Abstract—Implicit Computational Complexity (ICC) aims at giving machine-free characterisations of complexity classes. Because it is usually sound but not complete, it actually provides certificates that a given program can be run within a given amount of resources. ICC is usually applied on toy languages with restricted expressivity, we show here that it can be performed on real programming languages.

Because it is usually a static, syntactical analysis of the programs, ICC is well-suited to be performed at compile time. The bounds given by ICC can then be used to fuel some optimisation or to produce certificates of good behaviour. Modern compilers do most of their work in a modular sequences of passes done on some Intermediate Representation (IR) language. The IR is a generic typed assembly-like language and thus very well suited to express ICC criteria. The modularity of the passes make it easy to add one and to re-use existing ones at will.

We focus here on the relatively simple analysis of Non-Size Increasing (NSI) programs. We've implemented a NSI analysis for the LLVM compiler. This can be seen as a proof of concept that ICC and compilers are able to interact productively.

I. INTRODUCTION

In this article, we present an experiment in bringing analysis from Implicit Computational Complexity into real life compilers.

A. Context and Motivations

ICC aims at finding syntactic criterion on programs that guarantee some semantic property (usually some complexity bound). It emerged with the Bounded Recursion of Cobham [Cob62] but was really created by the breakthrough result on Safe Recursion by Bellantoni and Cook [BC92]. Since then, many different directions have been studied in ICC. The main ideas revolve around following dataflow (the Tiering of Leivant and Marion [LM95], the Size Change Termination of Lee, Jones and Ben-Amram [LJBA01], the Non-Size Increasing programs of Hofmann [Hof99], \ldots), performing a static check on values (the Quasi-interpretations of Bonfante, Marion and Moyen [BMM11], the mw-polynomials of Kristiansen and Jones [KJ09], \ldots) or enforcing a strict type checking (variations on Girard’s Linear Logic [Gir87] such as Baillot and Terui’s DLAB [BT09]). Schöpp introduced a more restricted Bounded Linear Logic: the Stratified Bounded Affine Logic [Sch07]. Hofmann and Jost [HJ03] furnish upper bounds on the heap usage in functional programming by accepting some restrictions.

Most of these results usually concern “toy” languages such as Term Rewriting Systems [AM13], λ-calculus or the LOOP language. Even if such languages do have a strong utility in Theoretical Computer Science, they are not daily used by programmers. On the other hand, actual languages use much more constructions (e.g. objects, pattern matching, exceptions, \ldots) which make analysis complicated. Thus, even with 20 years of ICC, it is not possible today to apply its results on actual programs. We start filling the gap.

The analysis we described here, based on NSI programs, it simple enough to be expressed on a small assembly-like language. Since it only focus on memory allocation and deallocation, we can concentrate on these operations (that is, the malloc and free in a C program) and on the control flow, and forget all the complicated constructions that may be used by the programming language. Since this is a purely syntactical analysis (as all ICC), it is perfectly suited to happen at compile time.

From the other end of the gap, we’ll use the intermediate representation in a compiler. During the compilation process, the source code is first translated in an intermediate language where optimisations are performed before being translated again into the target code in assembly language. This intermediate representation has few constructions and is simple enough to perform all the optimisations steps. Especially for our practical case, it strips all the constructions of the programs but keep the control flow and the allocations that we want to study.

Moreover, compilers already contain many analysis and optimisation tools that we can reuse. Most of these tools are spread in modular passes that can be applied in various order. Typically, there is no need to rebuild the control flow of the program. It is something that is already used by many compiler optimisations and thus that already exists as a standalone pass. We just need to call this pass and use its result. This limits the amount of code we have to write in order to perform our analysis.

B. Analysis and Optimisation

Compilers are usually focused on optimisation. Indeed, the goal is to produce an efficient code in order to have a fast program. ICC mostly provides analysis without much
optimisation of the code. However, analysis and optimisation are not so far apart...

Firstly, an analysis can be used to fuel further optimisations. Typically, building the Control Flow Graph of a program is an analysis that is used for many optimisations afterwards. Here, knowing the precise amount of memory that a function or program will need can help optimising system calls: rather than using the standard library to find free memory and allocate it, it becomes possible to let the program reuse its own memory efficiently.

