si') : bas, si) &= (b, si') : \texttt{filter\_bases}'(bas, si)
\end{aligned}
$$

Figure 3.3: Auxiliary functions for filtering location bases

Example of an evaluation under the C-in-$\mathbb{K}$ semantics

To demonstrate how the semantics detects undefined behavior, reconsider the introductory example in listing 3.2. We have added a client `main` which invokes `foo` with aliasing arguments, *i.e.* both p and q point to x. The code is annotated to show how applying the C-in-$\mathbb{K}$ semantic rules mutate the state. For simplicity, only a single location is tracked.

```
1  // Scope si_foo: R = [(si_foo, ∅), (si_main, ∅)]
2  int foo(int* restrict p, int* restrict q) {
3      // p is stored at sl_p = (b_p, 0). M(sl_p) = Ptr (sl_x, {(b_p, si_foo)})
4      // q is stored at sl_q = (b_q, 0). M(sl_q) = Ptr (sl_x, {(b_q, si_foo)})
5
6      *p = 10;    // Store via p,
7      // OK: Restricted {(b_p, si_foo)} ⊔ ⊥ = Restricted {(b_p, si_foo)}
8      // R = [(si_foo, {sl_x ↦ Restricted {(b_p, si_foo)}}), (si_main, ∅)]
9
10     *q = 11;  // Store via q,
11     // UB: Restricted {(b_q, si_foo)} ⊔ Restricted {(b_p, si_foo)} = ☣
12     return *p;
13 }
14
15 // Scope si_main: R = [(si_main, ∅)]
16 int main() {
17     int x; // x is stored at sl_x
18     return foo(&x, &x);
19 }
```

Listing 3.2: Annotated load optimization undefined behavior example

When starting the execution of the program with the invocation of `main` at line 15, there is a single map in the restrict stack: $(\texttt{si}_{\texttt{main}}, \emptyset)$. This means that the scope identifier assigned to the scope of `main` is $\texttt{si}_{\texttt{main}}$, and the restrict map is empty (as no accesses have occurred). The local variable x is stored at the location sl_x.

At the start of `foo` at line 1 a new scope is entered, reflected in the restrict stack by pushing the item $(\texttt{si}_{\texttt{foo}}, \emptyset)$. Because p and q are restrict pointers, the corresponding bases $(b_p, \texttt{si}_{\texttt{foo}})$ and $(b_q, \texttt{si}_{\texttt{foo}})$ are added as *provenance* to the pointer values they store. At line 8 a store via p occurs, so the restrict state of sl_x is updated to $\mathsf{Restricted}\ \{(b_p, \texttt{si}_{\texttt{foo}})\}$. Then, when the next store access occurs via q at line 11, the current state is joined with $\mathsf{Restricted}\ \{(b_q, \texttt{si}_{\texttt{foo}})\}$. As $\{(b_q, \texttt{si}_{\texttt{foo}})\} \neq \{(b_p, \texttt{si}_{\texttt{foo}})\}$, joining these states results in ☣, *i.e.* the program is assigned undefined behavior.

Accesses in different scopes

In the presented introductory example, all accesses leading up to the invalid restrict state occur in the scope $\texttt{si}_{\texttt{foo}}$. To detect violations that occur due to nested scopes (*e.g.* function calls), the C-in-$\mathbb{K}$ semantics does a *deferred check* when a scope is exited. The idea is that by iterating over the restrict map rm_m which is popped off the restrict stack R, all memory locations sl are found that were accessed within the scope si_m. Now, if $\texttt{block}(sl) \in \mathit{locals}$ (where *locals* contains the memory blocks of all variables local to that scope) nothing needs to happen, as these variables cannot have been accessed in scopes that started prior to si_m. Otherwise, states are joined in order to propagate the new restrict state onto rm_n that is now on top of R. This join does one additional operation compared to a regular join: a restrict state in rm_m is passed through the `filter_bases` function (defined in figure 3.3) to remove all bases going out of scope. Then, the restrict map is updated if the join did not result in ☣: $rm_n\{sl \leftarrow (\texttt{filter\_bases}(rm_m(sl), si_m)) \sqcup rm_n(sl)\}$. An example program that is assigned undefined behavior by these operations is given in listing 3.3.

```
1  // Scope si_foo, R = [(si_foo, ∅), (si_main, {sl_x ↦ OnlyRead {(b_p, si_main)}})]
2  void foo(int* q) {
3      *q = 0; // Store via q,
4              // R = [(si_foo, {sl_x ↦ Restricted ∅}), (si_main, {sl_x ↦ OnlyRead {(b_p, si_main)}})]
5  }
6
7  // Scope si_main, R = [(si_main, ∅)]
8  int main() {
9      int x = 5;              // x is stored at sl_x
10     int* restrict p = &x; // p is stored at sl_p = (b_p, 0). M(sl_p) = Ptr (sl_x, {(b_p, si_main)})
11     int y = *p; // Load via p, R = [(si_main, {sl_x ↦ OnlyRead {(b_p, si_main)}})]
12
13     foo(&x); // As sl_x is non-local to foo, join the state depicted at line 4
14              // UB: (filter_bases(Restricted ∅, si_foo)) ⊔ OnlyRead {(b_p, si_main)} = ☣
15     return y;
16 }
```

Listing 3.3: Undefined behavior through different scopes

In the function `main` the restrict pointer p is used to store to the location sl_x, changing the restrict state of sl_x in the restrict map of $\mathtt{si}_{\mathtt{main}}$ to Restricted $\{(b_p, \mathtt{si}_{\mathtt{main}})\}$. The call to `foo` lets the location sl_x escape via the expression $\&x$, which is **not** based on p. The store via q in `foo` changes the restrict state of sl_x in the restrict map of $\mathtt{si}_{\mathtt{foo}}$ to Restricted $\emptyset$, (because the pointer value provenance is the empty set).

When `foo` exits, the restrict map $rm_{\mathtt{foo}}$ is popped from R. The only item in the domain of $rm_{\mathtt{foo}}$ is sl_x, which is non-local to `foo`. This means that the state of sl_x in $rm_{\mathtt{foo}}$ is filtered and then joined with the state of sl_x in $rm_{\mathtt{main}}$. This leads to undefined behavior: $(\mathtt{filter\_bases}\ (\text{Restricted}\ \emptyset, \mathtt{si}_{\mathtt{foo}})) \sqcup \text{OnlyRead}\ \{(b_p, \mathtt{si}_{\mathtt{main}})\} = ☣$.

3.3 Incorrectness

We argue that some aspects of the C-in-$\mathbb{K}$ semantics for the restrict type qualifier are incorrect, by providing four example programs that we consider incorrectly accepted (too little UB, TLU) and two example programs that we consider wrongfully rejected (too much UB, TMU). For each of these programs we explain how the C standard applies, and whether it is clear if the program should be accepted or rejected according to the text. When applicable, we will instantiate the quadruple (declaration D, object P, designated object X, restrict block B) as previously defined in section 2.4. Furthermore, we will abbreviate `int* restrict` to IRP (as if `typedef int* restrict IRP;` is included in the program). Finally, we limit program annotations to the parts relevant for explaining the problem (*e.g.* the addresses of unrelated variables and restrict states of unrelated locations are omitted).

3.3.1 (TMU) Aliasing loads

A problem arises when a program loads from a memory location via two different restrict pointers in the same scope (resulting in Unrestricted), and this state is joined with the Restricted state from a prior scope. In the example depicted in listing 3.4, a restrict pointer p is created to y at line 15. This pointer gets aliased in scope $\mathtt{si_h}$ by the invocation of `h`, which is allowed because all accesses of sl_y in scope $\mathtt{si_h}$ are loads.

The loads occur at line 6 via pointers r and s, and regardless of the evaluation order the resulting restrict state of sl_y within scope $\mathtt{si_h}$ becomes Unrestricted, as $\{(b_p, \mathtt{si_{main}}), (b_r, \mathtt{si_h})\} \neq \{(b_p, \mathtt{si_{main}}), (b_s, \mathtt{si_h})\}$.

When scope $\mathtt{si_h}$ ends (after termination of `h`), the restrict states of sl_y in scope $\mathtt{si_h}$ and scope $\mathtt{si_{main}}$ have to be joined. Line 8 shows the restrict stack at this point of the execution. But this is problematic, as merging the states incorrectly assigns undefined behavior to the program (line 24).

```
1  // Scope si_h, R = [(si_h, ∅), (si_main, {sl_y ↦ Restricted {(b_p, si_main)}})]
2  void h(int* q, int* restrict r, int* restrict s)
3  {
4      // r is stored at b_r. M(b_r) = Ptr (sl_y, {(b_p, si_main), (b_r, si_h)})
5      // s is stored at b_s. M(b_s) = Ptr (sl_y, {(b_p, si_main), (b_s, si_h)})
6      *q = *r + *s; // Two load accesses via r and s:
7              // OnlyRead {(b_p, si_main), (b_r, si_h)} ⊔ OnlyRead {(b_p, si_main), (b_s, si_h)} = Unrestricted
8              // R = [(si_h, {sl_y ↦ Unrestricted}), (si_main, {sl_y ↦ Restricted {(b_p, si_main)}})]
9  }
10
11 // Scope si_main, R = [(si_main, ∅)]
12 int main()
13 {
14     int x, y;              // y is stored at sl_y
15     int* restrict p = &y; // p is stored at sl_p = (b_p, 0). M(sl_p) = Ptr (sl_y, {(b_p, si_main)})
16
17     *p = 0; // Store via p,
18             // R = [(si_main, {sl_y ↦ Restricted {(b_p, si_main)}})]
19
20     // Defined behavior because no modifications via p happen
21     h(&x, p, p);
22
23     // When h terminates, join states as sl_y is non-local to h:
24     // False negative: (filter_bases (Unrestricted, si_h)) ⊔ Restricted {(b_p, si_main)} = ☣
25
26     return 0;
27 }
```

Listing 3.4: Joining Unrestricted and Restricted between scopes (TMU)

Standard

- $(D_1, P_1, X_1, B_1) = (\mathsf{IRP}\ p, sl_p, sl_y, \mathtt{si_{main}})$
- $(D_2, P_2, X_2, B_2) = (\mathsf{IRP}\ r, sl_r, sl_y, \mathtt{si_h})$
- $(D_3, P_3, X_3, B_3) = (\mathsf{IRP}\ s, sl_s, sl_y, \mathtt{si_h})$

There are two places in the standard that support our claim that this program is well-defined. The first place is the section stating "and X is also modified (by any means)..." [15, 6.7.3.1, p4]. Such a modification does not occur during the execution of $\mathtt{si_h}$, so no undefined behavior occurs via r and s. In the scope $\mathtt{si_{main}}$, such a modification *does* occur via p, but because all accesses during the execution of $\mathtt{si_{main}}$ happen via lvalues derived from p this is well-defined.

Secondly, the program can be considered a simplified version (we use pointers to integers instead of arrays) of example 3 of the section on restrict in the standard [15, 6.7.3.1]. The purpose of this example is the same as ours, *i.e.* to emphasize that loads via aliasing restrict pointers are well-defined if the object is not modified.

3.3.2 (TLU) Nested restrict pointers

A particular construct with a subtle semantics is a nested restrict pointer. In the example depicted in listing 3.5, a compiler may want to optimize line 10 to `return 10;` (in fact, GCC 13.02 with optimization level 3 does this). However, the client `main` that invokes `foo` makes p and q and thus $*p$ and $*q$ alias.

The problem with the C-in-$\mathbb{K}$ semantics is that it does not update the restrict state of sl_{xp} to Restricted when writing to sl_x via the lvalues $**p$ and $**q$ (which both go through the restrict qualified object sl_{xp}). This should happen due to a subtle subclause of the standard. As a result, it fails to give undefined behavior.

```
1  // Scope si_foo
2  int foo(int *restrict *restrict p, int *restrict *restrict q)
3  {
4      **p = 10; // Load via p, store via *p = xp
5      // R = [(si_foo, {sl_x ↦ Restricted {(b_xp, si_foo)}, sl_xp ↦ OnlyRead {(b_p, si_foo)}}), (si_main, ∅)]
6
7      **q = 11; // Load via q, store via *q = xp
8      // R = [(si_foo, {sl_x ↦ Restricted {(b_xp, si_foo)}, sl_xp ↦ Unrestricted}), (si_main, ∅)]
9
10     return **p; // Optimization candidate
11 }
12
13 // Scope si_main
14 int main() {
15     int x = 0;     // x is stored at sl_x
16     int* xp = &x; // xp is stored at sl_xp = (b_xp, 0)
17
18     int res = foo(&xp, &xp);
19
20     return 0;
21 }
```

Listing 3.5: Undetected undefined behavior through nested restrict pointers (TLU)

Standard

- $(D_1, P_1, X_1, B_1) = (\mathsf{IRPRP}\ p,\ p = sl_p, sl_{xp}, \mathtt{si}_{\mathtt{main}})$
- $(D_2, P_2, X_2, B_2) = (\mathsf{IRPRP}\ p, *p = sl_{xp}, sl_x, \mathtt{si}_{\mathtt{main}})$
- $(D_3, P_3, X_3, B_3) = (\mathsf{IRPRP}\ q,\ q = sl_q, sl_{xp}, \mathtt{si}_{\mathtt{main}})$
- $(D_4, P_4, X_4, B_4) = (\mathsf{IRPRP}\ q, *q = sl_{xp}, sl_x, \mathtt{si}_{\mathtt{main}})$

The following excerpt describes why this program has undefined behavior: "Every access that modifies X shall be considered also to modify P, for the purposes of this subclause." [15, 6.7.3.1, p4].

In the example program, this means that $P_2 = P_4 = sl_{xp}$ is considered modified, but not by lvalues derived from the same restrict qualified object: once via sl_p and once via sl_q, thus inducing undefined behavior[3].

3.3.3 (TLU) Semantic preservation under inlining

Due to the deferred restrict check between scopes (previously explained in section 3.2), the place where undefined behavior is assigned to a program may be too late. More specifically, we can write a program in which the semantics of restrict is not preserved under inlining. Consider a slightly modified version of example 3.3 presented in listing 3.6: we have added a non-terminating while loop[4] at line 7. This program is fine according to the semantics: because $\mathtt{si}_{\mathtt{foo}}$ never terminates, no join is performed that would result in ☣.

```
1  // Scope si_foo
2  void foo(int* q) {
3      *q = 0;          // Store via q,
4                       // R = [(si_foo, {sl_x ↦ Restricted ∅}), (si_main, {sl_x ↦ OnlyRead {(b_p, si_main)}})]
5                       // should eventually result in ☣ because q is not based on p
6
7      while (1) {}     // Defined behavior because this loop is never exited
8  }
9
10 // Scope si_main
11 int main() {
12     int x = 5;               // x is stored at sl_x
13     int* restrict p = &x;    // p is stored at sl_p = (b_p, 0). M(sl_p) = Ptr (sl_x, {(b_p, si_main)})
14
15     int y = *p;     // Load via p, R = [(si_main, {sl_x ↦ OnlyRead {(b_p, si_main)}})]
16
17     foo(&x);        // But as si_foo never exits the semantics never gives ☣
18                     // because (filter_bases(Restricted ∅, si_foo)) ⊔ OnlyRead {(b_p, si_main)}
19                     // is never performed
20
21     return y;
22 }
```

Listing 3.6: Undetected undefined behavior through different scopes (TLU)

Now, the same program with `foo` inlined into `main` *does* give undefined behavior. This version is presented in section 3.7. Because the restrict states of sl_x are directly joined, the semantics detects undefined behavior at the right place. From this example we may conclude that function inlining does not preserve the restrict semantics.

[3] This is also corroborated by GCC: `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=14192#c8`

[4] This non-terminating while-loop is allowed by the standard, as its controlling expression is a constant expression [15, 6.8.5, p6].

Standard

- (D, P, X, B) is (IRP $p, sl_p, x, \texttt{si}_{\texttt{main}}$)

The restrict specification does not distinguish between conflicting accesses in deeper scopes of B and conflicting accesses within the same scope. So in the program of listing 3.6 it is irrelevant whether the conflicting access occurs in the deeper scope $\texttt{si}_{\texttt{foo}}$ or in the same scope $\texttt{si}_{\texttt{main}}$, because $\texttt{si}_{\texttt{foo}}$ is part of the scope $\texttt{si}_{\texttt{main}}$.

```
1  // Scope si_main
2  int main() {
3      int x = 5;
4      int* restrict p = &x;
5
6      int y = *p;
7
8      int* q = &x;
9      *q = 0;          // OnlyRead {(b_p, si_main)} ⊔ Restricted ∅ = ☣
10                      // Undefined behavior!
11
12     while (1) {}
13
14     return y;
15 }
```

Listing 3.7: Detected undefined behavior when foo is inlined

3.3.4 (TLU) Indistinguishable restrict pointers

Another thing we can do with the current semantics for assigning bases is creating different restrict pointers to the same location, that also have the same provenance. This is problematic: provenance is meant to distinguish restrict pointers to the same location, so having the same base for two different restrict pointers is completely contradictory.

An example of such a problematic program is given in listing 3.8. This program creates an array a containing two aliased restrict pointers to x at line 4. Because both pointers share the same *Block* b_a, both pointers have the same value: Ptr $(sl_x, \{(b_a, \texttt{si}_{\texttt{main}})\})$. The actual problem with the semantics occurs at line 7: after a store via the restrict pointer in $a[0]$ all other accesses in this scope are required to also happen via pointer $a[0]$ (or pointers based on it): the store via the restrict pointer $a[1]$ should thus give undefined behavior. But because the two pointers have the same provenance, the join operation accepts the store via $a[1]$ at line 7 and therefore fails to account for undefined behavior.

