

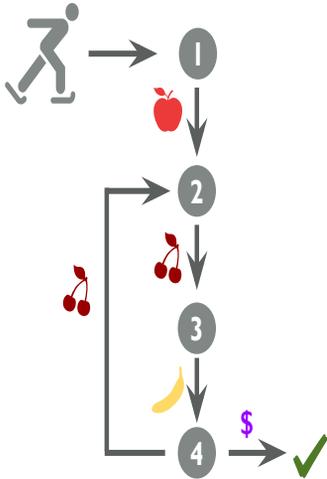
AUTOMATIC BLACKBOX EQUIVALENCE CHECKING

Sorav Bansal
IIT Delhi and CompilerAI Labs

Joint work with Manjeet Dahiya, Shubhani, Abhishek Rose
and several other past members of our research group

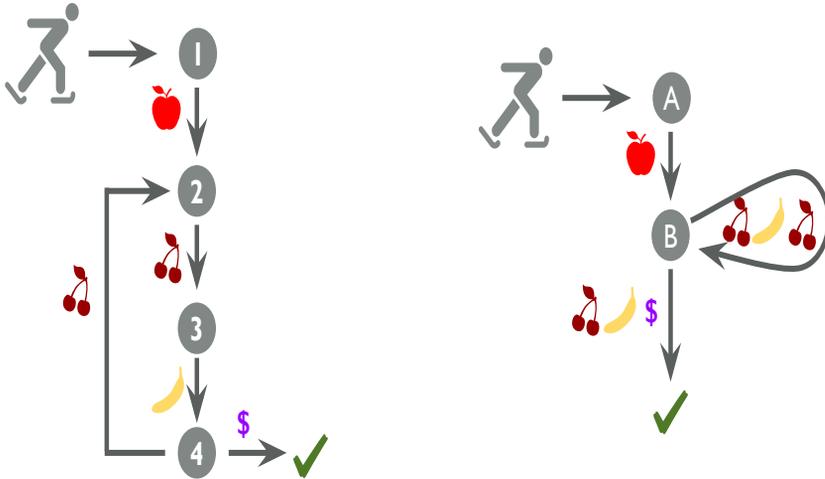
Hello everyone. This is a talk on automatic blackbox equivalence checking. This is joint work with Manjeet Dahiya, Shubhani, and Abhishek Rose, all of whom are current or past PhD students at IIT Delhi. Also there are several past Masters and Bachelors students who have contributed to the work that I will present today.

DETERMINISTIC FINITE AUTOMATON



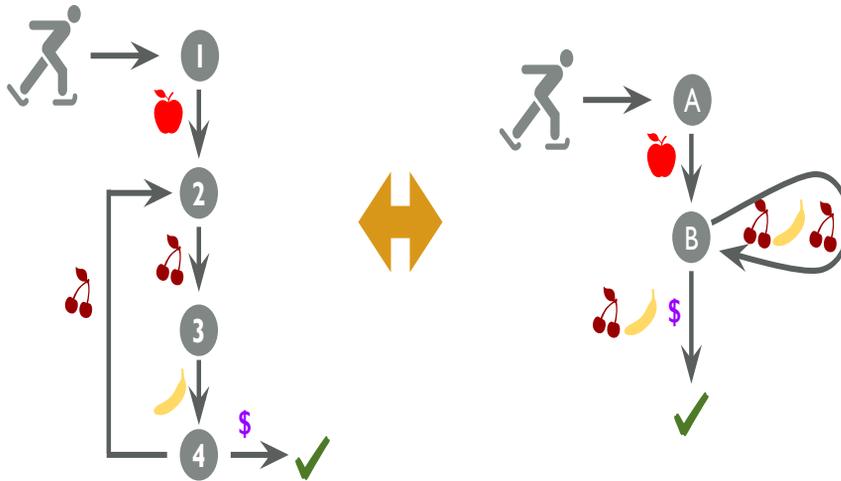
I will start with a Deterministic Finite Automaton (DFA) that I will assume you are already familiar with. Here I show a cartoon of a DFA where a person, representing a machine, starts at state 1 and starts consuming input. If he finds an apple, he moves to state 2; then if he finds a cherry, he moves to state 3, and then if he finds a banana, he moves to state 4. Thereafter, if he has reached the end of input (indicated using the dollar sign), he stops and exits. Otherwise, if he sees another cherry, he moves back to state 2. If he sees any other sequence of input fruits, the machine gets stuck which is equivalent to an error state.

TWO DFAS



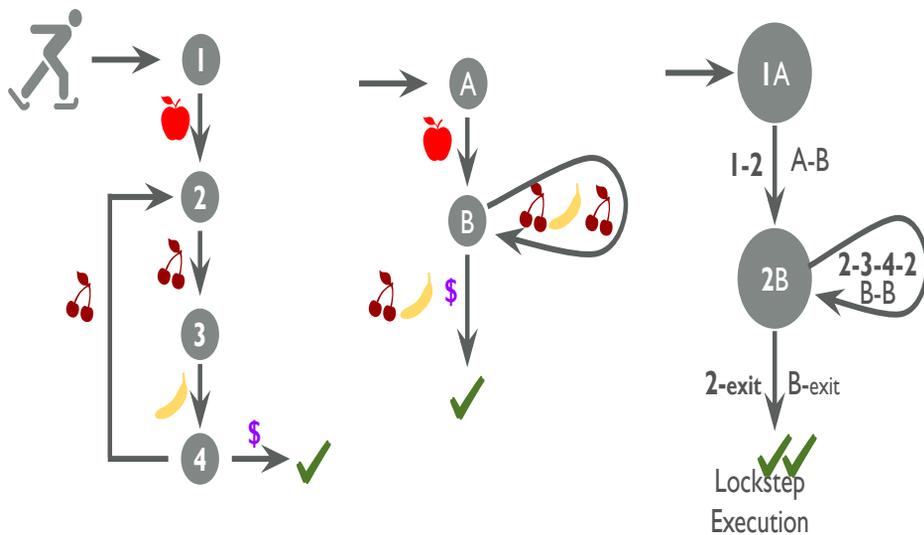
Now here is another machine where there are only two states A and B. The person simulating the machine execution starts at state A. If he eats an apple, he moves to state B, then if he sees a sequence of a cherry banana cherry, then he stays at state B; and if he sees a sequence of a cherry banana and an end-of-input (marked by dollar), then he successfully exits the machine.

EQUIVALENCE



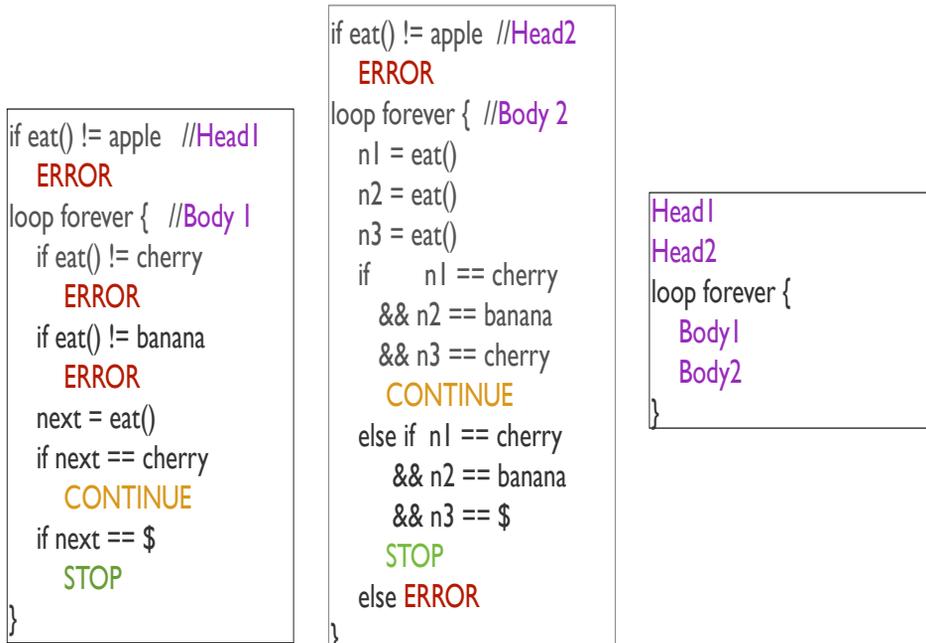
We are interested in showing that the two machines are equivalent. In this case, this means that the language of the sequence of input fruits accepted by the two machines are identical. In other words, a sequence of fruits is accepted by the first machine if and only if it is accepted by the second machine. In this simple example, this can be done by summarizing the language accepted by each DFA, through regular expressions and then computing their equivalence. However, in general, it may not be possible to summarize the machine's behaviour, and so we need easier ways to determine equivalence.

BISIMULATION AS A PRODUCT DFA



One approach to identifying such equivalences is bisimulation. Here try to find a lockstep execution of the two machines which specifies that whenever the first machine transitions through a sequence of states, the second machine transitions through a corresponding sequence of states. And vice-versa. The third automaton drawn on the right shows a DFA that demonstrates this lockstep execution. If the first machine starts at state 1, then the second machine starts at state A and vice-versa. This is encoded by the state 1A. Similarly, the first machine transitions across the edge 1-2 if and only if the second machine transitions across the edge A-B. To reach state 2B. The second machine transitions from B to itself if and only if the first machine cycles through the states 2, 3, 4, and back to 2. And finally the second machine transitions from state B to exit if and only if the first machine transitions from 2, 3, 4, to exit. This third automaton is also called a “product DFA” because it involves nodes and edges drawn from the product graph of the first two DFAs.

IMPERATIVE LANGUAGE SYNTAX



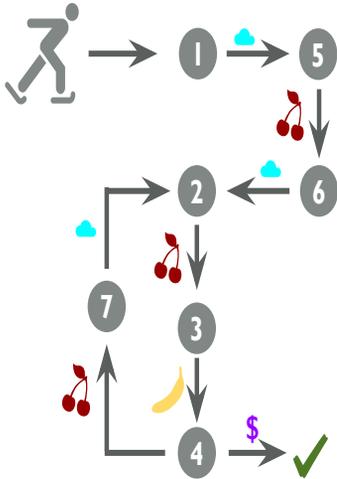
These automata can also be specified in an imperative language syntax. I use the “eat” function to simulate input consumption. For example, the first program involves eating one fruit at a time: it first expects to eat an apple, then it expects to eat a cherry, then a banana, then either it is done or it expects another cherry and so on. The second program on the other hand can potentially consume three fruits in one go: and depending on the sequence of fruits consumed, it can decide the next action. The action could be either to continue executing, stop executing, or raise an error. The corresponding product DFAs can be represented by merging the two programs as shown on the right. We divide each program into a head which is the part before the loop, and a body which is the body of the for loop. Thus, the product DFA encodes the execution of the two heads in lockstep, and the two bodies in lockstep. For example, body1 executes if and only if body2 executes (for the same input sequence). Similarly, body1 exits if and only if body2 exits.

IMPERATIVE LANGUAGE SYNTAX

<pre>if eat() != apple //Head1 ERROR loop forever { //Body 1 if eat() != cherry ERROR if eat() != banana ERROR next = eat() if next == cherry CONTINUE if next == \$ STOP }</pre>	<pre>if eat() != apple //Head2 ERROR loop forever { //Body 2 n1 = eat() n2 = eat() n3 = eat() if n1 == cherry && n2 == banana && n3 == cherry CONTINUE else if n1 == cherry && n2 == banana && n3 == \$ STOP else ERROR }</pre>	<pre>Head1 Head2 loop forever { Body1 Body2 }</pre> 
		<pre>Head1 Head2 loop forever { Body1 } loop forever { Body2 }</pre> 

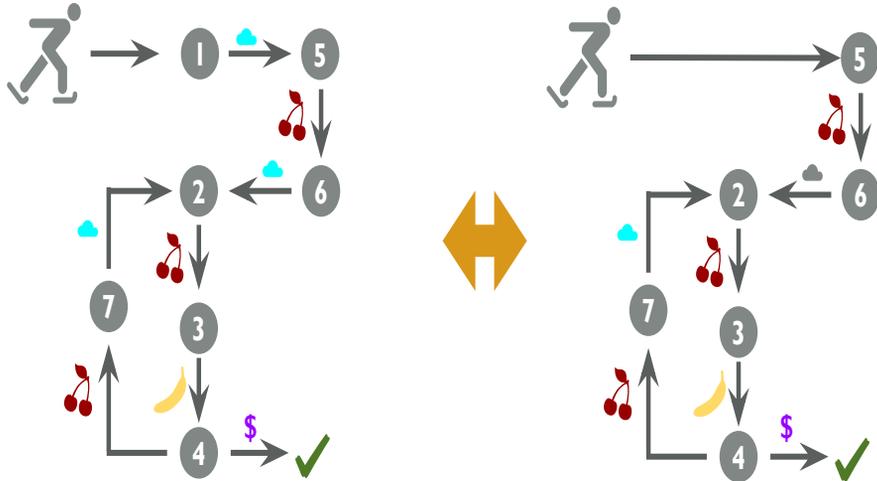
Notice that the product DFA that we used correlates one iteration of the first program's body with one iteration of the second program's body. This is much easier to do than trying to summarise each program separately and then trying to compare the summaries of the two programs. This first construction of lockstep correlations of small execution snippets is our intended bisimulation relation. We will use bisimulation relations for deciding equivalence.

INTERNAL ACTION



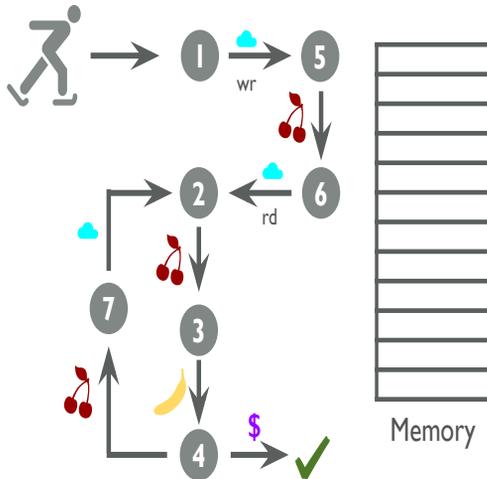
It is much easier to find such bisimulation relations for the kind of DFAs that I have shown so far, because every edge is associated with an input action. However, in general, program transitions may not consume an input or generate an output. To represent this in a labeled transition system, internal actions are used, which I represent using this cloud shaped figure. For example, the transition from state 1 to state 5 involves an internal action. In this case, even though the machine changes state, this change cannot be *observed* because it does not involve any interaction with the outside environment. On the other hand, consumption of a fruit is an observable action.

EQUIVALENCE IN THE PRESECE OF INTERNAL ACTIONS



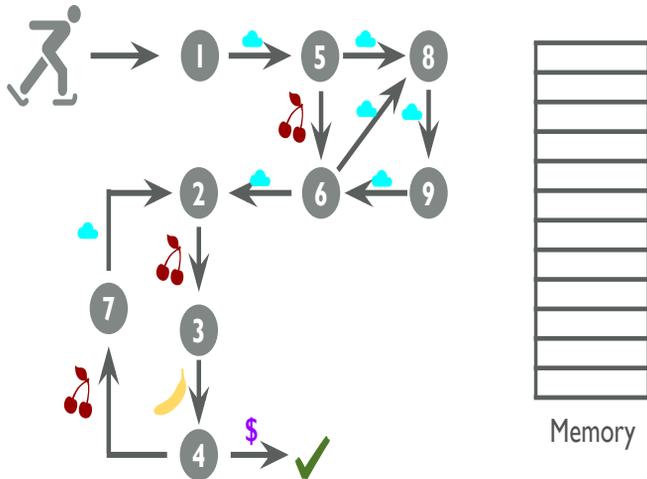
Because the internal actions are not observable, the DFA on the left is observationally equivalent to the DFA on the right, which does not have the transition from 1 to 5. In our work, we are interested in equivalence observable events, or events that interact with the external environment

MEMORY



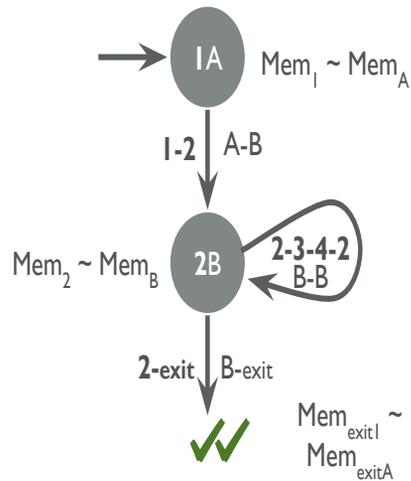
Moreover, there may be a larger state, also called memory, associated with the machine. For example, the Turing machine involves an infinitely long tape. In general, this memory state could be finite or infinite. Further the transitions of a machine may involve read and write to this memory. In our example, a transition from 1 to 5 involves a write to memory, while a transition from 6 to 2 involves a read from memory (indicated using wr and rd annotations). Reads and writes to memory are also non-observable events.

