Verifying Compiler Optimisation Passes

Brae Webb

Supervised by:
Ian Hayes
Mark Utting

Thanks to:
Kristian Thomassen
Ethan Bulmer
Kausthubram Rajesh

Roadmap

Roadmap

- Motivation

Roadmap

Motivation — Plan

Roadmap

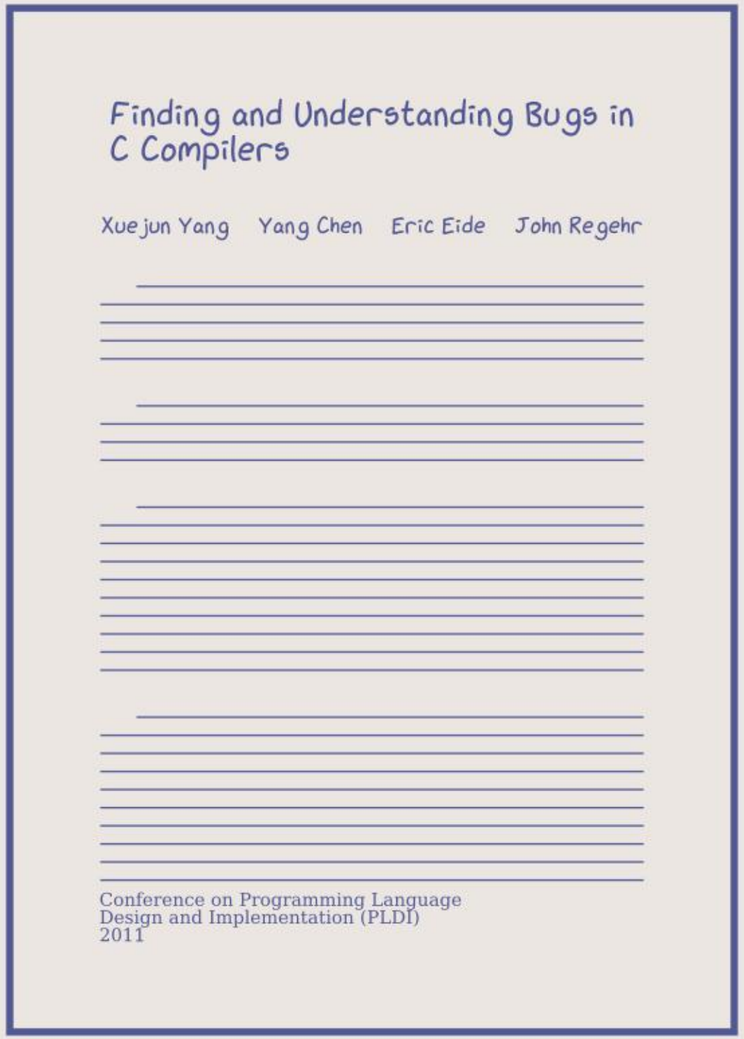
Roadmap

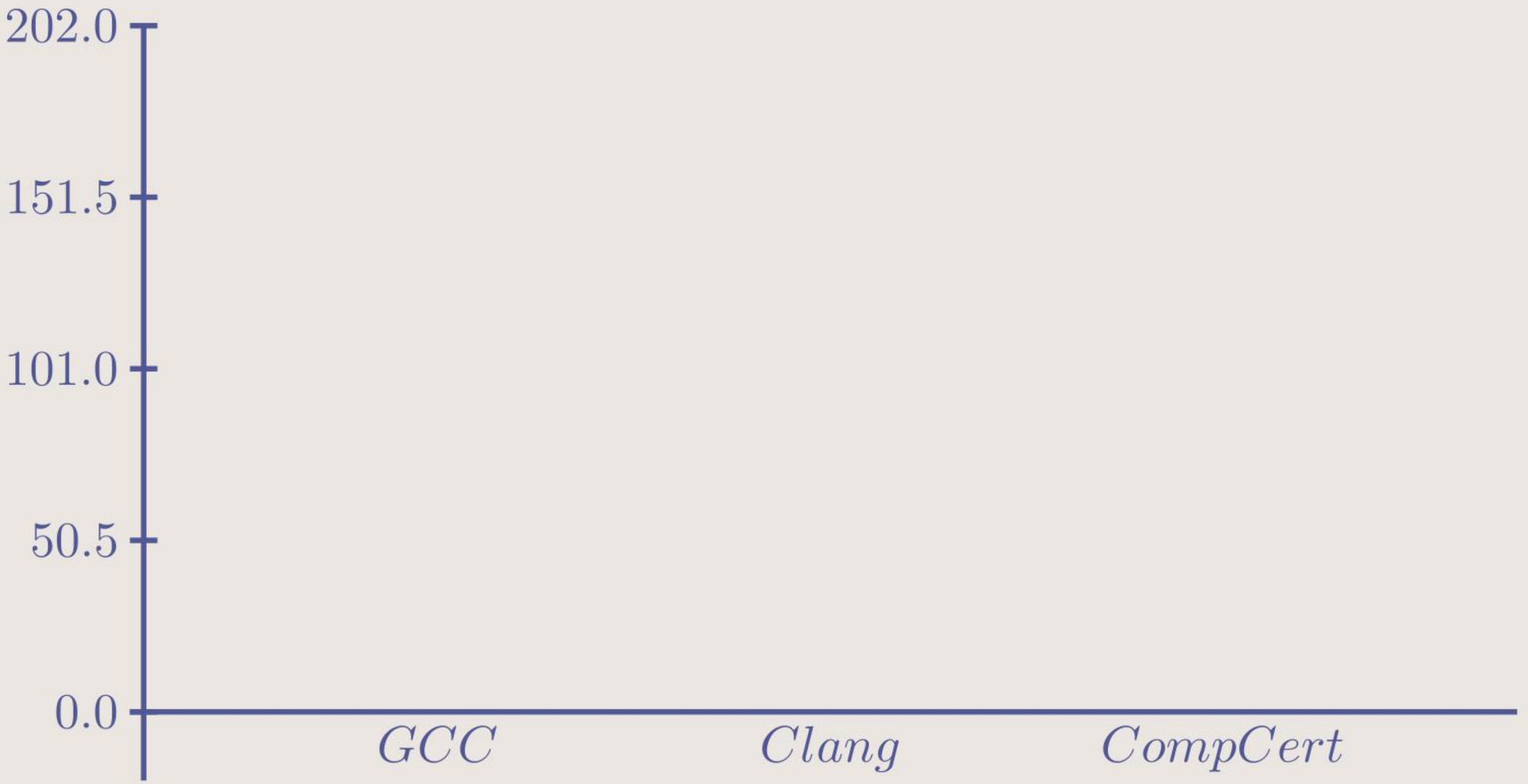
Motivation

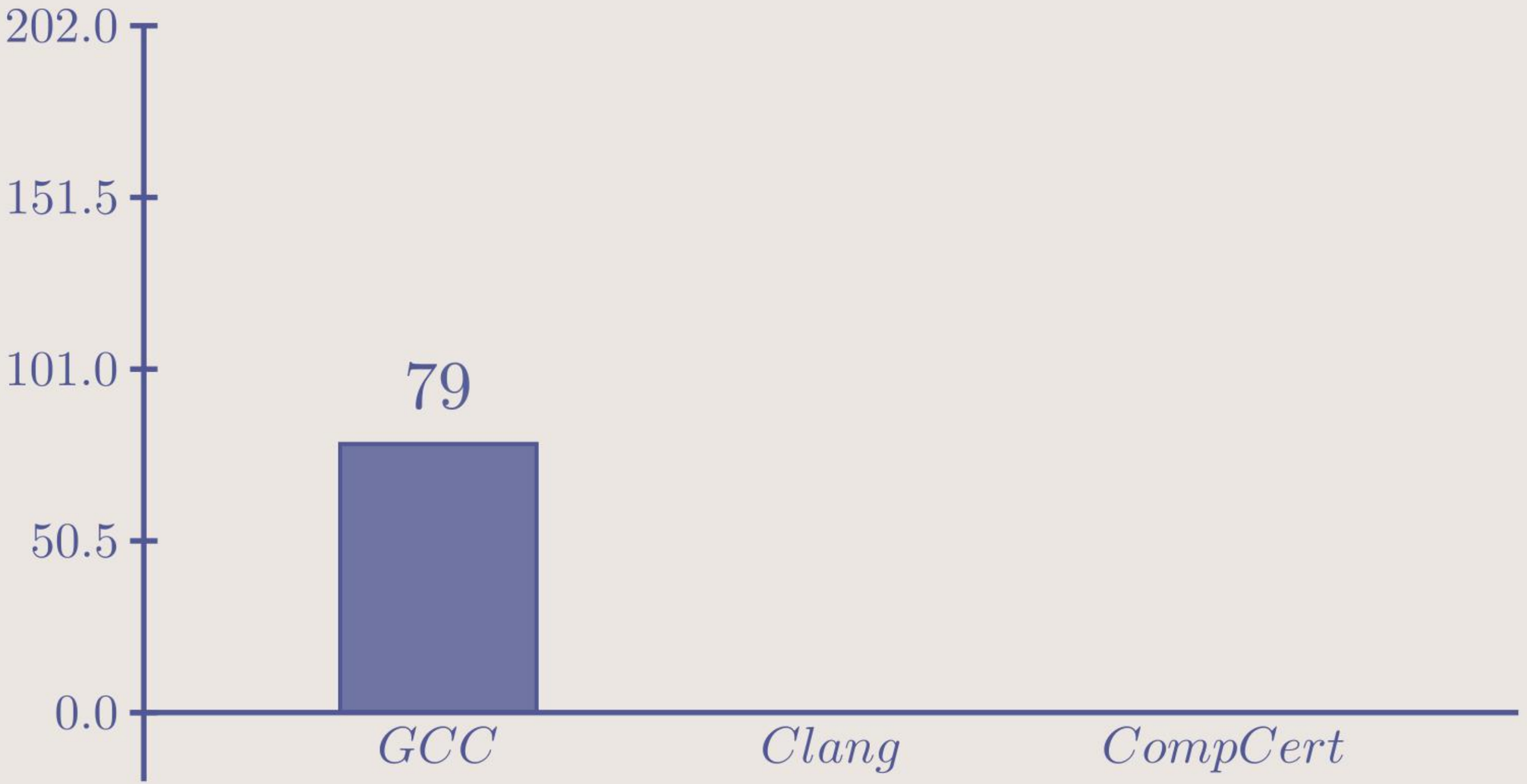
Verifying Compiler Optimization Passes

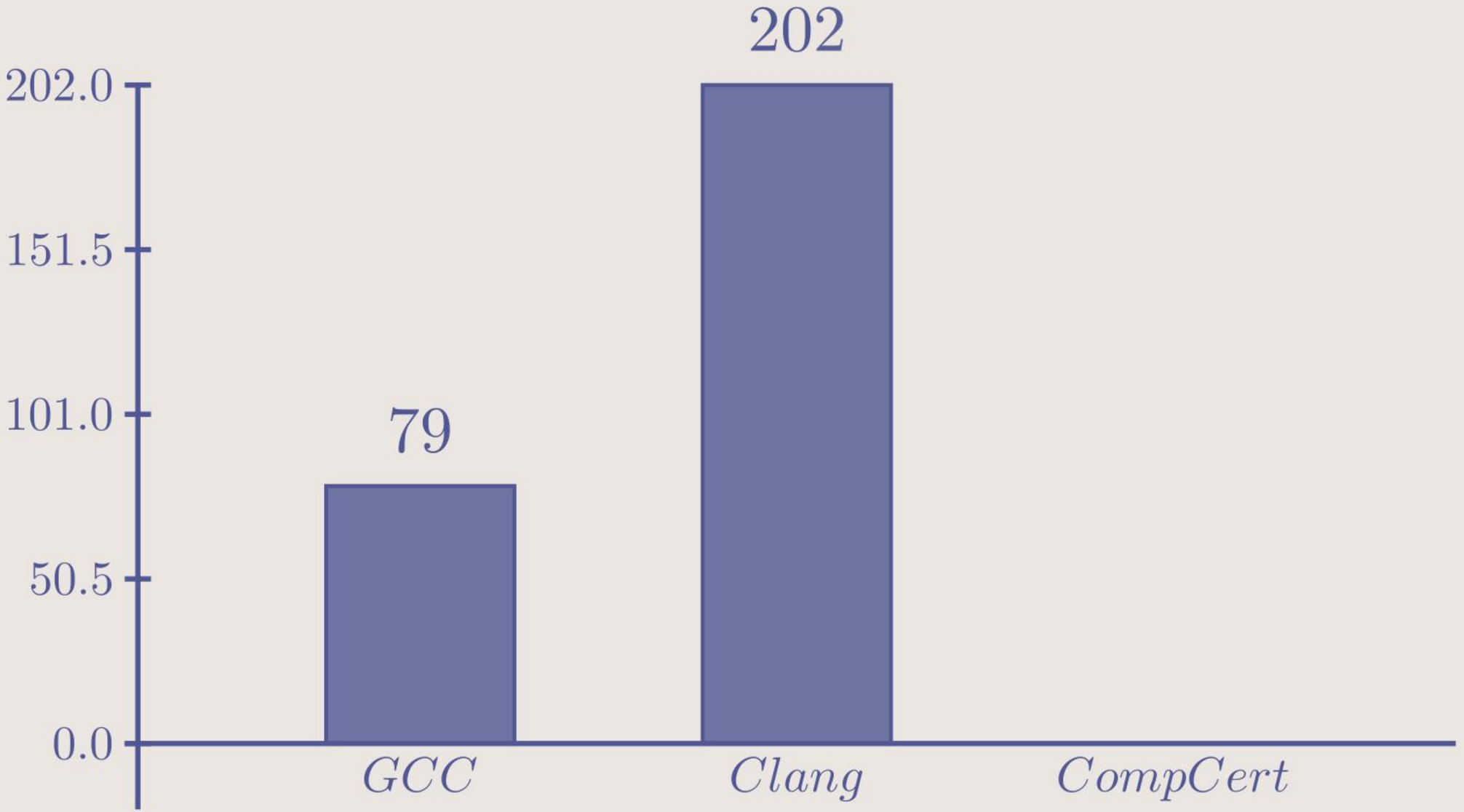
Verifying GraalVM Compiler Optimization Passes