Secondly, providing proven bounds on the time or space usage of a program is also a security property. If the program provably uses a fixed amount of memory, then it will not try to perform an attack by heap or stack overflow, this warns us if the program could try to exhaust system resources. Restricting the syntax in order to enforce (some) security is similar to what Facebook does with the restricted FBJS. Since analysis is complex but verification is (usually) easy, one can imagine a compiler that will provide a certificate for some property on the compiled code, in a Proof Carrying Code paradigm [Nec97]. The certificate could be checked, for example, before uploading an application to an application store for mobile devices to guarantee some safety to the user, or, at the other end, before downloading the application to the device to check if it has sufficient capacity to run it.

Next, some ICC analysis are known to also embed some program transformation in them. Notably, the Quasi-Interpretations method guarantee that the programs run in polynomial time if some sort of Dynamic programming is used. Thus, a program admitting a QI can run in exponential time but the analysis says that it will run in polynomial time after some (known) transformation. Bringing such an analysis in compiler will indicate which part of the code should be transformed by which method.

Lastly, these are also first steps in experimenting ICC into compilers. Thus, we chose to focus on a simple analysis that is easy to express in the compiler’s intermediate representation rather than on a powerful analysis/optimisation which require more work to be used. Thus, this can be seen as a proof of concept: yes, ICC and compilers can work together and can make more work to be used. Thus, this can be seen as a proof of concept: yes, ICC and compilers can work together and can fuel each other fruitfully. This opens the way for future works.

II. NON SIZE INCREASING PROGRAMS

A. Safe Recursion and Non Size Increasing

In Safe Recursion, Bellantoni and Cook analysed repeated iterations as a source of exponential growth. Typically, exponentiation can be computed by iterating doubling, itself an iteration.

In order to prevent repeated iterations, they designed a syntactical criterion (hence, a static analysis) based on splitting variables into normal and safe ones, which can be interpreted as with/without energy. Next, iteration must be performed on a normal variable (which provides the “energy” to run the program) and the result must be safe (the energy has been used). Thus, when computing the exponential with the usual recurrence $2^n = 2 \times 2^{n-1}$, the result of the recursion $(2^{n-1})$ must be safe and cannot be used to control the doubling $(2\times)$.

However, repeated iterations is a powerful expressive construction to build many reasonable programs. Indeed, writing a program by respecting the normal/safe tiering of arguments is often difficult. Typically, insertion sort works by iterating the insertion of an element into a sorted list, itself an iteration. The Safe Recursion prevents writing the insertion sort (or rather requires to write it in a non natural way).

Hofmann identified that the problem does not come from the exponential or sorting function but from the doubling or insertion function. Indeed, doubling produces an output twice as large as its input while insertion produces an output basically as large as the input. Thus, it is not harmful to iterate insertion, which does not increase the size of its data (or only by a constant factor) while it is harmful to iterate doubling which drastically increases the size of data. This justifies the detection of Non Size Increasing (NSI) programs.

In order to detect NSI programs, Hofmann introduced a new datatype, the diamond ($\lozenge$), with the particularity that this datatype has no constructor. That is, there is no closed term of type $\lozenge$ and only variables can have this type. Moreover, variables of type $\lozenge$ must be used linearly in the result of functions. Thus, $\lozenge$ in the result can be seen as “price” to be paid to compute and that can only be paid by diamonds already present in the arguments.

The next step is to make the type system aware of $\lozenge$. When working with lists, it is the cons that make the size of lists, Hofmann requires a diamond for each cons. Thus, instead of having the classical type $\alpha,\alpha \text{ list} \rightarrow \alpha \text{ list}$, cons is now of type $\lozenge,\alpha,\alpha \text{ list} \rightarrow \alpha \text{ list}$.

With this new type system, it is still possible to write the insertion sort in the usual way, but exponentiation is not possible anymore.

B. NSI and imperative programs

The diamonds have a very nice and natural interpretation in imperative programs, as shown by Hofmann.

The classical representation of lists in an imperative language is to have cells containing a value and a pointer to the next cell, the list itself being a pointer to the first cell. When performing a cons, a new cell must be created. For this, new memory must be allocated (malloc). This new memory is exactly the diamond we need to perform the cons! Indeed, if cons is given as a third argument a pointer (to a place in memory which is assumed to be free), then it does not need to allocate memory and can use the one that is provided.

Hofmann shown that NSI programs can be compiled into malloc-free C programs. The diamonds are essentially pointers and a program that is NSI does not need extra diamonds, hence does not need to allocate new memory.

