Lambda the Ultimate SSA!

Optimizing functional programs in SSA

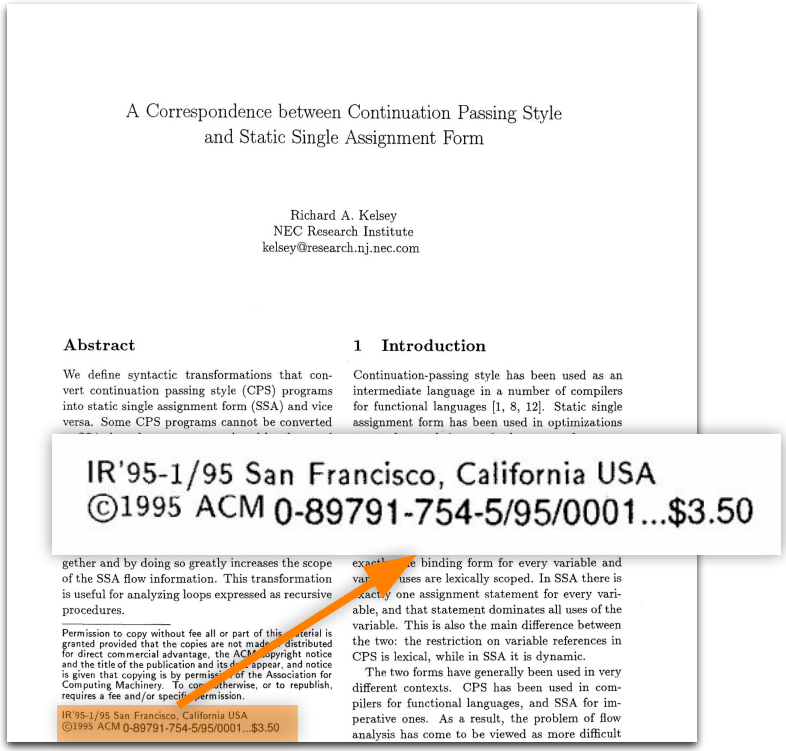
Siddharth Bhat, Tobias Grosser
University of Edinburgh

```
def out :=
  match True with
  | True → e
  | False → f
```

```
func out() {
  lp.switch true
    true → { e }
    false → { f }
}
```

```
func out() {
  %ve = rgn.val { e }
  %vf = rgn.val { f }
  %r = select true, %ve, %vf
  rgn.run %r
}
```

Continuations == SSA - A Solved Problem ?



A Correspondence between Continuation Passing Style and Static Single Assignment Form

Richard A. Kelsey
NEC Research Institute
kelsey@research.nj.nec.com

Abstract

We define syntactic transformations that convert continuation passing style (CPS) programs into static single assignment form (SSA) and vice versa. Some CPS programs cannot be converted

1 Introduction

Continuation-passing style has been used as an intermediate language in a number of compilers for functional languages [1, 8, 12]. Static single assignment form has been used in optimizations

IR'95-1/95 San Francisco, California USA
©1995 ACM 0-89791-754-5/95/0001...$3.50

gether and by doing so greatly increases the scope of the SSA flow information. This transformation is useful for analyzing loops expressed as recursive procedures.

IR'95-1/95 San Francisco, California USA
©1995 ACM 0-89791-754-5/95/0001...$3.50

exactly the binding form for every variable and variable uses are lexically scoped. In SSA there is exactly one assignment statement for every variable, and that statement dominates all uses of the variable. This is also the main difference between the two: the restriction on variable references in CPS is lexical, while in SSA it is dynamic.