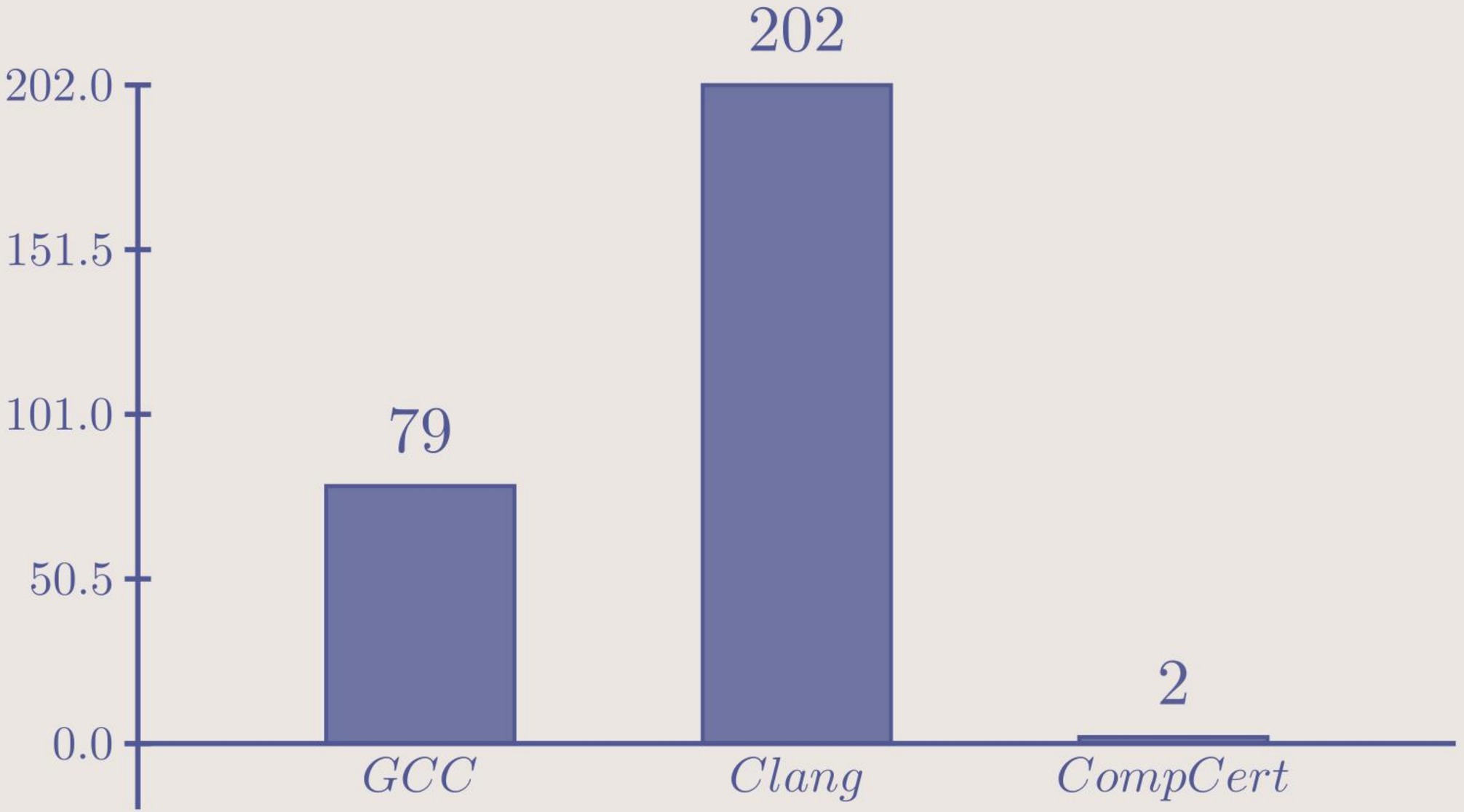
Verifying Compilers

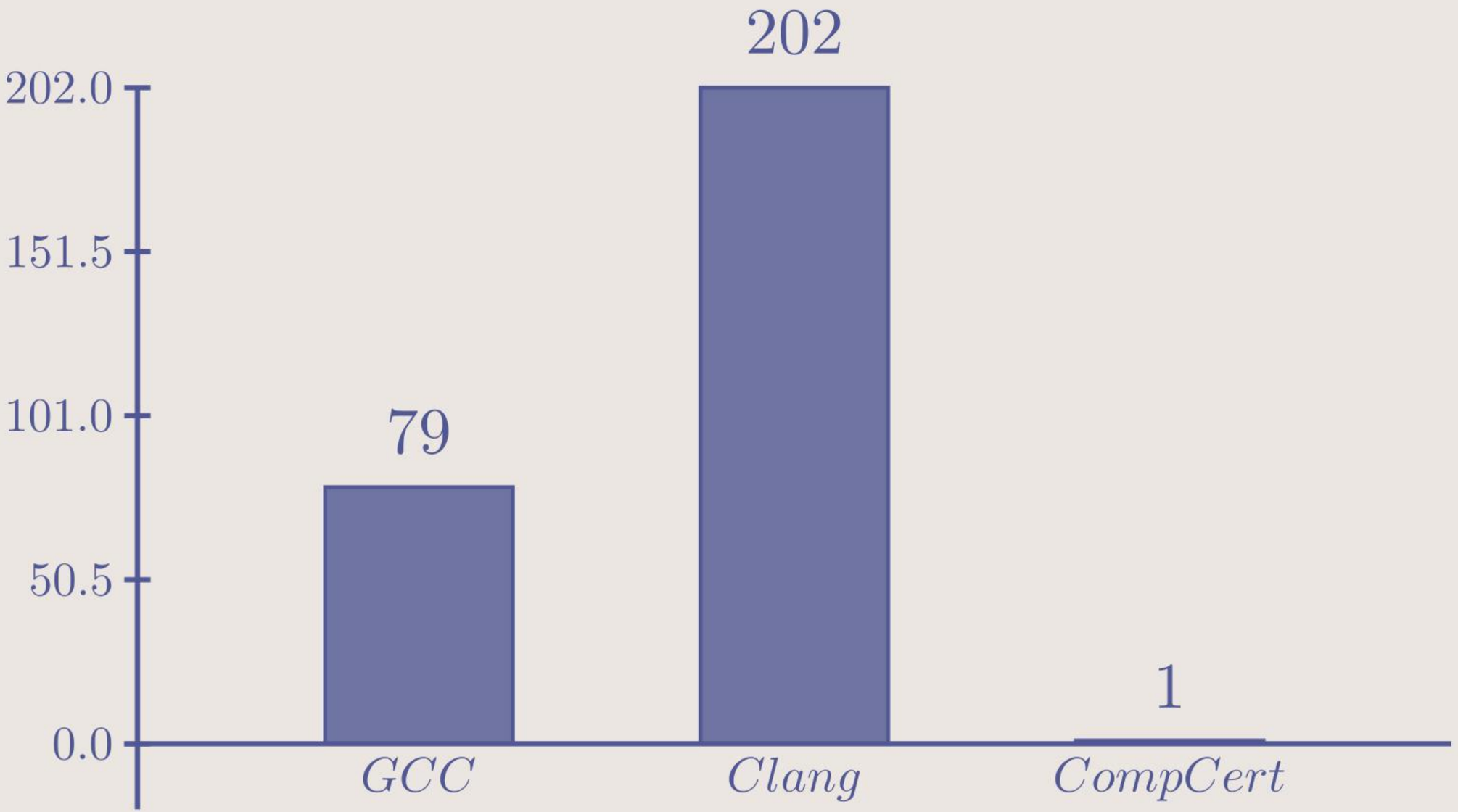

Finding and Understanding Bugs in C Compilers
Xuejun Yang Yang Chen Eric Eide John Regehr
Conference on Programming Language Design and Implementation (PLDI) 2011

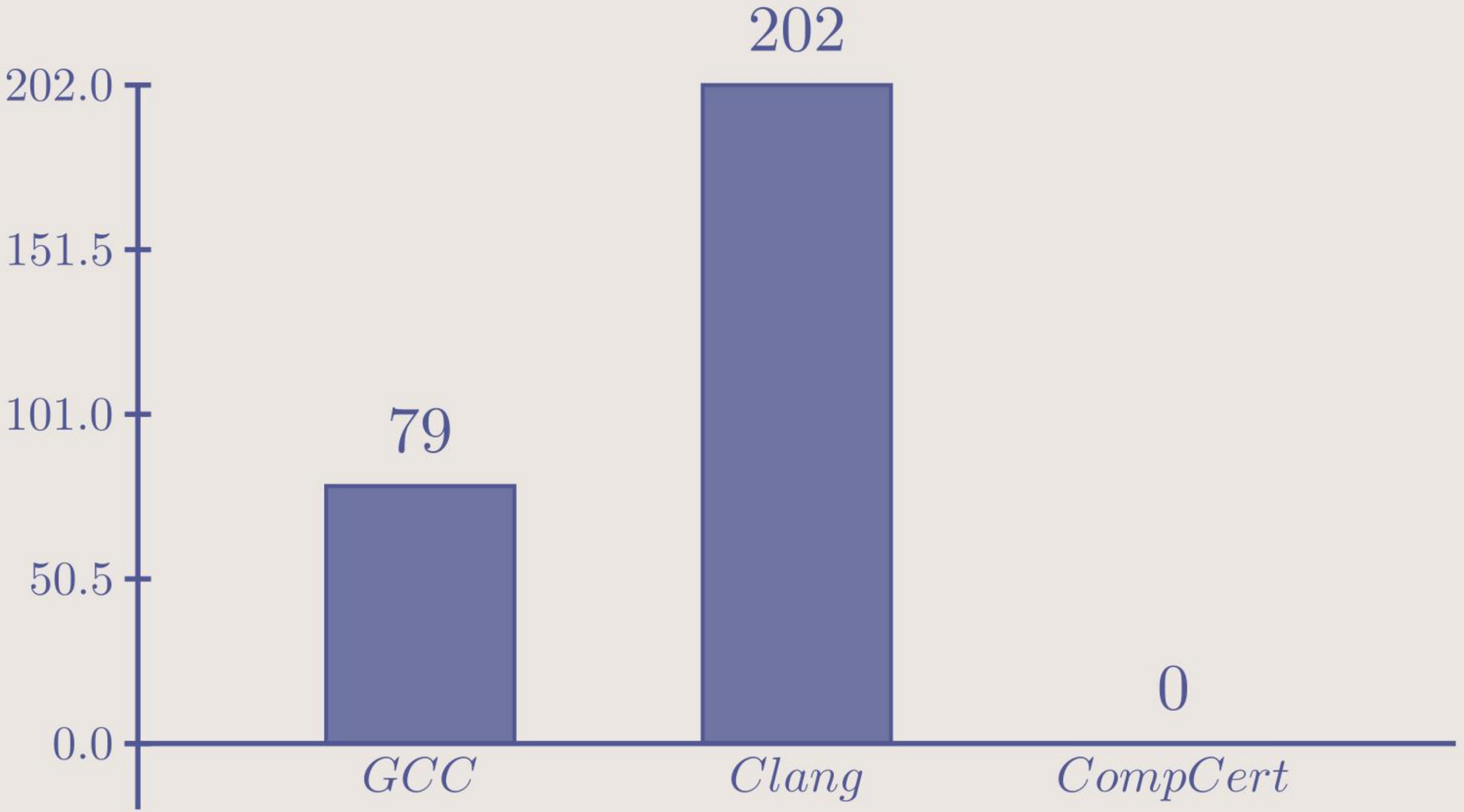

202.0
151.5
101.0
50.5
0.0
GCC
Clang
CompCert


202.0
151.5
101.0
50.5
0.0
79
GCC
Clang
CompCert


202
202.0
151.5
101.0
79
50.5
0.0
GCC
Clang
CompCert

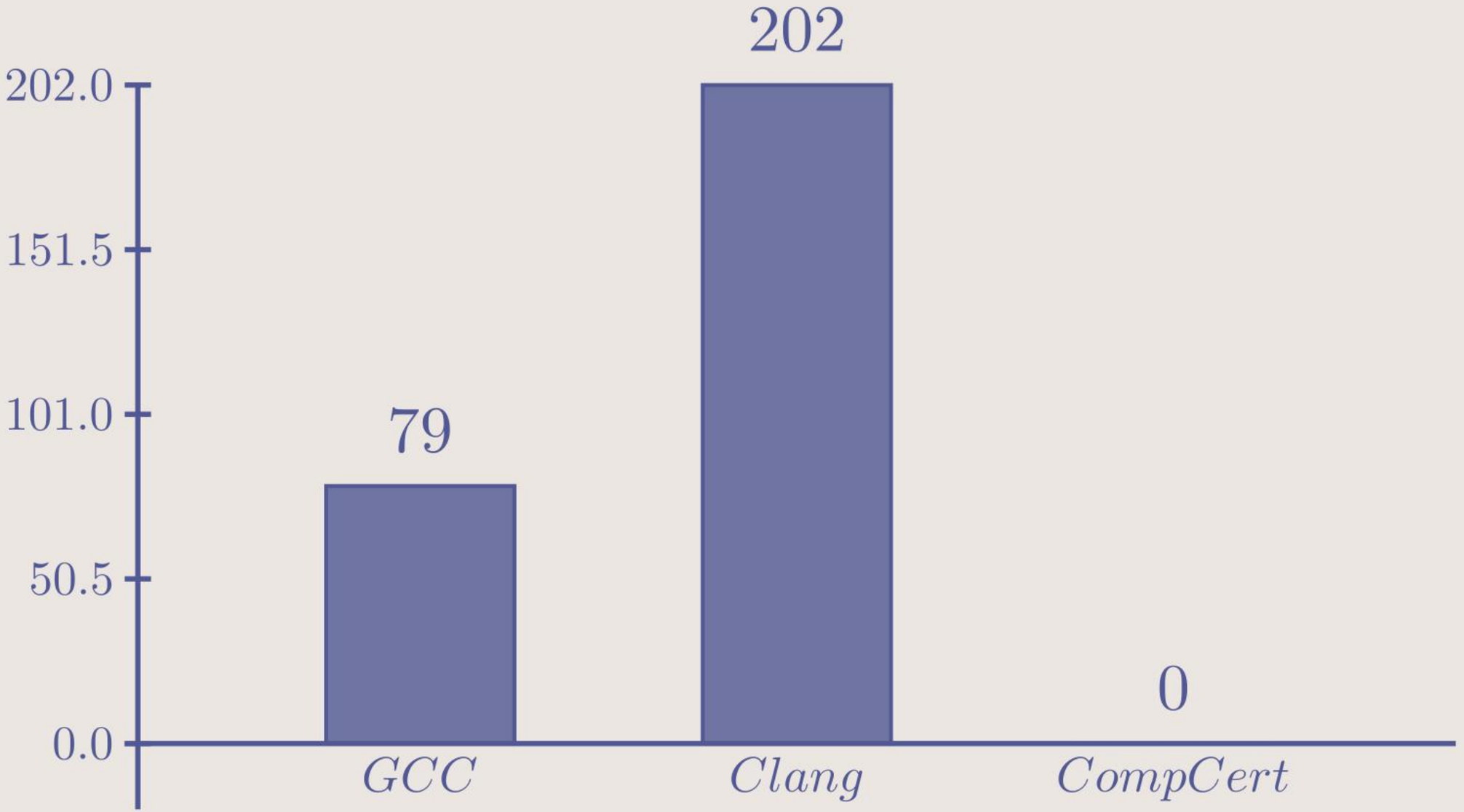

202
202.0
151.5
101.0
79
50.5
0.0
2
GCC
Clang
CompCert

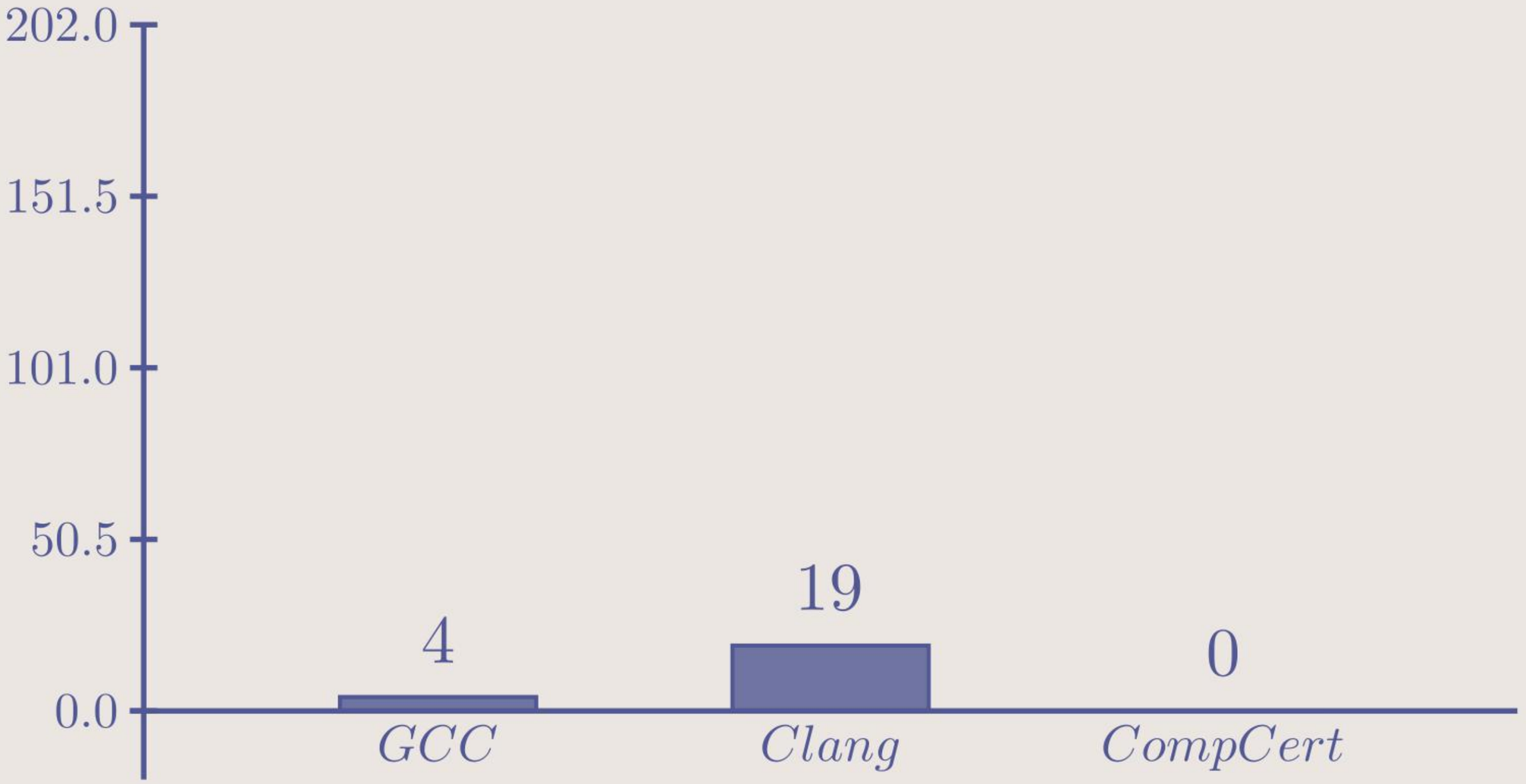

202
202.0
151.5
101.0
79
50.5
0.0
1
GCC
Clang
CompCert


202
202.0
151.5
101.0
79
50.5
0.0
0
GCC
Clang
CompCert

Verifying Compilers

Verifying Compiler Optimization Passes


202
202.0
151.5
101.0
79
50.5
0.0
0
GCC
Clang
CompCert


202.0
151.5
101.0
50.5
0.0
4
19
0
GCC
Clang
CompCert

Optimizing compilers are so difficult to get right
that we dare say that no optimizing compiler
is completely error-free!