Having a program which is guaranteed to be NSI is not only a complexity analysis. It also gives some security properties (the program won’t overflow memory and won’t cause memory leaks) and some possibilities for optimisation. It becomes indeed possible to completely remove the malloc from the
code and let the program efficiently reuse its memory. This will prevent several system calls and calls to the standard library that can slow down the program execution.

C. A Control Flow Graph analysis

To prove a bound on space usage, we only need to know the maximum amount of diamonds required at any time. That is, we can focus on the quantitative part of the analysis and forget about the qualitative part of how and where these diamonds are reused. Consider, for example, the following insertion sort function (where \(d\) and \(d'\) are ⋄):

```plaintext
insert(d, y, []) -> cons(d, y, [])
insert(d, y, cons(d', x, xs)) ->
  if x<y
    then cons(d', x, (insert(d, y, xs)))
  else cons(d, y, cons(d', x, xs))

sort([]) -> []
sort(cons(d, x, xs)) -> insert(d, x, sort(xs))
```

It is possible to have an overview of the diamonds (i.e. of the malloc and free in an imperative version of the program) behaviour during the recurrence. The recursion gets a diamond when pattern matching is performed to read and compare; if it’s the good place it uses two diamonds (calls cons two times): one to add the new element and another to replace the previous element; otherwise, it simply replaces the old element (with its own diamond, \(d'\)). This way, we understand that the insert function will globally constructs one element, and thus require and extra diamond, \(d\), which can be provided by sort.

It’s easy to do this analysis using a Control Flow Graph (CFG). A Control Flow Graph is a graph representation of all paths that might be traversed by a program during its execution. We can see each node as a program state and each edge as an instruction.

For our analysis, we need to augment this CFG by adding a weight (the diamond usage) to each instruction. This way it becomes the Resource Control Graph [Moy09] (RCG) of Figure 1. Note that if we drop the qualitative part of the analysis, we don’t need the ⋄ type anymore as counting variables of this type is equivalent to counting instructions producing or consuming it. This, however, only provides a proved bound on space usage but cannot actually remove malloc and free from the program.

Using this RCG we can find the most expensive path according to this weight. A maximum weighted path is quickly computable with a classical algorithm such as Dijkstra’s or Bellman-Ford’s. It’s equivalent to finding the shortest paths in a weighted graph. We also need to detect positive loop in a polynomial time. Here we are in the case where we have a single entry source. The Bellman-Ford algorithm can be used here to provide the shortest path instead of the Dijkstra’s one which is not able to deal with negative edge weights and detect negative loop.

We understand that, because the analysis is only static, it’s not accurate. We only consider the worst case to ensure the NSI property, this is why we prefer to have false negatives instead of false positives. Avoiding both is undecidable...
- and a back-end for each architecture which produces a machine code.

The Intermediate Representation is pretty similar to an assembly language. It's a lower-level programming language than the input one but it's a higher-level than real assembly language.

The optimizer (opt in LLVM) provides optimizations but also analysis, both are called passes.

This optimizer is mainly composed by a Pass Manager that keeps analysis information up to date, manages memory used, enforces enabled passes with a given order and make pass developer's life simple thank to its modularity. These passes visit and change the Intermediate Representation in the middle-end.

The optimizer is one of the several tools or modules provided by LLVM. Designed for more modularity, the optimizations are built into distinct libraries and the LLVM Intermediate Representation is preserved permanently, making it easy for other-ends to use them.

In our time, two mainly used compilers exist: GCC and LLVM. For our first prototype, our choice was LLVM because: first of all, LLVM is well documented; the community is huge and very active; it uses the same Intermediate Representation throughout the compilation; it's modular; it's more and more used. For instance, more, and more efforts have been done to build Debian with LLVM\(^1\).

By comparison, GCC remains more used but performances and accessibility are equivalents. However the LLVM community’s documentation and help are more appropriate. The modularity also helps to contribute without knowing the entire working flow. The analysis are, of course, feasible in GCC, CompCert\(^2\), etc. CompCert is a certified compiler using the Coq proof assistant, it guarantees that any transformations on each node. This can be done independently of the compilation cannot alter the program’s semantics. The produced assembly will compute exactly what the source said before compilation.

The LLVM Project [Lat02] is a collection of modular and reusable compiler and tool chain technologies. LLVM is an acronym for Low-Level Virtual Machine, but the scope of the project is not limited to the creation of virtual machines. As the scope of LLVM grew, it became an umbrella project that included a variety of other compiler and low-level tool technologies as well.