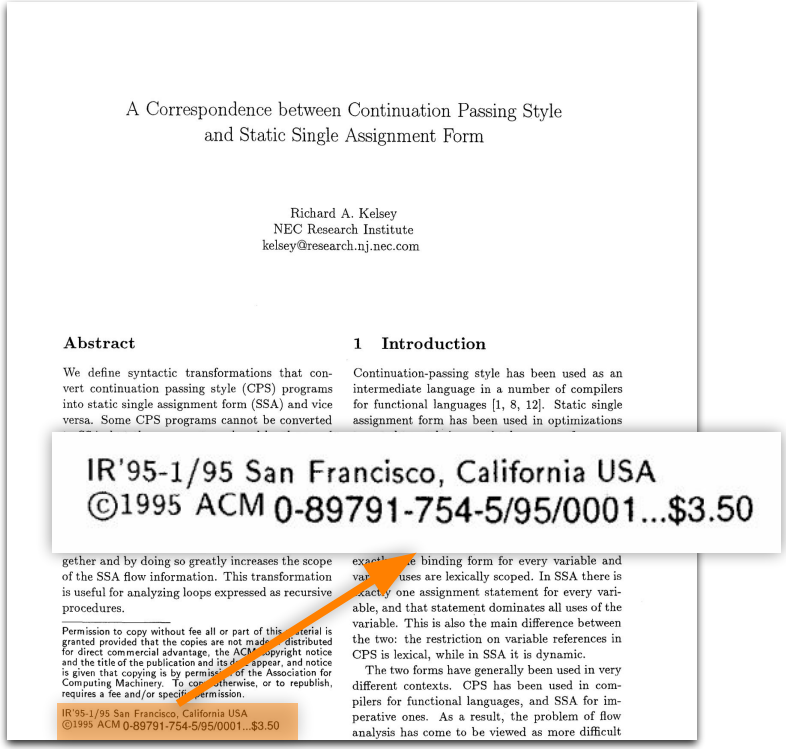
The two forms have generally been used in very different contexts. CPS has been used in compilers for functional languages, and SSA for imperative ones. As a result, the problem of flow analysis has come to be viewed as more difficult

$$\mathcal{G} : \mathrm{M}' \rightarrow \mathrm{B}$$
$$\mathcal{G}([\![(\texttt{let}\ ((x\ \mathrm{E}))\ \mathrm{M}')]\!]) = x \leftarrow \mathrm{E};\ \mathcal{G}([\![\mathrm{M}']\!])$$

$$\mathcal{G}([\![(\mathrm{E}\ \ldots\ k)]\!]) = \texttt{return}\ \mathrm{E}(\ldots);$$
$$\mathcal{G}([\![(k\ \mathrm{E})]\!]) = \texttt{return}\ \mathrm{E};$$

$$\mathcal{G}([\![(\texttt{if}\ \mathrm{E}\ \mathrm{M}'_1\ \mathrm{M}'_2)]\!]) = \texttt{if}\ \mathrm{E}\ \texttt{then}\ \mathcal{G}([\![\mathrm{M}'_1]\!])\ \texttt{else}\ \mathcal{G}([\![\mathrm{M}'_2]\!])$$
$$\mathcal{G}([\![(\texttt{letrec}\ (\ldots)\ \mathrm{M}')]\!]) = \mathcal{G}([\![\mathrm{M}']\!])$$

Continuations == SSA - A Solved Problem ?



A Correspondence between Continuation Passing Style and Static Single Assignment Form

Richard A. Kelsey
NEC Research Institute
kelsey@research.nj.nec.com

Abstract

We define syntactic transformations that convert continuation passing style (CPS) programs into static single assignment form (SSA) and vice versa. Some CPS programs cannot be converted

1 Introduction

Continuation-passing style has been used as an intermediate language in a number of compilers for functional languages [1, 8, 12]. Static single assignment form has been used in optimizations

IR'95-1/95 San Francisco, California USA
©1995 ACM 0-89791-754-5/95/0001...$3.50

gether and by doing so greatly increases the scope of the SSA flow information. This transformation is useful for analyzing loops expressed as recursive procedures.