"Optimizing compilers are so difficult to get right that we dare say that no optimizing compiler is completely error-free!"

- Aho, Lam, Sethi, Ullman in Compilers: Principles, Techniques, & Tools 2nd Edition

```
boolean isLessThan(Integer a, Integer b) {
    if (a == b) {
        return false;
    }

    if (a < b) {
        return true;
    }

    if (a > b) {
        return false;
    }

    return true;
}
```

```
boolean isLessThan(Integer a, Integer b) {
    if (a == b) {
        return false;
    }

    if (a < b) {
        return true;
    }

    if (a > b) {
        return false;
    }

    return true;
}
```

```
boolean isLessThan(Integer a, Integer b) {
    return a < b;
}
```

```
boolean isLessThan(Integer a, Integer b) {
    if (a == b) {
        return false;
    }

    if (a < b) {
        return true;
    }

    if (a > b) {
        return false;
    }

    return true;
}
```

```
boolean isLessThan(Integer a, Integer b) {
    return a < b;
}
```

```
assert lessThan(0, 0) == lessThanOpt(0, 0)
assert lessThan(0, 1) == lessThanOpt(0, 1)
assert lessThan(1, 0) == lessThanOpt(1, 0)
assert lessThan(-1, 100) == lessThanOpt(-1, 100)
assert lessThan(50, 50) == lessThanOpt(50, 50)
```

```
boolean isLessThan(Integer a, Integer b) {
    if (a == b) {
        return false;
    }

    if (a < b) {
        return true;
    }

    if (a > b) {
        return false;
    }

    return true;
}
```

```
boolean isLessThan(Integer a, Integer b) {
    return a < b;
}
```

```
assert lessThan(0, 0) == lessThanOpt(0, 0)
assert lessThan(0, 1) == lessThanOpt(0, 1)
assert lessThan(1, 0) == lessThanOpt(1, 0)
assert lessThan(-1, 100) == lessThanOpt(-1, 100)
assert lessThan(50, 50) == lessThanOpt(50, 50)
```

```
boolean isLessThan(Integer a, Integer b) {
    if (a == b) {
        return false;
    }

    if (a < b) {
        return true;
    }

    if (a > b) {
        return false;
    }

    return true;
}
```

```
boolean isLessThan(Integer a, Integer b) {
    return a < b;
}
```

```
assert lessThan(0, 0) == lessThanOpt(0, 0)
assert lessThan(0, 1) == lessThanOpt(0, 1)
assert lessThan(1, 0) == lessThanOpt(1, 0)
assert lessThan(-1, 100) == lessThanOpt(-1, 100)
assert lessThan(50, 50) == lessThanOpt(50, 50)
assert lessThan(2222, 2222) == lessThanOpt(2222, 2222)
```

Verifying Compiler Optimization Passes

Verifying GraalVM Compiler Optimization Passes

```
1  a + b - 0;
2  a - c;
```

```
1  a + b - 0;
2  a - c;
```

```
1   a + b - 0;
2   a - c;
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
1   a + b - 0;
2   a - c;
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    for (line : program.getExpressions()) {
        int value = line.execute(symtab);
        print(value);
    }
}
```

```
1   a + b - 0;
2   a - c;
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    for (line : program.getExpressions()) {
        int value = line.execute(symtab);
        print(value);
    }
}
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

    int execute(SymTable symtab) {
        return value;
    }
}

class VariableNode extends Expression {
    String identifier;

    int execute(SymTable symtab) {
        symtab.lookup(identifier);
    }
}
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    for (line : program.getExpressions()) {
        int value = line.execute(symtab);
        print(value);
    }
}
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

    int execute(SymTable symtab) {
        return value;
    }
}

class VariableNode extends Expression {
    String identifier;

    int execute(SymTable symtab) {
        symtab.lookup(identifier);
    }
}
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    int value = program.getExpressions().get(0).execute(s
    print(value);
    int value = program.getExpressions().get(1).execute(s
    print(value);
}
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

    int execute(SymTable symtab) {
        return value;
    }
}

class VariableNode extends Expression {
    String identifier;

    int execute(SymTable symtab) {
        symtab.lookup(identifier);
    }
}
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    int value =
        program.getExpressions().get(0).left.execute(symtab) -
        program.getExpressions().get(0).right.execute(symtab);
    print(value);
    int value =
        program.getExpressions().get(1).left.execute(symtab) -
        program.getExpressions().get(1).right.execute(symtab);
    print(value);
}
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

    int execute(SymTable symtab) {
        return value;
```

```
s VariableNode extends Expression {
tring identifier;

t execute(SymTable symtab) {
  symtab.lookup(identifier);
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    int value =
        (program.getExpressions().get(0).left.left.execute(symtab) +
        program.getExpressions().get(0).left.right.execute(symtab)) -
        program.getExpressions().get(0).right.value;
    print(value);
    int value =
        symtable.lookup(program.getExpressions().get(1).left.identifier) -
        symtable.lookup(program.getExpressions().get(1).right.identifier);
    print(value);
}
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

(SymTable symtab) {
lue;

Node extends Expression {
ifier;

(SymTable symtab) {
ookup(identifier);
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

                                Table symtab) {
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    int value =
        (symtable.lookup(program.getExpressions().get(1).left.left.identifier) +
        symtable.lookup(program.getExpressions().get(1).left.right.identifier)) -
        0;
    print(value);
    int value =
        symtable.lookup("a") -
        symtable.lookup("c");
    print(value);
}
```

```
                                extends Expression {


                                Table symtab) {
                                (identifier);
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    int value =
        (symtable.lookup("a") +
        symtable.lookup("b")) -
        0;
    print(value);
    int value =
        symtable.lookup("a") -
        symtable.lookup("c");
    print(value);
}
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

    int execute(SymTable symtab) {
        return value;
    }
}

class VariableNode extends Expression {
    String identifier;

    int execute(SymTable symtab) {
        symtab.lookup(identifier);
    }
}
```

```
program = new Program(
    SubNode(
        AddNode(VariableNode('a'),
                VariableNode('b')),
        ConstantNode(0)
    ),
    SubNode(
        VariableNode('a'),
        VariableNode('c')
    )
)
```

```
void executeProgram(Program program,
                    SymTable symtab) {
    int value =
        symtable.lookup("a") +
        symtable.lookup("b");
    print(value);
    int value =
        symtable.lookup("a") -
        symtable.lookup("c");
    print(value);
}
```

```
class AddNode extends Expression {
    Expression left;
    Expression right;

    int execute(SymTable symtab) {
        return left.execute() + right.execute();
    }
}

class ConstantNode extends Expression {
    int value;

    int execute(SymTable symtab) {
        return value;
    }
}

class VariableNode extends Expression {
    String identifier;

    int execute(SymTable symtab) {
        symtab.lookup(identifier);
    }
}
```

Plan

Verification

[specification]

[proof]

Verification

Using

[definitions]

Show

[property]

[proof]

Using

[definitions]

Show

An optimization transformation
preserves the program semantics

[proof]

Using

Program Semantics

Show

An optimization transformation
preserves the program semantics

[proof]

Using

Program Semantics

Optimization Transformation Definition

Show

An optimization transformation
preserves the program semantics

[proof]

Using

Program Semantics

Optimization Transformation Definition

Semantic Preservation Definition

Show

An optimization transformation
preserves the program semantics

[proof]

Using

Program Semantics

Optimization Transformation Definition

Semantic Preservation Definition

```
x = 4
x = 10
```

Show

An optimization transformation
preserves the program semantics

[proof]

Using

IR Semantics

Optimization Transformation Definition

Semantic Preservation Definition

Show

An optimization transformation
preserves the IR semantics

```
x = 4
x = 10
```

[proof]