We did not manage to find a compiler which performs optimizations for an array of restrict qualified pointers. However, GCC does utilize restrict qualified struct members for optimizations, a construct which suffers from the same problem in the C-in-$\mathbb{K}$ semantics[5].

```
1  // Scope si_main
2  int main() {
3      int x; // x is stored at sl_x
4      int* restrict a[2] = {&x, &x}; // a is stored at b_a
5                                     // M((b_a, 0)) = M((b_a, 1)) = Ptr (sl_x, {(b_a, si_main)})
6      *(a[0]) = 10;    // R = [(si_main, {sl_x ↦ Restricted {(b_a, si_main)}})]
7      *(a[1]) = 11;    // Restricted {(b_a, si_main)} ⊔ Restricted {(b_a, si_main)} = Restricted {(b_a, si_main)}
8                       // R = [(si_main, {sl_x ↦ Restricted {(b_a, si_main)}})]
9  }
```

Listing 3.8: Different restrict pointers with the same base (TLU)

[5] An example program which is optimized by GCC and has undefined behavior is given in `UB/unsupported/struct-aliasing.c` of the artifact [21]

Standard

- $(D, P_1, X_1, B_1) = (\mathsf{IRP}\ a[2], a[0] = (b_a, 0), sl_x, \mathtt{si_{main}})$
- $(D, P_2, X_2, B_2) = (\mathsf{IRP}\ a[2], a[1] = (b_a, 1), sl_x, \mathtt{si_{main}})$

The standard text does not give any specifics on the semantics of an array of restrict pointers. In the instantiation of P_1 and P_2 above, we say that the declaration D of the ordinary identifier a allows to designate *two* restrict objects, namely the array element objects $(b_a, 0)$ and $(b_a, 1)$. As these element objects are different restrict qualified objects, the program should be assigned undefined behavior.

3.3.5 (TMU) Returning a restrict pointer

When returning a restrict pointer, there are no special operations happening for the value that is returned. This means that the provenance of such a pointer value is unaffected. This allows us to write programs in which locations are derived from restrict pointers that are not in scope. An example of such an erroneous program is given in listing 3.9. The function `as_mut_ptr`[6] simply returns the restrict pointer parameter v as a normal pointer. But because the parameter is restricted, the value being returned now contains the base of v. By invoking this function twice with the same argument and assigning the result to two different pointers p_1 and p_2, we can now create a situation in which the semantics assigns undefined behavior to this program by performing two stores. At the second store (at line 20) we now get ☣, whereas this should have been accepted because p_1 and p_2 are allowed to alias each other because `main` is not part of the execution of any restrict block.

Similarly to returning restrict pointer values, we can let any global pointer (which is not restrict qualified) get based on a restrict pointer with a smaller scope[7]. This results in the same problem, *i.e.* at a certain time in the program the bases are no longer in scope, making the provenance invalid.

```
1  // Scopes si_as_mut_ptr_1, si_as_mut_ptr_2
2  int* as_mut_ptr(int* restrict v) {  // v is stored at sl_{v_1} = (b_{v_1}, 0) (invocation 1)
3                                      // v is stored at sl_{v_2} = (b_{v_2}, 0) (invocation 2)
4                                      // M(sl_{v_1}) = Ptr (sl_a, {(b_{v_1}, si_as_mut_ptr_1)})
5                                      // M(sl_{v_2}) = Ptr (sl_a, {(b_{v_2}, si_as_mut_ptr_2)})
6      return v;
7  }
8
9  // Scope si_main
10 int main() {
11     int a;                          // Stored at sl_a
12
13     int* p1;
14     int* p2;
15
16     p1 = as_mut_ptr(&a);          // M(p_1) = Ptr (sl_a, {(b_{v_1}, si_as_mut_ptr_1)})
17     p2 = as_mut_ptr(&a);          // M(p_2) = Ptr (sl_a, {(b_{v_2}, si_as_mut_ptr_2)})
18
19     *p1 = 0;          // R = [(si_main, {sl_a ↦ Restricted {(b_{v_1}, si_as_mut_ptr_1)}})]
20     *p2 = 0;          // UB: Restricted {(b_{v_1}, si_as_mut_ptr_1)} ⊔ Restricted {(b_{v_2}, si_as_mut_ptr_2)} = ☣
21
22     return 0;
23 }
```

Listing 3.9: Returning a restrict pointer (TMU)

[6] Inspired by `https://perso.crans.org/vanille/treebor/shared.html` (a restrict pointer is required but not actually used for accesses)

[7] Example `DB/thesis/thesis-filtering-bases-global.c` of the artifact [21]

Standard

- $(D, P, X, B_1) = (\mathsf{IRP}\ v, sl_{v_1}, sl_a, \mathtt{si}_{\mathtt{as\_mut\_ptr_1}})$
- $(D, P, X, B_2) = (\mathsf{IRP}\ v, sl_{v_2}, sl_a, \mathtt{si}_{\mathtt{as\_mut\_ptr_2}})$

The object designated by the restrict qualified objects sl_{v_1}, sl_{v_2} does not get modified within their associated scopes, $\mathtt{si}_{\mathtt{as\_mut\_ptr_1}}$, $\mathtt{si}_{\mathtt{as\_mut\_ptr_2}}$. It does get modified within the scope $\mathtt{si}_{\mathtt{main}}$, but as this scope is not part of the execution of a restrict block this program is well-defined.

3.3.6 (TLU) Call to free

The example program in listing 3.10 shows how the dynamically allocated object b_v is freed via the aliasing pointer r at line 3, which is not based on the restrict pointer q. Although the standard does not specifically mention the effect of such an operation on the semantics of restrict, simply ignoring it could be problematic.

The program in the current form does not lead to any problems when it gets compiled. However, as q is a restrict pointer, the compiler is allowed to assume that the assignment via q at line 12 does not affect the result of the call to `bar` at line 13. This means that reordering these lines would be allowed for the compiler. However, by doing so, the object b_v would be freed before it gets assigned to via q, thus leading to undefined behavior.

```
1  // Scope si_bar
2  void bar(int* s) {      // s is stored at sl_s
3      free(s);            // The restrict state is unaffected:
4                          // R = [(si_bar, ∅), (si_foo, {sl_v ↦ Restricted {(b_q, si_foo)}}), (si_main, ∅)]
5  }
6
7  // Scope si_foo
8  void foo(int* restrict q, int* r) {
9                  // q is stored at sl_q = (b_q, 0), r is stored at sl_r = (b_r, 0)
10                 // M(sl_q) = Ptr (sl_v, {(b_v, si_foo)}), M(sl_r) = Ptr (sl_v, ∅)
11
12     *q = 5;     // R = [(si_foo, {sl_v ↦ Restricted {(b_q, si_foo)}}), (si_main, ∅)]
13     bar(r);     // This is fine according to the C-in-𝕂 semantics,
14                 // even though b_v has been deallocated via a pointer
15                 // not based on b_q.
16 }
17
18 // Scope si_main
19 int main() {
20     int* p = malloc(sizeof(int));   // p is stored at sl_p, the allocated object at sl_v
21     foo(p, p);
22
23     return 0;
24 }
```

Listing 3.10: Free via an aliasing pointer (TLU)

Standard

- $(D, P, X, B) = (\mathsf{IRP}\ q, sl_q, sl_v, \mathtt{si}_{\mathtt{foo}})$

The standard does not explicitly state that the "modification" of an object in the context of restrict includes a call to `free`. We argue that it is fairly reasonable to take a slightly broader interpretation, and also perform the restrict checks when a dynamically allocated object is freed.

Chapter 4

The Crestrict semantics

In chapter 3 we have explained the C-in-$\mathbb{K}$ semantics for restrict and argued that it has some problems. In this chapter we will address these problems by proposing several refinements. We call the refined C-in-$\mathbb{K}$ restrict semantics the *Crestrict* semantics. The changes presented in this section are essential to present a more complete operational semantics of restrict, integrated in a small C-like language in chapter 5.

$$
\begin{array}{lcl}
b \in \mathit{Block}, \delta \in \mathit{Offset} & := & \mathbb{Z} \\
si \in \mathit{ScopeId} & := & \mathbb{Z} \\
sl \in \mathit{SimpleLoc} & := & \mathit{Block} \times \mathit{Offset} \\
l \in \mathit{Loc} & := & \mathit{SimpleLoc} \times \mathit{Bases} \\
ba \in \mathit{Base} & := & \mathit{Loc} \times \mathit{ScopeId} \\
bas \in \mathit{Bases} & := & \mathit{Set}(\mathit{Base}) \\
F_{bas} \in \mathit{BasesFam} & := & \mathit{Set}(\mathit{Bases}) \\
rs \in \mathit{RestrictState} & ::= & \mathsf{OnlyRead}\ F_{bas} \\
 & | & \mathsf{Restricted}\ bas \\
 & | & \text{☣} \\
 & | & \bot \\
rm \in \mathit{RestrictMap} & := & \mathit{SimpleLoc} \to \mathit{RestrictState} \\
R \in \mathit{RestrictStack} & := & \mathit{List}(\mathit{ScopeId} \times \mathit{RestrictMap})
\end{array}
$$

Figure 4.1: Crestrict related domains,
differences compared to the C-in-$\mathbb{K}$ domains are highlighted

In figure 4.1 a new version of the C-in-$\mathbb{K}$ domains is defined. Throughout the remaining sections of this chapter, we will explain and justify the changes compared to figure 3.1. In section 4.1 we address the removal of Unrestricted in favor of promoting the data of OnlyRead to a family of sets of bases. In section 4.2, we will explain why the *Block* of a base has been promoted to a *Loc*. Finally, section 4.3 and 4.4 explain how we improve upon the deferred restrict check and prevent locations with out of scope bases. These latter two sections are not related to changes within the domains, but to the actual operational rules.

4.1 Aliasing loads

Domain changes: *RestrictState*
Related problems: 3.3.1 (TMU)

The fundamental problem with the semantics for aliased loads is information loss, which leads to too much undefined behavior. The state representing loads via locations with different bases, Unrestricted, throws these bases away rather than keeping track of them. This makes it impossible to join this state with the Restricted *bas* state from a prior scope. More concrete, it excludes the possibility that the bases of the aliased pointers used for loading have the same subset of bases, *i.e.* they are based on the same restrict pointer somewhere in a prior scope.

The idea is to remove Unrestricted entirely (1), and promote the OnlyRead *bas* data to a family of sets of bases: OnlyRead F_{bas} (2). Every member of the family denotes a set of bases which was used to load from the location, so that we retain all information. Basically, the state OnlyRead $\{bas\}$ in the Crestrict semantics is equivalent to OnlyRead *bas* in the C-in-$\mathbb{K}$ semantics and OnlyRead F_{bas} with $|F_{bas}| > 1$ in the Crestrict semantics is similar to Unrestricted in the C-in-$\mathbb{K}$ semantics, with the only difference being that the Crestrict semantics tracks via which bases the location has been loaded from.

To show how the Crestrict semantics preserves the requirements of Unrestricted, consider the case when different pointers with bases bas_1 and bas_2 load from the same location. The restrict state is set to OnlyRead $\{bas_1, bas_2\}$. Attempting to store to this location means we attempt to join this restrict state with Restricted *bas*, which is *always* prohibited because $|\{bas_1, bas_2\}| > 1$ and thus results in $\not\downarrow$. This property justifies change 1. Similarly, when loading twice via a pointer with the same bases *bas*, the restrict state is set to OnlyRead $\{bas\}$. This is exactly what the operation OnlyRead *bas* $\sqcup$ OnlyRead *bas* in the original C-in-$\mathbb{K}$ semantics does: retain the OnlyRead state, allowing us to still be able to join with the Restricted state, which justifies change 2. Because we changed the *RestrictState* type, we have updated the join (figure 4.2) and `filter_bases` (figure 4.3) functions accordingly.

In figure 4.4b we show how the Crestrict semantics handles aliased loads, compared to the C-in-$\mathbb{K}$ semantics in figure 4.4a. This figure does not represent a specific program, but represents the fragment of the restrict stack corresponding to the fundamental problem. The stack symbol denotes the restrict stack R at some execution point of a program. Each tile in the stack is a restrict map *rm* (representing the right component of the restrict stack item), with the tile at the top representing the currently active scope. At the left of each tile the associated scope identifier is depicted, shaded in gray (representing the left component of the restrict stack item). The dashed arrow between two tiles represents the merge which happens when a scope exits, as was explained in section 3.2. The label at this arrow shows the result of the join operation for the relevant restrict state.

In scope $\mathtt{si_m}$ the restrict state of sl_x represents that aliased loads have occurred, so it is either Unrestricted (for the C-in-$\mathbb{K}$ semantics) or OnlyRead F_{bas} (for the Crestrict semantics). Scope $\mathtt{si_n}$ is some prior scope in which a store to sl_x was performed via a location with bases *bas*. Now, the Crestrict semantics allows for joining the restrict state of sl_x in $\mathtt{si_m}$ with the restrict state of sl_x in $\mathtt{si_n}$, if the precondition (`filter_bases` (OnlyRead $F_{bas}, \mathtt{si_m}$)) = $\{bas\}$ is met. This was exactly the objective of the changes: induce less undefined behavior by permitting aliased loads in a broader context.

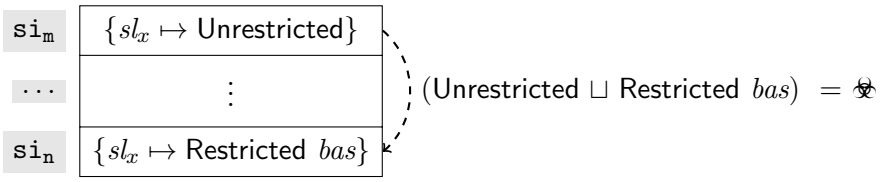
$$
\begin{aligned}
\mathsf{OnlyRead}\ F_{bas} \sqcup\ \ \mathsf{OnlyRead}\ F_{bas} &= \mathsf{OnlyRead}\ F_{bas} \\
\mathsf{OnlyRead}\ \{bas\} \sqcup \mathsf{OnlyRead}\ \{bas\} &= \mathsf{OnlyRead}\ \{bas\} \\
\mathsf{OnlyRead}\ F_{bas} \sqcup\ \ \mathsf{OnlyRead}\ {F_{bas}}' &= \mathsf{OnlyRead}\ (F_{bas} \cup {F_{bas}}') \\
\mathsf{OnlyRead}\ \{bas\} \sqcup\ \ \mathsf{Restricted}\ bas &= \mathsf{Restricted}\ bas \\
\mathsf{Restricted}\ bas \sqcup\ \ \mathsf{Restricted}\ bas &= \mathsf{Restricted}\ bas \\
\mathsf{OnlyRead}\ F_{bas} \sqcup\ \ \mathsf{Restricted}\ bas &= \text{☣} \\
\mathsf{Restricted}\ bas \sqcup\ \ \mathsf{Restricted}\ bas' &= \text{☣} \\
lhs \sqcup\ \ \text{☣} &= \text{☣} \\
\bot \sqcup\ \ rhs &= rhs
\end{aligned}
$$

Figure 4.2: Updated definition of the symmetric operation $\sqcup$, with $bas \neq bas'$, $F_{bas} \neq \{bas\}$ and $F_{bas} \neq {F_{bas}}'$

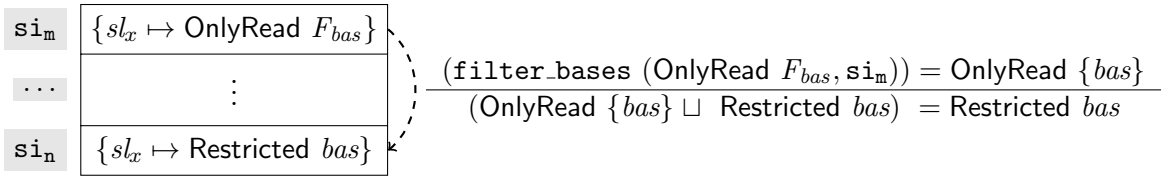
$$
\begin{aligned}
&\texttt{filter\_bases} : \mathit{RestrictState} \times \mathit{ScopeId} \to \mathit{RestrictState} \\
&\texttt{filter\_bases}\ (\mathsf{Restricted}\ bas, si) = \mathsf{Restricted}\ (\texttt{filter\_bases}'\ (bas, si)) \\
&\texttt{filter\_bases}\ (\mathsf{OnlyRead}\ F_{bas}, si) = \mathsf{OnlyRead}\ {F_{bas}}'\ \text{with}\ {F_{bas}}' := \\
&\qquad \{X \mid X = (\texttt{filter\_bases}'\ (bas, si)), bas \in F_{bas}\}
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{filter\_bases}' : \mathit{Bases} \times \mathit{ScopeId} \to \mathit{Bases} \\
&\texttt{filter\_bases}'\ (\emptyset, \_) = \emptyset \\
&\texttt{filter\_bases}'\ ((\_, si) : bas, si) = \texttt{filter\_bases}'(bas, si) \\
&\texttt{filter\_bases}'\ ((l, si') : bas, si) = (l, si') : \texttt{filter\_bases}'(bas, si)
\end{aligned}
$$

Figure 4.3: Updated auxiliary functions for filtering location bases



(a) C-in-$\mathbb{K}$ semantics



(b) Crestrict semantics

Figure 4.4: Restrict state of aliased loads

4.2 Promoting the block of a base

Domain changes: *Bases*
Related problems: 3.3.2, 3.3.4 (TLU)

The first component of a *Base* is no longer a *Block* but a complete *Loc*. The reason for this is best explain incrementally.