MEMORY WITH LOOPS



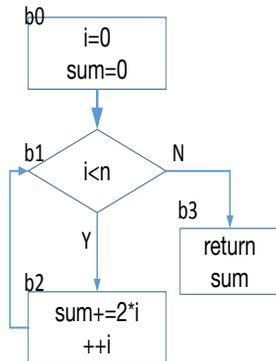
Once you have memory, the state transitions may depend on the the current memory state. For example, the state machine may now have loops. In our example, the machine at state 6 may deterministically decide to either transition to state 8 or to state 2, depending on the current state of the memory.

BISIMULATION WITH MEMORY RELATIONS



In this setting of memory and loops, bisimulation relations now also need to encode relations between memory states of the two programs being compared for equivalence. For example, in state 1A, we may want to constrain the relation between the corresponding memory states Mem_1 and Mem_A . In fact, it is likely that the lockstep execution guarantee holds only under the constraints imposed by these memory relations (e.g., Mem_1 should be equal to Mem_2). Similarly, we may want to constrain the memory states at the state 2B, and so on.

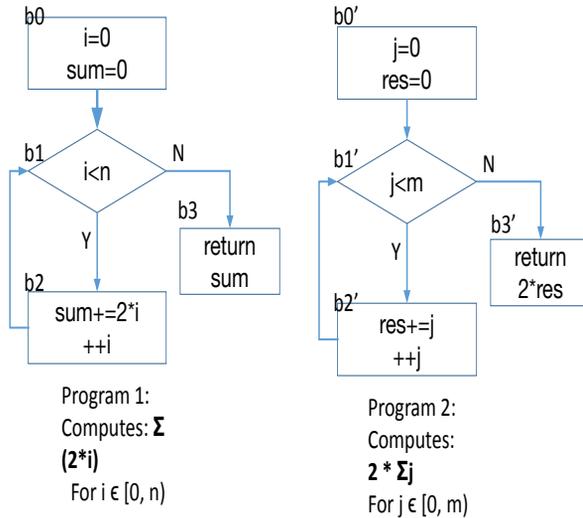
EXAMPLE 1



Program 1:
Computes: \sum
 (2^i)
For $i \in [0, n)$

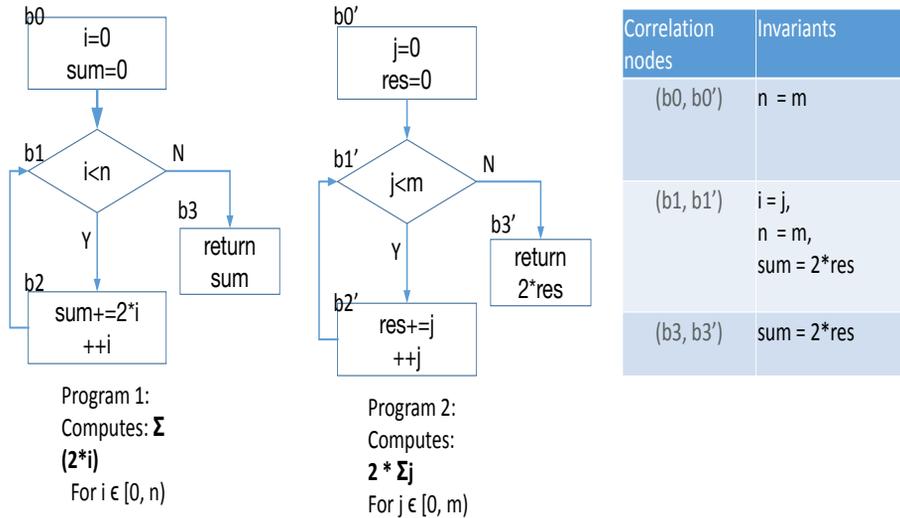
To see this with a concrete example, consider this program shown as a flowchart. This program initialises i and sum to 0. Then it executes a loop till i is less than N , incrementing i at each iteration of the loop. At each iteration, we add 2^i to sum . At the end of the loop, sum is returned. If I was to summarize, this program computes the sum of 2^i for i ranging from 0 to $n-1$. Recall that we are not interested in identifying summaries of programs, because every program may not have an easily expressible summary. But here, just for clarity in discussion, I also show the summary of the program.

EXAMPLE I



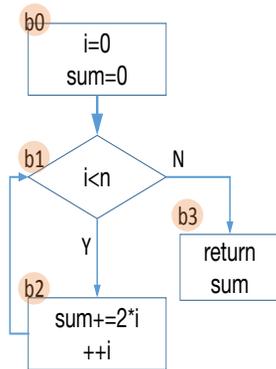
The second program involves an almost identical program, except that this time we use J instead of I , res instead of sum , and M instead of N . Also, we don't multiply j by 2 before adding it to res . Instead we multiply res by 2 at the very end before returning. Thus this program computes $2 * (\text{Sum over } j)$.

EXAMPLE I

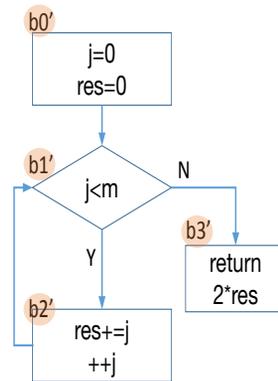


The two programs can be shown to be equivalent using a product program that executes both programs in lockstep. On the right I show a bisimulation relation. The first column of this bisimulation relation encodes the correlated PCs, e.g., b_0 is correlated with b_0' ; b_1 is correlated with b_1' , and b_3 is correlated with b_3' . Not every instruction needs to be correlated, e.g., we don't correlate b_2 and b_2' . The second column encodes the relations on memory that are required to hold when the two programs are at the respective PCs. Because the memory of these programs can be specified through the six variables, we have relations relating these variables in the second column.

EXAMPLE I



Program 1:
Computes: Σ
 $(2*i)$
For $i \in [0, n)$

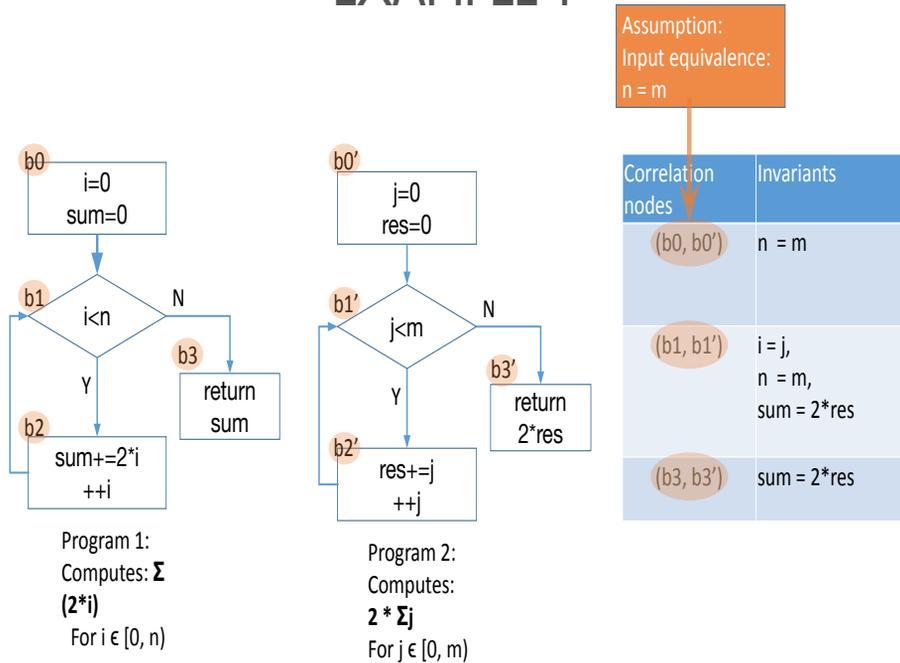


Program 2:
Computes:
 $2 * \Sigma j$
For $j \in [0, m)$

Correlation nodes	Invariants
$(b0, b0')$	$n = m$
$(b1, b1')$	$i = j,$ $n = m,$ $sum = 2 * res$
$(b3, b3')$	$sum = 2 * res$

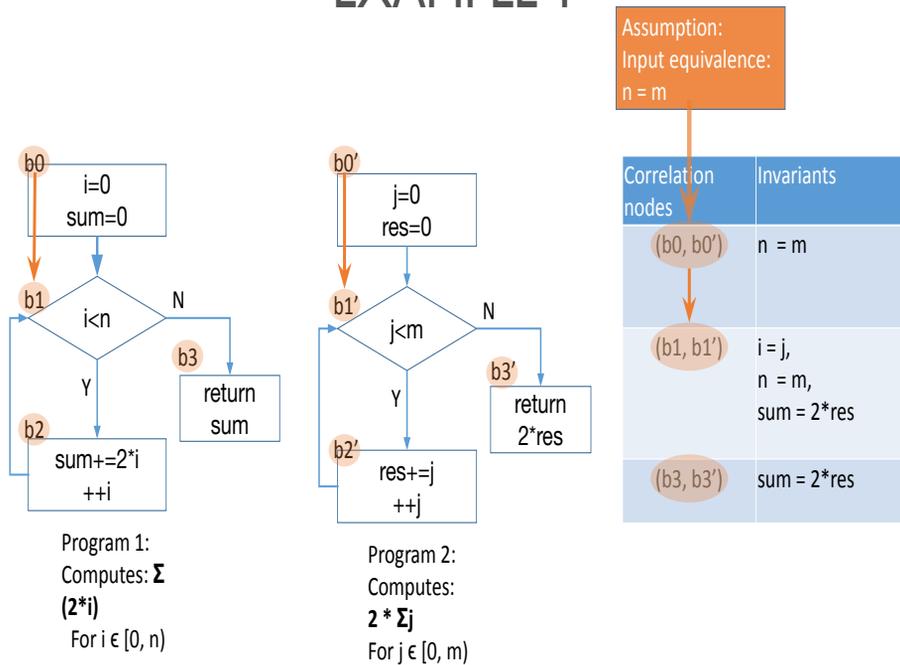
In this example, Program 1 is at $b0$ if and only if Program 2 is at $b0'$. Similarly, Program 1 is at $b1$ if and only if Program 2 is at $b1'$. And finally, Program 1 is at $b3$ if and only if Program 2 is at $b3'$. In general, it is possible for a bisimulation relation to have one-to-many mappings between PCs of the two programs. We will see some examples to this effect later.

EXAMPLE I



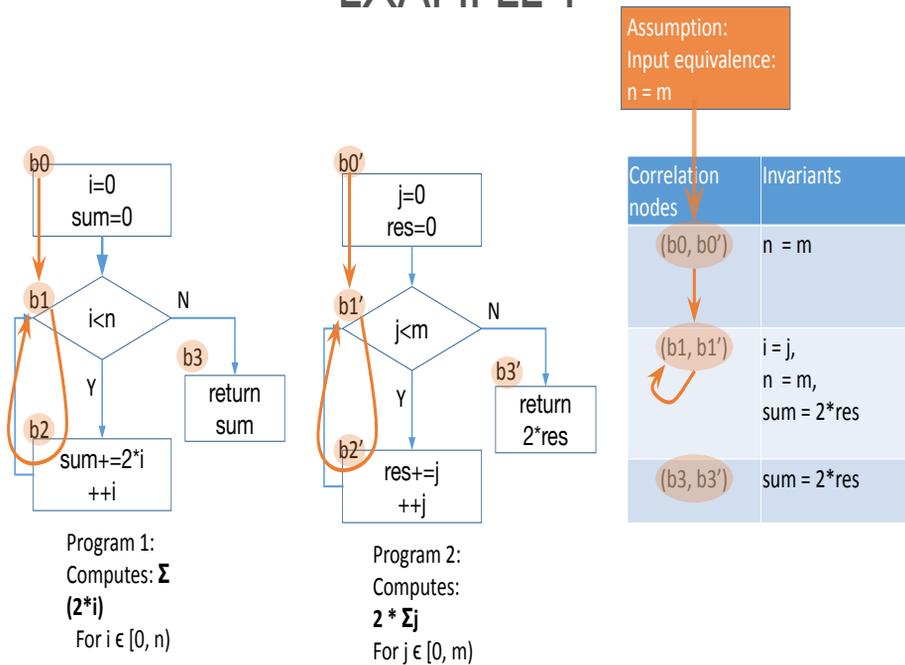
The memory relations can also be called the program invariants for the product program. The invariant at the start node specifies that the input variables n and m must be equal. This comes from the equivalence problem specification that states that the outputs should be equal for *equal inputs*.

EXAMPLE I



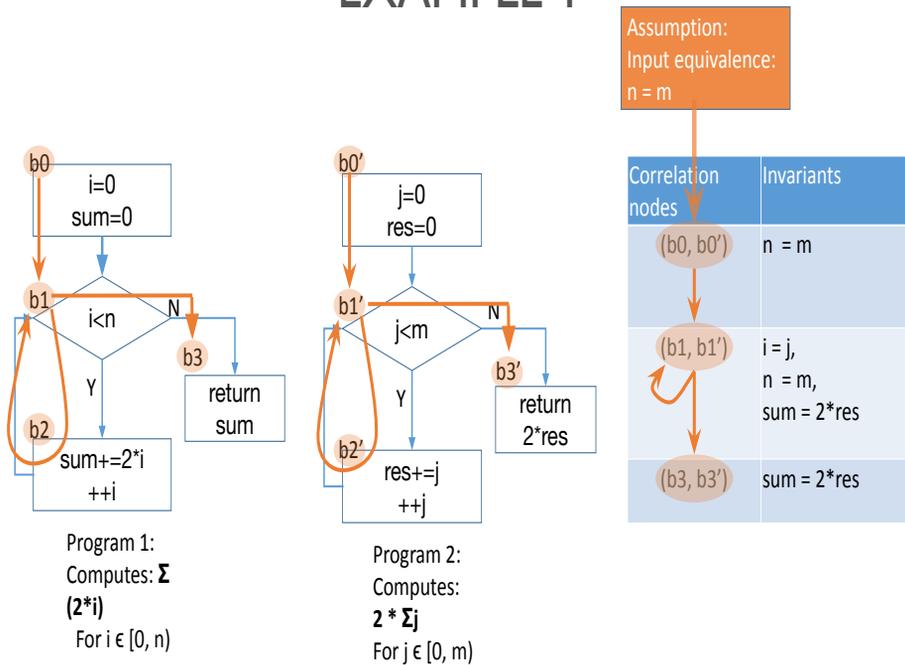
The edge (b_0, b_0') to (b_1, b_1') encodes the fact that Program1 transitions from b_0 to b_1 if and only if Program2 transitions from b_0' to b_1' . Moreover, we have set of invariants that we expect at (b_1, b_1') . I have not yet discussed how we obtain these invariants. Let's assume these invariants have been given to us magically. Based on these invariants, we are interested in proving that the programs have identical observable behaviour for identical inputs. In this case, we have three invariants at (b_1, b_1') : $i=j$, $n=m$, and $sum=2*res$. The proof involves showing that if we start at (b_0, b_0') such that $m=n$ at that point, and we transition to (b_1, b_1') then these three invariants would hold. This is easy to see because when the product program transitions from (b_0, b_0') to (b_1, b_1') , the two programs would transition from b_0 to b_1 and b_0' to b_1' respectively. In this case, the variables I and SUM would be set to 0. Similarly J and RES would be set to 0 too. If we look at the first invariant, $i=j$, it thus evaluates to $0=0$ which is trivially true. the second invariant $n=m$ is true because it was true at (b_0, b_0') and neither N nor M have changed. Finally, $sum=2*res$ evaluates to $0=2*0$, on this transition, which is also true. This part of the inductive bisimulation proof is similar to the "base case" of an induction-based proof..