Plan - Implementation

IR Semantics

IR Semantics

Semantics
Preservation
Definition

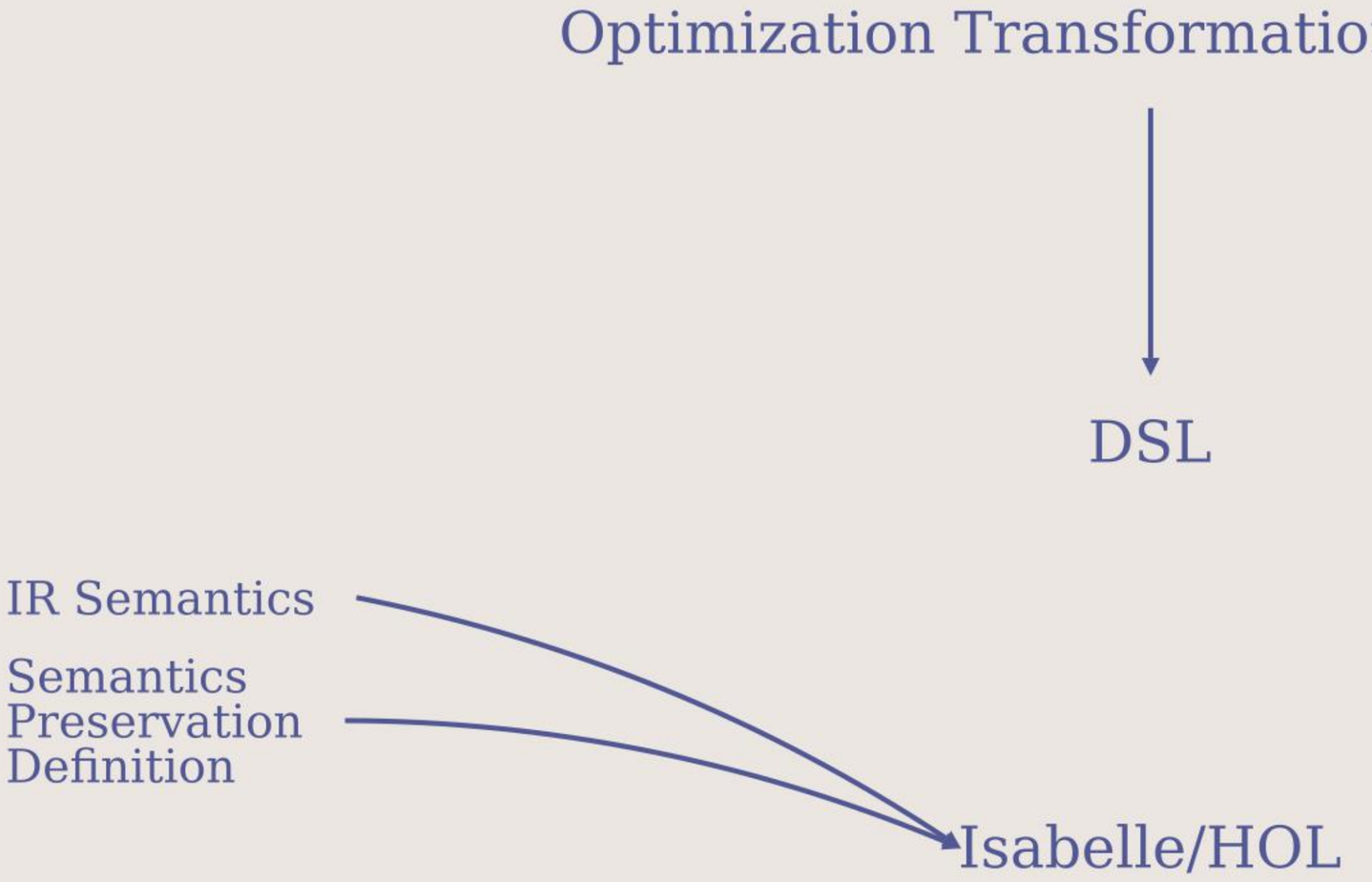
IR Semantics

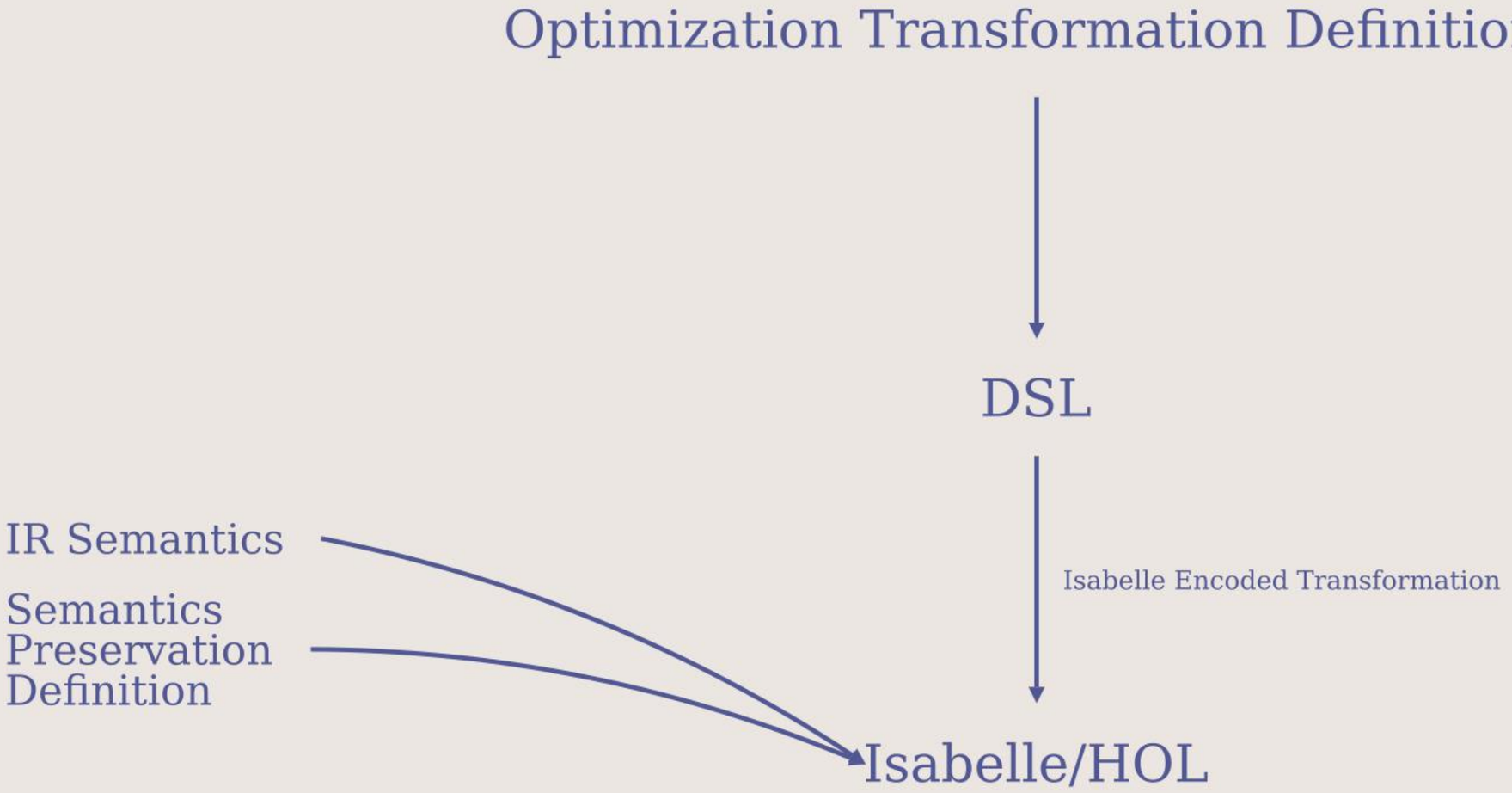
Semantics
Preservation
Definition

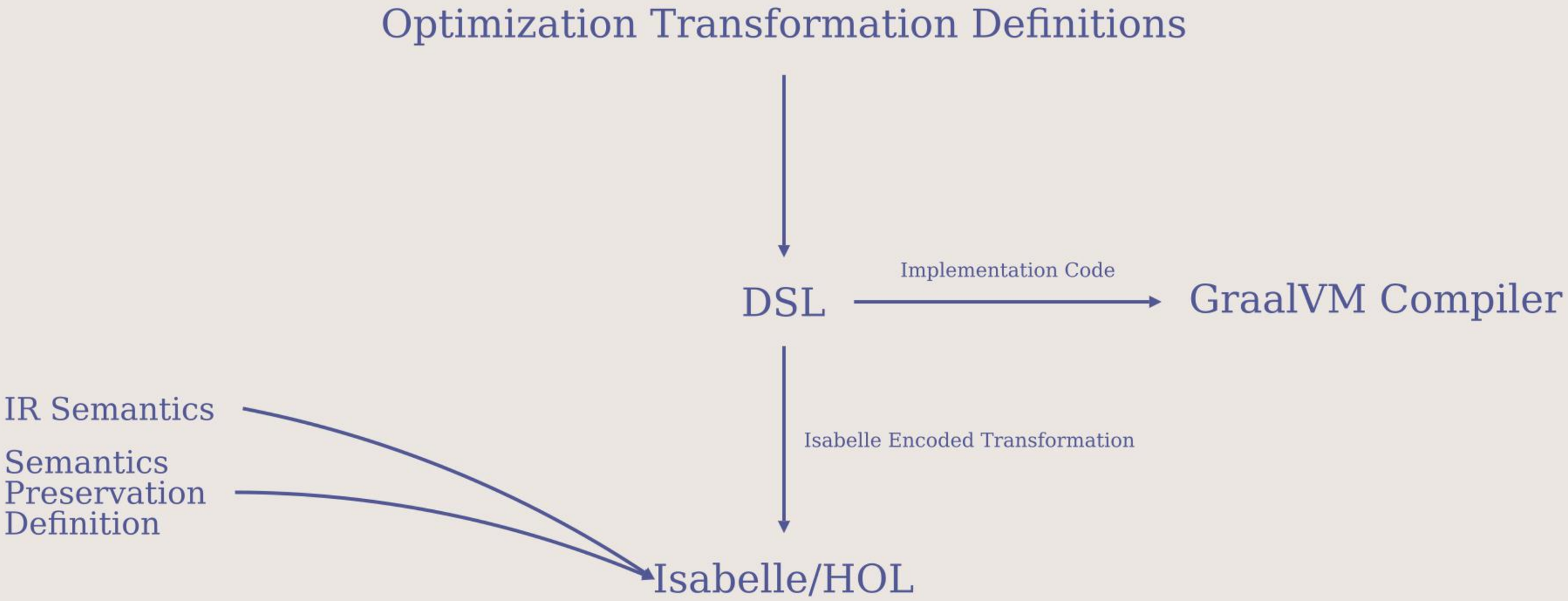
Isabelle/HOL

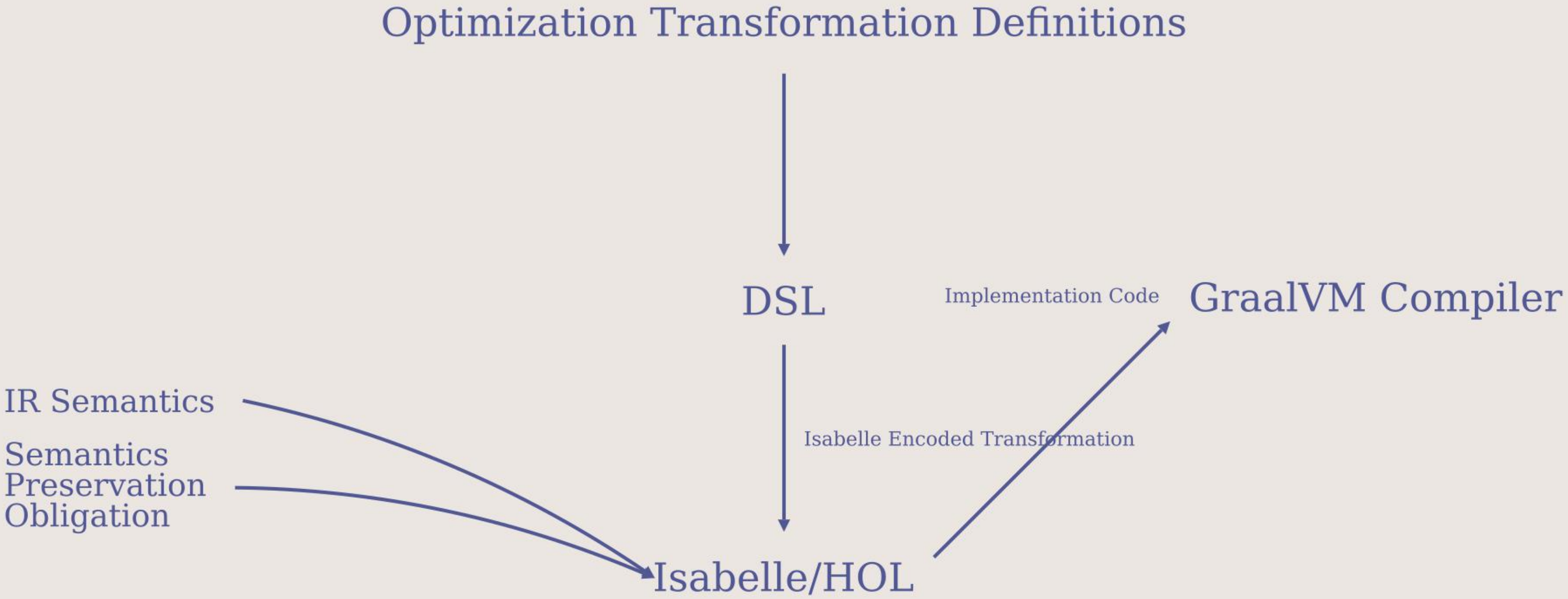

IR Semantics
Semantics
Preservation
Definition
Isabelle/HOL

Optimization Transformation Definitions

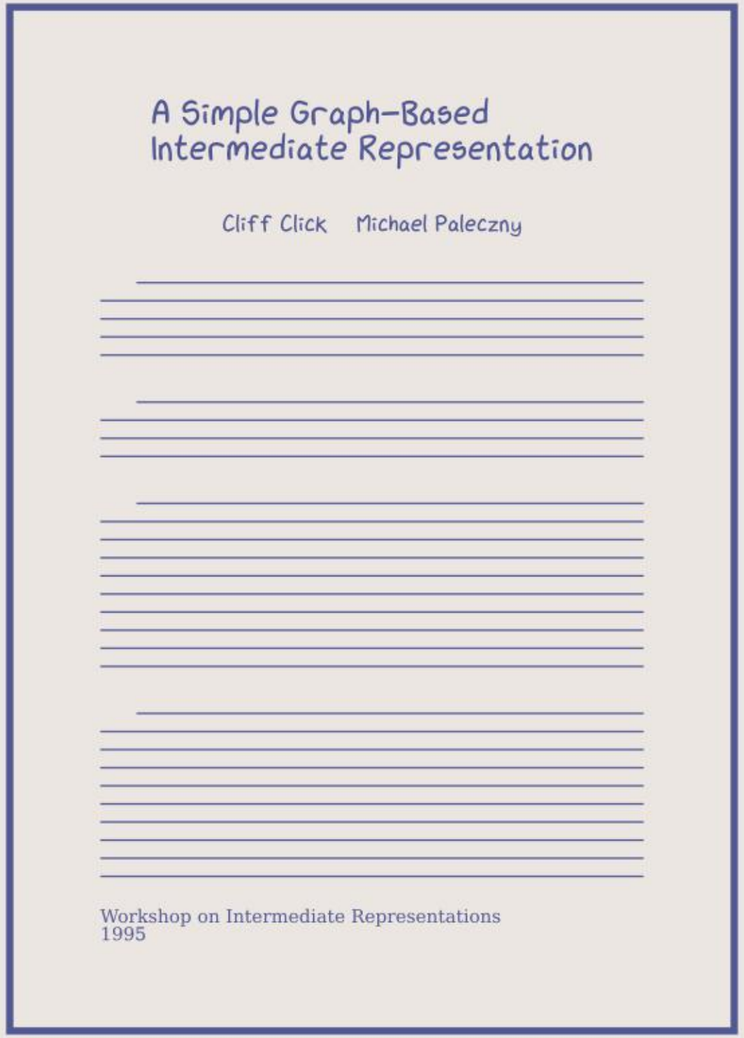

Optimization Transformation Definitions
DSL
IR Semantics
Semantics Preservation Definition
Isabelle/HOL


Optimization Transformation Definitions
DSL
Isabelle Encoded Transformation
IR Semantics
Semantics Preservation Definition
Isabelle/HOL


Optimization Transformation Definitions
DSL
Implementation Code
GraalVM Compiler
IR Semantics
Semantics Preservation Definition
Isabelle Encoded Transformation
Isabelle/HOL

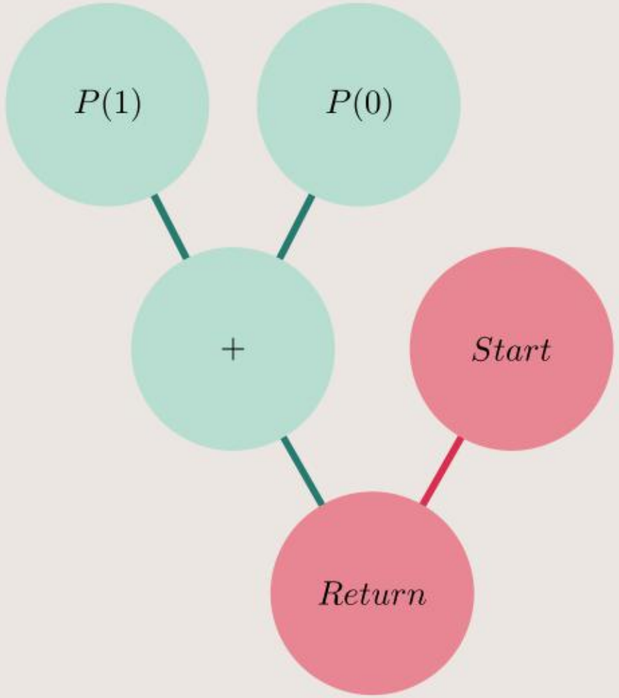

Optimization Transformation Definitions
DSL
Implementation Code
GraalVM Compiler
IR Semantics
Semantics Preservation Obligation
Isabelle Encoded Transformation
Isabelle/HOL

Progress


A Simple Graph-Based Intermediate Representation
Cliff Click Michael Paleczny
Workshop on Intermediate Representations
1995

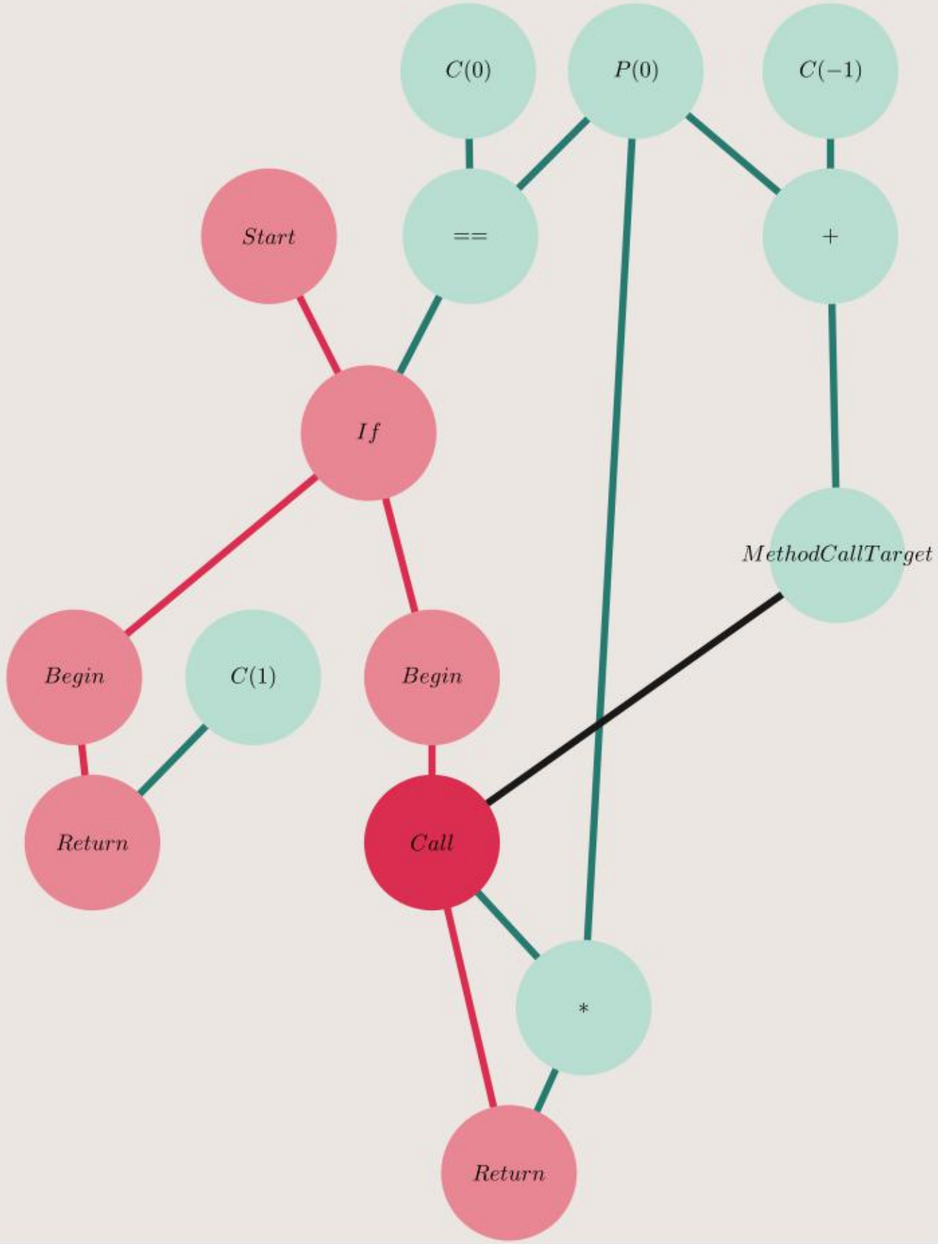
```
public static int add(int x, int y) {
   return x + y;
}
```

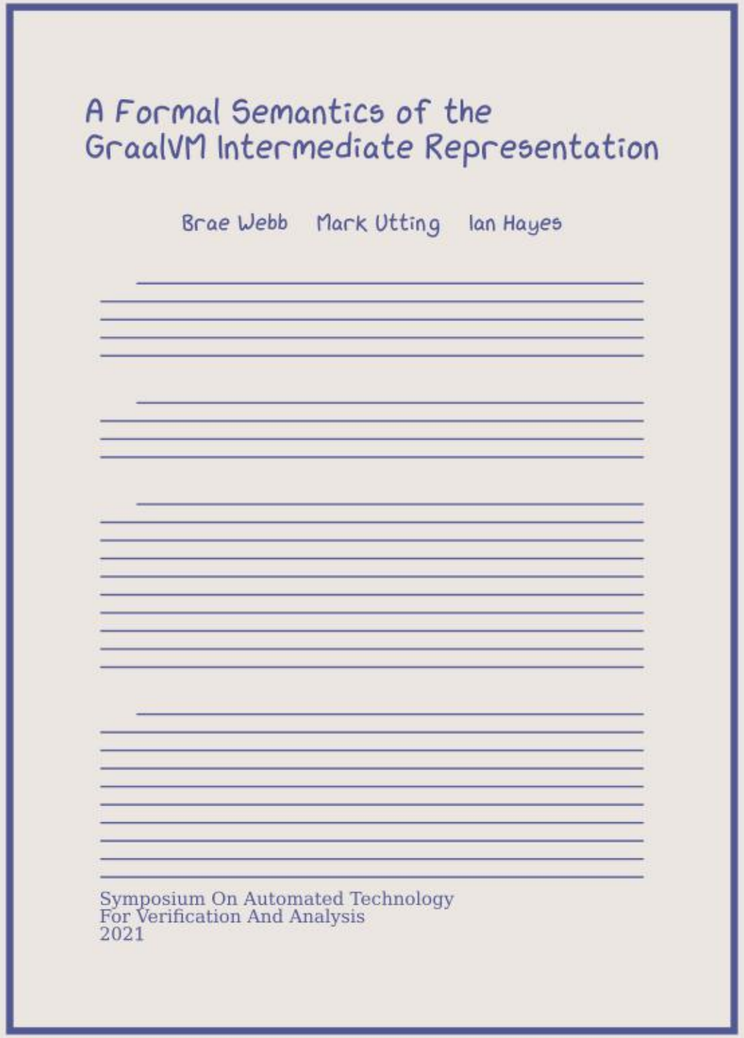
```
public static int add(int x, int y) {
    return x + y;
}
```