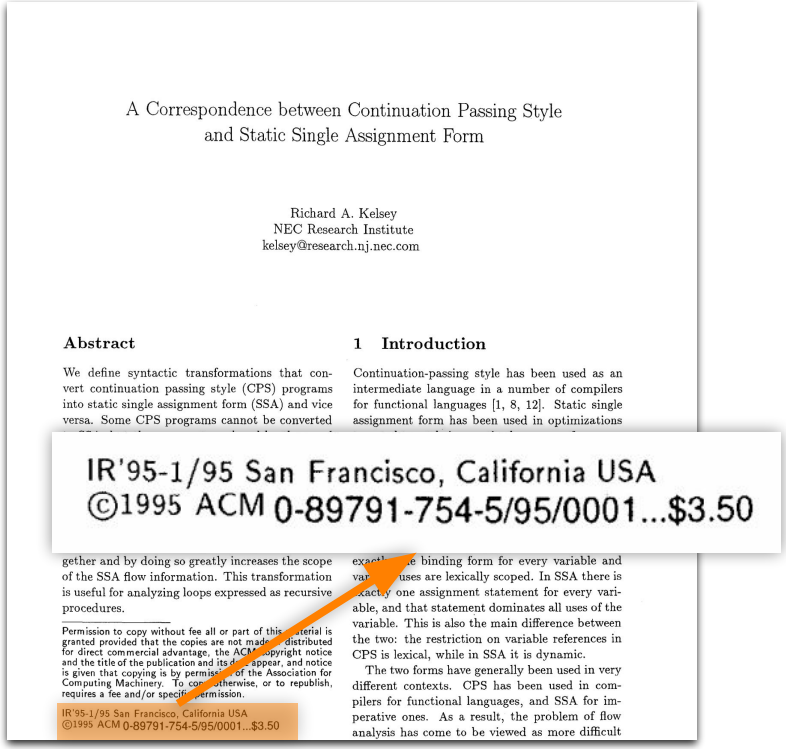
IR'95-1/95 San Francisco, California USA
©1995 ACM 0-89791-754-5/95/0001...$3.50

exactly one binding form for every variable and variable uses are lexically scoped. In SSA there is exactly one assignment statement for every variable, and that statement dominates all uses of the variable. This is also the main difference between the two: the restriction on variable references in CPS is lexical, while in SSA it is dynamic.

The two forms have generally been used in very different contexts. CPS has been used in compilers for functional languages, and SSA for imperative ones. As a result, the problem of flow analysis has come to be viewed as more difficult

$$\mathcal{G} : \mathrm{M}' \rightarrow \mathrm{B}$$
$$\mathcal{G}([\![(\texttt{let } ((x\ \mathrm{E}))\ \mathrm{M}')]\!]) = x \leftarrow \mathrm{E};\ \mathcal{G}([\![\mathrm{M}']\!])$$
$$\mathcal{G}([\![(\mathrm{E}\ \ldots\ (\lambda_{cont}\ (x)\ \mathrm{M}'))]\!]) = x \leftarrow \mathrm{E}(\ \ldots\ );\ \mathcal{G}([\![\mathrm{M}']\!])$$
$$\mathcal{G}([\![(\mathrm{E}\ \ldots\ k)]\!]) = \texttt{return } \mathrm{E}(\ldots);$$
$$\mathcal{G}([\![(k\ \mathrm{E})]\!]) = \texttt{return } \mathrm{E};$$
$$\mathcal{G}([\![(j\ \mathrm{E}_{0,i}\ \mathrm{E}_{1,i}\ \ldots)]\!]) = \texttt{goto } j_i;$$
$$\mathcal{G}([\![(\texttt{if E } \mathrm{M}'_1\ \mathrm{M}'_2)]\!]) = \texttt{if E then } \mathcal{G}([\![\mathrm{M}'_1]\!]) \texttt{ else } \mathcal{G}([\![\mathrm{M}'_2]\!])$$
$$\mathcal{G}([\![(\texttt{letrec } (\ \ldots)\ \mathrm{M}')]\!]) = \mathcal{G}([\![\mathrm{M}']\!])$$
$$\mathcal{G}_{proc} : \mathrm{P}' \rightarrow \mathrm{P}$$
$$\mathcal{G}_{proc}([\![(\lambda_{proc}\ (x\ \ldots)\ \mathrm{M}')]\!]) = \texttt{proc}(x\ \ldots\ k)\{\ \mathcal{G}([\![\mathrm{M}']\!])\ \mathcal{G}_{jum}$$
$$\mathcal{G}_{jump} : j \times (\lambda_{jump}\ (x\ \ldots)\ \mathrm{M}') \rightarrow \mathrm{L}$$
$$\mathcal{G}_{jump}([\![j,\ (\lambda_{jump}\ (x\ \ldots)\ \mathrm{M}')]\!]) = j :\ x \leftarrow \phi(\mathrm{E}_{0,0}, \mathrm{E}_{0,1}, \ldots)$$