EXAMPLE I



The next part of the proof involves an induction hypothesis and an induction step. The inductive hypothesis here is that the invariants already hold at (b_1, b_1') and the induction step involves showing that after one step transition from (b_1, b_1') to itself, the invariants continue to hold. Let's look at the first invariant $I=J$: assuming $I=J$, and one iteration of the loop transition (b_1, b_1') to itself, we get $I+1$ for the new value of I , and $J+1$ for the new value of J . Given $I=J$, it is easy to see that $I+1$ would also be equal to $J+1$. Hence, we have completed the induction step for this first invariant. For the second invariant $M=N$, the induction step is trivial because neither M nor N are modified across the loop edge. Finally, $SUM=2*RES$ gets transformed to $SUM+2*I=RES+J$ after one iteration of the loop in both programs. Given $I=J$ and $SUM=2*RES$ (from the inductive hypothesis), it is easy to see that $SUM+2*I=2*(RES+J)$. Thus we have completed the induction step

EXAMPLE I



Finally, the bisimulation proof encodes that Program1 exits to b_3 if and only if Program2 exits to b_3' . Moreover when this happens, then $SUM=2*RES$ holds at (b_3, b_3') . This completes our equivalence proof because the invariants at the exit states (b_3, b_3') ensure the equivalence of the return values SUM and $2*RES$

STEPPING BACK...

An Equivalence Checker is a proof finder



This talk

An Inequivalence Checker is a bug finder

→

1. Try to Find an Equivalence Proof

←

2. Try to Find a Distinguishing Input

3. Neither found, give up :-)

We have just seen how an equivalence proof between two programs can be determined. We are interested in identifying such proofs automatically. Thus an equivalence checker is a proof finder and that's what I will focus on in this talk.

In contrast, an inequivalence checker would be interested in identifying a distinguishing input that proves that the programs are inequivalent. Both equivalence checking and inequivalence checking are undecidable problems. A typical tool would first try to find an equivalence proof. If found, we are done. Else, it would try and find a distinguishing input or an inequivalence proof. If found, we are done. Else, we have neither found equivalence nor inequivalence, and we simply give up.

EQUIVALENCE CHECKING LITERATURE

Theoretical basis	{	Simulation Relation, cut points	A. Turing. Checking a large routine. In <i>The early British computer conferences</i> , pages 70-72. MIT Press, Cambridge, MA, USA, 1989 (reproduction) Milner, R. "Program Simulation: An Extended Formal Notion", Memo 17, Computers and Logic Research Group, University College of Swansea, U.K. (1961). Milner, R. "A Formal Notion of Simulation Between Programs", Memo 14, Computers and Logic Research Group, University College of Swansea, U.K. (1970). Milner, R. "An algebraic definition of simulation between programs", ICAI 71 Proceedings of the 2nd international joint conference on Artificial Intelligence (1971) Manna, Z., "The Correctness of Programs", J. of Computer and Systems Sciences, Vol. 3, No. 2, 119-127 (1969). ...
		Loop free code	D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal verification of DSP software. In DAC, pages 130-135, 2000. X. Feng and A. J. Hu. Automatic formal verification for scheduled VLIW code. In LCTES-SCOPES, pages 85-92, 2002. X. Feng and A. J. Hu. Cutpoints for formal equivalence verification of embedded software. In EUSOFT, pages 307-315, 2005. T. Matsuzono, H. Saito, and M. Fujita. Equivalence checking of C programs by locally performing symbolic simulation on dependence graphs. In ISOED, pages 370-375, 2006. T. Aross, C. Elter, L. Fix, S. Mador-Haim, M. Misha'el, J. Shalev, E. Singerman, A. Tiemeyer, M. Y. Vardi, and D. Zuck. Formal verification of backward compatibility of microcode. In CAV, pages 185-198, 2005. Lopes, N. R., Menendez, D., Nagarakatto, S., Rehrh. J.: Provably correct peephole optimizations with alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 22-32. PLDI 2015, ACM (2015) ...
	{	Partial equivalence (with bounded unrolling)	D. Jackson and D. A. Ladd. SemanticDiff: A tool for summarizing the effects of modifications. In KSM, pages 243-252, 1994. Lahiri, S., Hawblitzel, C., Kawaguchi, M., Hebel, H., SymDiff: A language-agnostic semantic diff tool for imperative programs. In CAV '12, Springer (2012) Lahiri, S., Sinha, R., Hawblitzel, C.: Automatic root-raising for program equivalence failures in binaries. In Computer Aided Verification (CAV'15), Springer (2015) S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu. Differential symbolic execution. In SIGSOFT FSE, pages 226-237, 2008 D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In CAV, pages 669-685, 2011. Lopes, N. R., Monteiro, L.: Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. Int. J. Softw. Technol. Transf. 18(4), 359-374 (Aug 2016) ...
		Regression verification	Hawblitzel, C., Lahiri, S. K., Pawar, K., Hashmi, H., Gokbulut, S., Fernando, L., Detlefs, D., Wadsworth, S.: Will you still compile me tomorrow? static cross-version compiler validation. In: ESEC/FSE 2013, ACM (2013) Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Verified Software: Theories, Tools, Experiments, vol. 4171, pp. 496-501. Springer Berlin Heidelberg (2008) Feilberg, D., Grebing, S., Klebanov, V., Ruzminer, P., Ubrich, M.: Automating regression verification. In: ASE '14, ACM (2014) ...
	Application specific/limitations	{	Affine programs
Data Driven (test cases are given)			Churchill, B., Sharma, R., Bashien, J., Alken, A.: Sound loop superoptimization for google native client. In: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 319-326. ASPLOS '17, ACM (2017). Sharma, R., Schifka, E., Churchill, B., Alken, A.: Data-driven equivalence checking. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications, pp. 391-406. OOPSLA '13, ACM (2013) ...
{		Translation validation (Pass based, knowledge of transformations)	V. Menon, V. Pingali and N. Mateev. Fractal symbolic analysis. ACM Trans. Program. Lang. Syst., 25(5):776-813, 2003. Kuntz, S., Tietz, Z., Lerner, S.: Proving optimizations correct using parameterized program equivalence. In: PLDI '09, ACM (2009) Tate, R., Stepp, M., Tietz, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: POPL '09 Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for llvm. In: Proceedings of the 23rd International Conference on Computer Aided Verification, pp. 737-742. CAV '11, Springer-Verlag (2011) B. Goldberg, L. D. Zuck, and C. W. Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. Electr. Notes Theor. Comput. Sci., 132(1):53-71, 2005. G. C. Necula. Translation validation for an optimizing compiler. In PLDI, pages 63-94, 2000. A. Pruehl, M. Siegel, and E. Singerman. Translation validation. In TACAS, pages 151-166, 1998. Kanade, A., Sanyal, A., Khetler, U.P.: Validation of gcc optimizers through trace generation. Softw. Pract. Exper. 39(6), 611-639 (Apr 2009) Pruehl, A., Siegel, M., Singerman, E.: Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems, pp. 151-166. TACAS '98, Springer-Verlag (1998) Tristan, J.B., Gouveau, P., Morrisett, G.: Evaluating value-graph translation validation for llvm. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 395-395. PLDI '11, ACM (2011) Zaks, A., Pruehl, A.: Cover: Compiler validation by program analysis of the crossproduct. In: Proceedings of the 15th International Symposium on Formal Methods, pp. 35-51. FM '08, Springer-Verlag (2008) ...

There is extensive literature on equivalence checking, which is unsurprising given the fundamental nature of the problem. Literature ranges from theoretical basis for this problem related to simulation relations for example to application specific treatments in the context of translation validation, regression verification, etc.

STILL MISSING...

An **Automatic** Equivalence Checker

that works across a **long and unknown** sequence of transformations

for **practically useful** programs written in **commonly-used syntaxes**

in a **scalable** way

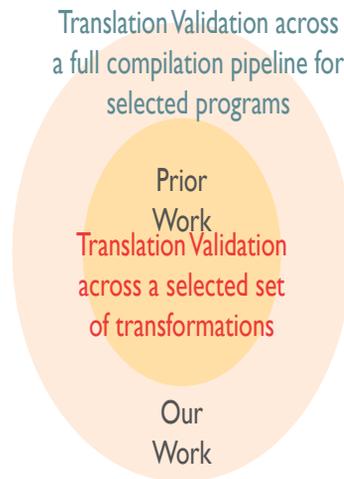
I will contend that there is still missing an automatic equivalence checker that works across a long and unknown sequence of transformations for practically useful programs written in commonly-used syntaxes in a scalable way. When I say “long and unknown”, I am interested in supporting at least the kinds of transformations supported by modern compilers. When I say “practically useful” and “commonly-used syntaxes”, I am referring to programs such as operating systems, web servers, analytic engines, embedded applications, etc written in PLs like C, Java, etc. When I say “automatic”, I mean that a machine-readable equivalence proof should be generated without manual assistance.

EQUIVALENCE CHECKING RESEARCH ROADMAP



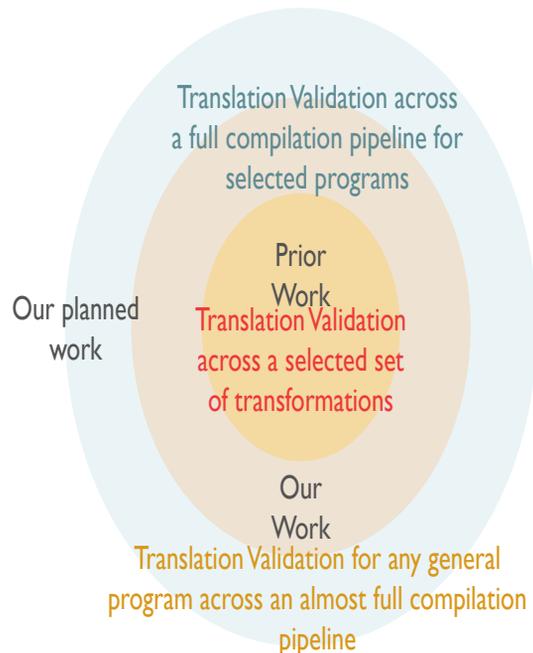
I want to first talk about the context of our work by discussing the current equivalence checking research and the future roadmap. Prior work has largely focussed on translation validation across a selected set of transformations.

EQUIVALENCE CHECKING RESEARCH ROADMAP



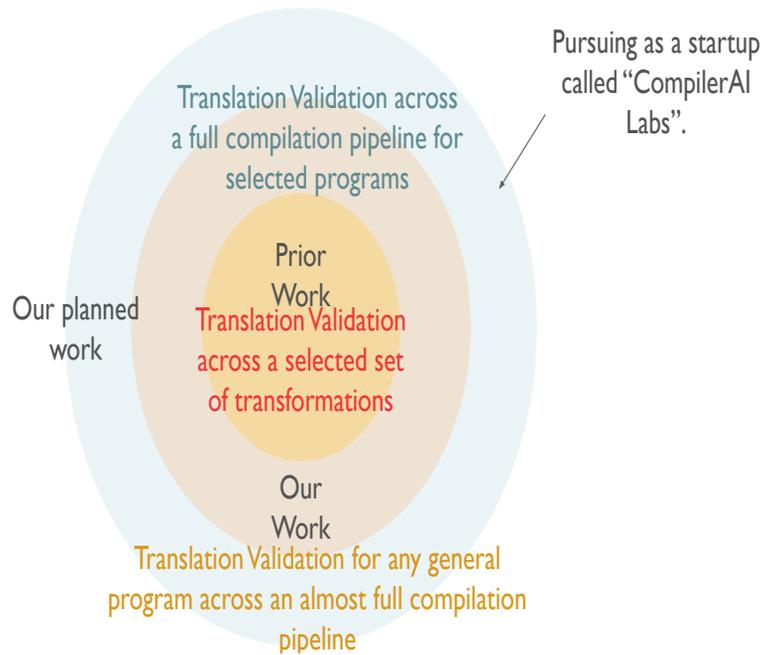
Our work, and also the work of some other research groups, has shown translation validation across a full compilation pipeline for a selected set of programs. The full compilation pipeline includes lowering to assembly and several high-level and low-level transformations. However, we are still not able to support all possible programs yet.

EQUIVALENCE CHECKING RESEARCH ROADMAP



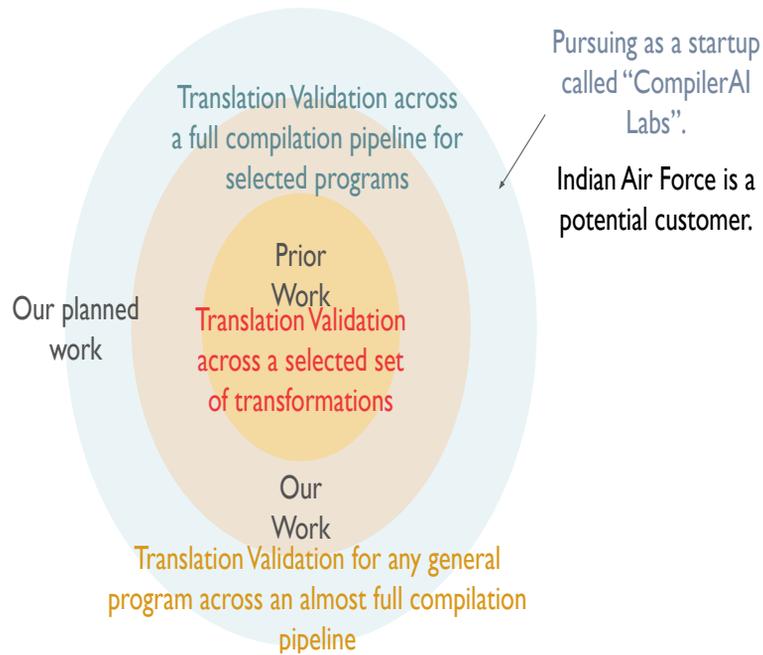
Through our planned future work over the next 1-2 years, we expect to be able to validate translations for any general program across an almost full compilation pipeline. We may not be able to support some very aggressive transformations, but we should be able to support a reasonably high degree of optimisation. Importantly, we should be able to validate compilations of any general program.