1. *Base* := *SimpleLoc* × *ScopeId* (first *Block* is replaced by a *SimpleLoc*)
   This preliminary substitution allows the semantics to distinguish restrict qualified array *element objects* and will be explained in section 4.2.1.

2. *Base* := *Loc* × *ScopeId* (in which *Block* is now a *Loc*)
   Taking the idea above one step further, this substitution also includes the "bases of a base" (*i.e.* the base of a *Loc* includes a *Loc*), making it a recursive definition. This permits the semantics to backtrack through which location a restrict qualified object itself was accessed, required to support a subclause of the standard which is especially relevant for nested restrict pointers and will be explained in section 4.2.2.

4.2.1 Arrays

As we explained in section 3.3.4, the semantics cannot distinguish restrict qualified objects which are array element objects because these objects have the same *Block*. This leads to too little undefined behavior given by the C-in-$\mathbb{K}$ semantics. This is resolved by also including the *Offset*, *i.e.* including the complete *SimpleLoc* of the restrict qualified object, into the base.

For example, the small fragment of C code depicted on the left in figure 4.5 contains an array a of restrict qualified integer pointers. The memory layout is depicted on the right, in which vertices represent memory cells, vertices with diagonal lines restrict qualified objects and edges pointer indirections. The text within a vertex represents the value of the memory cell, and the label just below it the memory address.

Using *SimpleLoc* in the *Base*, we can now distinguish these pointers because their provenance differs: $sl_{a_0} \neq sl_{a_1}$, whereas $(\texttt{block}\ sl_{a_0}) = (\texttt{block}\ sl_{a_1}) = b_a$. This means we can now give undefined behavior to restrict violations via restrict pointers in an array, which was exactly the objective of the change.

```
1  // Scope si
2  {
3      int x = 0; // Stored at sl_x
4                 // Stored at b_a
5      int* restrict a[2] = {&x, &x};
6  }
```

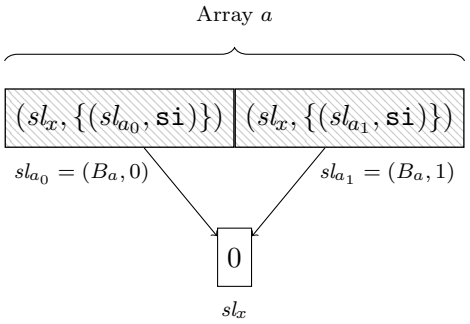


Figure 4.5: Distinguishing restrict qualified objects within an array

4.2.2 Modification of the restrict object itself

In section 3.3.2 we showed that nested restrict pointers have a subtle interaction with a subclause of the standard. This clause describes that accesses modifying the object X designated by the restrict qualified object P are also considered to modify P itself. As this clause is not covered by the C-in-$\mathbb{K}$ semantics, too little undefined behavior is given. In order to incorporate this clause in the semantics, two properties need to be taken into account:

1. "P is also considered modified" means the restrict state of P must be joined with Restricted.

2. The definition is recursive, *i.e.* if P_1 is used to load P_2 and P_2 is used to modify X, P_2 is considered modified and therefore P_1 is also considered modified.

To see how these properties can best be incorporated into the semantics, consider an example program with nested restrict pointers and its memory layout in figure 4.6. Besides the standard vertices and edges that were previously introduced for figure 4.5, pointer indirections are now also labelled to indicate whether the pointer value is used for a store or load (including the corresponding bases used for the access). Secondly, the dashed edge represents that the memory location pointed towards is considered modified due to (1), *i.e.* because the memory location the edge originated from was modified.

The program creates a pointer xp to x at line 4. Then, two nested restrict pointers p and q are created, which both point to xp but have different provenance. The first memory access of sl_x (after initialization) occurs at line 14. This assignment actually results in several changes to the restrict stack. The first dereference $*p$ loads the lvalue sl_{xp} from sl_p, updating the restrict state of sl_p to OnlyRead $\emptyset$. The second dereference $**p$ then updates the restrict state of sl_{xp} to OnlyRead $\{(sl_p, \texttt{si})\}$. The loaded location is $(sl_x, \{(sl_{xp}, \texttt{si})\})$, which is used for a store so the restrict state of sl_x becomes Restricted $\{(sl_{xp}, \texttt{si})\}$.

As this store is a modification, the semantics should also change the restrict state of sl_{xp} by property (1). But this means that the restrict state which represents this modification must be joinable with the current restrict of sl_{xp}, which is OnlyRead $\{(sl_p, \texttt{si})\}$. The only restrict state that meets this requirement is Restricted $\{(sl_p, \texttt{si})\}$. But currently, it is not possible to create this restrict state because the location $(sl_x, \{(sl_{xp}, \texttt{si})\})$ does not contain enough information to set the correct bases (*i.e.* it does not know the base location sl_{xp} is derived from sl_p). Therefore, we must extend the provenance with the information that the lvalue $*p$ is derived from $\{(sl_p, \texttt{si})\}$.

```
1  // Scope si
2  {
3      int x = 0;     // Stored at sl_x
4      int* xp = &x; // Stored at sl_xp
5                    // M(sl_xp) = Ptr (sl_x, ∅)
6      // Stored at sl_p
7      int *restrict *restrict p = &xp;
8      // M(sl_p) = Ptr (sl_xp, {(sl_p, si)})
9      // Stored at sl_q
10     int *restrict *restrict q = &xp;
11     // M(sl_q) = Ptr (sl_xp, {(sl_q, si)})
12     // The first load *q should result in ☣
13
14     **p = 10; // Load from *p, store via **p
15     **q = 11; // Load from *q, store via **q
16 }
```

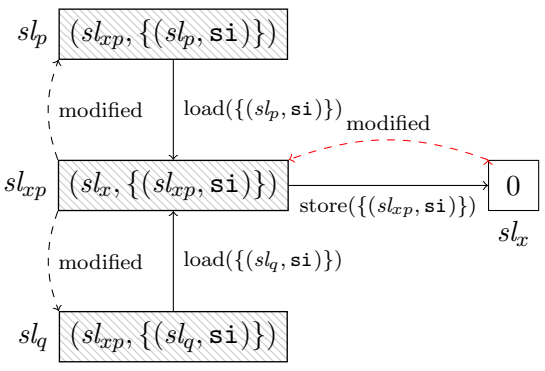


Figure 4.6: Memory layout of nested restrict pointers

This is the point where we promote the *SimpleLoc* component of a base to a *Loc*. By including the bases of each base, the semantics now has all the information needed to determine how the location of a restrict qualified object was constructed.

Recall the new definition of *Loc* from figure 4.1. The evaluation of $**p$ as lvalue now results in: $l = (sl_x, \{((sl_{xp}, \{((sl_p, \emptyset), \texttt{si})\}), \texttt{si})\})$. Now, because that lvalue is used to modify sl_x we have to also modify the restrict state of its bases, *i.e.* for every $(l', \_) \in \texttt{bases}(l)$ the semantics acts as if a store via l' has occurred. This matches the desired recursive definition (2) and results in the following restrict map: $\{sl_x \mapsto \mathsf{Restricted}\ \{((sl_{xp}, \{((sl_p, \emptyset), \texttt{si})\}), \texttt{si})\}$, $sl_{xp} \mapsto \mathsf{Restricted}\ \{((sl_p, \emptyset), \texttt{si})\}, sl_p \mapsto \mathsf{Restricted}\ \emptyset\}$.

When we continue program execution at line 15, we now have an access which induces undefined behavior. In fact, the first dereference $*q$ is the problematic access: we attempt to join $\mathsf{OnlyRead}\ \{((sl_q, \emptyset), \texttt{si})\}$ with the current restrict state of sl_{xp} $\mathsf{Restricted}\ \{((sl_p, \emptyset), \texttt{si})\}$, resulting in ☣. This means if we replace line 15 with $*q$ (a simple load), the program also has undefined behavior. This shows that the objective of the changes is achieved: induce more undefined behavior by modelling the subclause stating the restrict object itself is considered modified.

A tree of lvalue bases As a recursive structure, a location can be depicted as a tree. Consider the program depicted on the left in figure 4.7, which shows how the tree can grow both in breadth and depth. At line 5, the lvalue $**q$ is the location $(sl_x, \{((sl_{xp}, \{((sl_q, \emptyset), \texttt{si}_{\texttt{bar}}), ((sl_p, \emptyset), \texttt{si}_{\texttt{foo}})\}), \texttt{si}_{\texttt{bar}})\})$. Its tree structure is depicted on the right, in which the root node is the *SimpleLoc* of the actual location, the first child its base and the children of the child represent the bases of the child.

```
1  // Scope si_bar
2  int bar(int *restrict *restrict q) {
3                  // Stored at sl_q
4                  // M(sl_q) = Ptr (sl_xp, {((sl_q, ∅), si_bar), ((sl_p, ∅), si_foo)})
5      return **q; // What is the lvalue here?
6  }
7
8  // Scope si_foo
9  int foo() {
10     int x;          // Stored at sl_x
11     int* xp = &x;   // Stored at sl_xp
12                     // M(sl_xp) = Ptr (sl_x, ∅)
13
14     int **restrict p = &xp;
15                     // Stored at sl_p
16                     // M(sl_p) = Ptr (sl_xp, {((sl_p, ∅), si)})
17
18     return bar(p);
19 }
```

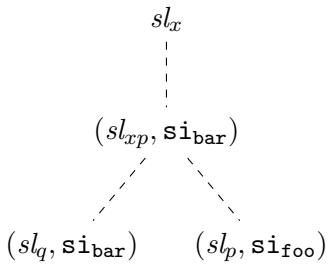


Figure 4.7: Tree depiction of lvalue $**q$ at line 5

4.3 Moving up the deferred check

Domain changes: N.A.
Related problems: 3.3.3 (TLU)

The deferred restrict check is meant to perform the restrict checks for accesses occurring in different scopes. The problem is that, as the name suggests, the check is not performed straight away. As previously shown, this means that optimizing a program by inlining a function may change whether it contains undefined behavior or not, which is undesirable. The idea is to detect these cases of undefined behavior directly when they occur, by performing joins not only with the top item on the restrict stack but also with every other map that contains a *RestrictState* $\neq \bot$. While moving through the restrict stack from top to bottom, the restrict state *rs* is filtered to remove bases whose scope has terminated.

The reason we only consider restrict states which are unequal to $\bot$ is two-fold. Firstly, when updating the restrict stack due to memory accesses, we do not know in which scopes a location is still valid. So if we were to also join with restrict state $\bot$, we would not know at which restrict map we should stop. Secondly, we need at least two memory accesses to induce undefined behavior, so only considering restrict states where a prior access has occurred is therefore an appropriate choice.

A visual representation of this operation is depicted in figure 4.8. Applying this operation to the example of listing 3.6 means that this program is now assigned undefined behavior and that the function inlining optimization now preserves the semantics.

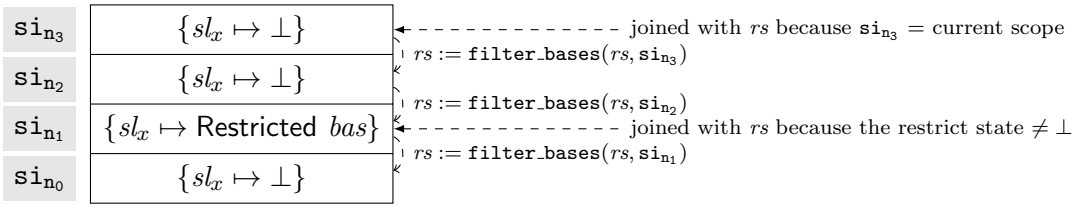


Figure 4.8: Joining *rs* with multiple *RestrictMap*s

4.4 Filtering bases of pointer values

Domain changes: *ScopeMap*
Related problems: 3.3.5 (TMU)

In the C-in-$\mathbb{K}$ semantics the bases of restrict states are filtered when a scope terminates, but the bases of pointer values are never filtered. We have shown that this is insufficient because we can create pointer values with expired bases by returning a restrict pointer or assigning a restrict pointer value to a global pointer, leading to too much undefined behavior. To this end, the idea is to track all active scopes by a map $S \in \mathit{ScopeMap} := \mathit{ScopeId} \to \mathbb{B}$. A scope identifier *si* is mapped to `true` if it is active, and `false` if the scope has terminated or the scope identifier is unused. Now, every base with an inactive scope is removed from its location before the restrict check is performed, thus ensuring only bases that are still active are included.

Listing 4.9 demonstrates this on an example program. At line 7 the scope $\mathtt{si}_{\mathtt{foo}}$ terminates, and the state is updated in S accordingly. The next memory access is a store via p at line 14 in which the `filter_inactive` function is first called on the lvalue, removing the base $((sl_q, \emptyset), \mathtt{si}_{\mathtt{foo}})$ as $\mathtt{si}_{\mathtt{foo}}$ is now out of scope. The filtered location is $(sl_x, \emptyset)$ because no bases remain, making the store well-defined instead of giving undefined behavior which was the objective of the changes.

```
1  int x;  // Stored at sl_x
2  int* p; // Stored at sl_p
3
4  // Scope si_foo
5  void foo(int* restrict q) { // q is stored at sl_q
6      p = q;                  // M(sl_p) = Ptr (sl_x, {((sl_q, ∅), si_foo)})
7  }                           // S(si_foo) = false
8
9  // Scope si_main
10 int main() {
11     p = &x;      // M(sl_p) = Ptr (sl_x, ∅)
12     *p = 10;     // R(sl_x) = Restricted ∅
13     foo(&x);     // p gets based on sl_q
14     *p = 11;     // Store via p, (filter_inactive ((sl_x, {((sl_q, ∅), si_foo)}), S)) = (sl_x, ∅)
15                  // Restricted ∅ ⊔ Restricted ∅ = Restricted ∅
16     return 0;
17 }
```

Figure 4.9: Filtering out terminated bases

Chapter 5

The Crestrict language

In chapter 4 we proposed several refinements for the C-in-$\mathbb{K}$ semantics. To show how we integrate the refined semantics into a programming language, this chapter presents the Crestrict language, a small C-like language which aims to provide sufficiently many features for defining a relevant restrict semantics. The language is based on CompCert's *Clight* language, which is one of the intermediate target languages of the verified CompCert compiler [3].

In section 5.1 the syntax of the language is presented. In section 5.2 the different kinds of evaluation judgments and operational semantics are presented. Section 5.3 defines and explains the different kinds of memory operations. Finally, section 5.4 defines and explains the operations on the *RestrictStack*.

The semantics we present in this chapter has also been implemented in an interpreter to make it executable. Section 6.1 will elaborate on this implementation, but throughout this chapter we already mention some differences between the operational semantics and the implementation.

5.1 Syntax

The syntax of the language and semantic constructs are presented in figure 5.1. Extensions and changes compared to the Clight language are mostly based on the domains that were presented in chapter 4.

All expressions are annotated with their type, which is left implicit in this thesis. Instead, (`type` e) is used to refer to the type which annotates e. This information is used by both overloaded operators (*e.g.* + works on both integers and pointers) and special rules for restrict qualified lvalues, which will be explained later.

An expression is either in *lvalue position* (at the left-hand side of the assignment operator) or *rvalue position* (anywhere else). The only expressions that may be placed in lvalue position are id and $*e$, and every expression may be placed in rvalue position. In our language, this is not enforced by the syntax but by the operational semantics (which will be explained later). The interpreter *does* check for this before interpreting the program, because the program is first type checked after elaboration.

Assignments $e_1 = e_2$ assign the evaluated rvalue e_2 to the evaluated lvalue e_1 and are included in the statements. Furthermore, procedure and function calls are also part of the statements. The for-loop notation $\mathsf{for}(s_1, e_2, s_3)\ s$ means that s_1 is executed at the beginning of the loop, e_2 is the condition, s_3 is executed at the end of each loop iteration and s is the body. Dynamic memory allocations and deallocations via malloc and free are included as statements.

A complete *Type* is composed of a *SimpleType* τ and a *TypeQualifier* τ_q. The supported variable types are I32 (signed 32-bit integers), Ptr τ (pointers to type τ) and Array τ n (ar-

rays of element type τ and size n). Furthermore, Function τ τ^* denotes function types with return type τ and parameter types τ^*, and Void denotes the void type. Three types of type qualifiers are distinguished: NoRestrict means that the type is unqualified, GlobalRestrict means that the type is restrict qualified within a global declaration and Restrict means the type is restrict qualified within a local declaration. The distinction between global and local restrict qualifiers is made to be able to distinguish the restrict declaration scope, which will become more apparent by the definition of `get_restrict_scope` which will be explained in the next section. Function definitions (τ, dcl_1, dcl_2, s) denote a function with parameter declarations dcl_1, local variable declarations dcl_2, function body s and return type τ. All local variable declarations must occur at the start of the function. Finally, a program is composed of a list of global variable declarations and a list of function definitions. The identifier denoting the program entry point is fixed to `main`. This means that if no function definition for main is given, the program will get stuck immediately (*i.e.* the syntax does not enforce main to be defined).

Compared to Clight, floats, structs, unions, type casts, switch statements and do loops are the largest omitted features of the language. The most important reason for this is that we want a small language to focus on how the restrict feature affects its semantics, while being large enough to be able to write some realistic example programs.