Continuations == SSA - A Solved Problem ?



A Correspondence between Continuation Passing Style and Static Single Assignment Form

Richard A. Kelsey
NEC Research Institute
kelsey@research.nj.nec.com

Abstract

We define syntactic transformations that convert continuation passing style (CPS) programs into static single assignment form (SSA) and vice versa. Some CPS programs cannot be converted

1 Introduction

Continuation-passing style has been used as an intermediate language in a number of compilers for functional languages [1, 8, 12]. Static single assignment form has been used in optimizations

IR'95-1/95 San Francisco, California USA
©1995 ACM 0-89791-754-5/95/0001...$3.50

gether and by doing so greatly increases the scope of the SSA flow information. This transformation is useful for analyzing loops expressed as recursive procedures.

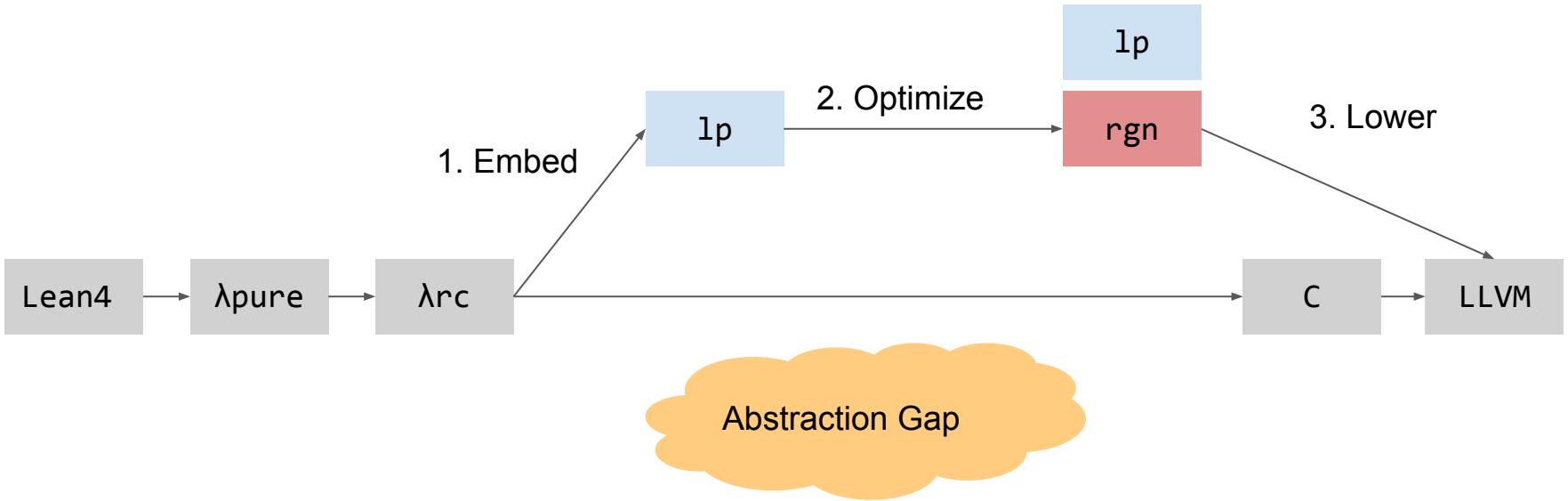
IR'95-1/95 San Francisco, California USA
©1995 ACM 0-89791-754-5/95/0001...$3.50

exactly one binding form for every variable and variables are lexically scoped. In SSA there is exactly one assignment statement for every variable, and that statement dominates all uses of the variable. This is also the main difference between the two: the restriction on variable references in CPS is lexical, while in SSA it is dynamic.

