Verified expression graph optimization in GraalVM

Formal Methods Australia 2022

Brae J. Webb   Ian J. Hayes   Mark Utting

6^{th} June 2022

Outline

1. Introduce *GraalVM*.
2. Introduce and *motivate* our research.
3. *Live demo* some of our work.
4. Explain the Isabelle/HOL representation.

Question

What is GraalVM™?

GraalVM

A State-of-the-Art *Optimizing* Compiler.

Key points on GraalVM

1. Open-source and written in Java.[1]

[1] https://github.com/oracle/graalvm
[2] Multiple languages

Key points on GraalVM

1. Open-source and written in Java.[1]
2. Polyglot[2] compiler through partial evaluation.

[1] `https://github.com/oracle/graalvm`
[2] Multiple languages

Key points on GraalVM

1. Open-source and written in Java.[1]
2. Polyglot[2] compiler through partial evaluation.
3. JIT and AOT compilation modes.

---

[1] `https://github.com/oracle/graalvm`

[2] Multiple languages

Our goal

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

Two questions

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

Why verify the optimizations?

Compiler bugs are *bad*.

Compiler bugs are bad

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

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

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

Two questions

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

Two questions

1. ~~Why verify 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. Unique/novel/challenging IR.

Why verify GraalVM?

1. Comprehensive optimization suite.
2. Actively developed.
3. Unique/novel/challenging IR.
4. Partial evaluation relies heavily on optimizations.

Two questions

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

Bonus question

Why *verify* compilers at all?

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

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

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

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

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

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

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

Why verify 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)_/¯

Question

How do we verify an optimization?

Question

How do we verify an optimization?

Answer

Use *Isabelle/HOL* to show that the optimized program refines the unoptimized program, thus the optimization is *semantics preserving*.

Steps to prove an optimization

1. A definition of what *any program* will do in *any state*.
2. Demonstrate that for *all states*: The unoptimized program will *behave the same* as the optimized program.

   i.e. semantics are preserved.

* Where we say *program* we mean the *intermediate representation* of the program

Steps to prove an optimization

1. ~~A definition of what *any program* will do in *any state*~~.
   [Webb et al., 2021]

2. Demonstrate that for *all states*: The unoptimized program will *behave the same* as the optimized program.

   i.e. semantics are preserved.

* Where we say *program* we mean the *intermediate representation* of the program

These proofs were hard…

Question

Why were these proofs hard?

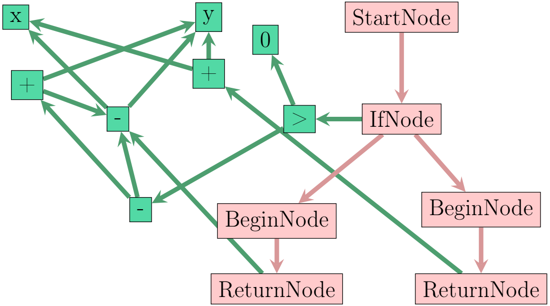
Back to GraalVM™...

A simple program

```
» cat PositiveAdd.java

static int positiveAdd(int x, int y) {
    if ((((x - y) + y) - (x - y)) > 0) {
        return x + y;
    } else {
        return x - y;
    }
}
```

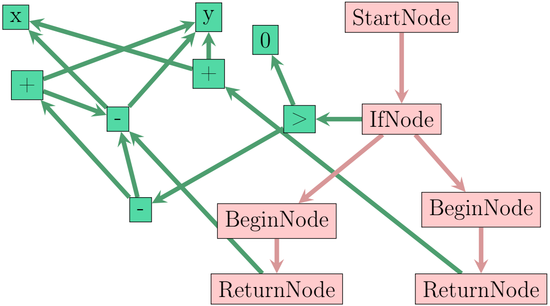
A simple program in GraalVM

```
» cat PositiveAdd.java

static int positiveAdd(int x,
    int y) {
    if ((((x - y) + y) - (x - y)
        ) > 0) {
        return x + y;
    } else {
        return x - y;
    }
}
```



A simple program in GraalVM

```
» cat PositiveAdd.java

static int positiveAdd(int x,
    int y) {
    if ((((x - y) + y) - (x - y)
        ) > 0) {
        return x + y;
    } else {
        return x - y;
    }
}
```



A mess!

