Verifying term graph optimizations using Isabelle/HOL

Certified Programs and Proofs 2023

Brae J. Webb   Ian J. Hayes   Mark Utting

16^{th} January 2023

Brae Webb

Ian Hayes

Mark Utting

Our research

Verifying the correctness of *optimizations* in the *GraalVM* compiler.

Question

Who has heard of *GraalVM*?

Outline

1. Introduce *GraalVM*.
2. *Motivate* our research.
3. Overview the *contributions* of this paper.

§ *GraalVM*

"One VM to rule them all"

Question

What is *GraalVM*?

Question

What is *GraalVM*?

Answer

- An *optimizing JIT compiler* on top of the JVM.

Question

What is *GraalVM*?

Answer

- An *optimizing JIT compiler* on top of the JVM.
- *Open-source* and written in *Java*.

Question

What is *GraalVM*?

Answer

- An *optimizing JIT compiler* on top of the JVM.
- *Open-source* and written in *Java*.
- *Polyglot* support through partial evaluation.

Polyglot

GraalVM implements multiple languages, including *Java*, *JavaScript*, *Ruby*, *Python*, *R*, *C*, *C++*, *Web Assembly*, *Ada*, *Haskell*, *Rust* and more!

Partial evaluation

```
» cat Interpreter.java

int add(Context c,
        Node lhs, Node rhs) {
  return lhs.eval(c)
    + rhs.eval(c);
}
```

+

```
» cat source.mylang

(+) 20 x
```

```
» cat source.class

int add(Context c,
        Node lhs, Node rhs) {
    return 20 + c.lookup("x");
}
```

Partial evaluation

```
» cat Interpreter.java

int add(Context c,
        Node lhs, Node rhs) {
  return lhs.eval(c)
    + rhs.eval(c);
}
```

+

```
» cat source.mylang

(+) 20 20
```

```
» cat source.class

int add(Context c,
        Node lhs, Node rhs) {
    return 20 + 20;
}
```

Partial evaluation

```
» cat Interpreter.java

int add(Context c,
        Node lhs, Node rhs) {
  return lhs.eval(c)
    + rhs.eval(c);
}
```

+

```
» cat source.mylang

(+) 20 20
```

```
» cat source.class

int add(Context c,
        Node lhs, Node rhs) {
    return 40;
}
```

Other notable features

- *Ahead-of-time* compilation.
- *Tool support* for *profiling* and *debugging*.

§ *Motivation*

Our goal

Verify the correctness of *optimizations* in the *GraalVM* compiler.

Two questions

1. Why verify *just* the *optimizations*?
2. Why focus on *GraalVM*?

Why verify just the optimizations?

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

Abstract

Compilers should be correct. To improve the quality of C compilers, we created Csmith, a randomized test-case generation tool, and spent three years using it to find compiler bugs. During this period we reported more than 325 previously unknown bugs to compiler developers. Every compiler we tested was found to crash and also to silently generate wrong code when presented with valid input. In this paper we present our compiler-testing tool and the results of our bug-hunting study. Our first contribution is to advance the state of the art in compiler testing. Unlike previous tools, Csmith generates programs that cover a large subset of C while avoiding the undefined and unspecified behaviors that would destroy its ability to automatically find wrong-code bugs. Our second contribution is a

```
int foo (void) {
  signed char x = 1;
  unsigned char y = 255;
  return x > y;
}
```

Figure 1. We found a bug in the version of GCC that shipped with Ubuntu Linux 8.04.1 for x86. At all optimization levels it compiles this function to return 1; the correct result is 0. The Ubuntu compiler was heavily patched; the base version of GCC did not have this bug.