All domains related to locations and provenance have previously been introduced in figure 4.1 and are merely repeated here for completeness.

Expressions in rvalue position may result in three types of values: I32 n (32-bit integer values n), Ptr l (a pointer value of location l) and Undef which represents the value at a memory location which is uninitialized. The `NULL` pointer, represented in Clight by the value (I32 0), is not modelled. The reason for this is that the interpreter checks type compatibility between the left and right-hand sides of the assignment operator (*i.e.* one cannot assign an integer value to a pointer type), and omitting the `NULL` pointer did not limit our flexibility in creating test programs.

As in Clight, statements are evaluated to an outcome *out* based on the approach of Norrish [38] and Huisman and Jacobs [13]. This can either be Normal, indicating the next statement can be executed; Continue, for executing the next iteration of the current loop; Break, for leaving the current loop, and Return / Return v for leaving a function (possibly with return value v).

The functions and variables are mapped to their memory locations by two environments. The global environment G is defined as a record and associates function and global variable identifiers to the memory blocks they are stored in, and also contains the function definitions. The local environment E is constructed per function and associates its parameter and local variable declarations with memory blocks. The memory model is based on CompCert's model [27], in which each allocation results in a different block reference $b \in$ *Block*. A memory instance $M \in$ *Mem* associates block references with a record $mb \in$ *MemBlock*. The upper bound hi denotes the highest offset (exclusive) which may be addressed (the lower bound cannot be configured unlike in Clight, and is treated as 0). The boolean dyn tracks whether a block was dynamically allocated (by means of a call to malloc) in order to determine whether it may be freed by the program. Finally, the third component contents associates block offsets with values. The *ScopeMap* tracks which scopes are active at a certain point of the execution, and its purpose has been explained in section 4.4.

Finally, the restrict related domains which have previously been introduced and explained in figure 4.1 are listed for completeness.

Expressions and statements

$$
\begin{array}{lcl}
id \in Id & := & String \\
e \in Expr & ::= & id \mid n \mid \mathsf{sizeof}(\tau) \mid op_1\ e \mid e_1\ op_2\ e_2 \mid *e \mid \&e \\
op_1 \in UnaryOp & ::= & \mathtt{!} \mid \sim \mid - \\
op_2 \in BinaryOp & ::= & + \mid - \mid * \mid / \mid \% \mid << \mid >> \mid \& \mid \mid \mid \hat{} \\
 & \mid & < \mid \leq \mid > \mid \geq \mid == \mid \mathtt{!=} \\
s \in Statement & ::= & \mathsf{skip} \mid \mathsf{break} \mid \mathsf{continue} \mid \mathsf{return}\ Opt(e) \\
 & \mid & e_1 = e_2 \mid e_1 = e_2(e^*) \mid e(e^*) \mid s_1; s_2 \\
 & \mid & \mathsf{if}(e)\ s_1\ \mathsf{else}\ s_2 \mid \mathsf{while}(e)\ s \mid \mathsf{for}(s_1, e_2, s_3)\ s \\
 & \mid & e_1 = \mathsf{malloc}(e_2) \mid \mathsf{free}(e)
\end{array}
$$

Locations and provenance

$$
\begin{array}{lcl}
b \in Block, \delta \in Offset & := & \mathbb{Z} \\
si \in ScopeId & := & \mathbb{Z} \\
sl \in SimpleLoc & := & Block \times Offset \\
l \in Loc & := & SimpleLoc \times Set(Base) \\
ba \in Base & := & Loc \times ScopeId \\
bas \in Bases & := & Set(Base) \\
F_{bas} \in BasesFam & := & Set(Bases)
\end{array}
$$

Values, outcomes, types, functions and programs

$$
\begin{array}{lcl}
v \in Val & ::= & \mathsf{I32}\ n \mid \mathsf{Ptr}\ l \mid \mathsf{Undef} \\
out \in Outcome & ::= & \mathsf{Normal} \mid \mathsf{Break} \mid \mathsf{Continue} \mid \mathsf{Return} \mid \mathsf{Return}\ v \\
st \in SimpleType & ::= & \mathsf{I32} \mid \mathsf{Ptr}\ \tau \mid \mathsf{Array}\ \tau\ n \mid \mathsf{Function}\ \tau\ \tau^* \mid \mathsf{Void} \\
\tau_q \in TypeQualifier & ::= & \mathsf{NoRestrict} \mid \mathsf{Restrict} \mid \mathsf{GlobalRestrict} \\
\tau \in Type & := & SimpleType \times TypeQualifier \\
dcl \in Declarations & := & List(Type \times Id) \\
Fd \in FunDef & := & Type \times Declarations \times Declarations \times Statement \\
fun\_dcl \in FunDeclarations & := & List(Id \times FunDef) \\
P \in Program & := & Declarations \times FunDeclarations
\end{array}
$$

Environments and state members

$$
\begin{array}{lcl}
E \in Env & := & Id \rightharpoonup Block \\
G \in Globals & := & \{\ \mathsf{env} : Id \rightharpoonup Block, \mathsf{defs} : Block \rightharpoonup FunDef\ \} \\
mb \in MemBlock & := & \{\ \mathsf{hi} : \mathbb{Z}, \mathsf{dyn} : \mathbb{B}, \mathsf{contents} : Offset \rightharpoonup Val\ \} \\
M \in Mem & := & Block \rightharpoonup MemBlock \\
S \in ScopeMap & := & ScopeId \to \mathbb{B} \\
rs \in RestrictState & ::= & \mathsf{OnlyRead}\ F_{bas} \mid \mathsf{Restricted}\ bas \mid \text{☣} \mid \bot \\
rm \in RestrictMap & := & SimpleLoc \to RestrictState \\
R \in RestrictStack & := & List(ScopeId \times RestrictMap)
\end{array}
$$

Figure 5.1: Syntax of language and semantic constructs

$$
\sigma \in State\ :=\ \left\{ \begin{array}{l} \mathsf{mem} : Mem, \\ \mathsf{rstack} : RestrictStack, \\ \mathsf{scopes} : ScopeMap \end{array} \right\}
$$

Figure 5.2: *State* record

5.2 Natural semantics

This section describes the semantics of the language as a *big-step operational semantics*, meaning expressions and statements are directly related to their results. This kind of semantics relates closely to the interpreter implementation. As in the Clight language, most judgments are parametrized by the global environment G and the local environment E. Secondly, most judgments also take a state σ which is defined as a record composed of three fields for the memory, restrict stack and scope map as shown in figure 5.2.

5.2.1 Evaluation judgments

Below the different kinds of evaluation judgments are defined. Every judgment relating a program construct to some other syntactic construct (explained below) is paired with state σ. The evaluation result of a program construct is paired with the updated state σ' (which resulted from executing the program construct).

$G, E \vdash$	$e, \sigma \Downarrow_{\mathsf{L}} l, \sigma'$	evaluation of expressions in lvalue position
$G, E \vdash$	$e, \sigma \Downarrow_{\mathsf{R}} v, \sigma'$	evaluation of expressions in rvalue position
$G, E \vdash$	$e^*, \sigma \Downarrow_{\mathsf{R}} v^*, \sigma'$	evaluation of lists of expressions
$G, E \vdash$	$s, \sigma \Downarrow out, \sigma'$	evaluation of statements
$G \vdash$	$Fd(v^*), \sigma \Downarrow v, \sigma'$	evaluation of function invocations
$\vdash$	$P \Downarrow n$	evaluation of programs

Expressions in lvalue position are evaluated to locations l and expressions in rvalue position are evaluated to values v. Statements are evaluated to outcomes *out*, which indicate how the execution ended: Normal when it completed normally, and Break, Continue or Return otherwise. Compared to Clight the expressions are no longer pure as they possibly modify the restrict stack (they still do not change the memory).

The original Clight language distinguished between diverging programs and undefined behavior by Leroy and Grall's coinductive approach [30]: they have separate evaluation rules for terminating programs and programs which diverge. Currently, they use a small-step semantics which also enables the semantics to make this distinction. In our operational semantics we do not distinguish between diverging programs and undefined behavior. However, the implemented interpreter does make this distinction as non-terminating programs will eventually lead to a stack overflow and programs with undefined behavior terminate immediately upon detection, resulting in an error message.

There are several auxiliary functions that will be used in the inference rules, which are defined in figure 5.3. The `add_prov` function adds a base to the set of bases of pointer values and is the identity function for other values. The function `get_restrict_scope` determines the declaration scope of a restrict qualified type, given the type qualifier and current state. The restrict declaration scope for globally restrict qualified variables is fixed to $\mathtt{si}_{\mathtt{main}}$ (due the associated restrict block previously mentioned in section 2.4). The function `by_reference` defines whether a type has Clight access mode "by reference", indicating that during lvalue conversion the address of the lvalue expression is returned as a pointer value. Finally, the `sizeof` function states that all types occupy one memory cell, except the array type (which occupies the amount of elements multiplied by the size of the element type). This reflects that the memory model works on the granularity of values rather than bytes. There are two auxiliary functions which are used in the inference rules and not defined in figure 5.3. These functions perform the evaluation of binary operations (`eval_binop`) and unary operators (`eval_unop`). As most evaluations are straightforward, only a partial definition is given in figure 5.4. The case for the logical not (!) operator is shown, as well as the overloaded + and == operators.

$$
\begin{aligned}
\texttt{add\_prov} &: \mathit{Val} \to \mathit{Base} \to \mathit{Val} \\
\texttt{add\_prov}\ (\mathsf{Ptr}\ (sl, bas))\ ba &\triangleq \mathsf{Ptr}\ (sl, bas \cup \{ba\}) \\
\texttt{add\_prov}\ v\ \_ &\triangleq v \\
\\
\texttt{si}_{\texttt{main}} &: \mathit{ScopeId} \\
\texttt{si}_{\texttt{main}} &\triangleq 0 \\
\\
\texttt{get\_restrict\_scope} &: \mathit{Type} \to \mathit{State} \to \mathit{Opt}(\mathit{ScopeId}) \\
\texttt{get\_restrict\_scope}\ (\_, \mathsf{NoRestrict})\ \_ &\triangleq \epsilon \\
\texttt{get\_restrict\_scope}\ (\_, \mathsf{GlobalRestrict})\ \_ &\triangleq \texttt{si}_{\texttt{main}} \\
\texttt{get\_restrict\_scope}\ (\_, \mathsf{Restrict})\ \sigma &\triangleq \textbf{if let}\ ((si, \_) : \_) = \sigma.\mathsf{rstack}\ \textbf{then}\ si\ \textbf{else}\ \epsilon \\
\\
\texttt{by\_reference} &: \mathit{Type} \to \mathbb{B} \\
\texttt{by\_reference}\ (st, \_) &\triangleq st = (\mathsf{Array}\ \_\ \_) \vee st = (\mathsf{Function}\ \_\ \_) \\
\\
\texttt{is\_restrict} &: \mathit{Type} \to \mathbb{B} \\
\texttt{is\_restrict}\ (\_, \tau_q) &\triangleq \tau_q = \mathsf{Restrict} \vee \tau_q = \mathsf{GlobalRestrict} \\
\\
\texttt{sizeof} &: \mathit{Type} \to \mathbb{Z} \\
\texttt{sizeof}\ ((\mathsf{Array}\ \tau\ n), \_) &\triangleq n \times \texttt{sizeof}(\tau) \\
\texttt{sizeof}\ \_ &\triangleq 1
\end{aligned}
$$

Figure 5.3: Auxiliary functions for expression and statement semantics

5.2.2 Expressions

The natural semantics of expressions are given in figure 5.5. The first three rules are for expressions in lvalue position, which evaluate to a *Loc*. The rules E-Id-Env and E-Id-Glob describe that a variable identifier *id* evaluates to $((b, 0), \emptyset)$ for the b associated with *id* in E or G, the offset 0 and the empty set of bases. The rule E-Deref describes that the dereference expression $*e$ evaluates to the location l, given that e evaluated to $\mathsf{Ptr}\ l$ by rvalue evaluation (*i.e.* only pointer values can be dereferenced).

The rule E-Sizeof does something special: the result of the auxiliary `sizeof` function is multiplied by four. As was previously explained, the granularity of the memory model is at value level, not at byte level. In order to give a sensible value we therefore act as if every memory cell occupies four bytes, which is reflected by this multiplication.

The remaining rules are for expressions in rvalue position, which evaluate to a *Val*.

There are three rules that are concerned with *lvalue conversion*, the occurrence of an lvalue expression e in rvalue position. The first rule is E-Lval-Conv which is applied for integer and pointer types (*i.e.* non reference types) that are not restrict qualified. The lvalue is evaluated into a location, the value at that location is loaded from memory and used for the result of the lvalue conversion. The E-Lval-Conv-Ref handles the special case when e is of type array or function. The value resulting from this rule is a pointer to the location of e. For example, when you pass an array variable as argument to a function the passed value becomes a pointer to the first element of the array due to this rule. The third rule, E-Lval-Conv-Restrict, makes a case distinction for the case where e is restrict qualified. Because only pointer types can be restrict qualified, loading from the location l will result in a pointer value v. The call to `add_prov` adds the base l with the restrict declaration scope to the value v. This rule ensures that all lvalues derived from a restrict qualified object get the correct provenance, because every time the pointer value of a restrict qualified pointer is *used* this rule applies. Use means that restrict qualified pointer expression `E` is evaluated as rvalue into the pointer value (for which lvalue conversion must take place). This happens either because the pointer expression is directly used for an access, *e.g.* $*$`E`, or because the pointer expression is

assigned to another pointer, *e.g.* `int* p = E;`.

The rules E-UnaryOp and E-BinaryOp show how unary and binary operations are evaluated. For binary operations, we have chosen the evaluation order to be left-to-right (which is the same in Clight). Finally, the rule E-AddressOf shows how the address of an expression results in a pointer value.

τ	v	(eval_unop ! τ v)
I32	I32 0	$(n == 0)$? I32 1 : I32 0

τ_1	τ_2	v_1	v_2	(eval_binop + τ_1 τ_2 v_1 v_2)
I32	I32	I32 n_1	I32 n_2	I32 $(n1 + n2)$
Ptr τ	I32	Ptr $((b, \delta), bas)$	I32 n	Ptr $((b, \delta + n \times (\texttt{sizeof}\ \tau)), bas)$
I32	Ptr τ	I32 n	Ptr $((b, \delta), bas)$	Ptr $((b, \delta + n \times (\texttt{sizeof}\ \tau)), bas)$
				(eval_binop == τ_1 τ_2 v_1 v_2)
I32	I32	I32 n_1	I32 n_2	$(n_1 == n_2)$? I32 1 : I32 0
Ptr τ	Ptr τ	Ptr $(sl_1, \_)$	Ptr $(sl_2, \_)$	$(sl_1 == sl_2)$? I32 1 : I32 0

Figure 5.4: Excerpt of `eval_unop` and `eval_binop` functions

Expressions in lvalue position:

E-Id-Env

$$\frac{E(id) = b}{G, E \vdash id, \sigma \Downarrow_{\mathsf{L}} ((b, 0), \emptyset), \sigma}$$

E-Id-Glob

$$\frac{id \notin \mathtt{dom}(E) \quad G.\mathsf{env}(id) = b}{G, E \vdash id, \sigma \Downarrow_{\mathsf{L}} ((b, 0), \emptyset), \sigma}$$

E-Deref

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{R}} \mathsf{Ptr}\ l, \sigma'}{G, E \vdash *e, \sigma \Downarrow_{\mathsf{L}} l, \sigma'}$$

Expressions in rvalue position:

E-Int

$$\frac{}{G, E \vdash n, \sigma \Downarrow_{\mathsf{R}} \mathsf{I32}\ n, \sigma}$$

E-Lval-Conv

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{L}} l, \sigma' \qquad (\texttt{load}\ \sigma'\ l) = (\sigma'', v) \qquad \neg(\texttt{by\_reference}\ (\texttt{type}\ e)) \quad \neg(\texttt{is\_restrict}\ (\texttt{type}\ e))}{G, E \vdash e, \sigma \Downarrow_{\mathsf{R}} v, \sigma''}$$

E-Sizeof

$$\frac{\texttt{sizeof}\ \tau = n}{G, E \vdash \mathsf{sizeof}(\tau), \sigma \Downarrow_{\mathsf{R}} \mathsf{I32}\ (n \times 4), \sigma}$$

E-Lval-Conv-Ref

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{L}} l, \sigma' \quad \texttt{by\_reference}\ (\texttt{type}\ e)}{G, E \vdash e, \sigma \Downarrow_{\mathsf{R}} \mathsf{Ptr}\ l, \sigma'}$$

E-Lval-Conv-Restrict

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{L}} l, \sigma' \qquad (\texttt{load}\ \sigma'\ l) = (\sigma'', v) \qquad \texttt{is\_restrict}\ (\texttt{type}\ e) \quad (\texttt{get\_restrict\_scope}\ (\texttt{type}\ e)\ \sigma) = si}{G, E \vdash e, \sigma \Downarrow_{\mathsf{R}} (\texttt{add\_prov}\ v\ (l, si)), \sigma''}$$

E-UnaryOp

$$\frac{G, E \vdash e_1, \sigma \Downarrow_{\mathsf{R}} v_1, \sigma' \quad (\texttt{eval\_unop}\ op_1\ (\texttt{type}\ e_1)\ v_1) = v}{G, E \vdash (op_1\ e_1), \sigma \Downarrow_{\mathsf{R}} v, \sigma'}$$

E-AddressOf

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{L}} l, \sigma'}{G, E \vdash \&e, \sigma \Downarrow_{\mathsf{R}} \mathsf{Ptr}\ l, \sigma'}$$

E-BinaryOp

$$\frac{G, E \vdash e_1, \sigma \Downarrow_{\mathsf{R}} v_1, \sigma' \quad G, E \vdash e_2, \sigma' \Downarrow_{\mathsf{R}} v_2, \sigma'' \quad (\texttt{eval\_binop}\ op_2\ (\texttt{type}\ e_1)\ (\texttt{type}\ e_2)\ v_1\ v_2) = v}{G, E \vdash (e_1\ op_2\ e_2), \sigma \Downarrow_{\mathsf{R}} v, \sigma''}$$

Figure 5.5: Natural semantics of expressions

5.2.3 Statements

The natural semantics of statements are given in figure 5.6 (everything except loops) and figure 5.7 (loops).