Question

Why the mess?

Question

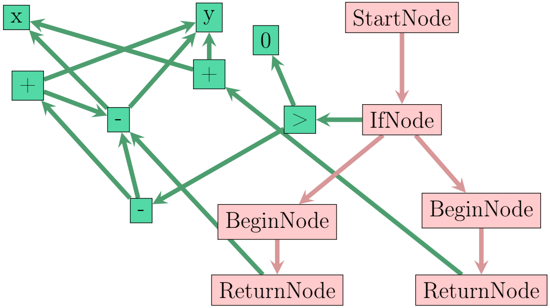
Why the mess?

Answer

It's very efficient [Click and Cooper, 1995].

Common subexpressions (or subgraphs) of the program can be *shared*.
Each unqiue expression is evaluated and stored once.

Important points on the GraalVM IR



- Expressions and control-flow are combined into one graph structure.
- Common sub-expressions are shared.
- Control-flow and data-flow (expressions) can have cycles.

Question

Why were these proofs hard?

Question

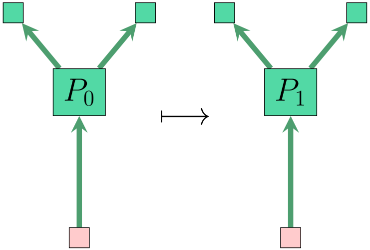
Why were these proofs hard?

Answer

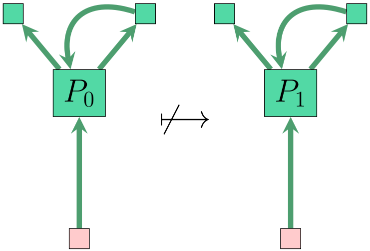
Graph updates can affect any expressions that share the updated expression.

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

Optimizing a graph



Optimizing a graph



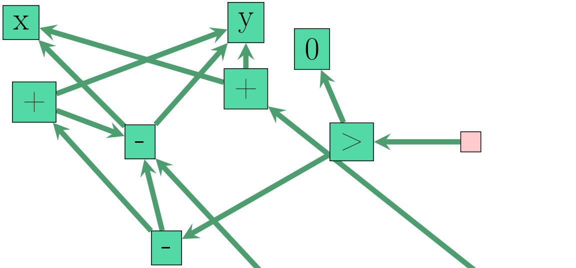
Optimizing a graph

Every optimization proof requires an *inductive proof* to satisfy *self-reference*.

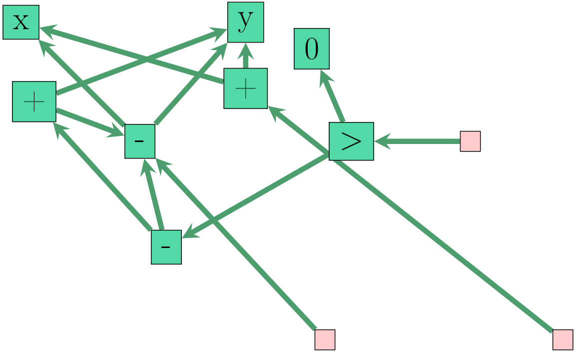
Solution

Represent expressions as *trees* rather than *graphs*.