We created Csmith, a randomized test-case generator that sup-

Why verify just the optimizations? [Yang et al., 2011]

GCC $\frac{75}{79} \approx 90\%$

Why verify just the optimizations? [Yang et al., 2011]

GCC $\dfrac{75}{79} \approx 90\%$

Clang $\dfrac{183}{202} \approx 95\%$

Why verify just the optimizations? [Yang et al., 2011]

GCC $\frac{75}{79} \approx 90\%$

Clang $\frac{183}{202} \approx 95\%$

CompCert [Leroy et al., 2016] $\frac{0}{2} \approx 0\%$

Why verify just the optimizations? *[Yang et al., 2011]*

GCC $\frac{75}{79} \approx 90\%$

Clang $\frac{183}{202} \approx 95\%$

CompCert *[Leroy et al., 2016]*

$\frac{0}{0} \approx$ ¯_(ツ)_/¯

Two questions

1. ~~Why verify *just* the *optimizations*?~~
2. Why verify *GraalVM*?

Why verify GraalVM?

1. Comprehensive optimization suite.

Why verify GraalVM?

1. Comprehensive optimization suite.
2. Actively developed.

Why verify GraalVM?

1. Comprehensive optimization suite.
2. Actively developed.
3. Widely used.

Why verify GraalVM?

1. Comprehensive optimization suite.
2. Actively developed.
3. Widely used.
4. Hosted languages rely on optimizations.

Two questions

1. ~~Why verify *just* the *optimizations*?~~
2. ~~Why verify *GraalVM*?~~

§ *Our Contributions*

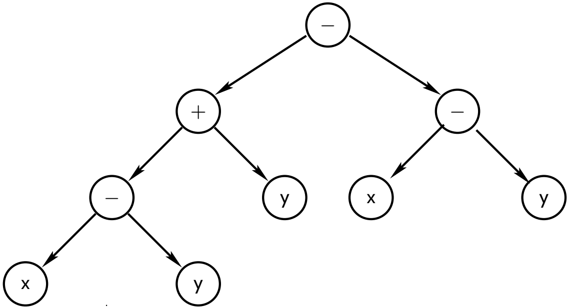
Contribution #1

Refinement proofs for GraalVM *expression* optimizations via *term rewriting*.

Consider

$$((x - y) + y) - (x - y)$$

$$(((x - y) + y) - (x - y))$$

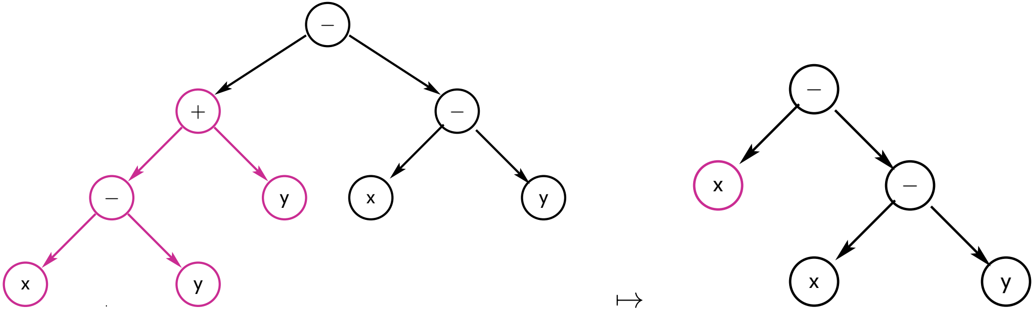


Optimizations as term rewriting

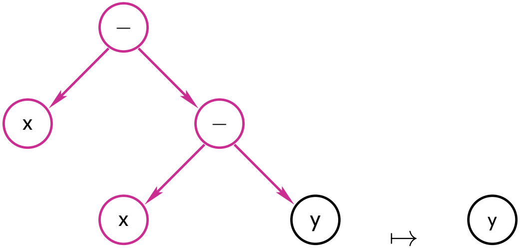
$$(x - y) + y \mapsto x \tag{1}$$

$$x - (x - y) \mapsto y \tag{2}$$

$(x - y) + y \mapsto x$



$x - (x - y) \mapsto y$



Theory vs Practice

Theory

- Expressions are *abstract syntax trees* or, *terms*.

Practice

Theory vs Practice

Theory

- Expressions are *abstract syntax trees* or, *terms*.
- Optimized by *term rewriting*.