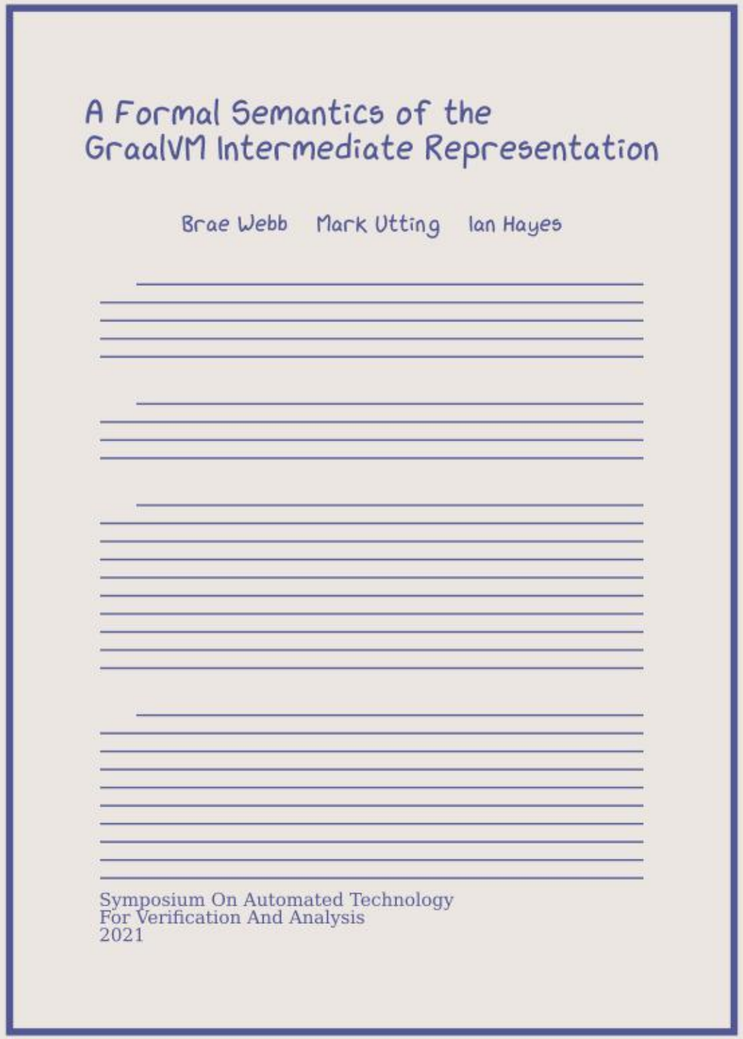
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

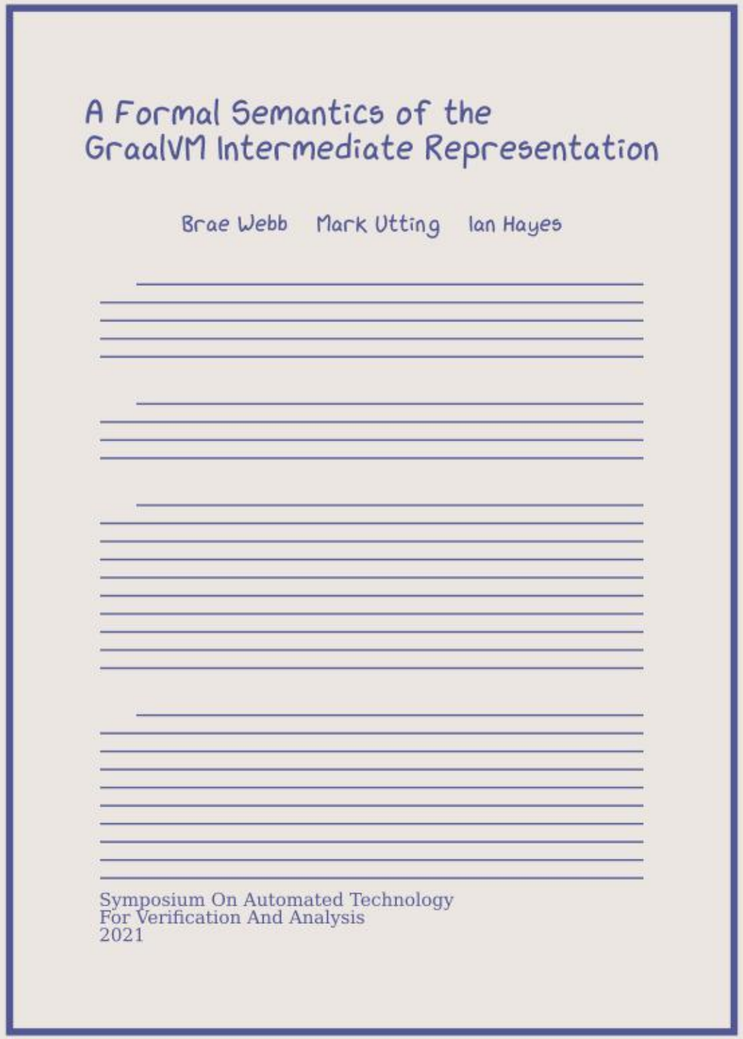


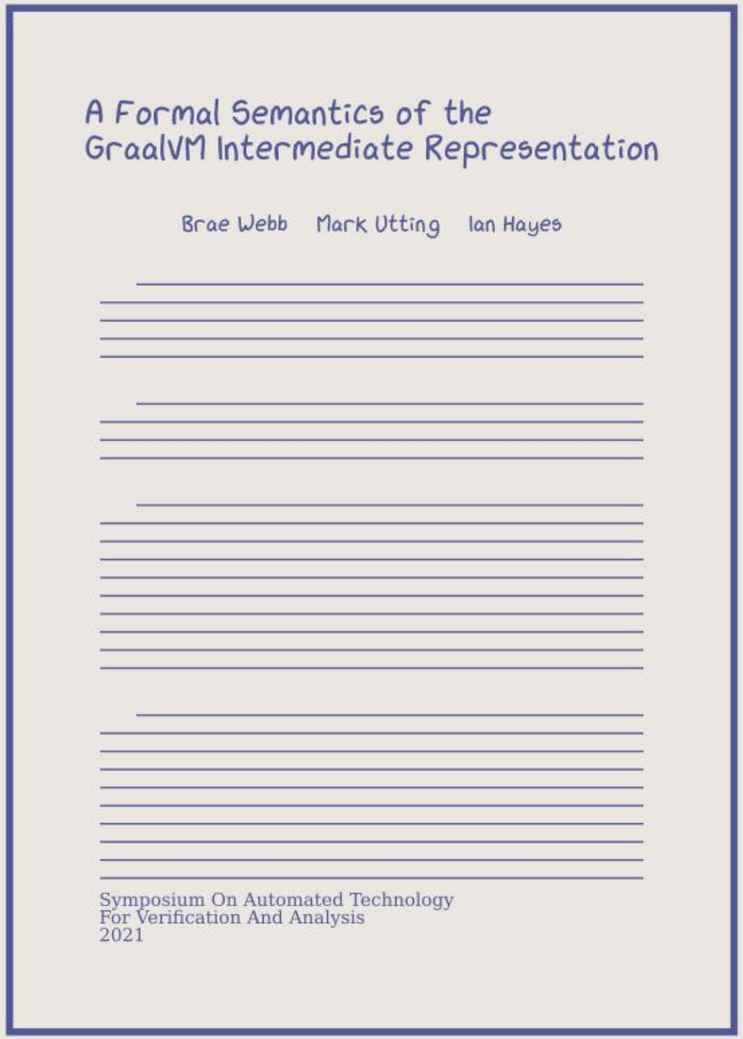

A Formal Semantics of the
GraalVM Intermediate Representation
Brae Webb Mark Utting Ian Hayes
Symposium On Automated Technology
For Verification And Analysis
2021

$$[g, m, p] \vdash n \mapsto v$$



$$[g, m, p] \vdash n \mapsto v \qquad [g, p] \vdash (nid, m, h) \rightarrow (nid', m', h')$$





$$[g,\ m,\ p] \vdash \mathit{ConstantNode}\ c \mapsto c$$

$$\frac{[g,\ m,\ p] \vdash g\langle\!\langle x \rangle\!\rangle \mapsto v1 \qquad [g,\ m,\ p] \vdash g\langle\!\langle y \rangle\!\rangle \mapsto v2}{[g,\ m,\ p] \vdash \mathit{AddNode}\ x\ y \mapsto v1 + v2}$$

$$[g,\ m,\ p] \vdash \mathit{ParameterNode}\ i \mapsto p_{[i]}$$

$$[g,\ m,\ p] \vdash \mathit{ValuePhiNode}\ nid\ uu\ uv \mapsto m\ nid$$

$$[g,\ m,\ p] \vdash \mathit{InvokeNode}\ nid\ vh\ vi\ vj\ vk\ vl \mapsto m\ nid$$

$$[g,\ m,\ p] \vdash \mathit{NewInstanceNode}\ nid\ vs\ vt\ vu \mapsto m\ nid$$

$$[g,\ m,\ p] \vdash \mathit{LoadFieldNode}\ nid\ vv\ vw\ vx \mapsto m\ nid$$

$$\frac{\mathit{is\text{-}sequential\text{-}node}\ g\langle\!\langle nid \rangle\!\rangle \qquad nid' = (\mathit{successors\text{-}of}\ g\langle\!\langle nid \rangle\!\rangle)_{[0]}}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m,\ h)}$$

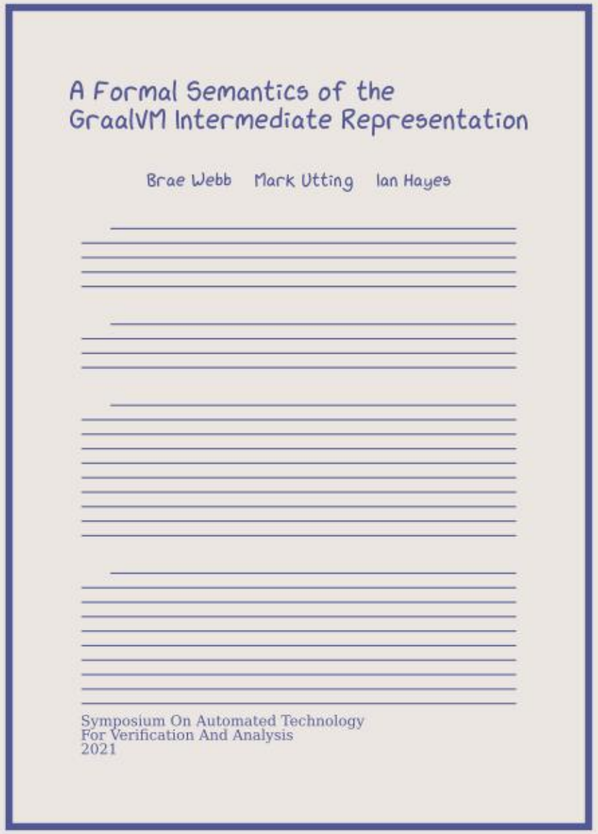
$$\frac{g\langle\!\langle nid \rangle\!\rangle = \mathit{IfNode}\ cond\ tb\ fb \qquad [g,\ m,\ p] \vdash g\langle\!\langle cond \rangle\!\rangle \mapsto val \qquad nid' = (\mathit{if}\ val\text{-}to\text{-}bool\ val\ \mathit{then}\ tb\ \mathit{else}\ fb)}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m,\ h)}$$

$$\frac{\begin{array}{c}\mathit{is\text{-}AbstractEndNode}\ g\langle\!\langle nid \rangle\!\rangle \\ merge = \mathit{any\text{-}usage}\ g\ nid \qquad \mathit{is\text{-}AbstractMergeNode}\ g\langle\!\langle merge \rangle\!\rangle \\ i = \mathit{find\text{-}index}\ nid\ (\mathit{inputs\text{-}of}\ g\langle\!\langle merge \rangle\!\rangle) \\ phis = \mathit{phi\text{-}list}\ g\ merge \qquad inps = \mathit{phi\text{-}inputs}\ g\ i\ phis \\ [g,\ m,\ p] \vdash inps \longmapsto vs \qquad m' = \mathit{set\text{-}phis}\ phis\ vs\ m\end{array}}{[g,\ p] \vdash (nid,\ m,\ h) \to (merge,\ m',\ h)}$$

$$\frac{g\langle\!\langle nid \rangle\!\rangle = \mathit{NewInstanceNode}\ nid\ class\ frame\ nid' \qquad (h',\ ref) = \mathit{h\text{-}new\text{-}inst}\ h \qquad m' = m(nid := ref)}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')}$$

$$\frac{g\langle\!\langle nid \rangle\!\rangle = \mathit{LoadFieldNode}\ nid\ f\ (\mathit{Some}\ obj)\ nid' \qquad [g,\ m,\ p] \vdash g\langle\!\langle obj \rangle\!\rangle \mapsto \mathit{ObjRef}\ ref \qquad \mathit{h\text{-}load\text{-}field}\ ref\ f\ h = v \qquad m' = m(nid := v)}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m',\ h)}$$

$$\frac{\begin{array}{c}g\langle\!\langle nid \rangle\!\rangle = \mathit{StoreFieldNode}\ nid\ f\ newval\ uu\ (\mathit{Some}\ obj)\ nid' \\ [g,\ m,\ p] \vdash g\langle\!\langle newval \rangle\!\rangle \mapsto val \\ [g,\ m,\ p] \vdash g\langle\!\langle obj \rangle\!\rangle \mapsto \mathit{ObjRef}\ ref \\ h' = \mathit{h\text{-}store\text{-}field}\ ref\ f\ val\ h \qquad m' = m(nid := val)\end{array}}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')}$$



$$[g,\ m,\ p] \vdash \mathit{ConstantNode}\ c \mapsto c$$

$$\frac{[g,\ m,\ p] \vdash g\langle\!\langle x \rangle\!\rangle \mapsto v1 \qquad [g,\ m,\ p] \vdash g\langle\!\langle y \rangle\!\rangle \mapsto v2}{[g,\ m,\ p] \vdash \mathit{AddNode}\ x\ y \mapsto v1\ +\ v2}$$

$$[g,\ m,\ p] \vdash \mathit{ParameterNode}\ i \mapsto p_{[i]}$$

$$[g,\ m,\ p] \vdash \mathit{ValuePhiNode}\ nid\ uu\ uv \mapsto m\ nid$$

$$[g,\ m,\ p] \vdash \mathit{InvokeNode}\ nid\ vh\ vi\ vj\ vk\ vl \mapsto m\ nid$$

$$[g,\ m,\ p] \vdash \mathit{NewInstanceNode}\ nid\ vs\ vt\ vu \mapsto m\ nid$$

$$[g,\ m,\ p] \vdash \mathit{LoadFieldNode}\ nid\ vv\ vw\ vx \mapsto m\ nid$$

$$\frac{\mathit{is\text{-}sequential\text{-}node}\ g\langle\!\langle nid \rangle\!\rangle \qquad nid' = (\mathit{successors\text{-}of}\ g\langle\!\langle nid \rangle\!\rangle)_{[0]}}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m,\ h)}$$

$$\frac{g\langle\!\langle nid \rangle\!\rangle = \mathit{IfNode}\ cond\ tb\ fb \qquad [g,\ m,\ p] \vdash g\langle\!\langle cond \rangle\!\rangle \mapsto val \qquad nid' = (\mathit{if}\ val\text{-}to\text{-}bool\ val\ \mathit{then}\ tb\ \mathit{else}\ fb)}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m,\ h)}$$

$$\frac{\begin{array}{c}\mathit{is\text{-}AbstractEndNode}\ g\langle\!\langle nid \rangle\!\rangle \\ merge = \mathit{any\text{-}usage}\ g\ nid \qquad \mathit{is\text{-}AbstractMergeNode}\ g\langle\!\langle merge \rangle\!\rangle \\ i = \mathit{find\text{-}index}\ nid\ (\mathit{inputs\text{-}of}\ g\langle\!\langle merge \rangle\!\rangle) \\ phis = \mathit{phi\text{-}list}\ g\ merge \qquad inps = \mathit{phi\text{-}inputs}\ g\ i\ phis \\ [g,\ m,\ p] \vdash inps \longmapsto vs \qquad m' = \mathit{set\text{-}phis}\ phis\ vs\ m\end{array}}{[g,\ p] \vdash (nid,\ m,\ h) \to (merge,\ m',\ h)}$$

$$\frac{g\langle\!\langle nid \rangle\!\rangle = \mathit{NewInstanceNode}\ nid\ class\ frame\ nid' \qquad (h',\ ref) = \mathit{h\text{-}new\text{-}inst}\ h \qquad m' = m(nid := ref)}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')}$$

$$\frac{g\langle\!\langle nid \rangle\!\rangle = \mathit{LoadFieldNode}\ nid\ f\ (\mathit{Some}\ obj)\ nid' \qquad [g,\ m,\ p] \vdash g\langle\!\langle obj \rangle\!\rangle \mapsto \mathit{ObjRef}\ ref \qquad \mathit{h\text{-}load\text{-}field}\ ref\ f\ h = v \qquad m' = m(nid := v)}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m',\ h)}$$

$$\frac{\begin{array}{c}g\langle\!\langle nid \rangle\!\rangle = \mathit{StoreFieldNode}\ nid\ f\ newval\ uu\ (\mathit{Some}\ obj)\ nid' \\ [g,\ m,\ p] \vdash g\langle\!\langle newval \rangle\!\rangle \mapsto val \\ [g,\ m,\ p] \vdash g\langle\!\langle obj \rangle\!\rangle \mapsto \mathit{ObjRef}\ ref \\ h' = \mathit{h\text{-}store\text{-}field}\ ref\ f\ val\ h \qquad m' = m(nid := val)\end{array}}{[g,\ p] \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')}$$