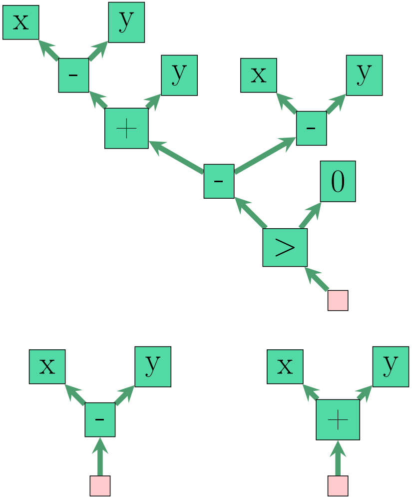
Recall our program graph



Removing sharing offers tree structures



$\longmapsto$



Added benefit

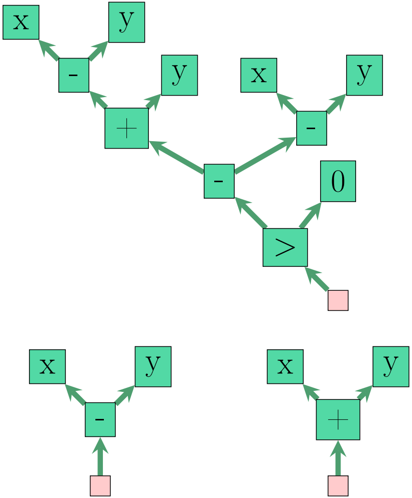
Trees enable natural infix encoding of optimization rules.

Added benefit

Trees enable natural infix encoding of optimization rules.

e.g. $x + 0 \longmapsto x$

How can we optimize these trees?



How about?

1. $(x - y) + y \longmapsto x$
2. $x - (x - y) \longmapsto y$

Wait...

Are these optimizations *correct*?

Wait...

Are these optimizations *correct*?

What about *integer overflow*?

Wait...

Are these optimizations *correct*?

What about *integer overflow*?

What about *floating-point* arithmetic?

Wait...

Are these optimizations *correct*?

What about *integer overflow*?

What about *floating-point* arithmetic?

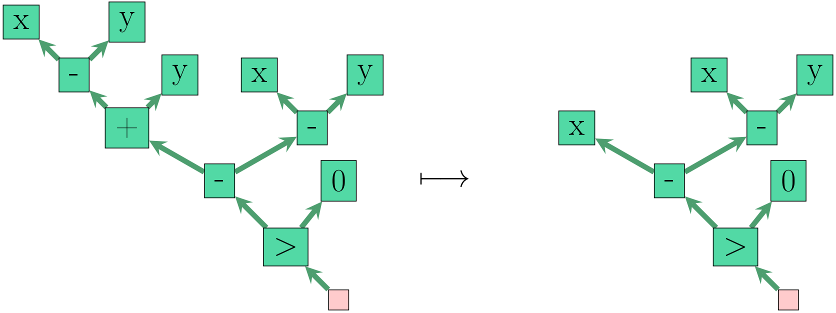
What about *side-effecting* operations?

Live demo

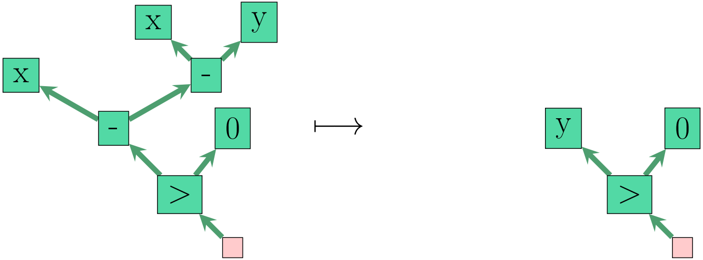
1. Show $(x - y) + y \longmapsto x$
2. Show $x - (x - y) \longmapsto y$

Now we have proofs

Let's apply the optimizations!



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



$$x - (x - y) \longmapsto y$$

The fun part

How we represent and prove these optimizations.