Practice

Theory vs Practice

Theory

- Expressions are *abstract syntax trees* or, *terms*.
- Optimized by *term rewriting*.
- *Semantics* are expressed over term structure.

Practice

Theory vs Practice

Theory

- Expressions are *abstract syntax trees* or, *terms*.
- Optimized by *term rewriting*.
- *Semantics* are expressed over term structure.

Practice

- Expressions are a *sea-of-nodes graph*. [Click and Paleczny, 1995]

Theory vs Practice

Theory

- Expressions are *abstract syntax trees* or, *terms*.
- Optimized by *term rewriting*.
- *Semantics* are expressed over term structure.

Practice

- Expressions are a *sea-of-nodes graph*. *[Click and Paleczny, 1995]*
- Optimized by *graph updates*.

Theory vs Practice

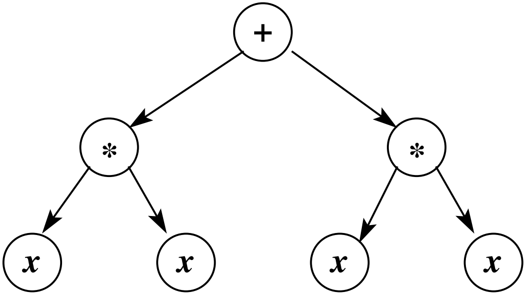
Theory

- Expressions are *abstract syntax trees* or, *terms*.
- Optimized by *term rewriting*.
- *Semantics* are expressed over term structure.

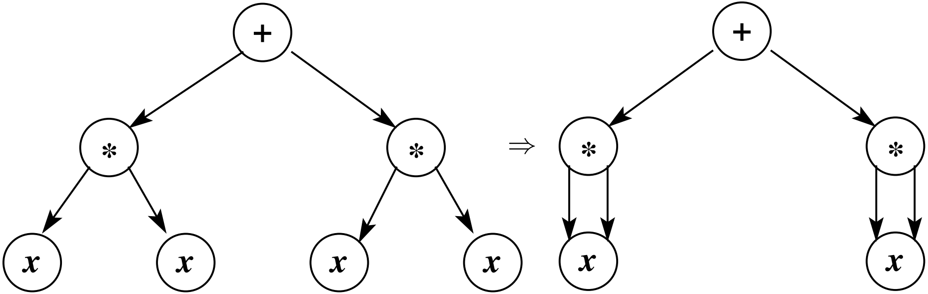
Practice

- Expressions are a *sea-of-nodes graph*. [Click and Paleczny, 1995]
- Optimized by *graph updates*.
- Efficiency is gained by *sharing*.

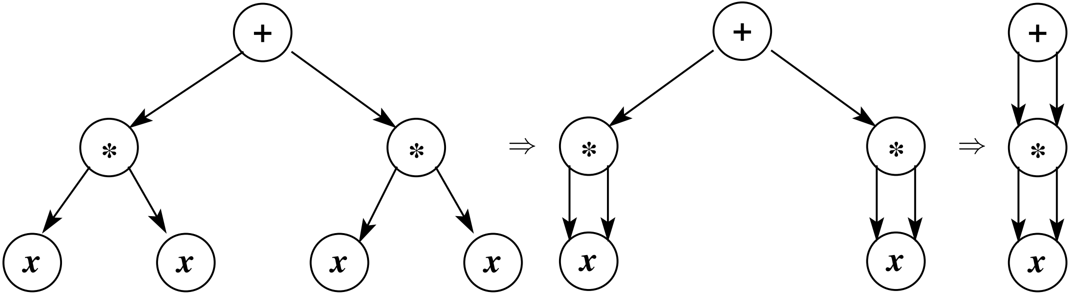
Sharing common sub-expressions



Sharing common sub-expressions



Sharing common sub-expressions



Our earlier work [Webb et al., 2021] defined semantics and optimizations on the *graph structure*.

Our earlier work [Webb et al., 2021] defined semantics and optimizations on the *graph structure*.

But these proofs were more complex than necessary...