The first interesting rule is the assignment rule, S-Assign. The expression on the left-hand side of the assignment operator is evaluated as expression in lvalue position, resulting in a location l. The expression on the right-hand side of the assignment operator is evaluated as expression in rvalue position, resulting in a value v. The `store` rule is then invoked to update the state accordingly.

The S-Procedure and S-Call rules denote the semantics for function calls. The location of the function definition is retrieved by evaluating the function identifier to a pointer. Then, the arguments of the function call are evaluated into a list of values. Finally, the function definition is invoked. In the case of S-Call the value resulting from the function invocation is saved at the evaluated location of the left-hand side expression.

The S-Malloc rule is for dynamic allocations. The evaluated location e_1 is assigned a pointer value to the freshly allocated memory object. Analogously to E-Sizeof, the amount of bytes n to be allocated is divided by four to adjust for the granularity of the memory model, which is why n is required to be a multiple of 4. The rule also allows n to be 0, in which case a block is reserved with upper bound 0 (*i.e.* the pointer value cannot be dereferenced).

The $\rightsquigarrow^{loop}$ relation describes how the *Outcome out* is updated when the loop is exited prematurely through a Break or Return outcome.

While loops are described by three rules: S-While-False is the normal terminating case of the loop when the condition evaluates to false. S-While-Abort is the premature abortion of the loop, through a return or break statement. Finally, the iteration is described by S-While-True: the condition evaluates to true and the execution of the body had a Normal or Continue outcome.

For loops are described by four rules. The S-For-False, S-For-Abort and S-For-True are very much similar to the corresponding while rules. The only different rule is S-For-Init, which evaluates the initial statement.

The natural semantics of function invocations and complete programs are given in figure 5.8. The auxiliary functions used in the inference rules are defined in figure 5.9.

A function invocation consists of a function definition *Fd* and a list of argument values v_{args}. Before executing the function body, the function `init_invocation` allocates memory for the parameters and local variables and binds the argument values to the corresponding memory locations. The result is a modified state σ' in which a new restrict map is pushed onto rstack, a new scope is set to active in scopes and the memory changes are updated in mem. The auxiliary function `fresh` is used to create a new scope identifier, and specializes for the case where no scopes are active yet. As we require the function main to be function from which the program starts, this means the scope identifier for main is always set to $\texttt{si}_{\texttt{main}}$ to correspond with the definition of `get_restrict_scope`. Besides the updated state, a local environment E (for the duration of the function invocation) and list of block references *bs* are returned. After executing the body s, the outcome *out* and function return type τ must have a compatible return value v_{res}, checked by $(out, \tau)\ \#\ v_{res}$. The result of the invocation is the value v_{res} and a new state in which the local variables have been freed from the memory, the restrict map is popped off the restrict stack and the scope identifier of the function scope is set to inactive in the scope map.

The execution of a complete program is done by constructing the *Globals* environment G. This allocates space for all global variables and functions, and also relates the block references where functions are stored to function definitions. Then, the function definition of the identifier *main* is retrieved, and invoked under the empty set `init_state`. The result is a number n, indicating the success status.

S-Skip

$$\frac{}{G, E \vdash \mathsf{skip}, \sigma \Downarrow \mathsf{Normal}, \sigma}$$

S-Break

$$\frac{}{G, E \vdash \mathsf{break}, \sigma \Downarrow \mathsf{Break}, \sigma}$$

S-Continue

$$\frac{}{G, E \vdash \mathsf{continue}, \sigma \Downarrow \mathsf{Continue}, \sigma}$$

S-Return

$$\frac{}{G, E \vdash (\mathsf{return}\ \epsilon), \sigma \Downarrow \mathsf{Return}, \sigma}$$

S-Return-Val

$$\frac{G, E, \vdash e, \sigma \Downarrow_{\mathsf{R}} v, \sigma'}{G, E \vdash (\mathsf{return}\ e), \sigma \Downarrow \mathsf{Return}\ v, \sigma'}$$

S-Assign

$$\frac{G, E \vdash e_1, \sigma \Downarrow_{\mathsf{L}} l, \sigma' \quad G, E \vdash e_2, \sigma' \Downarrow_{\mathsf{R}} v, \sigma'' \quad (\mathtt{store}\ \sigma''\ l\ v) = \sigma'''}{G, E \vdash (e_1 = e_2), \sigma \Downarrow \mathsf{Normal}, \sigma'''}$$

S-Procedure

$$\frac{\begin{array}{c} G, E \vdash e_{fun}, \sigma \Downarrow_{\mathsf{R}} \mathsf{Ptr}\ ((b, 0), \emptyset), \sigma' \quad G.\mathsf{defs}(b) = Fd \quad (\mathtt{type\_of\_fundef}\ Fd) = (\mathtt{type}\ e_{fun}) \\ G, E \vdash e_{args}, \sigma' \Downarrow_{\mathsf{R}} v_{args}, \sigma'' \quad G \vdash Fd(v_{args}), \sigma'' \Downarrow \_, \sigma''' \end{array}}{G, E \vdash e_{fun}(e_{args}), \sigma \Downarrow \mathsf{Normal}, \sigma'''}$$

S-Call

$$\frac{\begin{array}{ccc} G, E \vdash e, \sigma \Downarrow_{\mathsf{L}} l, \sigma' & G, E \vdash e_{fun}, \sigma' \Downarrow_{\mathsf{R}} \mathsf{Ptr}\ ((b, 0), \emptyset), \sigma'' & G.\mathsf{defs}(b) = Fd \\ (\mathtt{type\_of\_fundef}\ Fd) = (\mathtt{type}\ e_{fun}) & G, E \vdash e_{args}, \sigma'' \Downarrow_{\mathsf{R}} v_{args}, \sigma''' & G \vdash Fd(v_{args}), \sigma''' \Downarrow v, \sigma'''' \\ & (\mathtt{store}\ \sigma''''\ l\ v) = \sigma''''' & \end{array}}{G, E \vdash e = e_{fun}(e_{args}), \sigma \Downarrow \mathsf{Normal}, \sigma'''''}$$

S-Sequence

$$\frac{G, E \vdash s_1, \sigma \Downarrow \mathsf{Normal}, \sigma' \quad G, E \vdash s_2, \sigma' \Downarrow out, \sigma''}{G, E \vdash (s_1; s_2), \sigma \Downarrow out, \sigma''}$$

S-Sequence-Abort

$$\frac{G, E \vdash s_1, \sigma \Downarrow out, \sigma' \quad out \neq \mathsf{Normal}}{G, E \vdash (s_1; s_2), \sigma \Downarrow out, \sigma'}$$

S-Malloc

$$\frac{\begin{array}{c} G, E \vdash e_1, \sigma \Downarrow_{\mathsf{L}} l, \sigma' \qquad G, E \vdash e_2, \sigma' \Downarrow_{\mathsf{R}} \mathsf{I32}\ n, \sigma'' \\ n \geq 0 \wedge (n\ \%\ 4) = 0 \quad (\mathtt{alloc}\ \sigma''\ (n/4)\ \mathtt{true}) = (\sigma''', b) \\ (\mathtt{store}\ \sigma'''\ l\ (\mathsf{Ptr}\ ((b, 0), \emptyset))) = \sigma'''' \end{array}}{G, E \vdash e_1 = \mathsf{malloc}(e_2), \sigma \Downarrow \mathsf{Normal}, \sigma''''}$$

S-Free

$$\frac{G, E \vdash e_1, \sigma \Downarrow_{\mathsf{L}} l, \sigma' \quad \sigma'.\mathsf{mem}(\mathtt{block}\ l).\mathsf{dyn} = \mathtt{true} \quad (\mathtt{free}\ \sigma'\ l) = \sigma''}{G, E \vdash \mathsf{free}(e_1), \sigma \Downarrow \mathsf{Normal}, \sigma''}$$

Figure 5.6: Natural semantics of statements

Updates of statement outcomes:

$$\mathsf{Break} \rightsquigarrow^{loop} \mathsf{Normal} \qquad \mathsf{Return} \rightsquigarrow^{loop} \mathsf{Return} \qquad \mathsf{Return}\ v \rightsquigarrow^{loop} \mathsf{Return}\ v$$

While loops:

S-While-False

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{R}} v, \sigma' \quad \texttt{is\_false}\ v}{G, E \vdash (\mathsf{while}(e)\ s), \sigma \Downarrow \mathsf{Normal}, \sigma'}$$

S-While-Abort

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{R}} v, \sigma' \quad \texttt{is\_true}\ v \quad G, E \vdash s, \sigma' \Downarrow out, \sigma'' \quad out \rightsquigarrow^{loop} out'}{G, E \vdash (\mathsf{while}(e)\ s), \sigma \Downarrow out', \sigma''}$$

S-While-True

$$\frac{G, E \vdash e, \sigma \Downarrow_{\mathsf{R}} v, \sigma' \quad \texttt{is\_true}\ v \quad G, E \vdash s, \sigma' \Downarrow (\mathsf{Normal}|\mathsf{Continue}), \sigma'' \quad G, E \vdash (\mathsf{while}(e)\ s), \sigma'' \Downarrow out, \sigma'''}{G, E \vdash (\mathsf{while}(e)\ s), \sigma \Downarrow out, \sigma'''}$$

For loops:

S-For-Init

$$\frac{s_1 \neq \mathsf{skip} \quad G, E \vdash s_1, \sigma \Downarrow \mathsf{Normal}, \sigma' \quad G, E \vdash (\mathsf{for}(\mathsf{skip}, e_2, s_3)\ s), \sigma' \Downarrow out, \sigma''}{G, E \vdash (\mathsf{for}(s_1, e_2, s_3)\ s), \sigma \Downarrow out, \sigma''}$$

S-For-False

$$\frac{G, E \vdash e_2, \sigma \Downarrow_{\mathsf{R}} v, \sigma' \quad \texttt{is\_false}\ v}{G, E \vdash (\mathsf{for}(\mathsf{skip}, e_2, s_3)\ s), \sigma \Downarrow \mathsf{Normal}, \sigma'}$$

S-For-Abort

$$\frac{G, E \vdash e_2, \sigma \Downarrow_{\mathsf{R}} v, \sigma' \quad \texttt{is\_true}\ v \quad G, E \vdash s, \sigma' \Downarrow out, \sigma'' \quad out \rightsquigarrow^{loop} out'}{G, E \vdash (\mathsf{for}(\mathsf{skip}, e_2, s_3)\ s), \sigma \Downarrow out', \sigma''}$$

S-For-True

$$\frac{G, E \vdash e_2, \sigma \Downarrow_{\mathsf{R}} v, \sigma' \quad \texttt{is\_true}\ v \quad G, E \vdash s, \sigma' \Downarrow (\mathsf{Normal}|\mathsf{Continue}, \sigma'') \quad G, E \vdash s_3, \sigma'' \Downarrow \mathsf{Normal}, \sigma''' \quad G, E \vdash (\mathsf{for}(\mathsf{skip}, e_2, s_3)\ s), \sigma''' \Downarrow out, \sigma''''}{G, E \vdash (\mathsf{for}(\mathsf{skip}, e_2, s_3)\ s), \sigma \Downarrow out, \sigma'''}$$

Figure 5.7: Natural semantics of loops

Fun-Invocation

$$\frac{(\tau, dcl_1, dcl_2, s) = Fd \quad (\texttt{init\_invocation}\ \sigma\ dcl_1\ dcl_2\ v_{args}) = (\sigma', E, bs) \quad G, E \vdash s, \sigma' \Downarrow out, \sigma'' \quad (out, \tau)\ \#\ v_{res} \quad (\texttt{rmerge}\ \sigma''.\mathsf{rstack}\ bs) = R \quad (\texttt{free\_locals}\ \sigma''.\mathsf{mem}\ bs) = M \quad \sigma''' = [\mathsf{mem} := M, \mathsf{rstack} := R, \mathsf{scopes} := \sigma''.\mathsf{scopes}\{si \leftarrow \texttt{false}\}]}{G \vdash Fd(v_{args}), \sigma \Downarrow v_{res}, \sigma'''}$$

Program

$$\frac{(dcl, fun\_dcl) = P \quad \sigma = \texttt{init\_state} \quad G = [\mathsf{env} := \emptyset, \mathsf{defs} := \emptyset] \quad (\texttt{alloc\_globals}\ \sigma\ G\ dcl) = (\sigma', G') \quad (\texttt{alloc\_fun\_defs}\ \sigma'\ G'\ fun\_dcl) = (\sigma'', G'') \quad G''.\mathsf{env}(\text{“}\texttt{main}\text{”}) = b \quad G''.\mathsf{defs}(b) = Fd \quad G'' \vdash Fd(), \sigma'' \Downarrow \mathsf{I32}\ n, \_}{\vdash P \Downarrow n}$$

Figure 5.8: Natural semantics of function invocations and programs

$$
\begin{aligned}
(\#) &: (\mathit{Outcome} \times \mathit{Type}) \to \mathit{Val} \to \mathbb{B} \\
(\mathsf{Normal}, \mathsf{Void}) \# \mathsf{Undef} &\triangleq \mathtt{true} \\
(\mathsf{Return}, \mathsf{Void}) \# \mathsf{Undef} &\triangleq \mathtt{true} \\
(\mathsf{Return}\ v, \tau) \# v &\triangleq \textbf{if } \tau \neq \mathsf{Void} \textbf{ then } \mathtt{true} \textbf{ else } \mathtt{false} \\
\_ \# \_ &\triangleq \mathtt{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{fresh} &: \mathit{ScopeMap} \to (\mathit{ScopeMap} \times \mathit{ScopeId}) \\
\mathtt{fresh}\ \emptyset &\triangleq (\{\mathtt{si}_{\mathtt{main}} \mapsto \mathtt{true}\}, \mathtt{si}_{\mathtt{main}}) \\
\mathtt{fresh}\ S &\triangleq (S\{si \leftarrow \mathtt{true}\}, si) \textbf{ where } si \notin \mathtt{dom}(S)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{deallocate} &: \mathit{Mem} \to \mathit{Block} \to \mathit{Mem} \\
\mathtt{deallocate}\ M\ b &\triangleq M\{b \leftarrow M(b) \textbf{ with } [\mathsf{hi} := 0]\}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{free\_locals} &: \mathit{Mem} \to \mathit{List}(\mathit{Block}) \to \mathit{Mem} \\
\mathtt{free\_locals}\ M\ [] &\triangleq M \\
\mathtt{free\_locals}\ M\ (b : bs) &\triangleq (\mathtt{free\_locals}\ (\mathtt{deallocate}\ M\ b)\ bs)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{alloc\_vars} &: \mathit{State} \to \mathit{Declarations} \to \mathit{Opt}(\mathit{State} \times \mathit{Env} \times \mathit{List}(\mathit{Block})) \\
\mathtt{alloc\_vars}\ \sigma\ [] &\triangleq (\sigma, [], []) \\
\mathtt{alloc\_vars}\ \sigma\ ((\tau, id) : dcl) &\triangleq \textbf{let } (\sigma', b) = (\mathtt{alloc}\ \sigma\ (\mathtt{sizeof}\ \tau)\ \mathtt{false})? \textbf{ in} \\
&\quad \textbf{let } (\sigma'', E, bs) = (\mathtt{alloc\_vars}\ \sigma'\ dcl)? \textbf{ in} \\
&\quad\quad (\sigma'', E\{id \leftarrow b\}, (b : bs))
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{bind\_params} &: \mathit{Mem} \to \mathit{Env} \to \mathit{Declarations} \to \mathit{List}(\mathit{Val}) \to \mathit{ScopeId} \to \mathit{Mem} \\
\mathtt{bind\_params}\ M\ \_\ []\ []\ \_ &\triangleq M \\
\mathtt{bind\_params}\ M\ E\ ((\_, id) : dcl)\ (v : vs)\ si &\triangleq \textbf{let } sl = (E(id), 0) \textbf{ in} \\
&\quad \textbf{in } (\mathtt{bind\_params}\ M\{sl \leftarrow v\}\ E\ dcl\ vs\ si)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{init\_invocation} &: \mathit{State} \to \mathit{Declarations} \to \mathit{Declarations} \to \mathit{List}(\mathit{Val}) \to \\
&\quad \mathit{Opt}(\mathit{State} \times \mathit{Env} \times \mathit{List}(\mathit{Block})) \\
\mathtt{init\_invocation}\ \sigma\ dcl_1\ dcl_2\ v_{args} &\triangleq \textbf{let } (\sigma', E, bs) = (\mathtt{alloc\_vars}\ \sigma\ dcl_1 + dcl_2)? \textbf{ in} \\
&\quad \textbf{let } (S, si) = (\mathtt{fresh}\ \sigma'.\mathsf{scopes}) \textbf{ in} \\
&\quad\quad \textbf{let } M = (\mathtt{bind\_params}\ \sigma'.\mathsf{mem}\ E\ dcl_1\ v_{args}\ si) \textbf{ in} \\
&\quad\quad\quad \textbf{let } R = (\mathtt{rnew}\ \sigma'.\mathsf{rstack}\ si) \textbf{ in} \\
&\quad\quad\quad\quad ([\mathsf{mem} := M, \mathsf{rstack} := R, \mathsf{scopes} := S], E, bs)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{alloc\_globals} &: \mathit{State} \to \mathit{Globals} \to \mathit{Declarations} \to \mathit{Opt}(\mathit{State} \times \mathit{Globals}) \\
\mathtt{alloc\_globals}\ \sigma\ G\ [] &\triangleq (\sigma, G) \\
\mathtt{alloc\_globals}\ \sigma\ G\ (\tau, id) : dcl &\triangleq \textbf{let } (\sigma', b) = (\mathtt{alloc}\ \sigma\ (\mathtt{sizeof}\ \tau)\ \mathtt{false})? \textbf{ in} \\
&\quad \textbf{let } G' = G \textbf{ with } [\mathsf{env} := G.\mathsf{env}\{id \leftarrow b\}] \textbf{ in} \\
&\quad\quad (\mathtt{alloc\_globals}\ \sigma'\ G'\ dcl)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{alloc\_fun\_defs} &: \mathit{State} \to \mathit{Globals} \to \mathit{FunDeclarations} \to \mathit{Opt}(\mathit{State} \times \mathit{Globals}) \\
\mathtt{alloc\_fun\_defs}\ \sigma\ G\ [] &\triangleq (\sigma, G) \\
\mathtt{alloc\_fun\_defs}\ \sigma\ G\ ((id, Fd) : fun\_dcl) &\triangleq \textbf{let } (\sigma', b) = (\mathtt{alloc}\ \sigma\ 1\ \mathtt{false})? \textbf{ in} \\
&\quad \textbf{let } G' = [\mathsf{env} := G.\mathsf{env}\{id \leftarrow b\}, \mathsf{defs} := G.\mathsf{defs}\{b \leftarrow Fd\}] \textbf{ in} \\
&\quad\quad (\mathtt{alloc\_fun\_defs}\ \sigma'\ G'\ fun\_dcl)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{init\_state} &: \mathit{State} \\
\mathtt{init\_state} &\triangleq [\mathsf{mem} := \emptyset, \mathsf{rstack} := [], \mathsf{scopes} := \emptyset]
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{is\_false} &: \mathit{Val} \to \mathbb{B} \\
\mathtt{is\_false}\ v &\triangleq \textbf{if } v = (\mathsf{I32}\ 0) \textbf{ then } \mathtt{true} \textbf{ else } \mathtt{false}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{is\_true} &: \mathit{Val} \to \mathbb{B} \\
\mathtt{is\_true}\ v &\triangleq \textbf{if } v = \mathsf{Undef} \textbf{ then } \mathtt{false} \textbf{ else } \neg(\mathtt{is\_false}\ v)
\end{aligned}
$$