Expression Trees

datatype *IRExpr* =
  UnaryExpr IRUnaryOp IRExpr
  | *BinaryExpr IRBinaryOp IRExpr IRExpr*
  | *ConditionalExpr IRExpr IRExpr IRExpr*
  | *ParameterExpr nat Stamp*
  | *LeafExpr nat Stamp*
  | *ConstantExpr Value*

Expression Trees

datatype *IRExpr* =
UnaryExpr IRUnaryOp IRExpr
| *BinaryExpr IRBinaryOp IRExpr IRExpr*
| *ConditionalExpr IRExpr IRExpr IRExpr*
| *ParameterExpr nat Stamp*
| *LeafExpr nat Stamp*
| *ConstantExpr Value*

datatype *IRUnaryOp* =
UnaryAbs
| *UnaryNeg*
| *UnaryNot*
...

datatype *IRBinaryOp* =
BinAdd
| *BinMul*
| *BinSub*
...

Expression Semantics

Evaluating expressions depends on

- p a list of values of the parameters to the method
- m a mapping storing already evaluated leaf nodes
  - such as results from method calls

Expression Semantics

Evaluating expressions depends on

- p a list of values of the parameters to the method
- m a mapping storing already evaluated leaf nodes
  - such as results from method calls

Evaluating expression e to value v is represented by

Method parameters

Value

$$[\boxed{\mathrm{m}}, \boxed{\mathrm{p}}] \vdash \boxed{\mathrm{e}} \mapsto \boxed{\mathrm{v}}$$

Pre-calculated values

Expression

Expression Semantics

$$\frac{\mathit{valid\text{-}value}\ (\mathit{constantAsStamp}\ c)\ c}{[m,p] \vdash \mathit{ConstantExpr}\ c \mapsto c} \tag{1}$$

$$\frac{i < |p| \qquad \mathit{valid\text{-}value}\ s\ p_{[i]}}{[m,p] \vdash \mathit{ParameterExpr}\ i\ s \mapsto p_{[i]}} \tag{2}$$

$$\frac{[m,p] \vdash \mathit{xe} \mapsto v \qquad \mathit{unary\text{-}eval}\ \mathit{op}\ v \neq \mathit{UndefVal}}{[m,p] \vdash \mathit{UnaryExpr}\ \mathit{op}\ \mathit{xe} \mapsto \mathit{unary\text{-}eval}\ \mathit{op}\ v} \tag{3}$$

$$\frac{[m,p] \vdash \mathit{xe} \mapsto x \qquad [m,p] \vdash \mathit{ye} \mapsto y \qquad \mathit{bin\text{-}eval}\ \mathit{op}\ x\ y \neq \mathit{UndefVal}}{[m,p] \vdash \mathit{BinaryExpr}\ \mathit{op}\ \mathit{xe}\ \mathit{ye} \mapsto \mathit{bin\text{-}eval}\ \mathit{op}\ x\ y} \tag{4}$$

$$\frac{\mathit{val} = m\ n \qquad \mathit{valid\text{-}value}\ s\ \mathit{val}}{[m,p] \vdash \mathit{LeafExpr}\ n\ s \mapsto \mathit{val}} \tag{5}$$

Expression refinement

For $e2$ to be an correct optimization of $e1$ we require that

- whenever $e1$ evaluates to a value v in some context $[m, p]$, so does $e2$.
- $e2$ *refines* $e1$,

$$(e2 \leq e1) = (\forall\, m\ p\ v.\ [m,p] \vdash e1 \mapsto v \longrightarrow [m,p] \vdash e2 \mapsto v)$$

- $e2$ may be well formed in more contexts than $e1$, e.g. $x - x \geq 0$

Expression Semantics

$$\frac{\mathit{valid\text{-}value}\ (\mathit{constantAsStamp}\ c)\ c}{[m,p] \vdash \mathit{ConstantExpr}\ c \mapsto c} \tag{6}$$

$$\frac{i < |p| \qquad \mathit{valid\text{-}value}\ s\ p_{[i]}}{[m,p] \vdash \mathit{ParameterExpr}\ i\ s \mapsto p_{[i]}} \tag{7}$$