EQUIVALENCE CHECKING RESEARCH ROADMAP



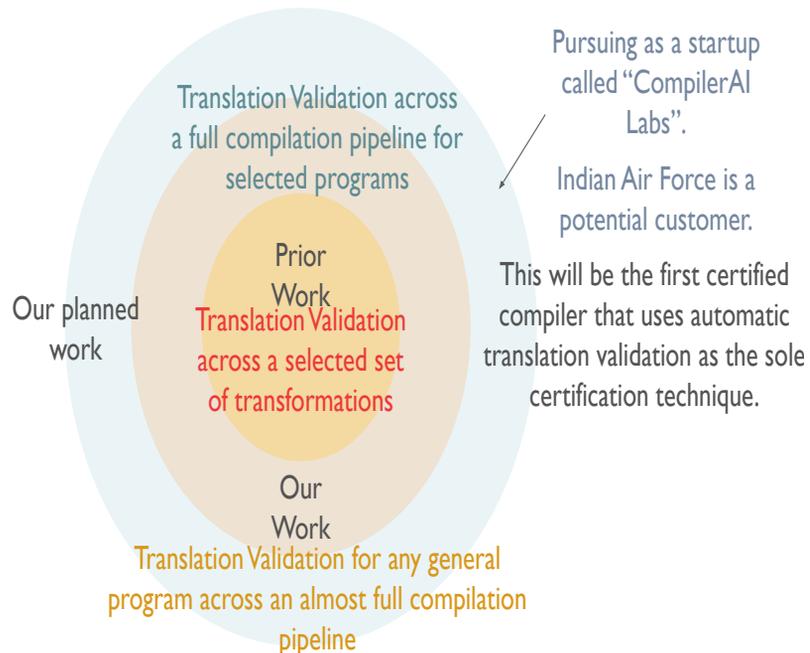
As an aside, we are pursuing this effort as a start called CompilerAI Labs

EQUIVALENCE CHECKING RESEARCH ROADMAP



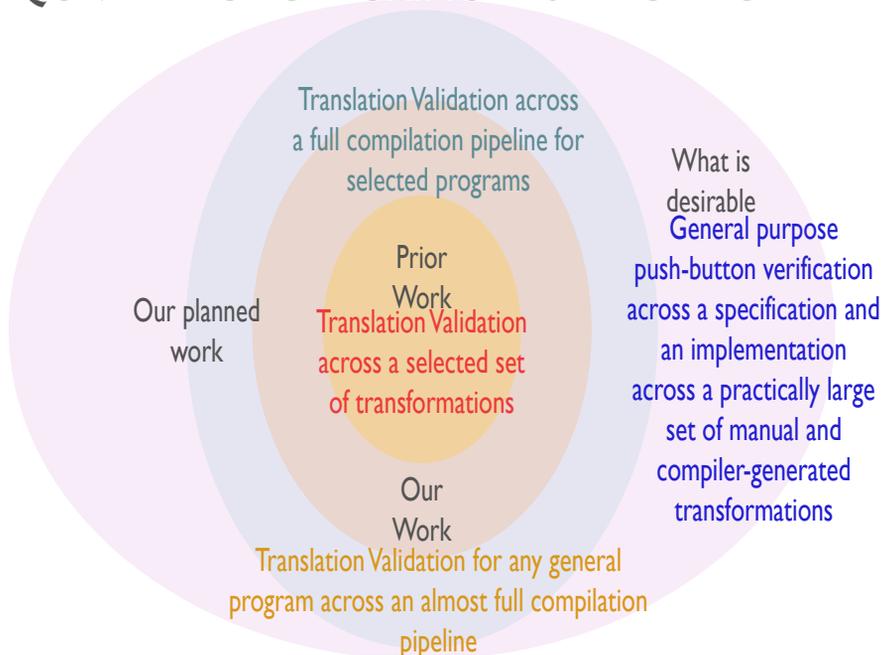
Indian Air Force is a potential customer for this capability

EQUIVALENCE CHECKING RESEARCH ROADMAP



If we are successful (which I expect us to be), this will be the first certified compiler that uses automatic translation validation as the sole certification technique

EQUIVALENCE CHECKING RESEARCH ROADMAP



But what is truly desirable is a more general push-button verification across a high-level specification and an implementation. For example, it would be really nice if an OS was specified using a high level language like OCaml and an efficient C implementation could be compared to its Ocaml specification automatically

EQCHECK SETTINGS

- Executable Binary Code vs. Executable Binary Code
 - C like PL vs. Executable Binary Code
 - C like PL vs. C like PL
-
- LLVM IR vs. LLVM IR
 - OCaml like PL vs. C like PL

APLAS17, HVC17, SAT18,
PLDI20, OOPSLA20

Ongoing

One way to classify equivalence checking efforts is the settings in which these tools operate. Prior work has involved equivalence checking between two executable binary programs; equivalence between a C like Programming language and executable binary code; between two different programs written in C-like programming language; between two different programs written in the LLVM IR syntax, and finally between an OCaml-like functional programming language and a C-like programming language. Some of our papers, as shown in red, have looked at the first three settings. More recently, we have been also looking at the other two settings. Interestingly, different settings provide different challenges and opportunities that stem from the semantics associated with each level of syntax.

EQCHECK APPLICATIONS

- Executable Binary Code vs. Executable Binary Code

- WYSIWYX
- Binary Code Analysis for Security
- Superoptimization

Before I dive into the algorithms, let me quickly go over the applications of equivalence checking in each of these syntaxes. For executable-to-executable equivalence checking, we have applications related to security and optimisation, e.g., WYSIWYX refers to what you see is what you execute. Superoptimization involves identifying faster binary rewrites automatically.

EQCHECK APPLICATIONS

- Executable Binary Code vs. Executable Binary Code

- C like PL vs. Executable Binary Code

- Translation Validation and Certified Compilation
- Superoptimization

Checking equivalence across a programming language like C and executable binary code has direct applications to translation validation and certified compilation, to rule out compiler bugs. Also, this capability can also be used for identifying higher-level optimizations through supeorptimization.

EQCHECK APPLICATIONS

- Executable Binary Code vs. Executable Binary Code
- C like PL vs. Executable Binary Code
- C like PL vs. C like PL
- Regression Verification
- Verifying Library Implementations against each other

Checking two different source programs, in the same syntax, against each other has applications in regression verification and verification of library implementations

EQCHECK APPLICATIONS

Translation Validation in the presence of non-deterministic values like Undefined and Poison

- LLVM IR vs. LLVM IR
- OCaml like PL vs. C like PL

LLVM presents its own non-intuitive challenges related to non-deterministic values such as Undefined and Poison, and so translation validation for such syntax becomes even more attractive and useful.

EQCHECK APPLICATIONS

- Push-button Verification
- Implementation Synthesis

- OCaml like PL vs. C like PL

Finally, if you can derive automatic proofs of equivalence between a higher level functional language and an imperative language like C, you would make progress towards important problems in push-button verification and implementation synthesis.

RESEARCH CHALLENGES

- Modeling Undefined Behaviour APLAS17, HVC17
- Identifying Correlations between Program Transitions OOPSLA20
- Efficient Encoding and Discharge of Proof Obligations SAT18
- LLVM UB / OCaml vs. C Ongoing

I will talk about three different challenges that we tackled and were of significance in our research on equivalence checking. The first is on the need to model the language-level undefined behaviour semantics. The second is about identifying correlations between program transitions automatically. The third is about efficient encoding and discharge of proof obligations using SMT Solvers. And finally I will briefly introduce some of the challenges related to LLVM UB and OCaml vs. C.

RESEARCH CHALLENGES

- Modeling Undefined Behaviour **APLAS17, HVC17**
- Identifying Correlations between Program Transitions **OOPSLA20**
- Efficient Encoding and Discharge of Proof Obligations **SAT18**
- LLVM UB / OCaml vs. C **Ongoing**

I will start with modeling undefined behaviour

UNDEFINED BEHAVIOUR (UB)

- A Part of the High-Level Language Specification
- More specifically, a part of the erroneous conditions specification
- Erroneous conditions without specification are called UB
- Some languages like C are UB-heavy

Undefined Behaviour semantics are a part of the high-level language specification related to error conditions. Error conditions without specification are called UB. Some languages like C are UB heavy because they are performance sensitive. For example, a language like Java would say that an error should explicitly raise an exception or print an error message. In contrast, a language like C would leave the behaviour unspecified on an error; in other words, the machine is allowed to do anything if an erroneous program is executed. These “can do anything” semantics allow better compiler optimization for performance, but make it harder to reason about the correctness of a program.

UB IN C : EXAMPLE I

- $a+b$ is UB on overflow (for int a, b) : Signed Integer Overflow

One example of Undefined Behaviour, or UB, in C is Signed Integer Overflow. Consider two integers in C, A and B. What should happen if $A+B$ is evaluated and it overflows. The C standard deems it to be UB.

Implications of UB on Compilers

`int c = INT_MAX + 1;`  `sys("rm -rf /")`

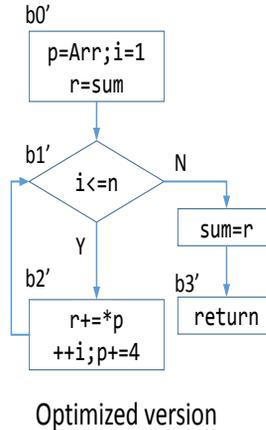
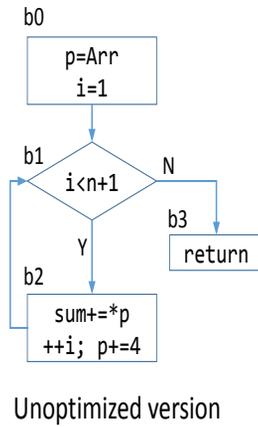
Academic example but this is a
legal transformation

To better understand the implications of such semantics, consider this snippet of code that computes `INT_MAX+1`. Clearly, this code will always result in UB, because `INT_MAX+1` is guaranteed to overflow. This means that the compiler+runtime are free to do anything if this program is executed. For example, they may execute this command to erase all files in the filesystem. Of course this is an academic example because no compiler/runtime would actually do this. But the point is that the UB semantics allow the machine to do anything, and so this would be a perfectly legal transformation.

EXAMPLE 2

```
// computes the sum of  
// first n integers of Arr
```

```
int Arr[256];  
int sum = 0;  
void compute_sum(int n)  
{  
    int* p = Arr;  
    for(int i = 1; i < n+1; ++i) {  
        sum = sum + *p;  
        ++p;  
    }  
}
```



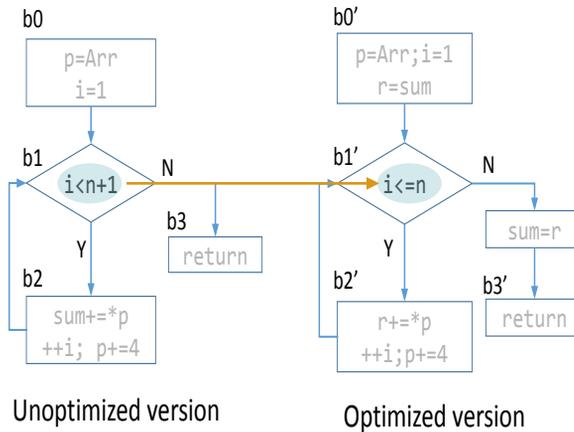
To see this in a more practical setting, consider this example program, “compute_sum”. A pointer P is initialised to a global array A; and the loop iterates from 1 to N+1, each time dereferencing and incrementing p. SUM holds the sum of the values dereferenced through P.

The unoptimised version of the program is a CFG representation of the original program. The optimised version of the program involves some transformations. For example, it changes $i < n+1$ to $i \leq n$. Also it register allocates sum to r.

EXAMPLE 2

```
// computes the sum of  
// first n integers of Arr
```

```
int Arr[256];  
int sum = 0;  
void compute_sum(int n)  
{  
    int* p = Arr;  
    for(int i = 1; i < n+1; ++i) {  
        sum = sum + *p;  
        ++p;  
    }  
}
```

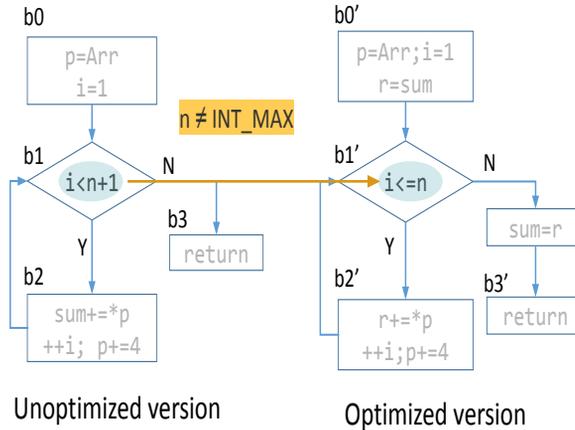


The first transformation involves converting $i < N+1$ to $i \leq N$. Notice that this transformation is only legal if we assume that $N+1$ does not overflow. For example, if N was `INT_MAX`, then the unoptimised program would check $i < 0$ whereas the optimised program would check $i \leq \text{INT_MAX}$, and the two would not be equivalent. Fortunately, thanks to UB semantics, we can assume that $N+1$ cannot overflow and so such a transformation becomes feasible because for all values of N other than `INT_MAX`, this transformation preserves the program behaviour.

EXAMPLE 2

```
// computes the sum of  
// first n integers of Arr
```

```
int Arr[256];  
int sum = 0;  
void compute_sum(int n)  
{  
    int* p = Arr;  
    for(int i = 1; i < n+1; ++i) {  
        sum = sum + *p;  
        ++p;  
    }  
}
```



In other words, the UB semantics guarantee that n cannot be equal to INT_MAX and that allows the compiler to perform this transformation legally.

UB IN C : EXAMPLE 2

- `Arr[i]` is UB if `i` belongs outside the bounds of `Arr`

Another example of UB in C specifies that if you index into an array `Arr` through an index `I`, then `I` must belong to the bounds of the array `Arr`. If it indexes outside `Arr`'s bounds, then this would trigger UB.

IMPLICATIONS OF UB ON COMPILERS

```
while (...) {  
  A[i] = ...  
  B[j] = ...  
  j++  
}
```



```
A[i] = ...  
while (...) {  
  B[j] = ...  
  j++  
}
```

Aliasing Assumptions can be made even when we don't know the bounds on i and j

To see this with an example, consider this loop where we are making two different array write accesses $A[i]$ and $B[j]$, and if the computation involving $A[i]$ is loop invariant, then that computation can be hoisted up. But such hoisting is only legal if we are assured that $A[i]$ cannot alias with $B[j]$. It turns out that if A and B are global variables, then even if we don't know the bounds on i and j , we can make such no-alias assumptions, because accessing an array outside their bounds would be UB and so we don't need to bother about that case. If both indices i and j are within bounds of A and B respectively for a non-UB program, then we can be sure that $A[i]$ cannot alias with $B[j]$ and thus we can make this transformation legally.

EXAMPLE 2

```
// computes the sum of
// first n integers of Arr

int Arr[256];
int sum = 0;
void compute_sum(int n)
{
    int* p = Arr;
    for(int i = 1; i < n+1; ++i) {
        sum = sum + *p;
        ++p;
    }
}
```

Based-on Analysis

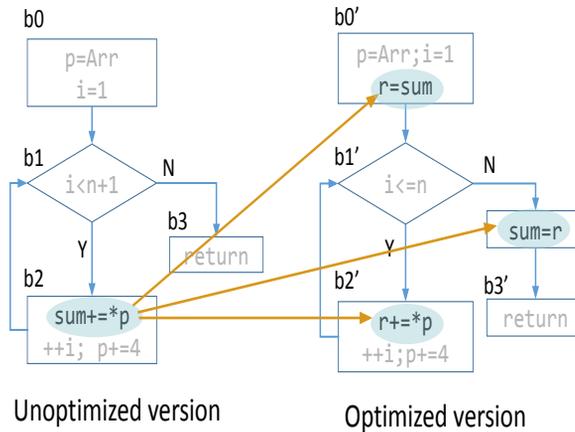
- Inside the loop, p is based on Arr
- If a pointer is based on an object X, then it must point within X
- In this example, we can infer that p cannot alias with 'sum'

In our `compute_sum` example, the access into the array `Arr` is not through a direct index but through `P`. Yet we can reason about the fact that `P` must point within `Arr`. This is due to the based-on semantics in C. Because `P` is initialized to `Arr` and is only incremented thereafter, `P` must be always based on `Arr`. Consequently, the compiler that the pointer `P` is based on the variable `A`. After that, it can use the C semantics to assume that because `P` is based on `A`, `P` must point within `A`; and thus `P` cannot alias with `SUM`.

EXAMPLE 2

```
// computes the sum of  
// first n integers of Arr
```

```
int Arr[256];  
int sum = 0;  
void compute_sum(int n)  
{  
    int* p = Arr;  
    for(int i = 1; i < n+1; ++i) {  
        sum = sum + *p;  
        ++p;  
    }  
}
```

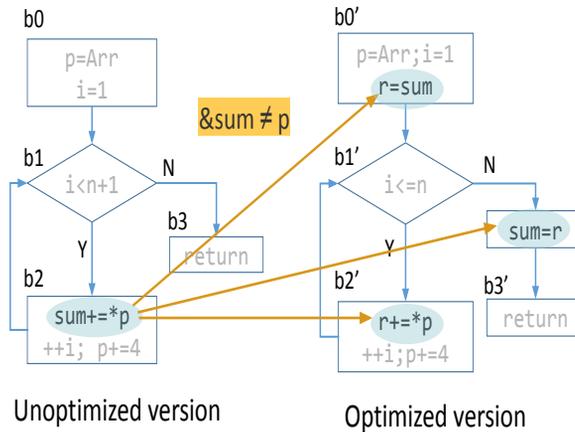


Thus, the compiler transformation that involves the register allocation of the sum variable across the for loop is a legal transformation based on these based-on semantics and associated UB conditions.