How hard?

$$(!c)\ ?\ t\ :\ f \longmapsto c\ ?\ f\ :\ t \tag{3}$$

$$true\ ?\ t\ :\ f \longmapsto t \tag{4}$$

$$false\ ?\ t\ :\ f \longmapsto f \tag{5}$$

$$c\ ?\ x\ :\ x \longmapsto x \tag{6}$$

$$(u < v)\ ?\ t\ :\ f \longmapsto t \tag{7}$$

$$\text{when } upperBound(u) < lowerBound(v)$$

Why hard?

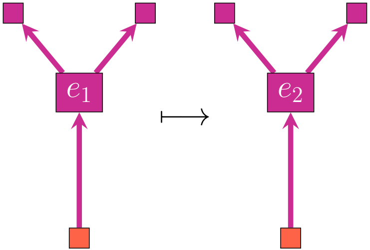
Graph updates can affect any expressions that share the updated expression.

Why hard?

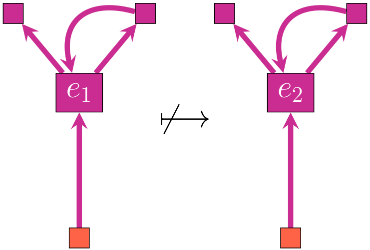
Graph updates can affect any expressions that share the updated expression.

We have to explicitly show that potential *cycles and self-reference* maintain semantic preservation.

Optimizing a graph



Optimizing a graph



Our approach

Abstract the *graph* representation to a *term* representation.

Term graph to term

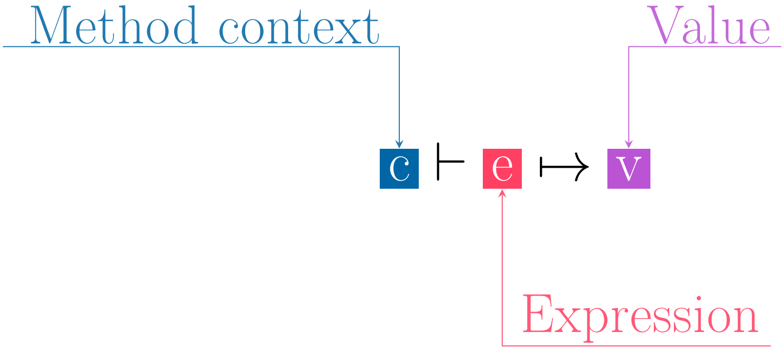
Node identifier

$$g \vdash n \simeq e$$

IR Graph

Expression

Expression semantics



Expression refinement

For e_2 to be a *sound* optimization of e_1 we require that

- whenever e_1 evaluates to a value v in some context c, so does e_2.
- e_2 *refines* e_1,

$$(e_1 \sqsupseteq e_2) = (\forall c\ v.\ c \vdash e_1 \mapsto v \Rightarrow c \vdash e_2 \mapsto v)$$

- e_2 may be well formed in more contexts than e_1, e.g. $x - x \sqsupseteq 0$

Term graph semantics preservation

$$\frac{e_1 \sqsupseteq e_2 \qquad \{n\} \trianglelefteq g_1 \subseteq g_2 \qquad g_1 \vdash n \simeq e_1 \qquad g_2 \vdash n \simeq e_2}{g_1 \sqsupseteq g_2}$$

Refinements on *terms* are easier to *define* and *prove* than equivalent proofs on *graphs*.

Contribution #2

A domain specific language to express *expression* optimizations in *Isabelle/HOL* then

1. generate *soundness*, and
2. *termination* proof obligations.

```
» cat SubNode.thy

phase SubNode
    terminating trm
begin

optimization SubAfterAddRight: "(x + y) - y  ↦  x"
    sorry

end
```

```
» cat SubNode.thy

phase SubNode
    terminating trm
begin

optimization SubAfterAddRight: "(x + y) - y  ↦  x"
    sorry

end
```