$$\frac{[m,p] \vdash xe \mapsto v \qquad \mathit{unary\text{-}eval}\ op\ v \neq \mathit{UndefVal}}{[m,p] \vdash \mathit{UnaryExpr}\ op\ xe \mapsto \mathit{unary\text{-}eval}\ op\ v} \tag{8}$$

$$\frac{[m,p] \vdash xe \mapsto x \qquad [m,p] \vdash ye \mapsto y \qquad \mathit{bin\text{-}eval}\ op\ x\ y \neq \mathit{UndefVal}}{[m,p] \vdash \mathit{BinaryExpr}\ op\ xe\ ye \mapsto \mathit{bin\text{-}eval}\ op\ x\ y} \tag{9}$$

$$\frac{\mathit{val} = m\ n \qquad \mathit{valid\text{-}value}\ s\ \mathit{val}}{[m,p] \vdash \mathit{LeafExpr}\ n\ s \mapsto \mathit{val}} \tag{10}$$

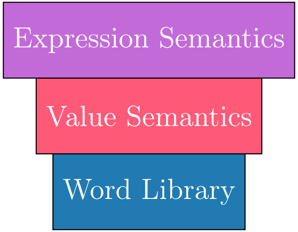
Value Semantics

$$
\begin{array}{l}
\textbf{fun } \textit{intval-add} :: \textit{Value} \Rightarrow \textit{Value} \Rightarrow \textit{Value} \textbf{ where}\\
\quad \textit{intval-add } (\textit{IntVal32 } v1)\ (\textit{IntVal32 } v2) = (\textit{IntVal32 } (v1+v2))\ |\\
\quad \textit{intval-add } (\textit{IntVal64 } v1)\ (\textit{IntVal64 } v2) = (\textit{IntVal64 } (v1+v2))\ |\\
\quad \textit{intval-add } \text{-}\ \text{-} = \textit{UndefVal}
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun } \textit{intval-xor} :: \textit{Value} \Rightarrow \textit{Value} \Rightarrow \textit{Value} \textbf{ where}\\
\quad \textit{intval-xor } (\textit{IntVal32 } v1)\ (\textit{IntVal32 } v2) = (\textit{IntVal32 } (v1\ \textit{XOR}\ v2))\ |\\
\quad \textit{intval-xor } (\textit{IntVal64 } v1)\ (\textit{IntVal64 } v2) = (\textit{IntVal64 } (v1\ \textit{XOR}\ v2))\ |\\
\quad \textit{intval-xor } \text{-}\ \text{-} = \textit{UndefVal}
\end{array}
$$

$$
\begin{array}{l}
\textbf{fun } \textit{bin-eval} :: \textit{IRBinaryOp} \Rightarrow \textit{Value} \Rightarrow \textit{Value} \Rightarrow \textit{Value} \textbf{ where}\\
\quad \textit{bin-eval BinAdd } v1\ v2 = \textit{intval-add } v1\ v2\ |\\
\quad \textit{bin-eval BinXor } v1\ v2 = \textit{intval-xor } v1\ v2\ |\\
\ldots
\end{array}
$$

Our Proof Approach



With those definitions

We can start proving!

In summary

- enables more *concise specification* of optimization rules,

In summary

- enables more *concise specification* of optimization rules,
- *reduces the proof burden* by a factor of 5 (by line count), while proving a significantly stronger property, and

In summary

- enables more *concise specification* of optimization rules,
- *reduces the proof burden* by a factor of 5 (by line count), while proving a significantly stronger property, and
- includes *termination* as a proof-obligation.

Future work

- Expand on *proof automation*.

Future work

- Expand on *proof automation*.
- Extend the DSL to *factor out common side conditions*.

Future work

- Expand on *proof automation*.
- Extend the DSL to *factor out common side conditions*.
- Resume work *control-flow optimizations*.

Thank you!

Any questions?

References

[Click and Cooper, 1995] Click, C. and Cooper, K. D. (1995). Combining analyses, combining optimizations. *TOPLAS*, 17(2):181–196.

[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 *ERTS 2016*, 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.