EXAMPLE 2

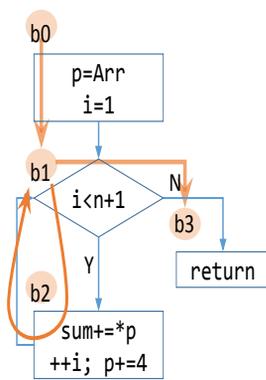
```
// computes the sum of  
// first n integers of Arr
```

```
int Arr[256];  
int sum = 0;  
void compute_sum(int n)  
{  
    int* p = Arr;  
    for(int i = 1; i < n+1; ++i) {  
        sum = sum + *p;  
        ++p;  
    }  
}
```

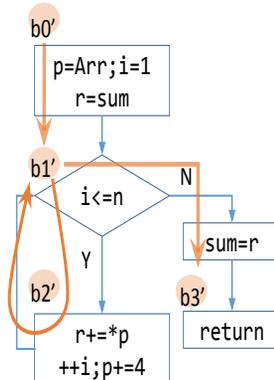


In particular, we are able to infer that because `P` is based on `Arr`, it cannot alias with the address of `SUM` for a non-UB program.

EXAMPLE 2



Unoptimized version
(Program 1)

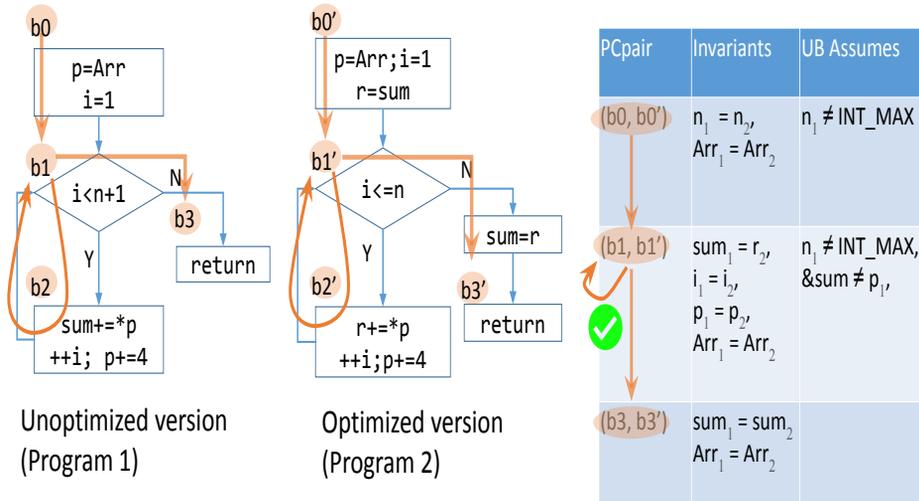


Optimized version
(Program 2)

PCpair	Invariants
(b_0, b_0')	$n_1 = n_2$, $Arr_1 = Arr_2$
(b_1, b_1')	$sum_1 = r_2$, $i_1 = i_2$, $p_1 = p_2$, $Arr_1 = Arr_2$
(b_3, b_3')	$sum_1 = sum_2$, $Arr_1 = Arr_2$

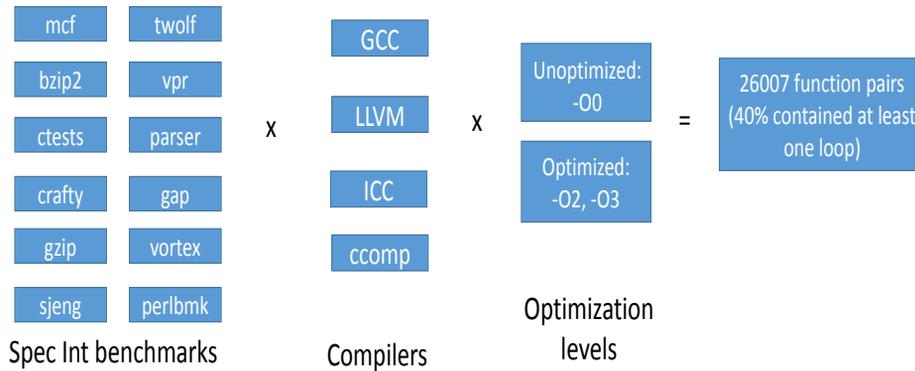
If we try to check the equivalence of these two versions of the program, we will never be able to determine equivalence until we model the corresponding UB semantics. In particular, the equivalence proof would fail on the inductive step, i.e. the loop edge from (b_1, b_1') to itself. This is because the theorem prover would return counterexamples that would set N to INT_MAX , or P to alias with $\&SUM$, because in these cases, we will be unable to complete the induction step.

EXAMPLE 2



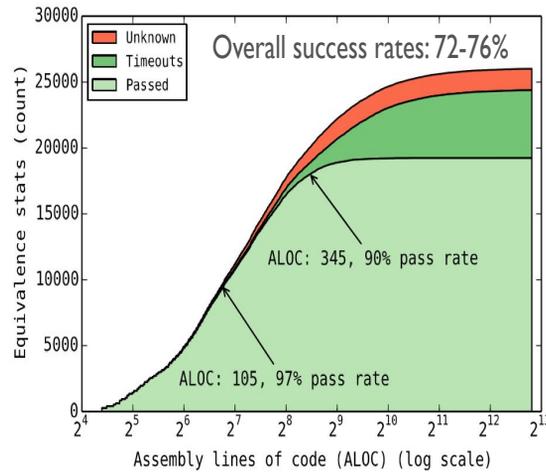
Thus, we add another column in our bisimulation relation table that encodes the conditions for the absence of UB. These conditions are derived from the programming language's UB semantics, e.g., no signed overflow. We call these conditions, UB assumes, and they are encoded using the bitvector operators available in SMT-like syntaxes.

EXPERIMENTAL SETUP



Before I discuss the algorithm any further, I will first share the results of our attempts at computing equivalence across compiler transformations produced by four different compilers, namely GCC, LLVM, ICC, and CompCert. We used the SPEC Integer benchmarks as the source programs. We performed equivalence checks for unoptimised O0 vs. optimised O2/O3 compilations. Overall, this experiment involved computing equivalence for over 26000 function pairs.

EQUIVALENCE CHECKING SUCCESS RATES



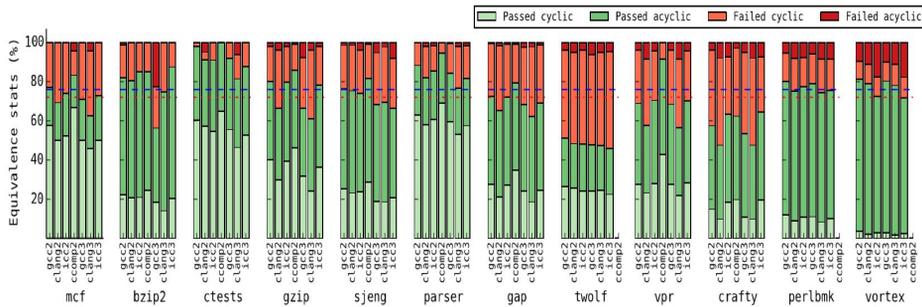
Published in: APLAS 2017. 15th Asian Symposium on Programming Languages and Systems.
Manjeet Dahiya, Sorav Bansal: Black-box equivalence checking across compiler optimizations.

Here is a summary of the equivalence checking statistics. On the X axis we show the number of Assembly lines of code (ALOC) for the function pair being checked for equivalence. On the Y axis, we show the cumulative count of function pairs for which the equivalence checker was able to generate a proof of equivalence (also called a success or a pass result). The success rate is the fraction of equivalence tests that returned a pass result (with a proof of equivalence). Till ALOC 100, the success rates for the equivalence checker are fairly high, but the success rates decrease rapidly for larger functions. Most of the equivalence failures for large functions are attributed to timeouts, and in some cases we could not ascertain the cause of the equivalence failure. Overall our success rates were 72-76%. This is not very encouraging especially given that most of the passes are for smaller functions.

SUCCESS RATE BREAKDOWN

- Success rate across O0-O2: 76%
- Success rate across O0-O3: 72%
- Verification runtime for passing: 313 s (avg), 8.5 s (median)

- Verification runtime for passing + failing: ~1hr (avg), 22 s (median)
- Largest verified function: 4754 ALOC
- Most complex function: 31 edges in the simulation relation



In our paper we have a more detailed breakdown of per benchmark and per compiler/optimization. In the interest of time, I will skip that discussion here.

ESTIMATING THE IMPACT OF UB ON OPTIMIZATION

Equivalence checking with modelling:	Success rate %	Drop in success rate
All UB	81	NA
All UB except <i>type based strict aliasing</i>	76	5
All UB except <i>signed integer overflow</i>	77	4
All UB except <i>out-of-bounds variable access</i>	50	31

The drop due to *out-of-bounds variable access* UB is significantly higher than other UBs

- Large number of global variables
- Register allocation or otherwise reordering of memory accesses are frequent and important optimizations

Published in: HVC 2017. 13th International Haifa Verification Conference

Manjeet Dahiya, Sorav Bansal: Modeling undefined behaviour semantics for checking equivalence across compiler optimizations.

Here are some interesting things you can do with such an equivalence checking capability. You can disable the modelling of specific types of UB and measure the effect on the success rates on our equivalence checker. This exercise provides insight into the relative importance of different types of UB. We find that disabling type-based strict aliasing assumptions reduces success rates by 5% from 81 to 76%. Similarly, disabling signed integer overflow assumptions decreases success rates by 4%. However, disabling the out-of-bounds variable access assumptions reduces success rates by a whopping 31%. This makes it clear that out of bounds memory accesses are the most consequential to compiler optimizations for C programs. This is not surprising because these latter UB assumptions are used for alias analysis which in turn is used for several transformations, including register allocation.

RESEARCH CHALLENGES

- Modeling Undefined Behaviour APLAS17, HVC17
- Identifying Correlations between Program Transitions OOPSLA20
- Efficient Encoding and Discharge of Proof Obligations SAT18
- LLVM UB / OCaml vs. C Ongoing

I will next move to our next major challenge that is on identifying correlation between program transitions automatically, towards the construction of a bisimulation relation.

EXAMPLE 3:VECTORIZATION

```
int LEN, a[LEN], b[LEN];
int c[LEN], d[LEN];
C0: void s441() {
C1:   for (int i = 0; i < LEN; i++) {
C2:     if (d[i] < 0) {
C3:       a[i] += b[i] * c[i];
C4:     } else if (d[i] == 0) {
C5:       a[i] += b[i] * b[i];
C6:     } else {
C7:       a[i] += c[i] * c[i];
C8:     }
C9:   }
C10:}
```

Consider this program. It involves a for loop on I which reads/writes four different arrays A, B, C, D. Depending on the value of d[I], it updates a[I] based on the values of b[I] and c[I]. This program is taken from the Testsuite for Vectorizing Compilers. Unsurprisingly, when this program is compiled, we obtain a highly vectored implementation

EXAMPLE 3 : VECTORIZATION

```
int LEN, a[LEN], b[LEN];
int c[LEN], d[LEN];
C0: void s441() {
C1:   for (int i = 0; i < LEN; i++) {
C2:     if (d[i] < 0) {
C3:       a[i] += b[i] * c[i];
C4:     } else if (d[i] == 0) {
C5:       a[i] += b[i] * b[i];
C6:     } else {
C7:       a[i] += c[i] * c[i];
C8:     }
C9:   }
C10:}

A0: s441:
A1:   r1 = 0
A2:   xmm1 = a[r1 .. r1+3]
A3:   xmm2 = xmm1 + b[r1 .. r1+3]*c[r1 .. r1+3]
A4:   xmm3 = xmm1 + b[r1 .. r1+3]*b[r1 .. r1+3]
A5:   xmm4 = xmm1 + c[r1 .. r1+3]*c[r1 .. r1+3]
      // pcmrgtd
A6:   xmm0 = (d[r1] < 0), .. , (d[r1+3] < 0)
A7:   xmm1 = xmm0 ? xmm2 : xmm1 // pblendvb
      // pcmpeqd
A8:   xmm0 = (d[r1] == 0), .. , (d[r1+3] == 0)
A9:   xmm1 = xmm0 ? xmm3 : xmm1 // pblendvb
      // pcmrgtd
A10:  xmm0 = (d[r1] > 0), .. , (d[r1+3] > 0)
A11:  xmm1 = xmm0 ? xmm4 : xmm1 // pblendvb
A12:  a[r1 .. r1+3] = xmm1
A13:  r1 += 4
A14:  if (r1 != LEN) goto A2
A15:  ret
```

This slide shows the vectorized x86 assembly implementation on the right. The vectorized implementation uses xmm registers and opcodes like pcmrgtd and pblend. A manual reading of the assembly code is rather difficult. But overall, this represents a four-unrolling of the loop on the left.

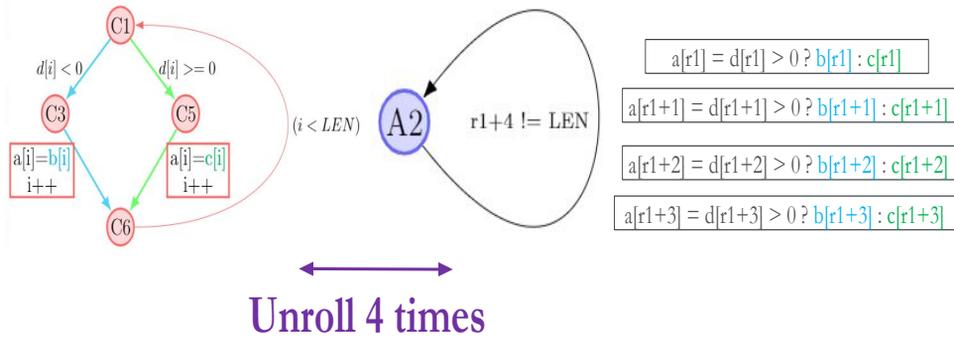
Counter Algorithm : Two Key Ideas

OOPSLA20

- Correlate “sets of paths” (pathsets) instead of individual paths
- Counterexample-Guided Best-First Search

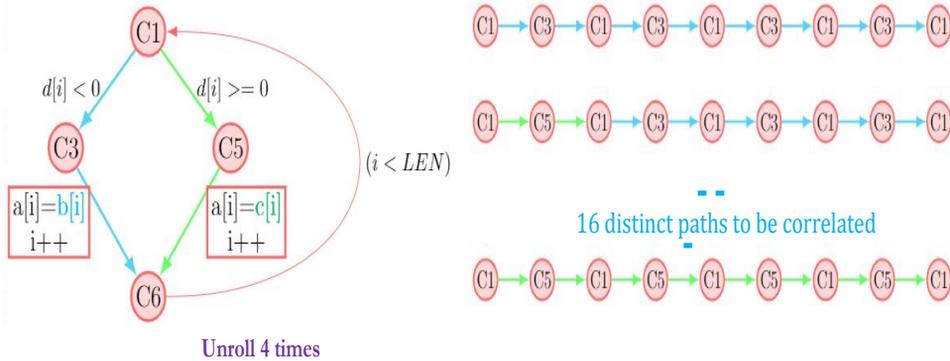
Our algorithm, called Counter, involves two key ideas: (1) correlate sets of paths, or pathsets, instead of individual paths. And (2) counterexample-guided best-first search. I will explain both these ideas through examples.

EXAMPLE 4 : CORRELATING PATHSETS



I will simplify the example that we had shown on the previous slide and just consider the loop paths of the two programs. Also let's assume that there is only a two-way if statement in the loop body: the if condition checks the value of $d[i]$. If it is positive, it assigns $c[i]$ to $a[i]$, else it assigns $b[i]$ to $c[i]$. In the assembly program, this single iteration of the loop is unrolled four times and vectorized. The corresponding operation of a single iteration of the assembly loop iteration involves updating four different elements of A, namely $r1$, $r1+1$, $r1+2$, and $r1+3$. The value used to update each of these elements could be independently derived from arrays B or C

UNROLLING RESULTS IN AN EXPONENTIAL NUMBER OF PATHS



If 3 level control flow and 8 unrolling, $3^8 = 6561$ distinct paths to be correlated

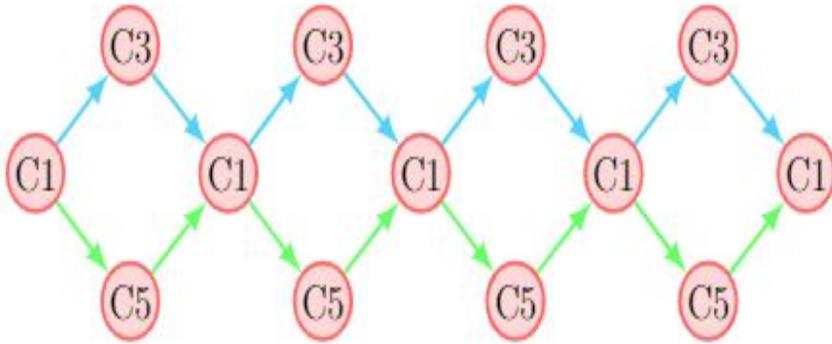
If we unroll this four times, there are actually 2^4 or 16 total number of potential paths that a program can take. In general, if there are k path inside the loop body that is unrolled n times, then we have potentially k^n paths that may be taken. For example, for $k=3$ and $n=8$, as in our previous example, this evaluates to over 6000 paths. Correlating every path separately would be very expensive.