ATVA 2021

$$x + y - y \rightarrow x$$

$$x + y - y \rightarrow x$$

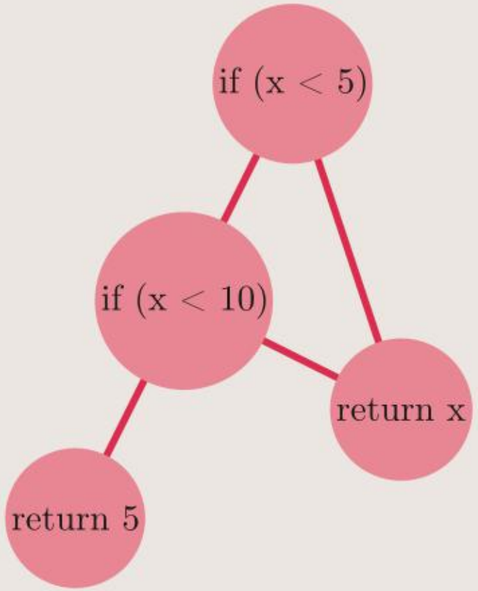
$$\forall s.(\text{eval } (node_1, s) = v) \leftrightarrow (\text{eval } (node_2, s) = v)$$

```
if (x < 5) {
    if (x < 10) {
        return 5;
    }
}
return x;
```

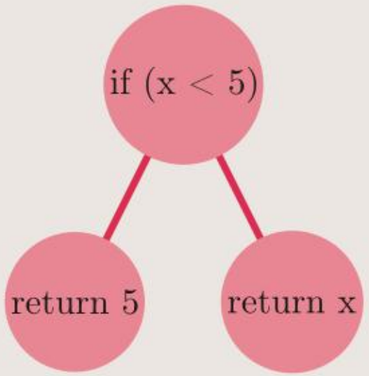
```
if (x < 5) {
    if (x < 10) {
        return 5;
    }
}
return x;
```

```
if (x < 5) {
    return 5;
}
return x;
```

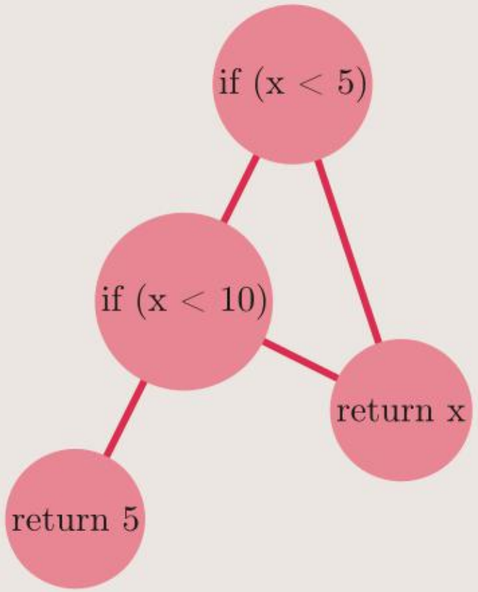
```
if (x < 5) {
    if (x < 10) {
        return 5;
    }
}
return x;
```



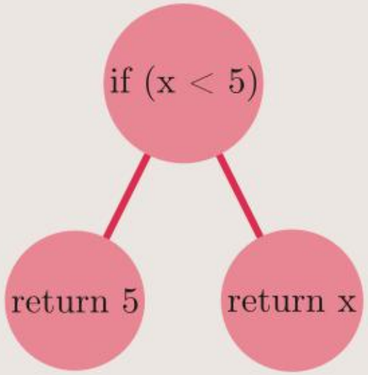

```
if (x < 5) {
    return 5;
}
return x;
```



```
if (x < 5) {
    if (x < 10) {
        return 5;
    }
}
return x;
```

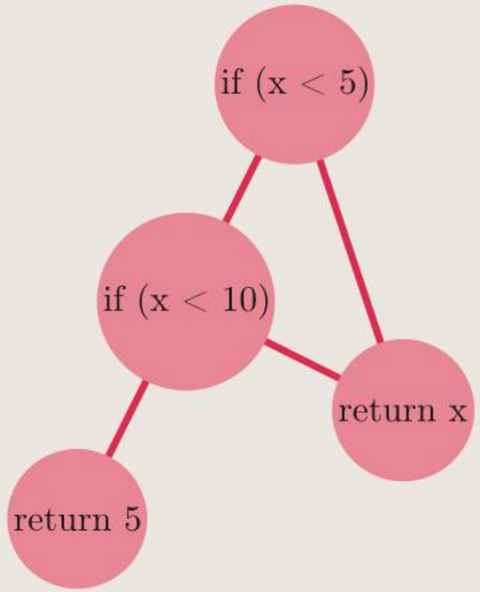



```
if (x < 5) {
    return 5;
}
return x;
```

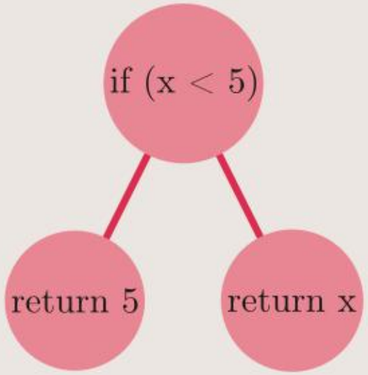


$$P \sim Q \leftrightarrow ((\forall P'.P \rightsquigarrow P' \rightarrow (\exists Q'.Q \rightsquigarrow Q' \wedge P' \sim Q'))$$
$$\wedge(\forall Q'.Q \rightsquigarrow Q' \rightarrow (\exists P'.P \rightsquigarrow P' \wedge P' \sim Q'))$$

```
if (x < 5) {
    if (x < 10) {
        return 5;
    }
}
return x;
```



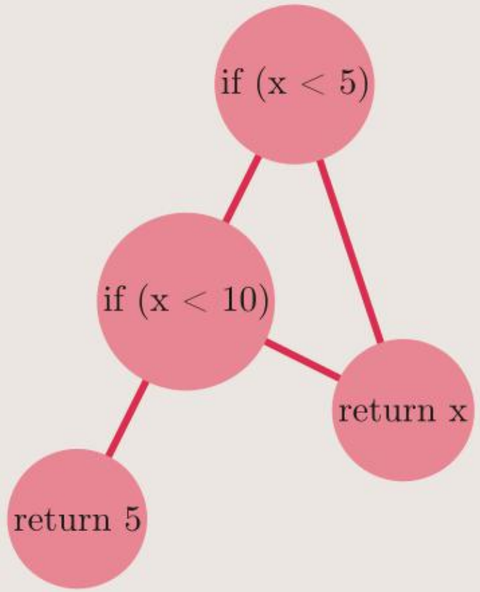

```
if (x < 5) {
    return 5;
}
return x;
```



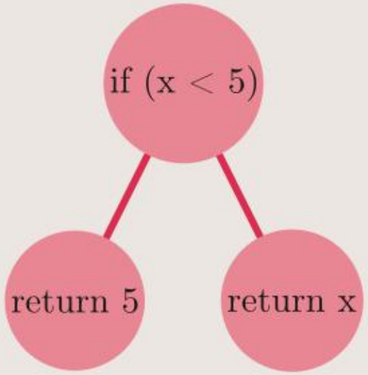
$$P \sim Q \leftrightarrow ((\forall P'.P \rightsquigarrow P' \rightarrow (\exists Q'.Q \rightsquigarrow Q' \wedge P' \sim Q')) \wedge (\forall Q'.Q \rightsquigarrow Q' \rightarrow (\exists P'.P \rightsquigarrow P' \wedge P' \sim Q'))$$

$$\frac{\forall\, P'.\ [g,\ p] \vdash (nid,\ m,\ h) \rightarrow P' \longrightarrow (\exists\, Q'.\ [g',\ p] \vdash (nid,\ m,\ h) \rightarrow Q' \wedge P' = Q') \quad \forall\, Q'.\ [g',\ p] \vdash (nid,\ m,\ h) \rightarrow Q' \longrightarrow (\exists\, P'.\ [g,\ p] \vdash (nid,\ m,\ h) \rightarrow P' \wedge P' = Q')}{nid\ m\ h \mid g \sim g'}$$

```
if (x < 5) {
    if (x < 10) {
        return 5;
    }
}
return x;
```




```
if (x < 5) {
    return 5;
}
return x;
```



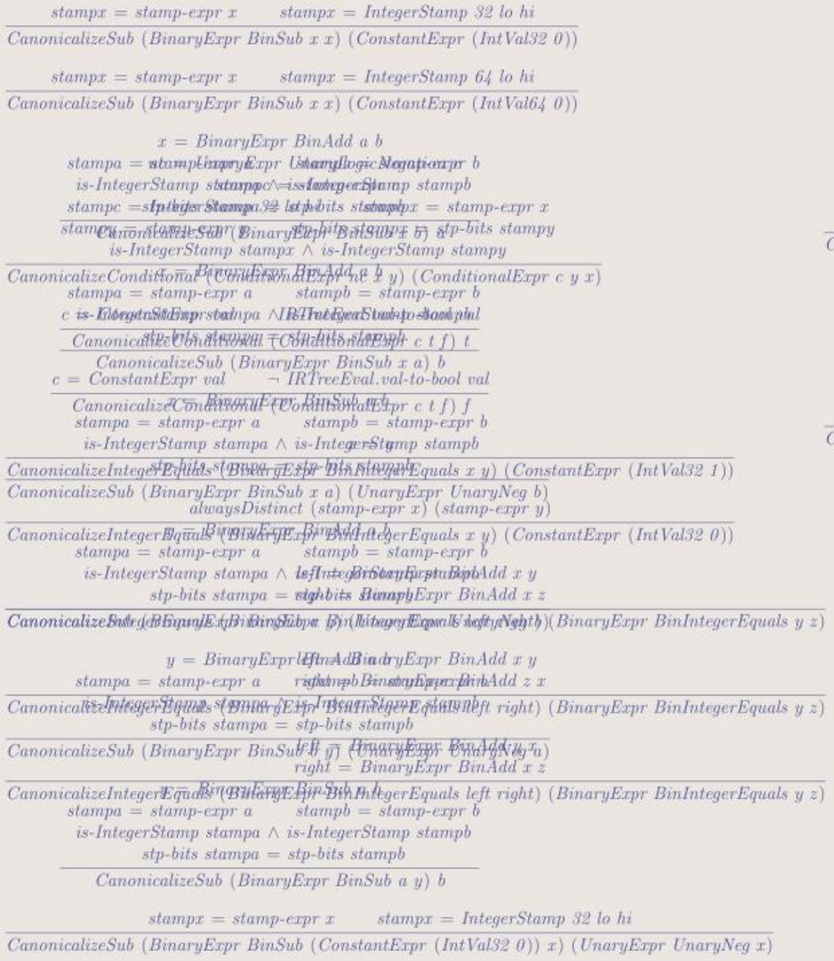
$$P \sim Q \leftrightarrow ((\forall P'.P \rightsquigarrow P' \rightarrow (\exists Q'.Q \rightsquigarrow Q' \wedge P' \sim Q')) \wedge (\forall Q'.Q \rightsquigarrow Q' \rightarrow (\exists P'.P \rightsquigarrow P' \wedge P' \sim Q'))$$

$$\frac{\forall\, P'.\ [g,\ p] \vdash (nid,\ m,\ h) \rightarrow P' \longrightarrow (\exists\, Q'.\ [g',\ p] \vdash (nid,\ m,\ h) \rightarrow Q' \wedge P' = Q') \quad \forall\, Q'.\ [g',\ p] \vdash (nid,\ m,\ h) \rightarrow Q' \longrightarrow (\exists\, P'.\ [g,\ p] \vdash (nid,\ m,\ h) \rightarrow P' \wedge P' = Q')}{nid\ m\ h \mid g \sim g'}$$

$$\frac{\forall\, P'.\ [g,\ m,\ p,\ h] \vdash nid \rightsquigarrow P' \longrightarrow (\exists\, Q'.\ [g',\ m,\ p,\ h] \vdash nid \rightsquigarrow Q' \wedge P' = Q') \quad \forall\, Q'.\ [g',\ m,\ p,\ h] \vdash nid \rightsquigarrow Q' \longrightarrow (\exists\, P'.\ [g,\ m,\ p,\ h] \vdash nid \rightsquigarrow P' \wedge P' = Q')}{nid\ .\ g \sim g'}$$

Canonicalization Phase

Canonicalization Phase



Conditional Elimination Phase

Conditional Elimination Phase

$$\frac{always Distinct\ (stamps\ x)\ (stamps\ y)}{tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ KnownFalse}$$

$$\frac{neverDistinct\ (stamps\ x)\ (stamps\ y)}{tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ KnownTrue}$$

$$\frac{is\text{-}IntegerStamp\ (stamps\ x) \qquad is\text{-}IntegerStamp\ (stamps\ y) \qquad stpi\text{-}upper\ (stamps\ x) < stpi\text{-}lower\ (stamps\ y)}{tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ KnownTrue}$$

$$\frac{is\text{-}IntegerStamp\ (stamps\ x) \qquad is\text{-}IntegerStamp\ (stamps\ y) \qquad stpi\text{-}upper\ (stamps\ y) \leq stpi\text{-}lower\ (stamps\ x)}{tryFold\ (IntegerLessThanNode\ x\ y)\ stamps\ KnownFalse}$$

$$\frac{g \vdash IntegerEqualsNode\ x\ y\ \&\ IntegerLessThanNode\ x\ y \mapsto KnownFalse}{ConditionalEliminationStep\ (set\ conds)\ (hdOr\ flow\ (stamp\ g))\ g\ nid\ g'}$$

$$\frac{g \vdash IntegerEqualsNode\ x\ y\ \&\ IntegerLessThanNode\ y\ x \mapsto KnownFalse}{ConditionalEliminationStep\ (set\ conds)\ (hdOr\ flow\ (stamp\ g))\ g\ nid\ g'}$$

$$\frac{g \vdash IntegerLessThanNode\ x\ y\ \&\ IntegerLessThanNode\ y\ x \mapsto KnownFalse}{ConditionalEliminationStep\ (set\ conds)\ (hdOr\ flow\ (stamp\ g))\ g\ nid\ g'}$$

$$\frac{g\langle nid\rangle = BeginNode\ nid' \qquad nid \notin seen \qquad seen' = \{nid\} \cup seen \qquad g \vdash IntegerLessThanNode\ x\ y\ \&\ IntegerEqualsNode\ x\ y \mapsto KnownFalse \qquad g\langle ifcond\rangle = IfNode\ cond\ t\ f \qquad s = find\text{-}index\ nid\ (successors\text{-}of\ g\langle ifcond\rangle) \qquad c = (\text{if } i = 0 \text{ then } g\langle cond\rangle \text{ else } LogicNegationNode\ cond) \qquad g \vdash x\ \&\ x \mapsto KnownTrue \qquad conds' = c \cdot conds \qquad flow' = registerNewCondition\ g\ c\ (hdOr\ flow\ (stamp\ g))}{ConditionalEliminationStep\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow'))\ g\ nid\ g'}$$

$$\frac{g \vdash x\ \&\ y \mapsto KnownTrue}{ConditionalEliminationStep\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow' \cdot flow))\ g\ nid\ g'}$$

$$\frac{g \vdash x\ \&\ LogicNegationNode\ y \mapsto KnownFalse}{ConditionalEliminationStep\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow' \cdot flow))\ g\ nid\ g'}$$

$$\frac{g\langle nid\rangle = EndNode \qquad nid' = any\text{-}usage\ g\ nid \qquad nid \notin seen \qquad seen' = \{nid\} \cup seen \qquad conds' = tl\ conds \qquad flow' = tl\ flow}{ConditionalEliminationStep\ (nid, seen, conds, flow)\ g\ (Some\ (nid', seen', conds', flow'))}$$

$$\frac{ConditionalEliminationPhase\ g\ (nid', seen', conds', flow')\ g'}{ConditionalEliminationPhase\ g\ (nid, seen, conds, flow)\ g'}\ ConditionalElimination.Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds', flow'))$$

$$\frac{\neg\ is\text{-}EndNode\ g\langle nid\rangle \qquad is\text{-}BeginNode\ g\langle nid\rangle \qquad nid \notin seen \qquad seen' = \{nid\} \cup seen \qquad Some\ nid' = ConditionalElimination.nextEdge\ seen'\ nid\ g}{ConditionalElimination.Step\ g\ (nid, seen, conds, flow)\ (Some\ (nid', seen', conds, flow))}$$

$$\frac{ConditionalElimination.Step\ g\ (nid, seen, conds, flow)\ None \qquad Some\ nid' = ConditionalElimination.pred\ g\ nid \qquad seen' = \{nid\} \cup seen}{ConditionalEliminationPhase\ g\ (nid, seen, conds, flow)\ g'}\ ConditionalEliminationPhase\ g\ (nid', seen', conds, flow)\ g'$$

$$\frac{\neg\ is\text{-}EndNode\ g\langle nid\rangle \qquad is\text{-}BeginNode\ g\langle nid\rangle \qquad nid \notin seen \qquad seen' = \{nid\} \cup seen \qquad None = ConditionalElimination.nextEdge\ seen'\ nid\ g}{ConditionalElimination.Step\ g\ (nid, seen, conds, flow)\ None}$$