Figure 5.9: Auxiliary functions for Crestrict function invocations, programs and loops

5.3 Memory operations

The memory operations, based on the CompCert memory model [1], are defined in figure 5.10. All memory operations are parametrized by a state σ, whose updated variant is included in the return value in order to propagate modifications. The memory instance mem is directly modified by the operations, and the restrict stack rstack indirectly by utilizing functions which operate on the *RestrictStack* (section 5.4).

Several specific notations are used in the definitions. Location validity in the memory is defined by $M \vDash sl$, which denotes for some *SimpleLoc* $sl = (b, \delta)$ that b is an allocated block and δ is a valid offset within the bounds of that block. Given that a location sl is valid under a memory M, we know that stores and loads of $M(sl)$ will not fail. Hence, we will not explicitly check for failure at places where we use these operations when location validity has been assured. The function `filter_inactive` filters out all bases with an inactive scope as was explained in section 4.4. Finally, the access (*i.e.* both loads and stores) $M(b).\mathsf{contents}(\delta)$ is abbreviated to $M((b, \delta))$.

Memory blocks are allocated through the `alloc` function, by providing the upper bound of the block capacity n (the number of values it must be able to store) and a boolean d indicating whether the allocation is a *dynamic* one (*i.e.* from a call to malloc). Tracking dynamic allocations is used to determine whether the program may free an object. All values of the block are initialized to Undef. The block identifier b is always fresh and never gets reused. The unspecified auxiliary function `new_alloc_id` simply uses the first unused block identifier (starting at 0 and increasing) and thus never fails in our implementation (but permits the possibility of failure for modelling limited memory storage).

When accessing a memory location via `load`, `store` or `free`, the first check is always whether the location sl used for the access is valid by $M \vDash sl$. Then, the restrict checks are applied by a call to the access-type specific function `rload`, `rstore` or `rfree`, which will be explained in section 5.4. These functions take as arguments the current restrict stack and a location whose bases are filtered by `filter_inactive`, based on the active scope map at the time of the access. The result is an optional *RestrictStack*, in which the some case indicates that no restrict related rules were violated and the value is the updated restrict stack. The none case represents that the program is assigned undefined behavior, *i.e.* the access has led to ☣ somewhere in the restrict stack.

If no error has occurred, the `load` function retrieves the value from memory. One additional check is performed to check whether the value does not equal Undef, *i.e.* the program does not load from uninitialized memory. If this check is satisfied, a tuple of the updated state σ' and the retrieved value is returned. Similarly, for the `store` function the value is stored and the updated memory instance is included in the returned state. The `free` function only utilizes the `rfree` function for dynamically allocated locations (as these must be freed by the user). Then, it simply sets the upper bound of the memory block to 0 by the auxiliary `deallocate` function, which indicates the block has been deallocated.

$$
\begin{aligned}
&\texttt{alloc} : \mathit{State} \to \mathbb{Z} \to \mathbb{B} \to \mathit{Opt}(\mathit{State} \times \mathit{Block}) \\
&\texttt{alloc}\ \sigma\ n\ d \triangleq \textbf{let}\ b = (\texttt{new\_alloc\_id}\ \sigma.\mathsf{mem})?\ \textbf{in}\ (\sigma', b) \\
&\quad \textbf{where}\ \sigma' = \sigma\ \textbf{with}\ [\mathsf{mem} := \\
&\quad\quad \sigma.\mathsf{mem}\{b \leftarrow [\mathsf{hi} := n, \mathsf{dyn} := d, \mathsf{contents} := \{\delta \leftarrow \mathsf{Undef} \mid \delta \in (0 \ldots n)\}]\} \\
&\quad]
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{load} : \mathit{State} \to \mathit{Loc} \to \mathit{Opt}(\mathit{State} \times \mathit{Val}) \\
&\texttt{load}\ \sigma\ (sl, bas) \triangleq \textbf{if}\ \sigma.\mathsf{mem} \vDash sl\ \textbf{then} \\
&\quad \textbf{let}\ R = (\texttt{rload}\ \sigma.\mathsf{rstack}\ (sl, (\texttt{filter\_inactive}\ bas\ \sigma.\mathsf{scopes})))?\ \textbf{in} \\
&\quad\quad \textbf{let}\ v = \sigma.\mathsf{mem}(sl)\ \textbf{in} \\
&\quad\quad\quad \textbf{if}\ v \neq \mathsf{Undef}\ \textbf{then}\ (\sigma', v) \\
&\quad\quad\quad\quad \textbf{where}\ \sigma' = \sigma\ \textbf{with}\ [\mathsf{rstack} := R] \\
&\quad\quad\quad \textbf{else}\ \epsilon \\
&\quad \textbf{else}\ \epsilon
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{store} : \mathit{State} \to \mathit{Loc} \to \mathit{Val} \to \mathit{Opt}(\mathit{State}) \\
&\texttt{store}\ \sigma\ (sl, bas)\ v \triangleq \textbf{if}\ \sigma.\mathsf{mem} \vDash sl\ \textbf{then} \\
&\quad \textbf{let}\ R = (\texttt{rstore}\ \sigma.\mathsf{rstack}\ (sl, (\texttt{filter\_inactive}\ bas\ \sigma.\mathsf{scopes})))?\ \textbf{in}\ \sigma' \\
&\quad\quad \textbf{where}\ \sigma' = \sigma\ \textbf{with}\ [\\
&\quad\quad\quad \mathsf{mem} := \sigma.\mathsf{mem}\{sl \leftarrow v\}, \\
&\quad\quad\quad \mathsf{rstack} := R \\
&\quad\quad] \\
&\quad \textbf{else}\ \epsilon
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{free} : \mathit{State} \to \mathit{Loc} \to \mathit{Opt}(\mathit{State}) \\
&\texttt{free}\ \sigma\ (sl, bas) \triangleq \textbf{if}\ \sigma.\mathsf{mem} \vDash sl\ \textbf{then} \\
&\quad \textbf{let}\ R = \\
&\quad\quad \textbf{if}\ \sigma.\mathsf{mem}(\texttt{block}\ sl).\mathsf{dyn}\ \textbf{then} \\
&\quad\quad\quad (\texttt{rfree}\ \sigma.\mathsf{rstack}\ (sl, (\texttt{filter\_inactive}\ bas\ \sigma.\mathsf{scopes})))? \\
&\quad\quad \textbf{else}\ \sigma.\mathsf{rstack} \\
&\quad \textbf{in}\ \sigma' \\
&\quad\quad\quad \textbf{where}\ \sigma' = \sigma\ \textbf{with}\ [\\
&\quad\quad\quad\quad \mathsf{mem} := (\texttt{deallocate}\ \sigma.\mathsf{mem}\ (\texttt{block}\ sl)), \\
&\quad\quad\quad\quad \mathsf{rstack} := R \\
&\quad\quad\quad] \\
&\quad \textbf{else}\ \epsilon
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{filter\_inactive} : \mathit{Bases} \to \mathit{ScopeMap} \to \mathit{Bases} \\
&\texttt{filter\_inactive}\ []\ \_ \triangleq [] \\
&\texttt{filter\_inactive}\ (ba : bas)\ S \triangleq \textbf{let}\ ((sl, bas'), si) = ba\ \textbf{in} \\
&\quad \textbf{if}\ S(si) = \texttt{true}\ \textbf{then} \\
&\quad\quad ((sl, (\texttt{filter\_inactive}\ bas'\ S)), si) : (\texttt{filter\_inactive}\ bas\ S) \\
&\quad \textbf{else} \\
&\quad\quad \texttt{filter\_inactive}\ bas\ S
\end{aligned}
$$

Figure 5.10: Memory operations over states σ and auxiliary function `filter_inactive`

5.4 Restrict stack operations

This section presents the operations which transform the restrict stack and model the semantics of the restrict type qualifier. The operations are based on the adaption of the functional style C-in-$\mathbb{K}$ semantics to accommodate the Crestrict language (chapter 3) and the proposed refinements for the Crestrict semantics (chapter 4).

All operations are parametrized by a restrict stack R, whose updated variant is included in the return value. The operations are performed either when a memory location is accessed or when the scope changes.

When a new scope is invoked (by a function call), the `rnew` function is utilized to push a new tuple onto the restrict stack, composed of the passed *ScopeId* and an empty *RestrictMap*. Analogously, when a scope terminates (by a function return), the function `rmerge` pops the *RestrictMap* rm_m currently on top off the restrict stack. The restrict states of all non-local locations which were modified (*i.e.* which are in the domain of rm_m) are filtered and merged into the *RestrictMap* rm_n which is now on top, if the restrict state of such a location is $\bot$. If such a location has another state, a join was already performed by the `rcheck_rec` rule, and the restrict state of the location within rm_m can simply be discarded.

The rules `rload` and `rstore` are invoked for load and store operations on the memory. Both functions utilize the helper function `rcheck_rec`, which performs the actual algorithm for checking undefined behavior due to a memory access. Basically, the algorithm always takes the map currently on top of the restrict stack, performs a join onto that map with the restrict state representing the memory event and calls itself recursively onto the remaining restrict stack. The recursive call represents the solution presented in section 4.3. Whenever a join resulted in ☣, the function returns ϵ immediately. For the recursive call, the restrict state is filtered in order to remove bases with the scope identifier of the scope which was popped off the restrict stack. The function distinguishes between recursive calls: in the case of a recursive call the state is only joined if the current value does not equal $\bot$, as the restrict stack does not track until which scope a location is valid this prevents assigning a restrict state to locations in scopes in which it is no longer valid.

The `rstore` performs an additional `rmodify` operation, which takes as additional argument the bases of the location used for the store. This operation represents the solution presented in section 4.2.2, which updates the restrict state of the (possibly multiple) restrict qualified object(s) the location is derived from.

Finally, the `rfree` operation also updates the restrict stack if the operation was invoked by a call to free in the program (for dynamically allocated memory locations only). This allows the semantics to give undefined behavior for the program of section 3.3.6.

$$
\begin{aligned}
&\texttt{rnew} : \mathit{RestrictStack} \to \mathit{ScopeId} \to \mathit{RestrictStack} \\
&\texttt{rnew}\ R\ si \triangleq (si, \emptyset) : R \\
\\
&\texttt{rload} : \mathit{RestrictStack} \to \mathit{Loc} \to \mathit{Opt}(\mathit{RestrictStack}) \\
&\texttt{rload}\ R\ (sl, bas) \triangleq \textbf{let}\ rs = (\mathsf{OnlyRead}\ \{bas\})\ \textbf{in}\ (\texttt{rcheck\_rec}\ R\ rs\ sl\ \texttt{false}) \\
\\
&\texttt{rstore} : \mathit{RestrictStack} \to \mathit{Loc} \to \mathit{Opt}(\mathit{RestrictStack}) \\
&\texttt{rstore}\ R\ (sl, bas) \triangleq \textbf{let}\ rs = (\mathsf{Restricted}\ bas)\ \textbf{in} \\
&\qquad \textbf{let}\ R' = (\texttt{rcheck\_rec}\ R\ rs\ sl\ \texttt{false})?\ \textbf{in} \\
&\qquad\quad (\texttt{rmodify}\ R'\ bas) \\
\\
&\texttt{rmodify} : \mathit{RestrictStack} \to \mathit{Bases} \to \mathit{Opt}(\mathit{RestrictStack}) \\
&\texttt{rmodify}\ R\ \emptyset \triangleq R \\
&\texttt{rmodify}\ R\ ((l, \_) : bas) \triangleq \textbf{let}\ R' = (\texttt{rstore}\ R\ l)?\ \textbf{in}\ (\texttt{rmodify}\ R'\ bas) \\
\\
&\texttt{rcheck\_rec} : \mathit{RestrictStack} \to \mathit{RestrictState} \to \mathit{SimpleLoc} \to \mathbb{B} \to \\
&\qquad \mathit{Opt}(\mathit{RestrictStack}) \\
&\texttt{rcheck\_rec}\ []\ \_\ \_\ \_ \triangleq [] \\
&\texttt{rcheck\_rec}\ (si, rm_{cur}) : R\ rs\ sl\ rec \triangleq \textbf{let}\ rm_{hd} = \\
&\qquad \textbf{if}\ (rm_{cur}(sl) = \bot) \wedge rec\ \textbf{then} \\
&\qquad\quad (si, rm_{cur}) \\
&\qquad \textbf{else} \\
&\qquad\quad \textbf{let}\ rs_{new} = rs \sqcup rm_{cur}(sl)\ \textbf{in} \\
&\qquad\qquad \textbf{if}\ rs_{new} \neq \text{☣}\ \textbf{then} \\
&\qquad\qquad\quad (si, rm_{cur}\{sl \leftarrow rs_{new}\}) \\
&\qquad\qquad \textbf{else}\ \epsilon \\
&\qquad \textbf{in}\ (rm_{hd}? : (\texttt{rcheck\_rec}\ R\ (\texttt{filter\_bases}\ (rs, si))\ sl\ \texttt{true})?) \\
\\
&\texttt{rfree} : \mathit{RestrictStack} \to \mathit{Loc} \to \mathit{Opt}(\mathit{RestrictStack}) \\
&\texttt{rfree}\ R\ (sl, bas) \triangleq \textbf{let}\ R' = (\texttt{rcheck\_rec}\ R\ (\mathsf{Restricted}\ bas)\ sl\ \texttt{false})?\ \textbf{in} \\
&\qquad (\texttt{remove}\ R'\ sl) \\
\\
&\texttt{remove} : \mathit{RestrictStack} \to \mathit{SimpleLoc} \to \mathit{RestrictStack} \\
&\texttt{remove}\ []\ \_ \triangleq [] \\
&\texttt{remove}\ (rm : R)\ sl \triangleq (rm \setminus \{(sl, \_)\}) : (\texttt{remove}\ R\ sl) \\
\\
&\texttt{rmerge} : \mathit{RestrictStack} \to \mathit{List}(\mathit{Block}) \to \mathit{RestrictStack} \\
&\texttt{rmerge}\ []\ \_ \triangleq [] \\
&\texttt{rmerge}\ (\_ : [])\ \_ \triangleq [] \\
&\texttt{rmerge}\ ((si_m, rm_m) : (si_n, rm_n) : R)\ bls \triangleq \textbf{let}\ sls = \{sl \mid sl \in \texttt{dom}(rm_m) \wedge sl \notin bls \wedge rm_n(sl) = \bot\}\ \textbf{in} \\
&\qquad \textbf{let}\ rm_o = \{sl \mapsto (\texttt{filter\_bases}\ (rm_m(sl), si_m)) \mid sl \in sls\} \cup \\
&\qquad\qquad\qquad \{sl \mapsto rm_n(sl) \mid sl \in \texttt{dom}(rm_n) \wedge sl \notin sls\}\ \textbf{in} \\
&\qquad\quad (si_n, rm_o) : R
\end{aligned}
$$

Figure 5.11: Restrict operations over restrict stacks R

Chapter 6

Evaluation

To be able to test whether a given Crestrict program contains undefined behavior due to violations of the restrict semantics, we have developed an interpreter which can run such programs and dynamically track whether the operations it performs lead to undefined behavior. This implementation makes the semantics *executable* and is described in section 6.1. We have created a test suite composed of 96 test programs. For each program we expect it to have either defined or undefined behavior according to the ISO/IEC standard definition of restrict. Furthermore, all tests we classify as *defined* do not lead to incorrect program outputs when compiled and optimized by GCC and Clang. The categorization and other details of this test suite are described in section 6.2.