Correlating Pathsets

- Correlating Individual Paths is not Scalable
- Counter identifies a correlation for a set of paths, or pathset, in a single step

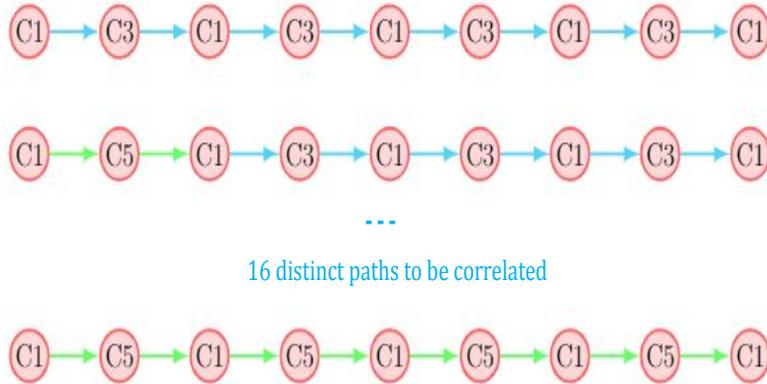
Instead we identify a correlation for a set of paths, or a pathset, in a single step.

Correlating Pathsets



We use a directed-acyclic-graph representation for a pathset, such as the one shown above. This directed acyclic graph encodes the possible paths that may be taken. Even though this represents an exponential number of different path possibilities, this graph representation is linear in the size of the unroll factor.

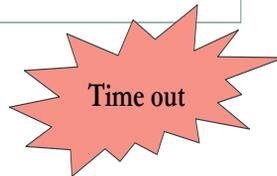
Correlating Pathsets



Yet this linear directed acyclic graph representation exactly represents an exponential number of paths, 16 in this example.

Correlating Pathsets

SMT-proof obligations	Correlating Pathsets	Correlating individual paths
Usual-case	Linear	Exponential
Worst-case	Exponential	Exponential



By representing a pathset using a linear-sized representation, and by correlating a pathset in a single step, we are able to make our algorithm scalable. In the usual-case, this approach of correlating a pathset in a single step would yield linear-time algorithms, assuming that the transformations performed by the compiler can be tackled using reasoning at pathset-granularity. However, the worst-case still remains exponential. On the other hand, the approach of correlating individual paths would be exponential even in the usual case.

Key Idea #2

Counterexample Guided Best-First Search

Our second key idea is counterexample guided best-first search

EXAMPLE 5 : LOOP TRANSFORMATIONS

```
#define LEN 1000
int original() {
    int sum = 0;
    int mid = LEN / 2;
    for ( int i = 0; i < LEN; i ++ ) {
        if ( i < mid ) sum += c[a[i]];
        if ( i >= mid ) sum += b[i];
    }
    return sum;
}

int loopSplitting() {
    int sum = 0;
    int mid = LEN / 2;
    for ( int i = 0; i < mid; i ++ ) {
        if ( i < mid ) sum += c[a[i]];
        if ( i >= mid ) sum += b[i];
    }
    for ( int i = mid; i < LEN; i ++ ) {
        if ( i < mid ) sum += c[a[i]];
        if ( i >= mid ) sum += b[i];
    }
    return sum;
}
```



Consider this example program on the left. Here there is a single loop that iterates i from 0 to LEN . Also, we have $mid = LEN/2$. The body of the loop does something different for $i < mid$ and something else for $i \geq mid$.

An optimizing compiler typically splits the loop into two loops as shown on the right. The first loop iterates from 0 to mid , while the second loop iterates from mid to LEN . The bodies of the two loops are identical to the body of the original loop.

EXAMPLE 5 : LOOP TRANSFORMATIONS

```
int loopSplitting() {
    int sum = 0;
    int mid = LEN / 2;
    for (int i = 0; i < mid; i++) {
        if (i < mid) sum += c[a[i]];
        if (i >= mid) sum += b[i];
    }
    for (int i = mid; i < LEN; i++) {
        if (i < mid) sum += c[a[i]];
        if (i >= mid) sum += b[i];
    }
    return sum;
}

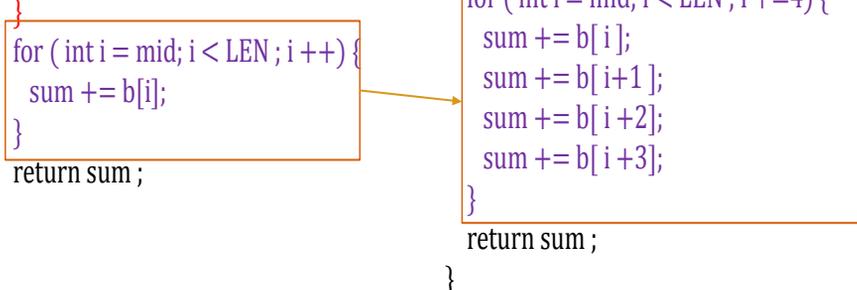
int loopUnswitching() {
    int sum = 0;
    int mid = LEN / 2;
    for (int i = 0; i < mid; i++) {
        sum += c[a[i]];
    }
    for (int i = mid; i < LEN; i++) {
        sum += b[i];
    }
    return sum;
}
```

Next, the optimizing compiler transforms both loops using the loop unswitching transformation. Notice that in the first loop, the second statement is never executed; and in the second loop, the first statement is never executed. The compiler takes advantage of this fact to get rid of the other statement, and also removes the if condition in the body of both loops.

EXAMPLE 5 : LOOP TRANSFORMATIONS

```
int loopUnswitching() {
    int sum = 0;
    int mid = LEN / 2;
    for (int i = 0; i < mid; i++) {
        sum += c[a[i]];
    }
    for (int i = mid; i < LEN; i++) {
        sum += b[i];
    }
    return sum;
}

int loopUnrolling() {
    int sum = 0;
    int mid = LEN / 2;
    for (int i = 0; i < mid; i++) {
        sum += c[a[i]];
    }
    for (int i = mid; i < LEN; i += 4) {
        sum += b[i];
        sum += b[i + 1];
        sum += b[i + 2];
        sum += b[i + 3];
    }
    return sum;
}
```



The next transformation of a vectorizing compiler involves unrolling the second loop because it has a nice pattern where consecutive elements in memory accessed in a sequence of iterations.

EXAMPLE 5 : LOOP TRANSFORMATIONS

```
int loopUnrolling() {
    int sum = 0;
    int mid = LEN /2;
    for ( int i = 0; i < mid; i ++ ) {
        sum += c[a[i]];
    }
    for ( int i = mid; i < LEN ; i +=4 ) {
        sum += b[ i ];
        sum += b[ i+1 ];
        sum += b[ i +2 ];
        sum += b[ i +3 ];
    }
    return sum ;
}
```

A0 : loopVectorizedAndRegAllocated :
A1 : r1 = 0; r2 = 0;
A2 : r2 += c [a [r1]]
A3 : r1 ++
A4 : if (r1 != mid) goto A2
A5 : r1 = &b[mid]; r3=& b[LEN]; xmm0 = 0
A6 : xmm0 += * r1 , .. , *(r1 +12)
A7 : r1 += 16
A8 : if (r1 != r3) goto A6
A9 : xmm0 += (xmm0 >> 8)
A10 : xmm0 += (xmm0 >> 4)
A11 : r2 += xmm0 [31:0]
EA : ret r2

Finally, the unrolled loop body of the second loop is implemented using a single vector operation in the assembly code. Also, in the assembly syntax, we have several types of register allocation and other transformations.

End-to-End Equivalence Check

```
#define LEN 1000
```

```
C0: int original() {
```

```
C1: int sum = 0;
```

```
C2: int mid = LEN / 2;
```

```
C3: for (int i = 0; i < LEN; i++) {
```

```
C4:   if (i < mid) sum += c[a[i]];
```

```
C5:   if (i >= mid) sum += b[i];
```

```
C6: }
```

```
EC: return sum;
```

```
}
```

```
A0: loopVectorizedAndRegAllocated:
```

```
A1: r1 = 0; r2 = 0;
```

```
A2: r2 += c[a[r1]]
```

```
A3: r1 ++
```

```
A4: if (r1 != mid) goto A2
```

```
A5: r1 = &b[mid]; r3 = &b[LEN]; xmm0 = 0
```

```
A6: xmm0 += *r1, ..., *(r1 + 12)
```

```
A7: r1 += 16
```

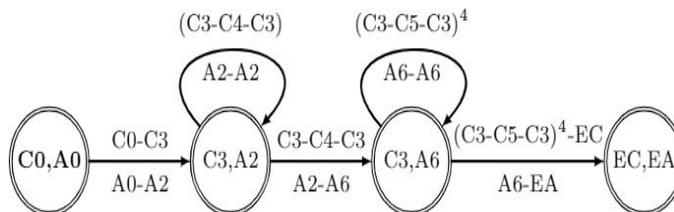
```
A8: if (r1 != r3) goto A6
```

```
A9: xmm0 += (xmm0 >> 8)
```

```
A10: xmm0 += (xmm0 >> 4)
```

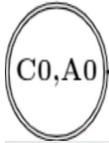
```
A11: r2 += xmm0 [31:0]
```

```
EA : ret r2
```



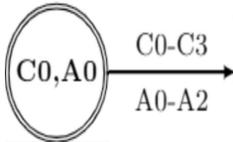
The two input programs that are given to our equivalence checker in this case are: the original program and the final vectorized assembly program, and it is expected to identify an equivalence proof automatically. Indeed, our equivalence checker works for this pair of examples, and the corresponding product program is shown as a CFG (control-flow graph) in the figure below. The entry node of the product CFG is (C0,A0). Then transition from C0 to C3 (entry to loop head) is correlated with the transition from A0 to A2 in assembly (entry to loop head for the first loop). The loop path C3-C4-C3 is correlated with A2-A2 under certain conditions. Under another mutually exclusive condition (e.g., $i = \text{mid}$), the C3-C4-C3 is correlated with A2-A6. The second loop in the product program correlates the loop path A6-A6 in the assembly program with four unrollings of C3-C5-C3 (the four unrollings are represented using the superscript). Finally if the assembly program exits from A6, the C program also exits from C3 albeit after making four more executions of the loop body.

Incremental Construction of the Product CFG



So how do we construct this product program, or product CFG (i.e., CFG of a product program). We do this incrementally. We already the start nodes of both programs, i.e., C_0 and A_0 , and so we correlate them first to add a node (C_0, A_0) to the product CFG.

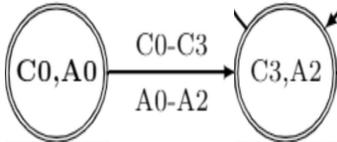
Incremental Construction of the Product CFG



We then identify a correlation for the assembly edge A0-A2. Here the correlation indicates that if any of the paths in the pathset A0-A2 is taken in assembly, then some path in C0-C3 is taken in the C program. Also, we relax the bidirectional condition: the condition holds only one way. For example, if one of the paths in C0-C3 is taken in the C program, it is not necessary for one of the paths in A0-A2 to be taken. This allows a pathset in the C program to be correlated with multiple pathsets of the assembly program.

Incremental Construction of the Product CFG

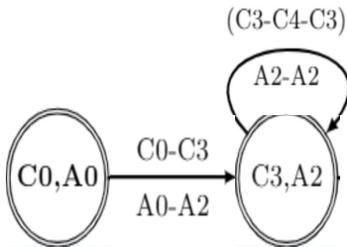
Use off-the-shelf invariant inference algorithms to infer affine, equality and inequality invariants on bitvectors and memory states



Infer Invariants at
(C3, A2)

Based on the pathset correlations, we can also correlate the end-points and add a new node, (C3, A2). Further, we use an off-the-shelf invariant inference algorithm to infer equality, inequality, and affine invariants. The only difference between prior work on invariant inference and our setting is that we infer these invariants on the product program. In contrast, most prior invariant inference work has tackled individual programs. The product program setting has no effect on the operation of the invariant inference algorithm.

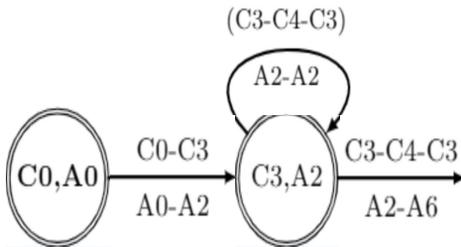
Incremental Construction of the Product CFG



Relax Invariants
at (C_3, A_2)

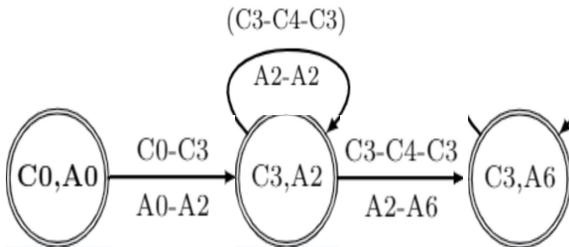
Then after each correlation, we relax invariants at the destination node of that correlation.

Incremental Construction of the Product CFG



We continue this incremental construction process.

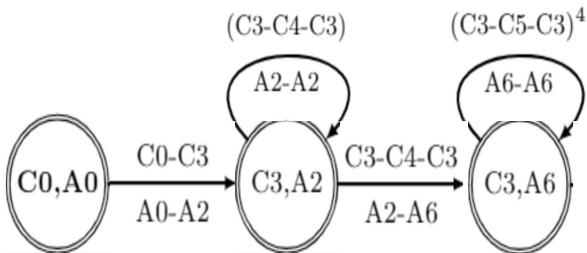
Incremental Construction of the Product CFG



Infer Invariants at
 (C_3, A_6)

Inferring invariants at any new node that is added.

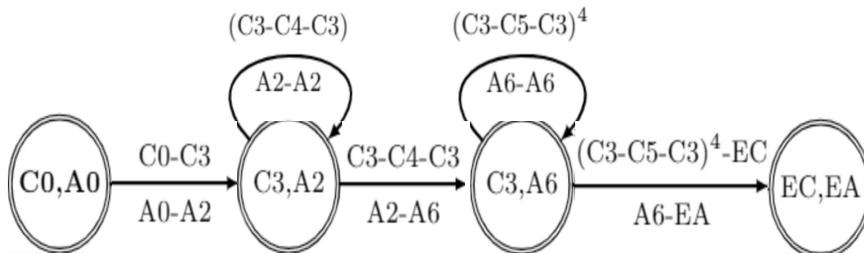
Incremental Construction of the Product CFG



Relax Invariants
at (C_3, A_6)

And relaxing invariants using any new edge that is added

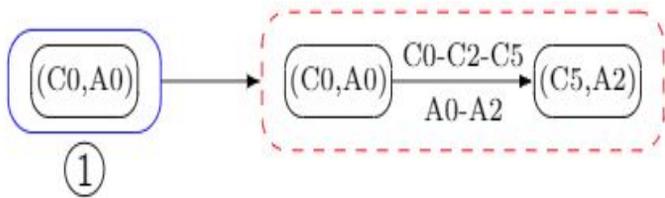
Incremental Construction of the Product CFG



Check equivalence of
return values under
inferred invariants

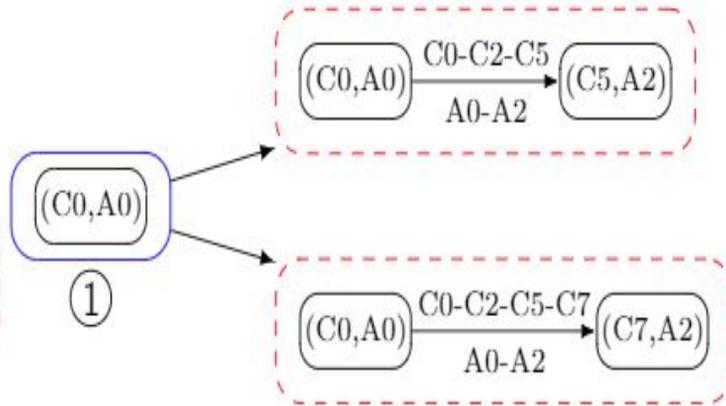
Finally, we check the equivalence of the return values (observables) under the inferred invariants. If the equivalence can be proven under the inferred invariants, we have obtained a proof of observable equivalence.