1. $(x + y) - y \sqsupseteq x$
2. $trm((x + y) - y) > trm(x)$

Contribution #3

A workflow to enable proofs to be expressed in the GraalVM compiler and generated as *Isabelle/HOL* proofs.

```
» cat SubNode.java

if (forX instanceof AddNode) {
    AddNode x = (AddNode) forX;
    if (x.getY() == forY) {
        return x.getX();
    }
}
```

```
» cat SubNode.java

if (forX instanceof AddNode) {
    AddNode x = (AddNode) forX;
    if (x.getY() == forY) {
        // veriopt: SubAfterAddRight: ((x + y) - y) |-> x
        return x.getX();
    }
}
```

Concluding

1. There are good *motivations* to verify GraalVM optimizations.

Concluding

1. There are good *motivations* to verify GraalVM optimizations.
2. We have a framework to *verify graph optimizations as term rewrites*.

Concluding

1. There are good *motivations* to verify GraalVM optimizations.
2. We have a framework to *verify graph optimizations as term rewrites*.
3. We have infrastructure to *express optimizations*, *generate proof obligations*, and *automatically* apply tactics.

Concluding

1. There are good *motivations* to verify GraalVM optimizations.
2. We have a framework to *verify graph optimizations as term rewrites*.
3. We have infrastructure to *express optimizations*, *generate proof obligations*, and *automatically* apply tactics.
4. We have started *integrating* into the GraalVM compiler.

Future work

1. Formalize *strategy* operators to *combine* optimization rules.

Future work

1. Formalize *strategy* operators to *combine* optimization rules.
2. Automatically *generate code* to implement optimizations.

Future work

1. Formalize *strategy* operators to *combine* optimization rules.
2. Automatically *generate code* to implement optimizations.
3. Investigate techniques to encode *control-flow* optimizations.

References

[Click and Paleczny, 1995] Click, C. and Paleczny, M. (1995).
A simple graph-based intermediate representation.
SIGPLAN Not., 30(3):35–49.

[Leroy et al., 2016] Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., and Ferdinand, C. (2016).
CompCert - a formally verified optimizing compiler.
In *Embedded Real Time Software and Systems, 8th European Congress*, ERTS '16, Toulouse, France. SEE.

[Webb et al., 2021] Webb, B. J., Utting, M., and Hayes, I. J. (2021).
A formal semantics of the GraalVM intermediate representation.
In Hou, Z. and Ganesh, V., editors, *Automated Technology for Verification and Analysis*, volume 12971 of *Lecture Notes in Computer Science*, pages 111–126, Cham. Springer International Publishing.

[Yang et al., 2011] Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011).
Finding and understanding bugs in C compilers.
In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA. Association for Computing Machinery.

How to you know it's faithful?

Differential Testing of a Verification Framework for Compiler Optimizations (Experience Paper)

Mark Utting
The University of Queensland
Australia
m.utting@uq.edu.au

Brae J. Webb
The University of Queensland
Australia
b.webb@uq.edu.au

Ian J. Hayes
The University of Queensland
Australia
ian.hayes@uq.edu.au

ABSTRACT

We want to verify the correctness of optimization phases in the GraalVM compiler, which consist of many thousands of lines of complex Java code performing sophisticated graph transformations. We have built high-level models of the data structures and operations of the code using the Isabelle/HOL theorem prover, and can formally verify the correctness of those high-level operations. But the remaining challenge is: how can we be sure that those high-level operations accurately reflect what the Java is doing? This paper addresses that issue by applying several different kinds of differential testing to validate that the formal model and the Java code have the same semantics. Many of these validation techniques should be applicable to other projects that are building formal models of

high-level optimizations (the GraalVM compiler also includes many low-level machine-dependent optimizations, but they are outside the scope of this paper).

This paper addresses two research questions relating to validation issues between formal models and the real world:

(1) How can we validate that our IR semantics in Isabelle matches the expected semantics of the GraalVM compiler IR? This is non-trivial because the compiler IR has no formal semantics and its nodes do not always directly correspond with JVM constructs because the IR has to be sufficiently general to support all its hosted languages.
(2) How can we be sure that our formal descriptions of each IR optimization *transformation* correctly match the transforma-