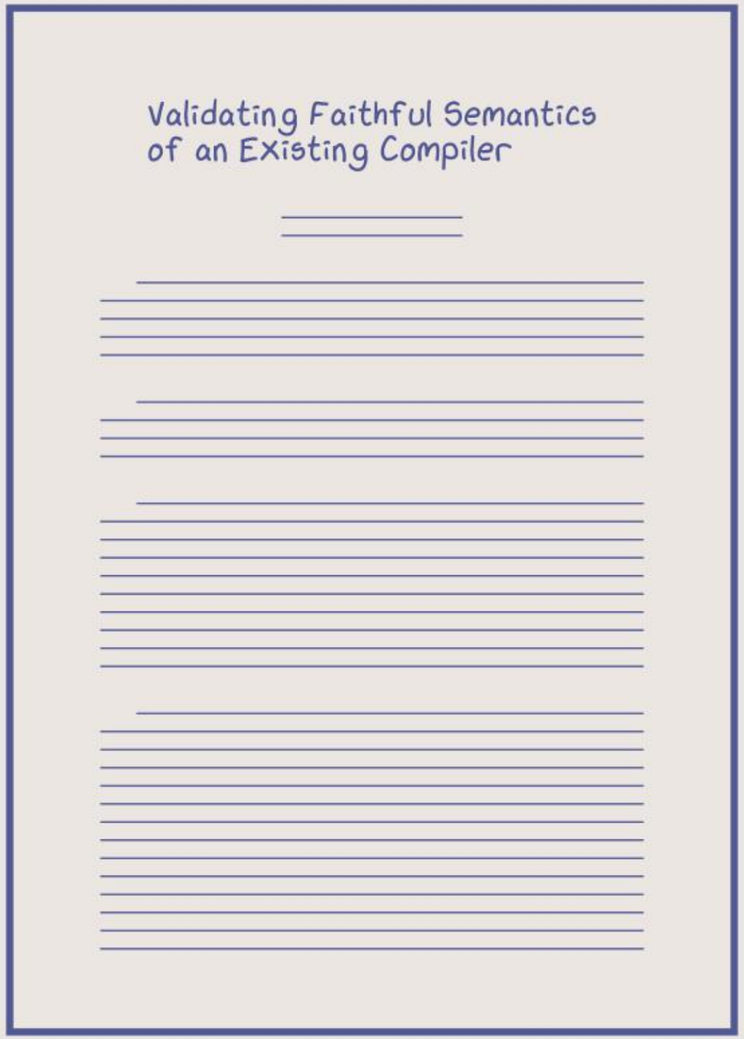
$$\frac{ConditionalElimination.Step\ g\ (nid, seen, conds, flow)\ None \qquad None = ConditionalElimination.pred\ g\ nid}{ConditionalEliminationPhase\ g\ (nid, seen, conds, flow)\ g}\ nid \in seen$$

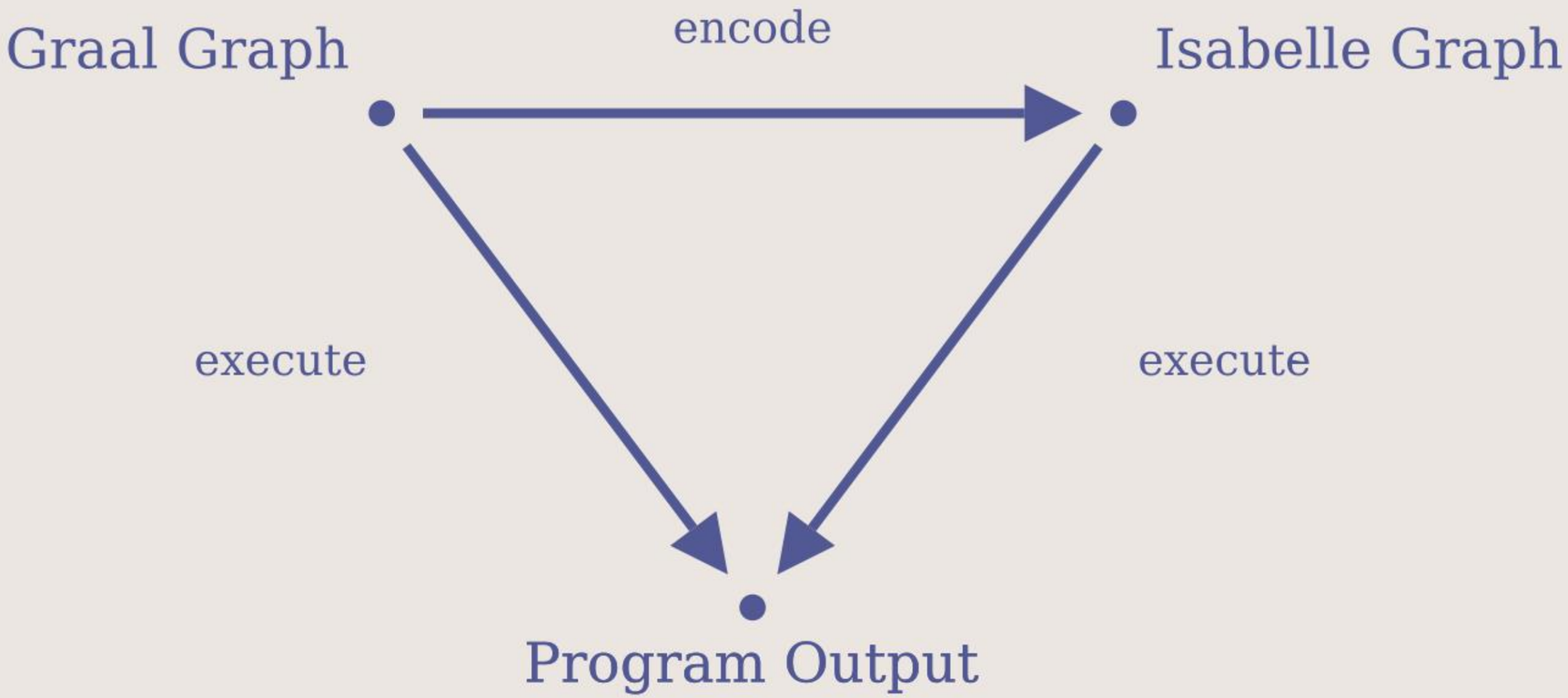
$$\frac{g\langle ifcond\rangle = IfNode\ cid\ t\ f \qquad cond = g\langle cid\rangle \qquad \exists\, c \in conds.\ g \vdash c\ \&\ cond \mapsto KnownTrue \qquad g' = constantCondition\ True\ ifcond\ g\langle ifcond\rangle\ g}{ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'}$$

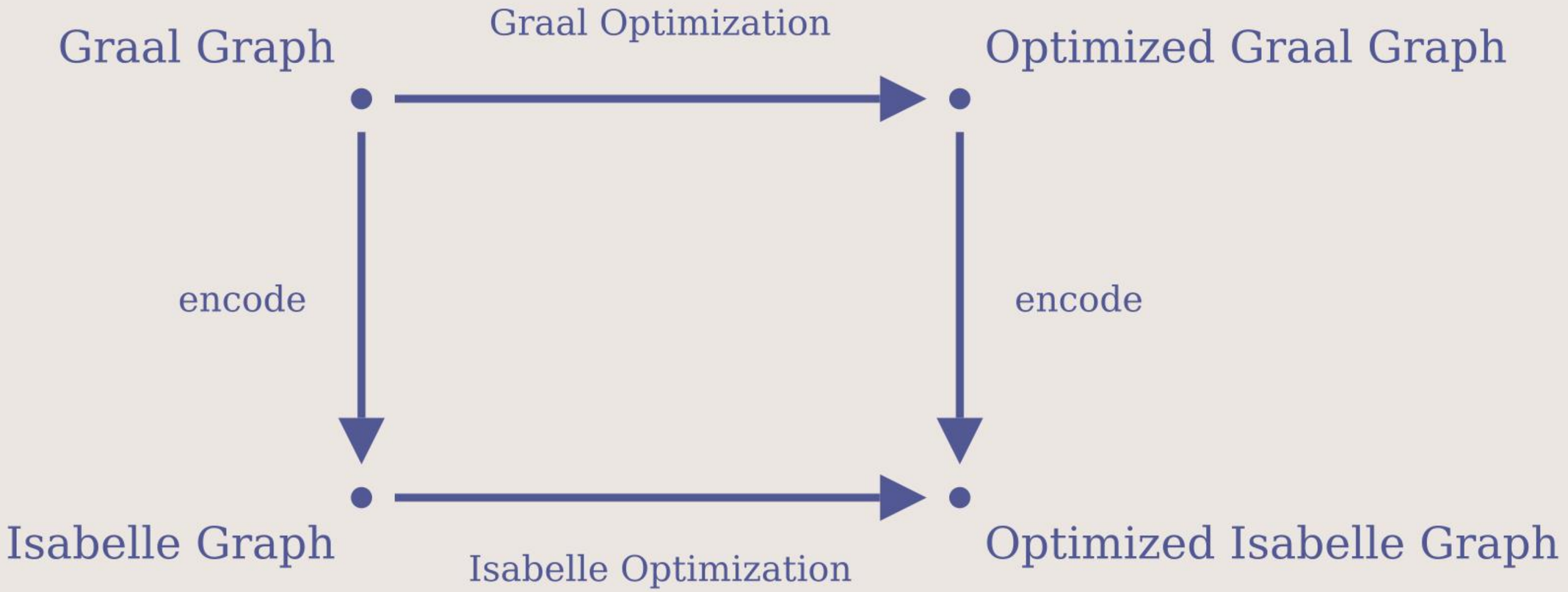
$$\frac{g\langle ifcond\rangle = IfNode\ cid\ t\ f \qquad cond = g\langle cid\rangle \qquad \exists\, c \in conds.\ g \vdash c\ \&\ cond \mapsto KnownFalse \qquad g' = constantCondition\ False\ ifcond\ g\langle ifcond\rangle\ g}{ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'}$$

$$\frac{g\langle ifcond\rangle = IfNode\ cid\ t\ f \qquad cond = g\langle cid\rangle \qquad tryFold\ g\langle cid\rangle\ stamps\ KnownTrue \qquad g' = constantCondition\ True\ ifcond\ g\langle ifcond\rangle\ g}{ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'}$$

$$\frac{g\langle ifcond\rangle = IfNode\ cid\ t\ f \qquad cond = g\langle cid\rangle \qquad tryFold\ g\langle cid\rangle\ stamps\ KnownFalse \qquad g' = constantCondition\ False\ ifcond\ g\langle ifcond\rangle\ g}{ConditionalEliminationStep\ conds\ stamps\ g\ ifcond\ g'}$$


Validating Faithful Semantics
of an Existing Compiler


Graal Graph
encode
Isabelle Graph
execute
execute
Program Output


Graal Graph
Graal Optimization
Optimized Graal Graph
encode
encode
Isabelle Graph
Isabelle Optimization
Optimized Isabelle Graph

What's Next?

```
boolean associative = op.isAssociative();
if (associative) {
    if (forX instanceof SubNode) {
        SubNode sub = (SubNode) forX;
        if (sub.getY() == forY) {
            // (a - b) + b
            return sub.getX();
        }
    }
}
```

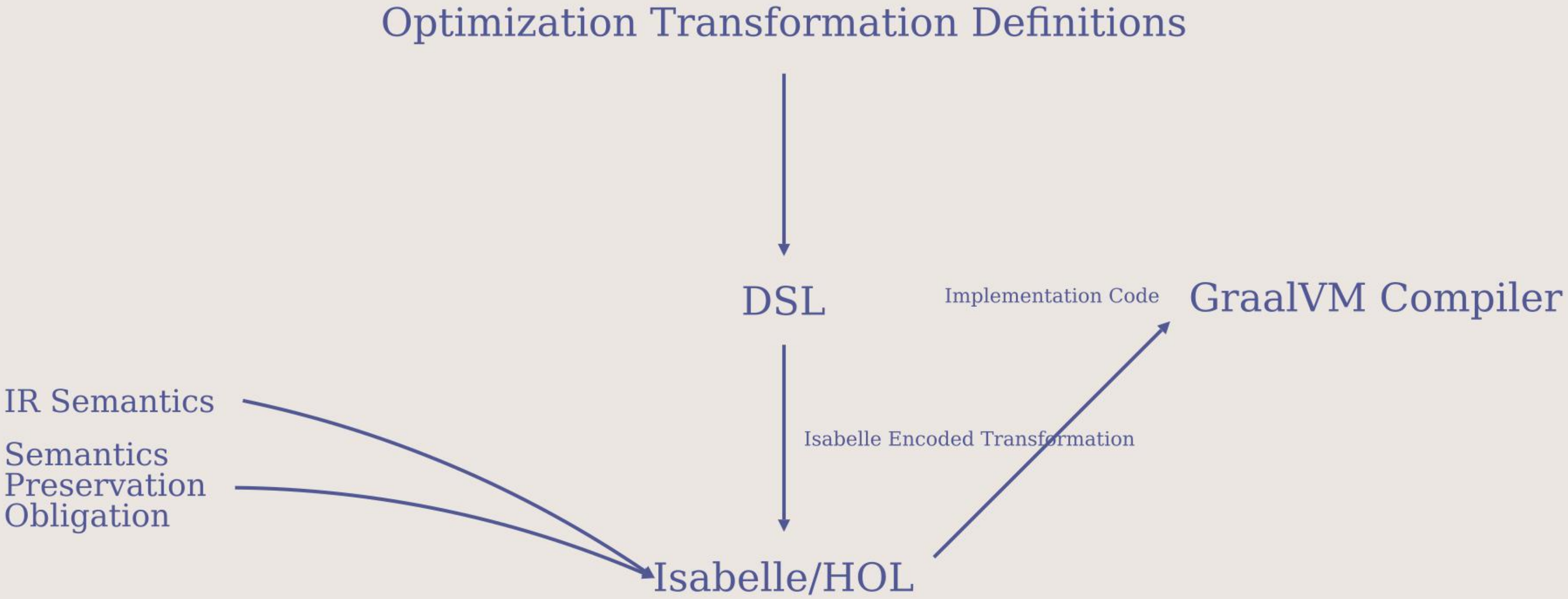
```
boolean associative = op.isAssociative();
if (associative) {
    if (forX instanceof SubNode) {
        SubNode sub = (SubNode) forX;
        if (sub.getY() == forY) {
            // (a - b) + b
            return sub.getX();
        }
    }
}
```