SEARCH SPACE



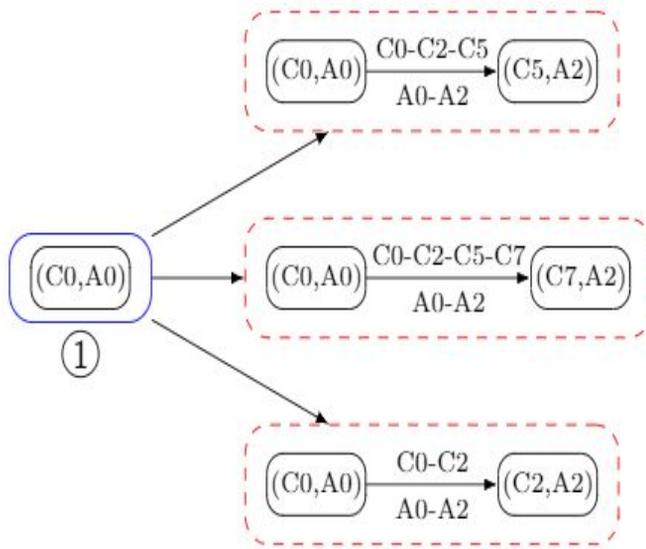
Of course, the incremental construction I described earlier seems too easy to be true. In reality, we don't know which assembly pathset to correlate with which C program pathset. For example, I could potentially correlate A0-A2 with C0-C2-C5

SEARCH SPACE



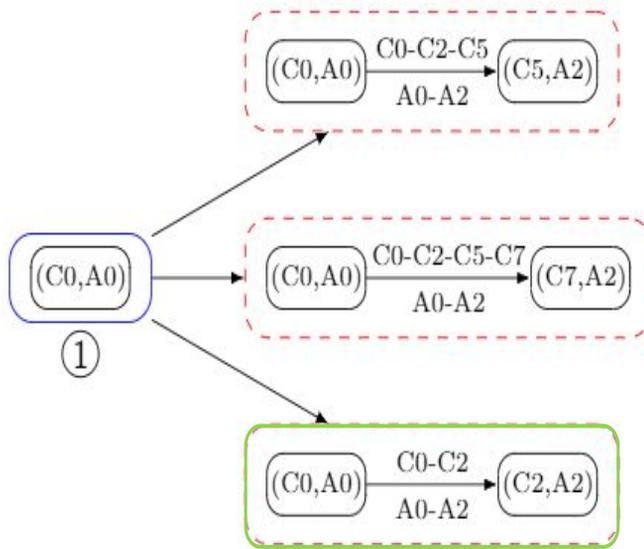
Or here is another correlation possibility that correlates $C0-C2-C5-C7$ with $A0-A2$

SEARCH SPACE



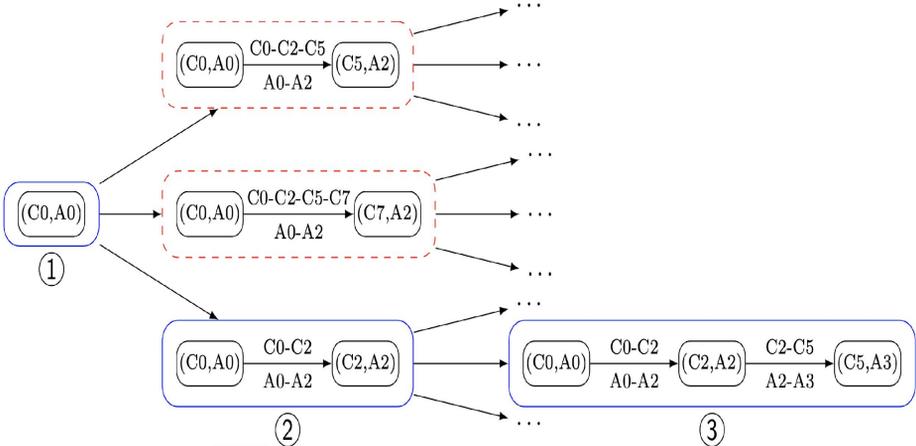
And so on...

SEARCH SPACE



If I pick one of these possibilities, then I have a similar choice at the next step.

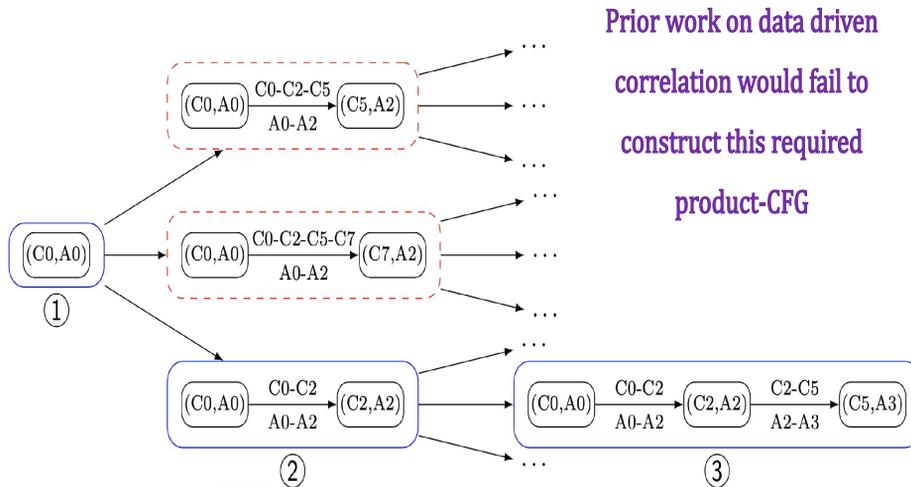
SEARCH SPACE



Exhaustive search would take years to compute equivalence

Overall, this is an exponential search space and an exhaustive search would take years to complete even for small examples.

SEARCH SPACE



Prior work on data driven correlation would fail to construct this required product-CFG

Exhaustive search would take years to compute equivalence

No prior work before ours can handle the types of transformations that I described using the previous example.

Counterexamples

During invariant inference, we make potential GUESSES for invariants. We try to prove a GUESS using an SMT Solver.

- If the GUESS is provable, we have found an invariant.
- If not, the SMT solver returns a counterexample



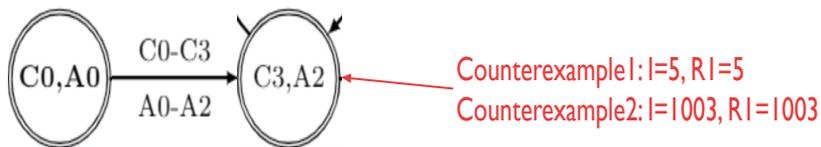
Infer Invariants at
(C3, A2)

A central idea that we use in our algorithm is the idea of Counterexamples. During invariant inference, we make potential GUESSES for a possible invariant. We then check a GUESS by trying to prove that it holds on the incoming edges. If it holds on all incoming edges, then we have found an invariant. If it does not hold on some incoming edge, then we obtain a counterexample from the SMT solver

Counterexamples

A counterexample at a node is a potential concrete machine state that may occur at that particular node during execution.

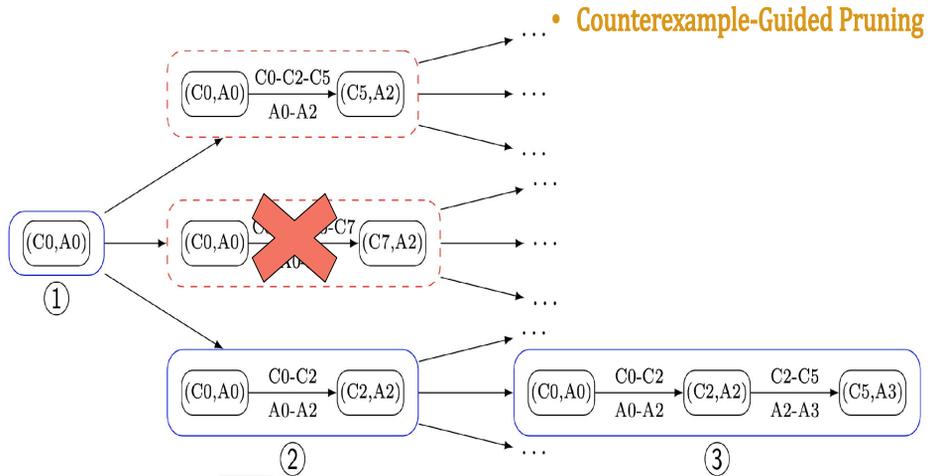
The concrete state would involve valuations for (related) variables of both C and A.



Infer Invariants at
 $(C3, A2)$

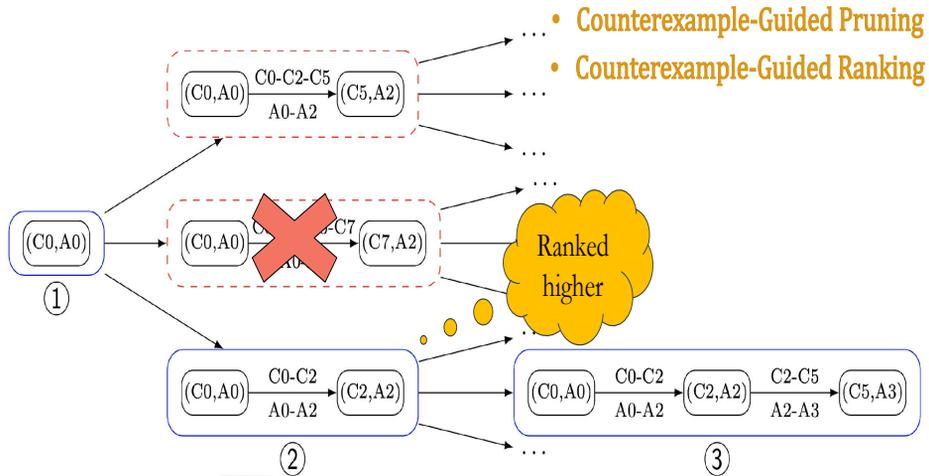
A counterexample at a node represents a potential concrete machine state that may occur at that particular node during execution. For a product CFG, this would involve valuations of variables for both programs C and A. Moreover, because the product program is executing in lockstep, we can expect the variables of the two programs to have some relations, e.g., I in C is always equal to R1 in A for all the counterexamples.

COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH



The two primary ideas that allow us to navigate this large search space are: (1) counterexample guided pruning. Using the counterexamples, we can potentially determine that some correlations are obviously incorrect if we find that the counterexample produces inconsistent behaviour on the two machines. I will show an example of an inconsistent behaviour.

COUNTEREXAMPLE GUIDED BEST-FIRST SEARCH



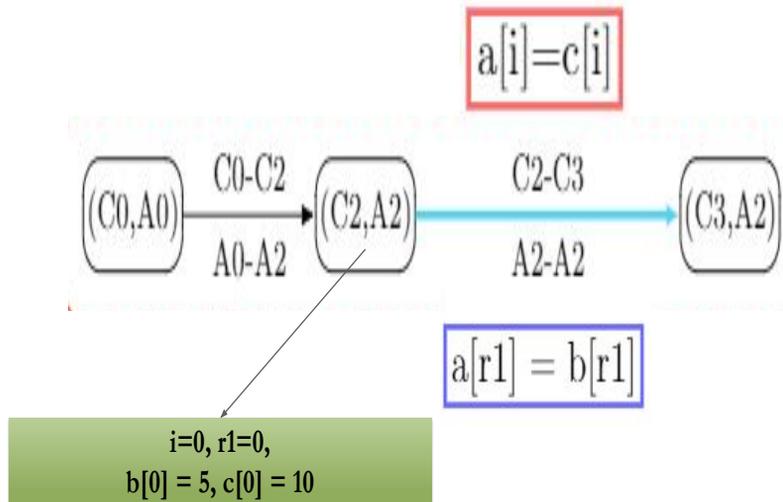
(2) Counterexample-Guided Ranking. We use a counterexample guided heuristic to rank some potential correlations higher than the other. Depending on the behaviour of the counterexamples, we can identify that some candidates are more promising than others. I will show this in more detail too.

Counterexample Guided Best-First Search

. Counterexample Guided Pruning

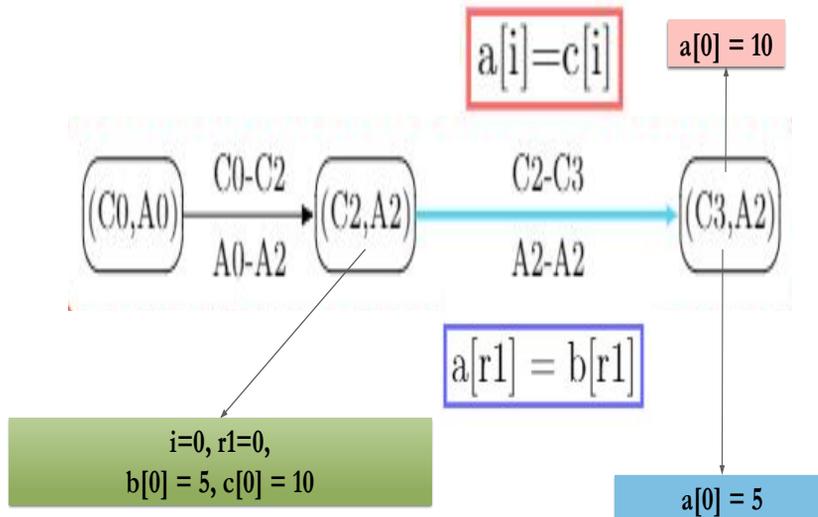
I will first show an example for counterexample guided pruning

COUNTEREXAMPLE EXECUTION



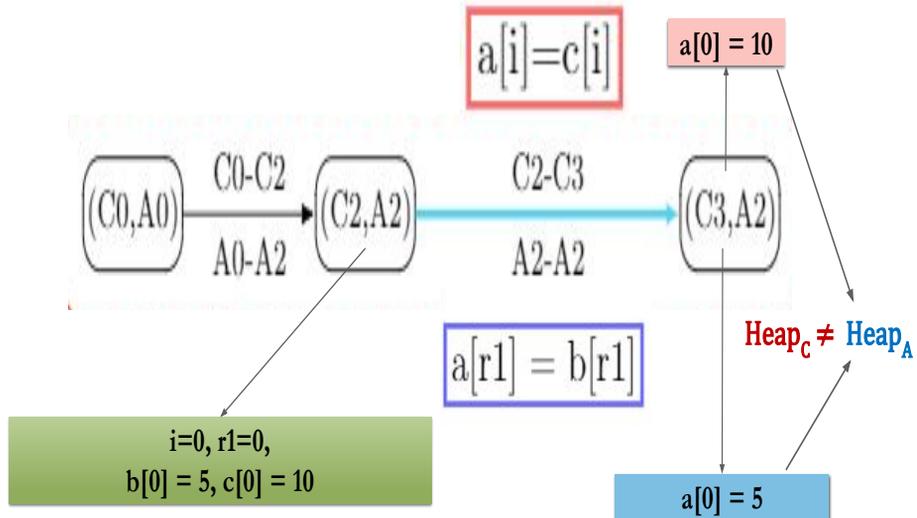
Consider this counterexample at $(C2, A2)$ shown in the green box. Also consider this product-CFG edge $C2-C3, A2-A2$. Let's say that $C2-C3$ executes the statement $a[i]=c[i]$. Similarly, $A2-A2$ executes $a[r1]=b[r1]$. Now, we will interpret both these operations of the product-CFG edge on the counterexample at $(C2, A2)$. Based on this, we will get a counterexample at $(C3, A2)$, the destination node of this product-CFG edge.