6.1 Implementation

The CRESTRICT interpreter is written in the Rust programming language. It utilizes the LANG-C parser[1] crate, which parses C source files into an abstract syntax tree (AST). This AST is converted into a Crestrict AST. If any unsupported language constructs were used, an error is returned immediately. The AST is then type checked. If type checking failed an error is returned immediately. The Crestrict program is then interpreted according to the operational rules, resulting in either a result code or an error message indicating what problem occurred, *i.e.* the execution aborts immediately upon detecting undefined behavior due to memory operations or a violation of the restrict semantics.

There are only a few differences between the implementation and the operational semantics presented in chapter 5 which are worth mentioning. None of these differences are fundamental for the semantics, but rather an implementation choice that had to be made.

Firstly, instead of an $Opt(\tau)$ type denoting failure, the implementation uses Rust's `Result`$\langle\tau, String\rangle$ type. The `Err` constructor is annotated with a string providing information as to where an error occurred.

Secondly, the implementation accepts a slightly larger language than the one presented in section 5.1: initializers for global and local variables (including function calls) are also supported. The initializer expressions are evaluated in declaration order and their values are directly stored into the memory. Importantly, initializer values do **not** alter the restrict stack (as is also the case for the C-in-$\mathbb{K}$ semantics). This choice is supported by the standard definition, *i.e.* an initializer does not *modify* an object as there was nothing beforehand. Finally, declarations inside the initial statement of a for loop are also supported. The variable declaration is moved into the list of variable declarations of the function (which may only occur at the beginning), and the initializer is treated is described above.

[1] Available at `https://github.com/vickenty/lang-c`

6.2 Feature categorization

To the best of our knowledge, there is no publically available test suite dedicated to the restrict type qualifier (although tests do exist as part of other test suites). Therefore, we created our own suite to evaluate CRESTRICT and KCC against (and when possible we have included tests from other test suites). The test suite only contains programs whose syntax is supported by the Crestrict language and was created simultaneously with the refined restrict semantics, due to which CRESTRICT passes all tests. All the examples which were presented in section 3.3 are included in the test suite. We note that we have no tests for assignments between restrict pointers, as this was out of scope for this thesis (section 2.4).

The features are categorized based on the *context* in which a restrict qualified pointer occurs, which affects how the presence of the type qualifier should be interpreted. For completeness, the restrict-related tests from the KCC interpreter and the examples from the ISO/IEC standard are also included. An overview of the different features and the number of defined/undefined tests is given in table 6.1. We will also point out for which kind of tests per feature KCC gives another result than CRESTRICT. For clarity, note that KCC is a much more mature tool in the sense that it is able to detect a lot of different kinds of undefined behavior and accepts a much larger language as input. CRESTRICT on the other hand has the sole focus of detecting undefined behavior which arises due to uses of restrict.

Feature	#DB	#UB	Description
Aggregate type	2	3	Restrict pointers within an aggregate datatype.
Global	5	4	Restrict pointers/lvalues with global scope.
Nested	9	29	Nested pointers with the restrict type qualifier on one of the inner types.
KCC	1	2	Tests for restrict from the KCC test suite[2].
Standard	2	1	Restrict examples from the ISO/IEC C11 standard [15].
Other	18	19	Other tests representing possible useful programs for restrict.
Total	38	58	Total: 96

Table 6.1: Feature categorization

In the context of Crestrict, only a single **Aggregrate type** exists: the array data type. The tests for this feature have one or more restrict pointers as part of an array. The undefined behavior tests fail for KCC, due to the problem described in section 3.3.4.

Global is the feature describing global restrict pointers and global pointers based on some restrict pointer. One of the defined behavior tests fails for KCC, due to the problem described in section 3.3.5.

Nested is the feature describing nested pointers with the restrict type qualifier on one of the inner pointer types. The defined behavior tests all pass for KCC, but ∼44% of the undefined behavior tests fail due to the problem described in section 3.3.2.

KCC has actually eight tests related to restrict. Four of them test assignments between restrict pointers and are therefore omitted. One test uses unsupported language features (const) and has therefore also been omitted. Finally, one test was moved from undefined behavior to defined behavior in our suite, because we do not support type casts.

Standard contains the example programs from the ISO/IEC standard. Only the two

[2] `https://github.com/kframework/c-semantics/tree/master/tests`

combinations of example two and example three are included, as the other examples use unsupported language features (struct) or demonstrate illegal assignments between restrict pointers. Furthermore, one of the defined behavior tests fails for KCC due to the problem described in section 3.3.1.

Finally, **Other** contains test programs of both realistic use cases of restrict and problematic programs not captured by the features listed above. Two of the defined behavior tests fail for KCC due to the aliasing loads problem discussed in section 3.3.1 and two of the undefined behavior tests fail for KCC due to the problem with the deferred check (section 3.3.3) and call to free (section 3.3.6).

Chapter 7

Related work

Developing a formal semantics for the C programming language has been quite an active research area. Due to the imprecision of natural language and the desire for a precise semantics (*e.g.* to state and prove theorems about the language), various works with their own respective goals have been created over the past decades. In section 7.1 we describe some of the memory models used by these semantics and how they relate to our work. In section 7.2 we compare our functional implementation of the restrict fragment of the C-in-$\mathbb{K}$ semantics with their original implementation. Finally, in section 7.3 we discuss two related semantics for the Rust programming language.

7.1 Memory models

In section 3.1 we have previously discussed the differences between concrete, abstract and hybrid memory models. In this section we will describe some of the memory models which have been created for C.

Norrish (1998) created one of the earlier formal C specifications by defining a structural operational semantics in the HOL theorem prover [38]. As he treats C89, restrict was not yet added to the language and thus is not part of his semantics. The operational semantics are composed of both a small-step semantics (for expressions) and a big-step semantics (for statements). Upon this semantics and the accompanying formalization of C's type system, Norrish proved type preservation and type safety for expressions. The memory model is a concrete one: it is implemented as a map from addresses to bytes.

Leroy *et al.* (2006) formalized a large part of C11 for the CompCert project in the Coq proof assistant [26, 29]. The main result of their work is the eponymous optimizing CompCert compiler, for which they proved a semantic equivalence between the source program (compiler input) and emitted machine code (compiler output).

The compiler has several passes, of which the first few (except the parser [18]) are not verified: the preprocessed C-source code is parsed, elaborated into an CompCert C AST and then type checked. The AST then is converted into a C-like language called Clight [3], in which expressions are pure. The next few passes are all verified: each pass compiles its input language into an intermediate language, up until one of the supported assembly languages PowerPC, ARM or x86. Assembling and linking the sources into an executable is also supported, but not verified.

If a compiler pass is verified, this means that the pass upholds the *semantics preservation* property: the observable behaviors produced by a correct output program are acceptable behaviors of the source program. This notion accounts for the unspecified evaluation orders of expressions, because the output program might have less observable

behaviors than the source program due to the compiler making a choice on the order. The project uses a small-step operational semantics for all the languages (including the C source, intermediate and assembly languages).

The memory model of CompCert is abstract and was previously explained in detail in section 3.1. This model does not support interpreting pointers as integer values, *e.g.* for pointer-to-integer casts or low-level idioms such as `mmap` returning -1 indicating that no memory is available.

Besson *et al.* propose a more concrete model in which abstract pointers are mapped to numeric addresses, allowing for reasoning about the binary encoding of pointers [1] while preserving deterministic allocations (which is required for proofs of compiler passes). They show that the existing memory model is an abstraction of the more concrete one.

Kang *et al.* propose a "quasi-concrete memory model" which gives semantics to bit-manipulation of pointer values [20]. Pointer values in their model are abstract by default and *realized* to a concrete pointer value upon pointer-integer cast. They argue that their semantics are easier than Besson *et al.*'s because normalization is a straightforward translation from pointers to concrete blocks and integers, whereas Besson *et al.* use an SMT solver and have more complex semantics in general. Integrating either of these semantics could be considered future work, we will discuss this more in the context of Memarian *et al.* [37], later in this section.

Campbell gives an executable semantics (in the form of an interpreter) for CompCert C to help "gaining faith in the CompCert C semantics" and shows its equivalence to the original semantics [4]. Using the interpreter for testing, he found several bugs in both the semantics and compiler.

Several ideas from CompCert have influenced this thesis. Firstly, the Crestrict source language we define our semantics upon (chapter 5) is based on the Clight language [3]. Secondly, the memory model we use is based on their abstract memory model.

Krebbers (2015) gives a formal semantics for a large part of the non-concurrent fragment of C11 in the CH_2O project [24]. Their memory model incorporates C's aliasing restrictions [22]. Similar to restrict, a compiler exploits the *strict aliasing rule*, which states that pointers to different types cannot alias, to perform optimizations. However, because C contains the *union* type (which is an untagged sum type) this rule cannot be statically enforced by the type system.

The relation with *type-punning* (reading a union through a variant it is not in) leads to subtle interactions which are not easy to capture in a formal semantics. Krebbers follows the GCC documentation, which states that type-punning is only allowed if "the memory is accessed through the union type". His memory model, based on CompCert, uses well-typed trees with arrays of bits as the contents of a memory object instead of an array of bytes. Bits are used as the smallest unit to be able to deal with bit fields of structs. The use of trees allows the model to capture the *effective type* of a memory area, which is represented by its state. Consequently, pointer values are pairs of object references and a path through the tree. The model is implemented in the Coq proof assistant and includes a proof of correctness for the strict-aliasing theorem (which captures the strict aliasing rule) and other properties of memory operations.

The semantics also have an executable variant, which computes the set of all allowed behaviors (which can be plural due to non-determinism) [23]. The executable semantics have been implemented in an interpreter, and were proven sound and complete with respect to the operational semantics.

Memarian *et al.* (2019) give an executable model for a large fragment of the "de facto" C11 semantics, called The Cerberus C semantics [36]. They observe that the values of C cannot be seen as purely concrete nor abstract. Their *concreteness* is required for C's features to manipulate of the underlying representation of values, such as integer-pointer casts and using `char` pointers to access individual bytes. Their *abstractness* is required for reasoning about provenance of pointers (a concept which was previously explained in section 3.1), exploited by compilers to justify optimizations. They point out that the ISO/IEC standard does not explicitly define a notion of provenance, but a response by the ISO WG14 C standards committee [16] does indicate that implementors may utilize such a concept.

Pointer values are represented by pairs (π, a), in which a is a concrete address and π is its associated provenance, which is either a fresh allocation identifier $@i$ or the empty provenance. In order to further explore the design space (also taking into account existing C code), they have developed two semantics for pointers and memory objects in C, with a focus on integer-pointer casts [37]. The **PVI** semantics associates a provenance to not only pointer values but also integer values. The provenance is preserved throughout integer-pointer casts. The **PNVI** semantics (for which three variants are defined) takes a different approach and instead assigns the provenance at integer-to-pointer cast points. They report that the PVI semantics suffers from the loss of algebraic properties of integer arithmetic and some other problems, while PNVI makes some existing compiler behaviors unsound. Because PNVI is simpler to define and explain than PVI, they argue PNVI is preferable. In 2022 a proposal for a "Provenance-aware Memory Object Model" under N3005 was submitted [10]. This model is, among other papers, based on the PNVI model.

The type of provenance considered in their semantics is used to distinguish addresses based on allocation. This allows the PNVI semantics to (re)construct provenance upon integer-pointer casts: the numerical address of the pointer must be consistent with its provenance in the sense that it points into a "live" allocation and has the correct type. For the type of provenance we use (bases) it is not clear if and how it is possible to apply this idea. We therefore expect that our semantics for restrict would best integrate into the PVI model. As a small example, consider the function `foo` below. Depending on the implementation of `to_int` the pointer r may be based on p. If the provenance is preserved throughout casts, q will have the correct provenance and after casting back to a pointer, this provenance is propagated to pointer expression r.

```
1  extern uintptr_t to_int(int*);
2
3  int foo(int* restrict p) {
4      uintptr_t q = to_int(p);
5      int* r = (int*) q; // What bases should *r have?
6
7      return *r;
8  }
```

7.2 Comparison with C-in-$\mathbb{K}$

The C-in-$\mathbb{K}$ semantics has been very fundamental for this thesis, as we have taken the restrict fragment of this semantics as a basis for developing a more complete semantics for restrict. There are two fundamental differences between our semantics and the original C-in-$\mathbb{K}$ semantics.

1. We went from a rewrite-based to a functional-based definition of the C-in-$\mathbb{K}$ semantics for restrict (chapter 3).

2. We introduced new semantic domains and rules (chapter 4 and 5) to refine the semantics in order to deal with the problems described in section 3.3.

As part of the first difference, we have made several changes to the representation of the C-in-$\mathbb{K}$ domains. These are either purely esthetic, for simplicity or to adapt the semantics for the functional-based semantics. An overview of these changes is given in table 7.1. Despite these discrepancies, the presented examples in this thesis exhibit the same behavior in both the functional implementation described in section 3.2 and the original generated KCC interpreter. Furthermore, the granularity of our memory model is at a higher level of abstraction than KCC, *i.e.* at the level of values and not bytes. This is appropriate because we consider only a small C-like language whereas KCC considers the complete C99 and a large fragment of the C11 language and therefore has needs to be able to manipulate the underlying representation of values.

The original KCC interpreter was evaluated under the GCC Torture Tests [6], a subset of the Juliet Test Suite for C/C++ and their own Undefinedness Test Suite [12]. Except the latter, whose restrict related tests we previously discussed in more detail in chapter 6, none of these test suites include tests which check for undefined behavior induced by restrict. This is also a clear difference with our evaluation, in which we included a lot more tests specifically for restrict.

The C-in-$\mathbb{K}$ project has been continued into the commercial tool RV-Match, "an improved tool for doing practical analysis of real C programs" [11]. The implementation is closed-source, so we do not know whether it has a different semantics for restrict than the original KCC interpreter (but judging from the paper it seems the original semantics are used). They evaluate the tool by running it on the Toyota ITC Benchmark [42]. This benchmark also does not have restrict related tests, so we do not know if it detects more undefined behavior induces by restrict.

```
1  int* restrict p; // Stored at sl_p = (b_p, 0), (type p) = ((Ptr I32), GlobalRestrict)
2
3  // Scope si_foo
4  void foo() {
5      int x;                  // Stored at sl_x,
6      p = &x;                 // Declaration scope ≠ si_foo: M(sl_p) = Ptr (sl_x, {(b_p, si_main)})
7  }
8
9  // Scope si_bar
10 void bar() {
11     int y;                  // Stored at sl_y,
12     int* restrict q = &y;   // Stored at sl_q = (b_q, si_bar)
13                             // Declaration scope = si_bar: M(sl_q) = Ptr (sl_y, {(b_q, si_bar)})
14 }
```

Listing 7.1: Restrict declaration scopes and bases

K-sources	**Functional representation**	**Justification**
⟨`restrict`⟩ cell	$R \in \mathit{RestrictStack}$	The restrict stack is an evaluation judgment parameter rather than a global state that can be operated on. The content type remains unchanged.
⟨`types`⟩/ ⟨`local-types`⟩	Expressions are annotated with their type, the supported type qualifiers distinguish Restrict and GlobalRestrict	The ⟨`types`⟩ cell maps identifiers to their types. There is a special relation with the restrict semantics: restrict qualified types are *tagged* with the scope (encapsulated in a *RestrictBlock*) in which they are declared. This is used to add the *BasedOn* provenance to restrict pointer values, in which the associated declaration scope is included. In the functional representation we do not have a mapping for types at "runtime", but type checked the program before execution. We distinguish GlobalRestrict and Restrict as type qualifiers. For GlobalRestrict we know the declaration scope of the provenance must be $\texttt{si}_{\texttt{main}}$. Because variables and parameters local to a function are not accessible from other scopes (compound expressions are not supported), the active scope in the program execution can be used when a scope identifier needs to be assigned to a base (in the case of type qualifier Restrict). An example is given in listing 7.1.
Scope ::= | FileScope | PrototypeScope | BlockScope(*functionId*, *functionLoc*, *block*)	$\mathit{si} \in \mathit{ScopeId}$	Scopes are simplified to unique scope identifier numbers *si* instead of distinguishing function, file, block and function prototype scopes [15, 6.2.1].
$\top$	☣	Emphasize that the undefined behavior state is to be seen as something negative rather than something positive.