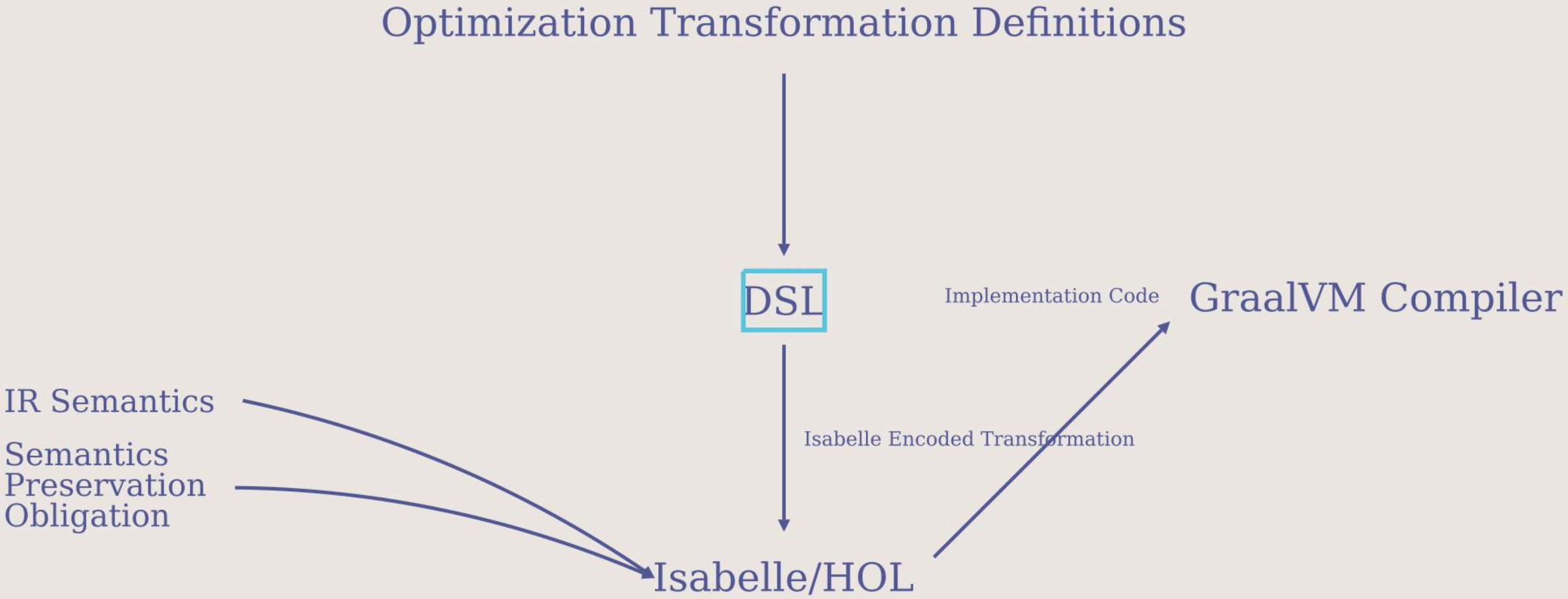
$$(\text{BinaryExpr BinAdd (BinaryExpr BinSub a b) b}) \mapsto \text{a}$$

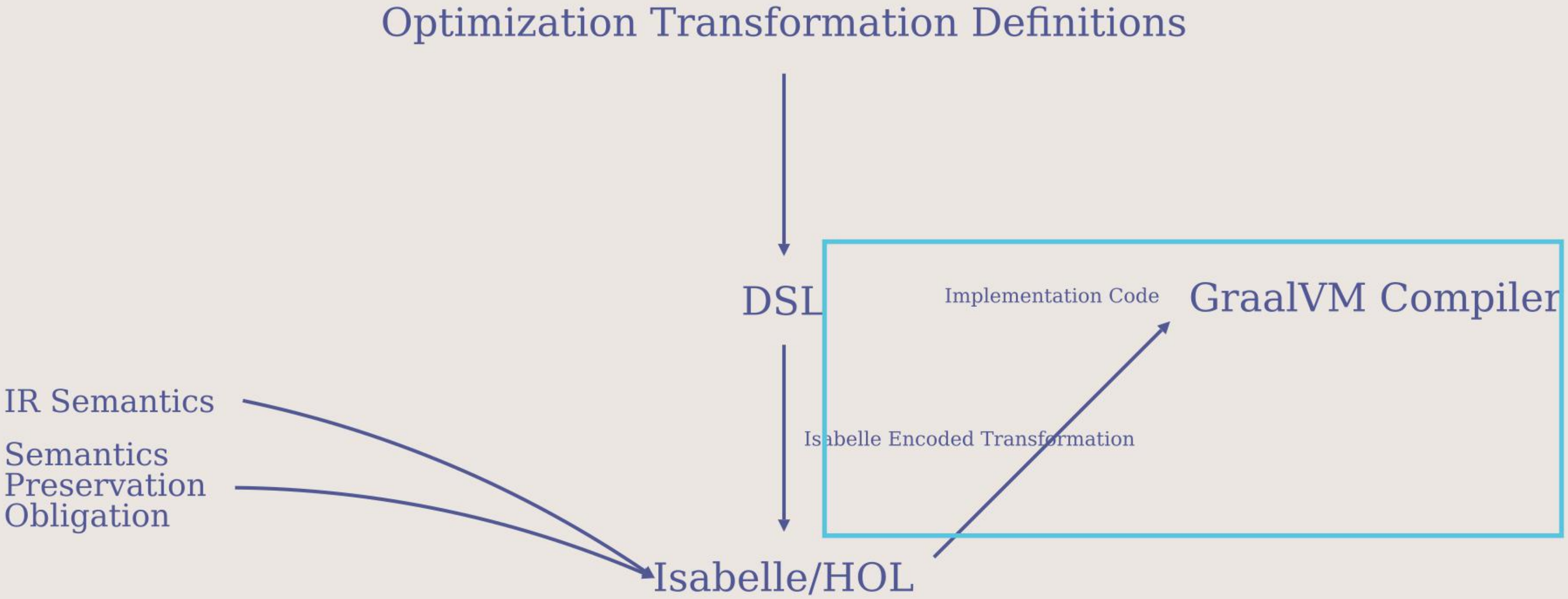
```
boolean associative = op.isAssociative();
if (associative) {
    if (forX instanceof SubNode) {
        SubNode sub = (SubNode) forX;
        if (sub.getY() == forY) {
            // (a - b) + b
            return sub.getX();
        }
    }
}
```

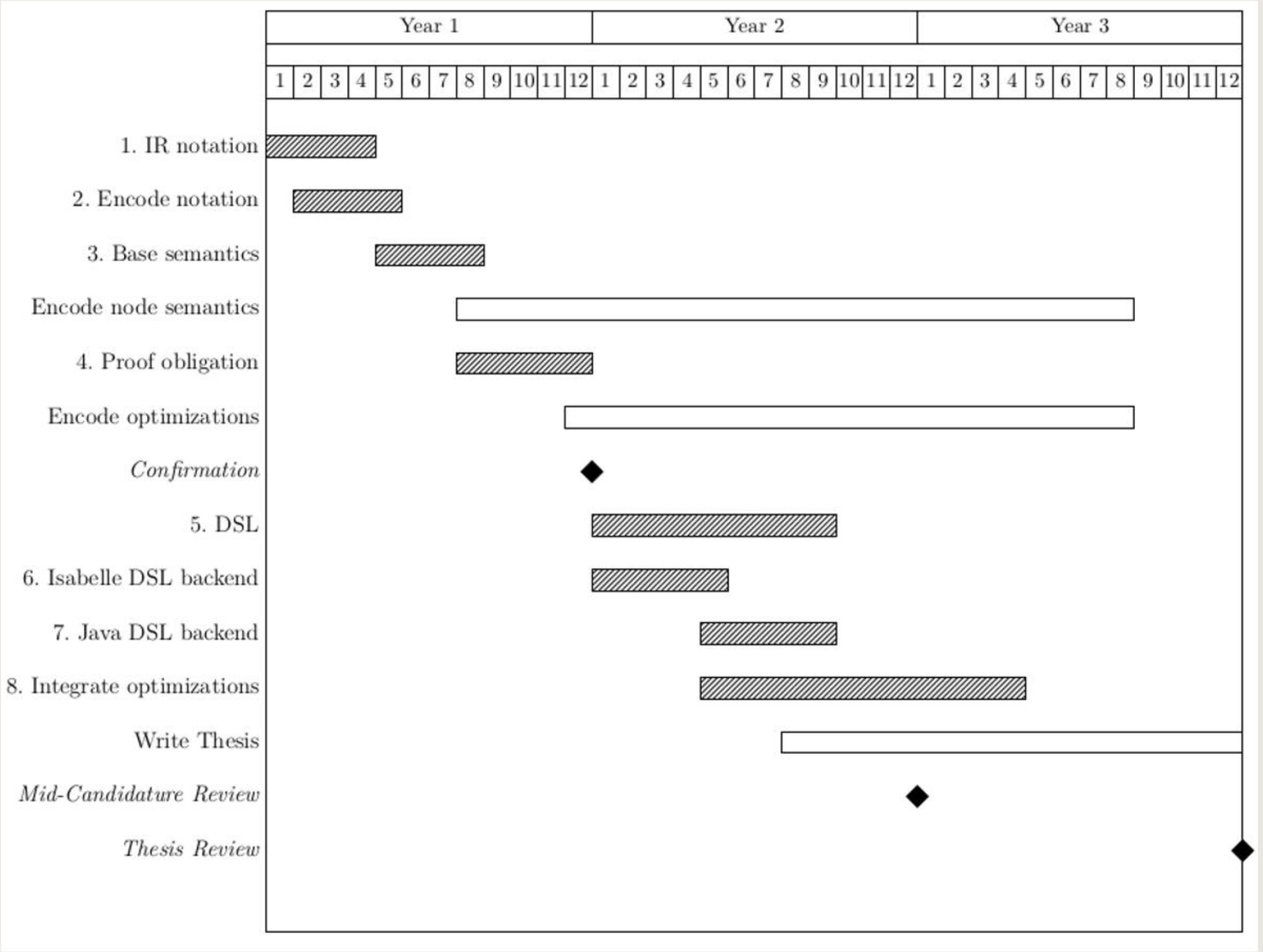
(BinaryExpr BinAdd (BinaryExpr BinSub a b) b) $\mapsto$ a

$$(a - b) + b \mapsto a$$


Optimization Transformation Definitions
DSL
Implementation Code
GraalVM Compiler
IR Semantics
Semantics Preservation Obligation
Isabelle Encoded Transformation
Isabelle/HOL


Optimization Transformation Definitions
DSL
Implementation Code
GraalVM Compiler
IR Semantics
Semantics Preservation Obligation
Isabelle Encoded Transformation
Isabelle/HOL


Optimization Transformation Definitions
DSL
Implementation Code
GraalVM Compiler
IR Semantics
Semantics Preservation Obligation
Isabelle Encoded Transformation
Isabelle/HOL


Year 1
Year 2
Year 3
1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9 10 11 12
1. IR notation
2. Encode notation
3. Base semantics
Encode node semantics
4. Proof obligation
Encode optimizations
Confirmation
5. DSL
6. Isabelle DSL backend
7. Java DSL backend
8. Integrate optimizations
Write Thesis
Mid-Candidature Review
Thesis Review

Thank You!

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

Conference on Programming Language Design and Implementation (PLDI) 2011

A Simple Graph-Based Intermediate Representation

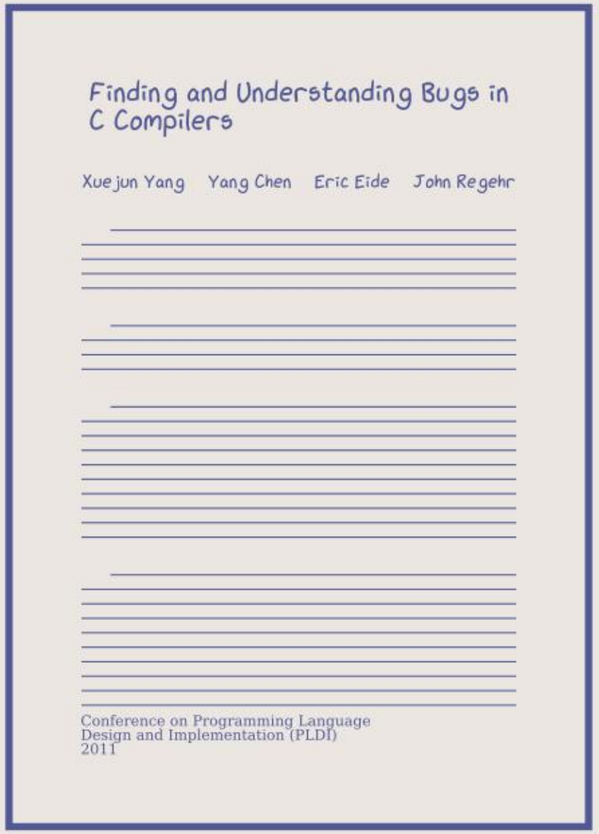
Cliff Click Michael Paleczny

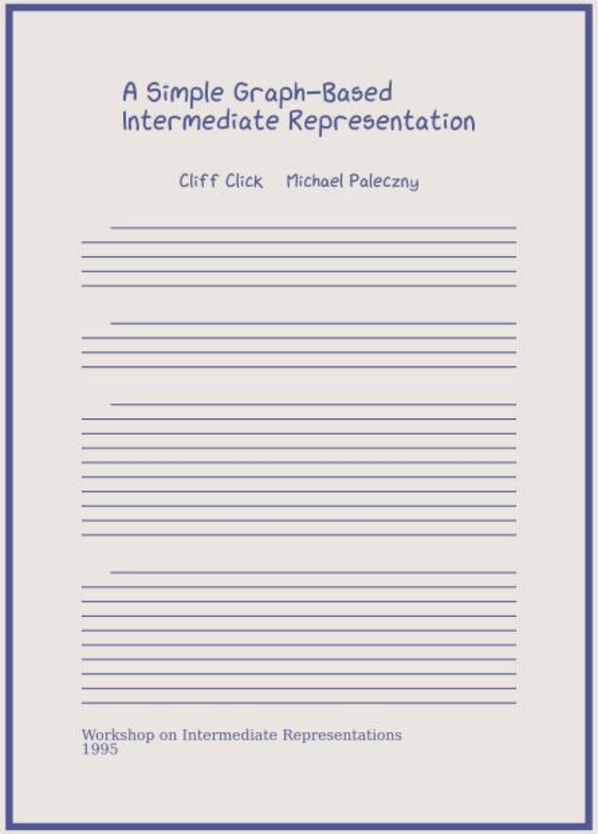
Workshop on Intermediate Representations 1995

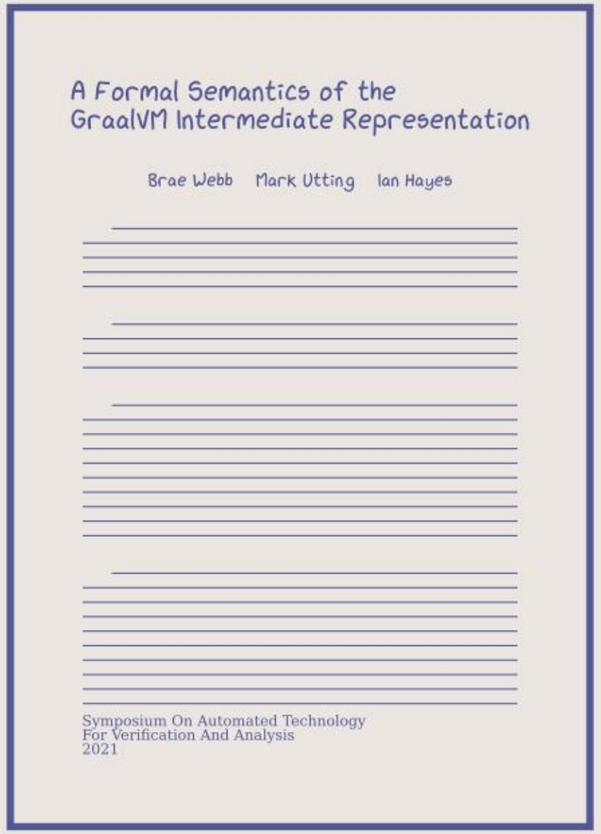
A Formal Semantics of the GraalVM Intermediate Representation

Brae Webb Mark Utting Ian Hayes

Symposium On Automated Technology For Verification And Analysis 2021







https://phd.brae.dev