COUNTEREXAMPLE EXECUTION



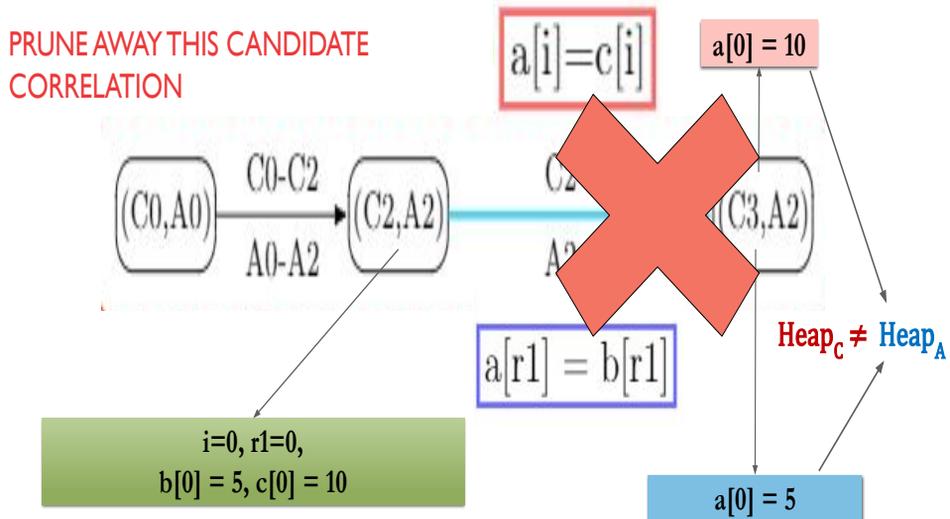
In this example, the evaluation of the edge on the green counterexample would cause different values for the “A” array. On the C program side, at $(C3, A2)$, we would have $a[0]$ evaluated to 10. Whereas on the assembly side $a[0]$ would evaluate to 5.

Counterexample Guided Pruning



This effectively means that the heaps would be unequal at $(C3, A2)$.

Counterexample Guided Pruning



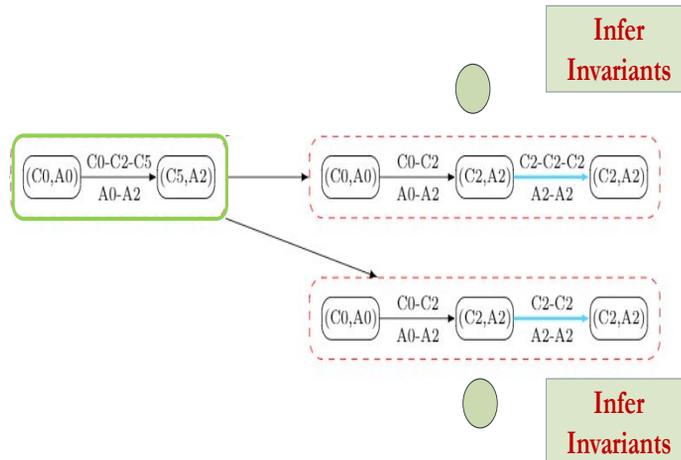
If we expect the heaps to be equal at all correlated nodes, then we can easily prune this candidate correlation because it does not produce identical heap states at the destination node of the product-CFG edge.

Counterexample Guided Best-First Search

- . Counterexample Guided Pruning
- . Counterexample Guided Ranking

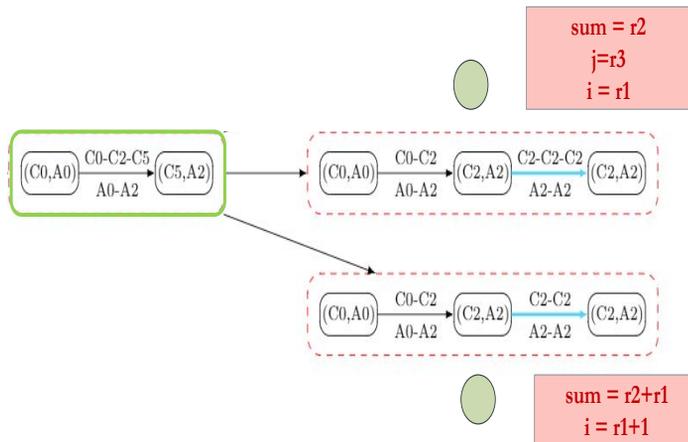
Next I will show an example for counterexample-guided ranking

Infer Invariant Covers for Executed Counterexamples



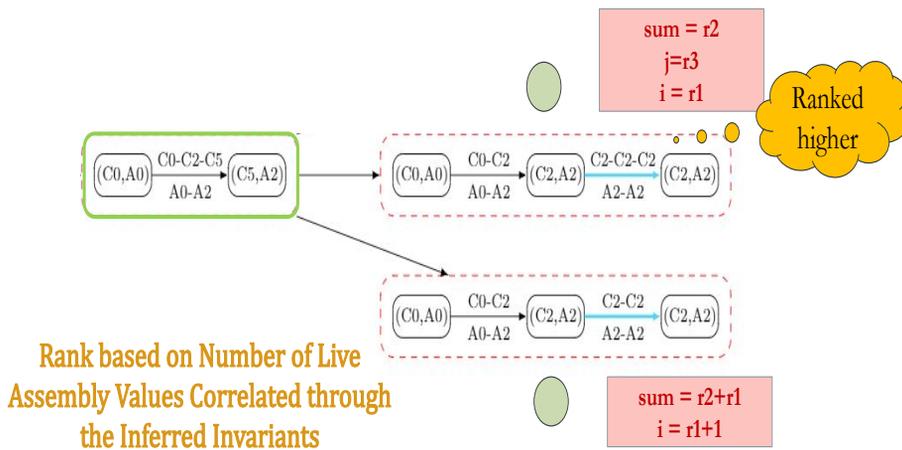
These circles represent counterexamples identified for two different correlations at the destination node (C_2, A_2) . Based on the counterexamples, we try and identify the possible affine relations between the variables of the two programs.

Infer Invariant Covers for Executed Counterexamples



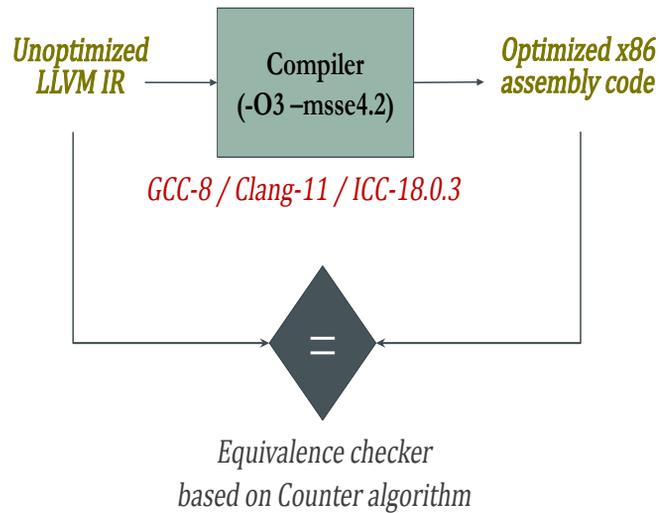
Let's say that the inferred relations from the counterexamples are:
 $sum=r2, j=r3, i=r1$ for the first correlation; and $sum=r2+r1, i=r1+1$ for the second correlation.

Infer Invariant Covers for Executed Counterexamples



One of them has affine relations for three different assembly registers, namely $r1$, $r2$, and $r3$. Whereas the other correlation has affine relations only for $r1$ and $r2$, but not for $r3$. We rank the first correlation higher because it relates a larger number of assembly registers. This is our ranking heuristic. We find that this performs remarkably well in practice, to identify the most promising candidate at each step of the incremental product-CFG construction.

Counter Evaluation



I will now discuss the experimental results of applying the Counter algorithm to C programs and their x86 assembly counterparts generated through three different optimizing compilers, GCC, Clang, and ICC, with O3 and SSE4.2 vectorization transformations enabled.

Counter Evaluation

- TSVC Benchmarks : TestSuite for Vectorizing Compilers
 - 208 function-compiler pairs tested
 - **175 function-compiler pairs pass**

We took C programs from the TSVC benchmarks, which being a testsuite for vectorizing compilers, represents one of the hardest set of equivalence checking problems because compilers produce very aggressive transformations for these programs. Our tool is able to successfully compute equivalence for 175 of the 208 function-compiler pairs tested in this testsuite. This is much larger than any previous equivalence checking effort for these benchmarks.

Counter Evaluation

- TSVC Benchmarks : TestSuite for Vectorizing Compilers
 - 208 function-compiler pairs tested
 - **175 function-compiler pairs pass**
- LORE Repository for Loop Nests
 - **27 different vectorizable loop patterns, all pass**
 - 16 with multiple potentially-nested loops
 - 6 where multiple control flow paths in the loop body
 - 17 use multi-dimensional arrays

We also tested on the LORE repository of loop nests which contain 27 different vectorizable loop patterns. Our equivalence checker is able to compute equivalence for all of these patterns successfully.

Bugs Discovered

<https://compiler.ai/bugs>

- [Bug in ICC-16.03 involving integer overflow](#)
- [Bug in ICC-16.03 related to incorrect reordering of memory accesses](#)
- [Bug in GCC-4.8 involving incorrect reordering of memory accesses](#)
- [Bug in Qemu machine emulator that is shipped with Linux/KVM hypervisor](#)
- [Three bugs in DietLibc related to missing unsigned-to-signed typecasts](#)
- [Bug in the Yices SMT Solver related to incorrect query result](#)

Over the years, we have found several bugs in compilers and other software such as the Qemu binary translator, a C library and an SMT solver. All these bugs were found using our equivalence checker --- in all these cases, when an expected equivalence proof failed, we tried to identify the reason for the failure. We would typically expect a shortcoming of our tool, but sometimes we found that the bug was in the program pair being tested. Except the GCC bug, all other bugs were previously unknown and were fixed immediately upon reporting by us. You can find more details on these bugs at <https://compiler.ai/bugs>

RESEARCH CHALLENGES

- Modeling Undefined Behaviour APLAS17, HVC17
- Identifying Correlations between Program Transitions OOPSLA20
- Efficient Encoding and Discharge of Proof Obligations SAT18
- LLVM UB / OCaml vs. C Ongoing

A central effort in making this equivalence checking possible is improving its scalability through efficient encoding and discharge of proof obligations. This effort involves a lot of engineering and it is usually not possible to explain all the different optimizations we perform in a single talk like this. Some of these approaches are discussed in our SAT18 paper. But I will skip this discussion in the interest of time.

QUERY DECOMPOSITION

Most proof obligations can be expressed as Hoare triples

$$\{Pre\} w \{src=dst\}$$

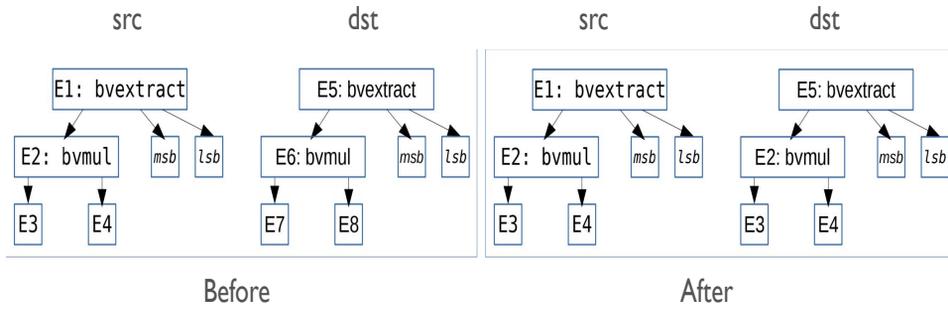
Which get lowered to

$$Pre \Rightarrow WP_w(src=dst)$$

Becomes easier if $WP_w(src=dst)$ is small

Skipped

QUERY DECOMPOSITION



If we can prove:
Pre \Rightarrow (E3=E7)
Pre \Rightarrow (E4=E8)

Counterexample-Guided Algorithm to Identify Equality Pairs

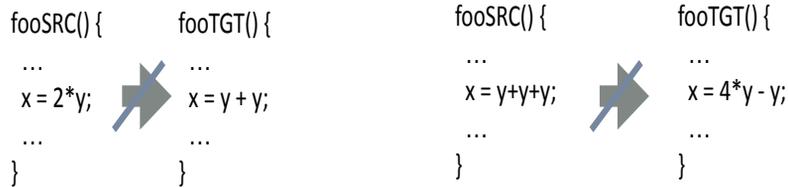
Skipped

RESEARCH CHALLENGES

- Modeling Undefined Behaviour **APLAS17, HVC17**
- Identifying Correlations between Program Transitions **OOPSLA20**
- Efficient Encoding and Discharge of Proof Obligations **SAT18**
- LLVM UB / OCaml vs. C **Ongoing**

Finally, I will briefly motivate some interesting problems that we are currently working on

LLVM UB AND COMPILER TRANSFORMATIONS



LLVM has weaker forms of undefined behaviour, such as undefined values and poison values. These non-deterministic values make some of the most basic transformations invalid. For example $2*y$ can no longer be strength-reduced to $y+y$. This is highly non-intuitive for human programmers and prior work has addressed this problem in limited settings of loop-free code or of bounded translation validation. We are interested in identifying translation validation algorithms for code with loops in the presence of such LLVM-style UB.

OCAML VS. C

```
exception Empty_list

type 'a my_list = Nil | List of 'a * 'a my_list

let head = function
  Nil -> raise Empty_list
  | List(e,_) -> e;;

let tail = function
  Nil -> Nil
  | List(_,t) -> t

let l = List(1, List(4, List(8, Nil)));;
```

```
// A linked list node
struct Node {
  int data;
  struct Node* next;
};

struct Node* head = NULL;
struct Node* second = NULL;
struct Node* third = NULL;

// allocate 3 nodes in the heap
head = (struct Node*)malloc(sizeof(struct Node));
second = (struct Node*)malloc(sizeof(struct Node));
third = (struct Node*)malloc(sizeof(struct Node));

head->data = 1; // assign data in first node
head->next = second; // Link first node with second

second->data = 2; // assign data to second node
second->next = third;

third->data = 3; // assign data to third node
third->next = NULL;
```

As I said earlier, the most interesting problem in this space is the automatic identification of equivalence proofs between a higher level of abstraction, such as a functional program, and a lower level of abstraction, such as a C program. This example shows a list implementation in OCaml on the left and C on the right. The C implementation has several implementation details like pointers, allocation, struct, etc. People have previously completed such equivalence proofs manually using proof assistants. However manual proofs are usually cumbersome and thus have low adoption. Automatically identifying equivalences would be a desirable capability - we are trying to generalize our bisimulation framework to these settings.

CONCLUSIONS

- Equivalence Checking is a fundamental problem with important applications in
 - Translation Validation, Push-button Verification, Program Synthesis and Superoptimization
- Much progress has been made over the past 20+ years
 - Invariant inference, automatic correlation, UB modelling, assembly-level modelling, ...
- Several problems are still open, some within shooting distance ...
 - Support for address-taken local variable modelling, scalability improvements, LLVM, ...

To conclude, (read from the slide).

COMPILERAI

<https://compiler.ai>



- Deep-tech start-up based on our research in the area of equivalence checking.
- Building the **first certified compiler** that uses automatic translation validation as the sole certification technique for **Indian Air Force**.
- Tools for **Source and Binary executable code analysis**.

Even after decades of research, formal verification tools have not become mainstream in the software development pipeline. We think that this is set to change soon. With the aim of “innovation to industry”, we have founded a deep-tech startup based on our research that develops certified compilers and tools for source and binary code analysis. We are looking for bright engineers who are passionate about the space of compilers and/or formal verification to join us.

THANK YOU
QUESTIONS?

See equivalence checking demo at
<https://compiler.ai/demo>

Thank you and I am happy to take questions. You can try the equivalence checker for yourself at compiler.ai/demo