LLVM is almost well designed for our work because it:
- Offers modularity, simplicity and a good research environment for compilers developers.
- Operates transparently to the developer.
- Provides a multi-stage optimization strategy.

B. LLVM IR, instruction set and Data structure

The LLVM Intermediate Representation is a Typed Assembly Language (TAL) and a Static Single Assignment (SSA) based representation which provides type safety, low-level operations, flexibility and capability to represent any high-level languages cleanly. As we said, this representation is used throughout all phases of the compilation in LLVM.

A lot of well known optimizations are already dealing with this IR: Dead Code Elimination, Loop Invariant Code Motion, Constant Propagation etc... for example: The Instruction Combination Pass is one of the simplest passes. It knows some optimizable patterns like “\(\text{add } X, 0 \rightarrow X\)”, “\(\text{xor } X, X \rightarrow 0\)” etc... and can detect and replace them.

The LLVM Intermediate Representation is source-language-independent, mainly because it uses a low-level instruction set slightly richer than assembly languages, it’s a RISC-like virtual instruction set. The instruction set consists of 31 opcodes, just enough to don’t loose type expressivity but still a low-level representation. Most of these operations are in a three-address form: that’s means that they take one or two operands and produce one result. But, unlike RISC instructions, LLVM-IR is strictly typed, then type mismatch can easily be detected. Types can be primitive or constructive (composed by several primitive types or constructive types). Each instruction has restrictions on the arguments types. Instructions can be polymorphic: for instance \(\text{add}\) can operate on different types, this widely reduces the number of opcodes. Here, we will only be interested in instructions for typed memory allocation.

The \(\text{malloc}\) instruction allocates one or more elements of a specific type on the heap, returning a typed pointer to the new memory. The \(\text{free}\) instruction releases memory allocated through \(\text{malloc}\). When the native code is generated, this instructions are converted to the appropriate native function calls, allowing also customizations. There are no implicit accesses to memory, this simplifies all memory access analysis.

LLVM has shown that an efficient low-level representation enriched with type information can support high-level analysis and transformations.

IV. RCG computation and positive loops detection

LLVM already builds the CFG of every function. LLVM provides some tools to visit and match instructions targeted in the entire graph given. This representation can give foundations in order to create a new analysis.

Each node in the CFG represents a basic block, i.e. a succession of instructions without any branching. Directed edges are used to represent jumps. A CFG starts with one entry-block and has one or several exit-blocks (or leaves). That builds the structured programming concept.

The RCG can be built by traversing the entire CFG once and counting the number of memories allocations and deallocations on each node. This can be done independently of the order of the blocks.

In order to do this, we use a LLVM tool: Basic Blocks visitor which goes through each basic block on the CFG. We can add a function to run for each basic block. Here we just compute their weight and map this to be used by another pass.

Now we can compute the maximal weight or worst case space that might be used by each function. We can use the

---

1. sy/lestre.ledru.info/blog/2014/09/11/rebuild-of-debian-using-clang-3.5
2. compcert.inria.fr/compcert-C.html
Bellman-Ford’s algorithm to find the heaviest path for our weighted graph in a polynomial time.

Basic Blocks are stocked in a list in the Function Class and not as a graph. We need to, recursively, travel through each successor of blocks, starting with the entry one. To fill up this new graph we will need to use a Depth-First Search to obtain our nodes in the correct order.

If we reconsider the analysis, it just provides an answer to the following question: “is the program NSI?” We actually don’t provide the accurate amount of space needed, but we detect if this amount is fixed. That is, we need to detect positive loops without regarding how many times they will occur. Thus we consider all positive loops as occurred a non-determined number of time. In fact we can be more precise by detecting static loops and upper bounds but it already exists passes that find invariants and unroll loops.

V. CONCLUSIONS AND FURTHER WORKS

We built a static analyzer in almost 250 lines of code mostly because it reuses the LLVM’s environment and tools. It can be split in two parts: the first builds a Resources Control Graph and the second computes functions weights and detects positives loops. This analysis has been tested on classical libraries like libc and, maybe, the “proof-carrying code” [Nec97] movement. By this way we could globalize the “proof-carrying code” [Nec97] movement, maybe by using an annotated system like the Clang Language Extensions3 or something similar for the Intermediate Representation [ASM13] or Resources Control Graphs [Moy09]. This last work is also based on weighted Control Flow Graphs or Resources Control Graphs [Moy09].

REFERENCES


3http://clang.llvm.org/docs/LanguageExtensions.html