Table 7.1: Representational discrepancies between the C-in-$\mathbb{K}$ domains and chapter 3

Annotations of pointer values in memory

In the annotations for clarification of programs we sometimes act as if a specific pointer value stored in memory has a provenance. For example, if x is stored at sl_x and p at $sl_p = (b_p, 0)$ in the snippet `int x; int* restrict p = &x;`, then we state that the memory at location sl_p contains Ptr $(sl_x, \{(b_p, si)\})$. Actually, the provenance will not be added until p is evaluated in rvalue position. This corresponds to the rule E-Lval-Conv-Restrict in our semantics. We have abstracted this away in order to simplify explaining problematic memory accesses with respect to the restrict semantics.

Omission of has-restrict

The kcc interpreter performs an optimization to speed up restrict related checks, as these can consume quite some time. This optimization is based on the ⟨`has-restrict`⟩ cell, which records whether restrict-qualified pointers are dynamically in scope. If no such pointers are in scope, the checks for restrict violations are disabled. Unfortunately, we have found that this optimization is incorrect for some programs, as disabling the checks might change whether a program has undefined behavior or not. That is, the restrict semantics are not preserved under this optimization.

A problematic program is given in listing 7.2. The accesses to sl_y in the scopes $\texttt{si}_\texttt{f}$ and $\texttt{si}_\texttt{g}$ lead to the restrict state Unrestricted. The store to sl_y at line 17 in the scope $\texttt{si}_\texttt{main}$ means that the restrict checks lead to ☣ due to the join of Restricted and OnlyRead $\emptyset$.

Now, if line 16 would not be present, the scope $\texttt{si}_\texttt{main}$ would not have any restrict pointers in scope and thus the optimization would disable all restrict checks for this scope. If the restrict checks are not executed for $\texttt{si}_\texttt{main}$, the program would become well-defined! Note that the trick at line 16 does not otherwise affect the semantics of the program. This shows that the optimization might change the restrict semantics.

For all example programs which we evaluated under the C-in-$\mathbb{K}$ semantics in thesis, one may assume we used the "most restrictive variant" of their semantics. That is, we have completely omitted the ⟨`has-restrict`⟩ cell and optimization performed by it.

```
1  int y; // Stored at sl_y
2  int z = 0;
3
4  // Scope si_g
5  void g() {
6      z = y; // Load from y, R = [(si_g, {sl_y ↦ OnlyRead ∅}), (si_f, ∅), (si_main, {sl_y ↦ Restricted ∅})]
7  }
8  // Scope si_f
9  int f(int* restrict x) {
10         // x is stored at sl_x = (b_x, 0). M(sl_x) = Ptr (sl_y, {(b_x, si_f)})
11     g(); // After the deferred check, R = [(si_f, {sl_y ↦ OnlyRead ∅}), (si_main, {sl_y ↦ Restricted ∅})]
12     return *x; // Load via x, R = [(si_f, {sl_y ↦ Unrestricted}), (si_main, {sl_y ↦ Restricted ∅})]
13 }
14 // Scope si_main
15 int main() {
16     int* restrict _;     // A trick to circumvent the <has-restrict> based optimization
17     y = 0;               // Store to y, R = [(si_main, {sl_y ↦ Restricted ∅})]
18
19     f(&y);               // Now, the deferred check attempts to perform the join
20                          // Unrestricted ⊔ Restricted = ☣
21                          // But without line 16 this check would not take place
22
23     return z;
24 }
```

Listing 7.2: An example program demonstrating the incorrectness of the ⟨`has-restrict`⟩ based optimization

7.3 Rust: Stacked and Tree Borrows

The Stacked Borrows Aliasing Model by Jung *et al.* [19] addresses a problem in the Rust programming language which is conceptually closely related to the one we address in this thesis. In Rust, mutable references `&mut T` provide even stronger aliasing guarantees than restrict pointers in C: they cannot alias with anything else in scope, which is statically enforced by the compiler. Like in C, the Rust compiler (rustc) gratefully exploits this aliasing information to justify optimizations. However, Rust also has *unsafe* code, which are explicitly annotated blocks in a program for which the compiler poses fewer restrictions than safe code. More specifically, using unsafe Rust, one can alias `&mut T` variables, thus circumventing the static rules for safe Rust and invalidating the aliasing guarantees.

Stacked Borrows addresses this problem by giving an operational semantics for memory accesses, which compose an *aliasing discipline* valid Rust programs have to adhere to and assigns undefined behavior to programs failing to do so (which do not have to be considered by the compiler). This discipline could be considered a dynamic version of the *borrow checker*, the static analysis performed by rustc to check that safe Rust adheres to the aliasing rules.

The key idea is that the memory model is extended with a per-location stack containing items for various kind of pointers that are able to access the memory location. Pointers are tagged with a unique identifier $t \in (\mathbb{N} \cup \{\bot\})$ in order to distinguish them. A set of rules describe how the stack progresses when new pointers are created and the memory is accessed, and what kind of memory operations are allowed given a state. The stack represents that correct usage of references follows a stack discipline: reborrows are pushed on top of the stack, and usage of a pointer pops all items above it.

As a small example, consider the program below. The variable l is allocated and then reborrowed from by x and y. When x is used, the pointer y is popped off the stack because it is above x. When the program attempts to use y for a write access, the model assigns undefined behavior to the program because y is not in the borrow stack.

```
1  let mut l = 0;     // [l]
2  let x = &mut l;    // [x, l]
3  let y = &mut l;    // [y, x, l]
4  *x = 1;            // Pop everything above x: [x, l]
5  *y = 2;            // UB: y is no longer in the borrow stack
```

The example program demonstrates only a small fragment of the operational semantics. The actual model distinguishes three kinds of pointers: *Unique(t)* (unique mutable access), *SharedRO(t)* (shared read-only access) and *SharedRW(⊥)* (shared mutable read and write access).

The semantics are also implemented in *Miri*, an interpreter for Rust, which either returns the program result for valid programs or an error indicating the cause of undefined behavior.

Both Clang and rustc aim to emit LLVM's *noalias* attribute for restrict qualified and `&mut T` variables. Stacked Borrows uses the Rust standard library test suite to ensure that it does not give overly much undefined behavior for existing code patterns and Coq proofs of compiler transformations to ensure it gives enough undefined behavior. This is a clear difference with our semantics, in which we try to follow the standard definition where possible.

Villani [44] proposes another aliasing model, called Tree Borrows, which is based on Stacked Borrows and aims to supersede it. He argues that the Stacked Borrows model is too strict, and a number of Rust crates would break if rustc were allowed to optimize based on Stacked Borrows. The main argument is that the stack data structure loses information: it does not distinguish reborrows from child pointers and parent pointers. The model is unfinished at the time of writing and still in the process of being evaluated.

Chapter 8

Conclusion and future work

In this work we have presented Crestrict, an operational semantics for the C99 restrict type qualifier based on the restrict fragment of the C-in-$\mathbb{K}$ semantics. We have seen arguments showing that the existing semantics inadequately models the type qualifier for six specific programs, and proposed fixes which refine the semantics. We incorporated the new semantics in a small but representative C-like language. The test suite we created demonstrates that for all test programs for which we argued the C-in-$\mathbb{K}$ semantics gives too little undefined behavior Crestrict gives more undefined behavior and vice versa. The interpreter makes the semantics executable and allows one to systematically test whether a given program utilizing restrict has undefined behavior. This achieves our goal of providing an alternative resource for restrict, which is not subject to the complex and error-prone definition in natural language of the ISO/IEC standard.

8.1 Future work

There are various ways in which one could extend our work and we will describe some of them in this section.

Assignments between restrict pointers The first obvious extension is complete support for restrict. As we pointed out in section 2.4 we have omitted the restriction for assignments between restrict pointers in our semantics. This restriction basically states that only "outer-to-inner" assignments between restrict pointers declared in nested blocks are well-defined [15, 6.7.3.1, p4]. For example, in the code below the assignments of q to p and r to p induce undefined behavior, but the assignment of p to r is well-defined. In this thesis, we have manually ensured that this restriction is not violated by any of the example programs, to make sure it does not conflict with the semantics we have presented for restrict.

```
int* restrict p;
int* restrict q;
p = q; // Undefined behavior
{
    int* restrict r = p; // Well-defined
    p = r;               // Undefined behavior
}
```

Extend to a larger language The language considered in this thesis is relatively small and has no support for *e.g.* structs and integer pointer casts. We expect incorporating structs and supporting restrict pointers as struct members should be compatible with our semantics due to the refinement for array types (section 4.2). The integer-pointer casts

problem seems more tricky, and an exploration combining our semantics and Memarian *et al.* [37] could be interesting. Furthermore, general type casts (*e.g.* casting the restrict qualifier away from a pointer type) and const (which in conjunction with restrict could possibly permit more optimizations) could be interesting.

Verification of program optimizations Another interesting road which can be explored is formally showing that the proposed Crestrict semantics permits some desired compiler optimizations. This could be done in the style of CompCert, *i.e.* a method similar to the semantic preservation property previously discussed in section 7.1. An alternative is Simuliris [8], a framework for the verification of concurrent program optimizations. Simuliris has a strong focus on concurrency and establishes "fair termination preservation" for many optimizations, meaning that under a fair scheduler a terminating program is not allowed to be turned into a diverging one. Similar to CompCert, a correct program optimization means that the result of the optimized program must be a possible result of the original program. The framework demonstrates its effectiveness by instantiating it for a new concurrent Stacked Borrows (the Rust aliasing discipline previously explained in section 7.3), and verifying the correctness of the original optimizations.

Program logic The development of a program logic for Crestrict could be used to formally verify that a program does not contain undefined behavior induced by restrict.

Krebbers [24] defines a separation logic for CH_2O core C, and proved it sound with respect to the operational semantics. Program verification employing this logic is done by proving a Hoare triple $\{P\}\ s\ \{Q\}$, in which s is a statement, P the precondition and S the postcondition. Proving such a triple also ensures the absence of undefined behavior.

Louwrink [32] defines a separation logic for Stacked Borrows. Similar to Krebbers, he also states the adequacy theorem, from which it follows that if a Hoare triple can be derived for a program it does not have undefined behavior.

Axiomatic semantics As we explained in chapter 1, the ISO/IEC definition of restrict uses a rather axiomatic description. We have chosen to create an operational semantics in this thesis as this type of semantics lends itself better for implementation. As a result, our semantics does not relate as closely to the standard description as one could accomplish with an *axiomatic semantics*. Defining such a semantics and possibly show an equivalence relation with our semantics could therefore be valuable.

Noalias As we pointed out in section 7.3, both Clang and rustc aim to emit the LLVM `noalias` attribute for C's `T * restrict` and Rust's `&mut T` types within parameter declarations. The LLVM description of noalias[1] states that its definition is intentionally similar to the definition of C's restrict, but also points out some differences (*e.g.* annotating return values with noalias has a special meaning). Creating a formal semantics for noalias, or extending existing formal semantics for LLVM such as Vellvm [46] or K-LLVM [31] would therefore be valuable. Having such a semantics would also pave the way to formally show whether Clang preserves the program semantics by replacing all occurrences of restrict in function parameter declarations with noalias.

[1] `https://llvm.org/docs/LangRef.html#parameter-attributes`

Bibliography

[1] Besson, F., Blazy, S., and Wilke, P. A concrete memory model for CompCert. In *Interactive Theorem Proving: 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings 6* (2015), Springer, pp. 67–83.

[2] Biener, R. [meta-bug] restrict qualification aliasing issues. `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=49774`.

[3] Blazy, S., and Leroy, X. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning 43*, 3 (2009), 263–288.

[4] Campbell, B. An executable semantics for CompCert C. In *International Conference on Certified Programs and Proofs* (2012), Springer, pp. 60–75.

[5] Dobbelaere, J. Full 'restrict' support in LLVM. `https://lists.llvm.org/pipermail/llvm-dev/2019-October/135672.html`, March 2019.

[6] Ellison, C., and Roşu, G. An executable formal semantics of C with applications. *ACM SIGPLAN Notices 47*, 1 (2012), 533–544.

[7] Fasselt, M. restrict, static & inline Keywords in C. `https://hps.vi4io.org/_media/teaching/wintersemester_2013_2014/epc-1314-fasselt-c-keywords-report.pdf`, 2014.

[8] Gäher, L., Sammler, M., Spies, S., Jung, R., Dang, H.-H., Krebbers, R., Kang, J., and Dreyer, D. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages 6*, POPL (2022), 1–31.

[9] Gustedt, J. The semantics of the restrict qualifier. `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3234.htm`, March 2024.

[10] Gustedt, J., Sewell, P., Kayvan, M., Gomes, V. B., and Uecker, M. A Provenance-aware Memory Object Model for C. `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3005.pdf`, June 2022.

[11] Guth, D., Hathhorn, C., Saxena, M., and Roşu, G. Rv-match: Practical semantics-based program analysis. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I 28* (2016), Springer, pp. 447–453.

[12] Hathhorn, C., Ellison, C., and Roşu, G. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), pp. 336–345.

[13] Huisman, M., and Jacobs, B. Java program verification via a Hoare logic with abrupt termination. In *International Conference on Fundamental Approaches to Software Engineering* (2000), Springer, pp. 284–303.

[14] IEEE Spectrum. The Top Programming Languages 2023. https://spectrum.ieee.org/the-top-programming-languages-2023, 2023.

[15] ISO. *ISO/IEC 9899:2018 Information technology — Programming languages — C*, fourth ed. June 2018.

[16] ISO/IEC JTC I, SC22, W. *Defect report #236.* Tech. rep. https://www.open-std.org/jtc1/sc22/wg14/www/docs/dr_260.htm, 2004.

[17] Johnson, T. A., and Homer, B. Clarifying the restrict Keyword v2. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2260.pdf, May 2018.

[18] Jourdan, J.-H., Pottier, F., and Leroy, X. Validating LR (1) parsers. In *European Symposium on Programming* (2012), Springer, pp. 397–416.

[19] Jung, R., Dang, H.-H., Kang, J., and Dreyer, D. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages 4*, POPL (2019), 1–32.

[20] Kang, J., Hur, C.-K., Mansky, W., Garbuzov, D., Zdancewic, S., and Vafeiadis, V. A formal C memory model supporting integer-pointer casts. *ACM SIGPLAN Notices 50*, 6 (2015), 326–335.

[21] Klappe, T., and Krebbers, R. Crestrict: an operational semantics for the C99 restrict type qualifier. https://doi.org/10.5281/zenodo.11031862, Apr. 2024.

[22] Krebbers, R. Aliasing restrictions of C11 formalized in Coq. In *International conference on certified programs and proofs* (2013), Springer, pp. 50–65.

[23] Krebbers, R., and Wiedijk, F. A typed C11 semantics for interactive theorem proving. In *Proceedings of the 2015 Conference on Certified Programs and Proofs* (2015), pp. 15–27.

[24] Krebbers, R. J. *The C standard formalized in Coq.* PhD thesis, [Sl]:[Sn], 2015.

[25] Leroy, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2006), pp. 42–54.

[26] Leroy, X. Formal verification of a realistic compiler. *Communications of the ACM 52*, 7 (2009), 107–115.

[27] Leroy, X., Appel, A. W., Blazy, S., and Stewart, G. *The CompCert memory model, version 2.* PhD thesis, Inria, 2012.

[28] Leroy, X., and Blazy, S. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning 41* (2008), 1–31.

[29] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress* (2016).

[30] Leroy, X., and Grall, H. Coinductive big-step operational semantics. *Information and Computation 207*, 2 (2009), 284–304.

[31] Li, L., and Gunter, E. L. K-LLVM: a relatively complete semantics of LLVM IR. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)* (2020), Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[32] Louwrink, D. A Separation Logic for Stacked Borrows. `https://eprints.illc.uva.nl/id/document/11131`, 2021.

[33] MacDonald, T., and Homer, B. Provenance-Style Specification of restrict. `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3058.htm`, October 2022.

[34] MacDonald, T., and Homer, B. Revision 2 Of Defect With Wording Of restrict Specification. `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3225.htm`, February 2024.

[35] MacDonald, T., Tong, H., and Uecker, M. Defect With Wording Of restrict Specification. `https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3025.htm`, July 2022.

[36] Memarian, K. The Cerberus C semantics. Tech. rep., University of Cambridge, Computer Laboratory, 2023.

[37] Memarian, K., Gomes, V. B., Davis, B., Kell, S., Richardson, A., Watson, R. N., and Sewell, P. Exploring C Semantics and Pointer Provenance. *Proceedings of the ACM on Programming Languages 3*, POPL (2019), 1–32.

[38] Norrish, M. C formalised in HOL. Tech. rep., University of Cambridge, Computer Laboratory, 1998.

[39] Roşu, G., and Ştefănescu, A. Towards a unified theory of operational and axiomatic semantics. In *International Colloquium on Automata, Languages, and Programming* (2012), Springer, pp. 351–363.

[40] Roșu, G., and Șerbănuță, T. F. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming 79*, 6 (2010), 397–434.

[41] Sewell, P., Memarian, K., and Gomes, V. B. C provenance semantics: detailed semantics (for PNVI-plain, PNVI address-exposed, PNVI address-exposed user-disambiguation, and PVI models).

[42] Shiraishi, S., Mohan, V., and Marimuthu, H. Test suites for benchmarks of static analysis tools. In *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2015), IEEE, pp. 12–15.

[43] TIOBE Software. TIOBE Index. `https://www.tiobe.com/tiobe-index/`, 2024.

[44] Villani, N. Tree Borrows. `https://github.com/Vanille-N/tree-borrows/blob/master/half/main.pdf`, 2023.

[45] Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., and Kaashoek, M. F. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems* (2012), pp. 1–7.

[46] Zhao, J., Nagarakatte, S., Martin, M. M., and Zdancewic, S. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2012), pp. 427–440.