The two forms have generally been used in very different contexts. CPS has been used in compilers for functional languages, and SSA for imperative ones. As a result, the problem of flow analysis has come to be viewed as more difficult

$$\mathcal{G} : \mathrm{M}' \rightarrow \mathrm{B}$$
$$\mathcal{G}([\![(\texttt{let } ((x\ \mathrm{E}))\ \mathrm{M}')]\!]) = x \leftarrow \mathrm{E};\ \mathcal{G}([\![\mathrm{M}']\!])$$
$$\mathcal{G}([\![(\mathrm{E}\ \ldots\ (\lambda_{cont}\ (x)\ \mathrm{M}'))]\!]) = x \leftarrow \mathrm{E}(\ \ldots\ );\ \mathcal{G}([\![\mathrm{M}']\!])$$
$$\mathcal{G}([\![(\mathrm{E}\ \ldots\ k)]\!]) = \texttt{return } \mathrm{E}(\ldots);$$
$$\mathcal{G}([\![(k\ \mathrm{E})]\!]) = \texttt{return } \mathrm{E};$$
$$\mathcal{G}([\![(j\ \mathrm{E}_{0,i}\ \mathrm{E}_{1,i}\ \ldots)]\!]) = \texttt{goto } j_i;$$
$$\mathcal{G}([\![(\texttt{if E } \mathrm{M}'_1\ \mathrm{M}'_2)]\!]) = \texttt{if E then } \mathcal{G}([\![\mathrm{M}'_1]\!]) \texttt{ else } \mathcal{G}([\![\mathrm{M}'_2]\!])$$
$$\mathcal{G}([\![(\texttt{letrec } (\ \ldots)\ \mathrm{M}')]\!]) = \mathcal{G}([\![\mathrm{M}']\!])$$
$$\mathcal{G}_{proc} : \mathrm{P}' \rightarrow \mathrm{P}$$
$$\mathcal{G}_{proc}([\![(\lambda_{proc}\ (x\ \ldots)\ \mathrm{M}')]\!]) = \texttt{proc}(x\ \ldots\ k)\{\ \mathcal{G}([\![\mathrm{M}']\!])\ \mathcal{G}_{jum}$$
$$\mathcal{G}_{jump} : j \times (\lambda_{jump}\ (x\ \ldots)\ \mathrm{M}') \rightarrow \mathrm{L}$$
$$\mathcal{G}_{jump}([\![j,\ (\lambda_{jump}\ (x\ \ldots)\ \mathrm{M}')]\!]) = j : x \leftarrow \phi(\mathrm{E}_{0,0}, \mathrm{E}_{0,1}, \ldots)$$

The Abstraction Gap of Functional IR vs SSA IR in Lean4



SSA Primitives for Strict Functional IRs + Ref-Counting

Algebraic Data Types

```
lp.construct : (!lp.t)* -> !lp.t
lp.construct %<arg>* { tag = <tag> }

lp.getlabel : !lp.t -> i8
lp.getlabel %<val>

lp.project: !lp.t -> !lp.t
lp.project %<data>, <constant-index>

lp.switch: i8 -> ()
lp.switch %<tag>
  <alt-0> → {
    // rhs-0
  },
  <alt-1> → {
    // rhs-1
  },
  ...
  @default → {
    // rhs-default
  }
```

Numerics

```
lp.int: i64 → !lp.t
lp.int %<v>

lp.bigint: !lp.t
lp.bigint <constant-bigint>
```

Closures

```
lp.pap: (!lp.t)* -> !lp.t
lp.pap @<fn-name> %<arg>*

lp.papextend: !lp.t x (!lp.t)* -> !lp.t
lp.papextend %<closure>, %<arg>*
```

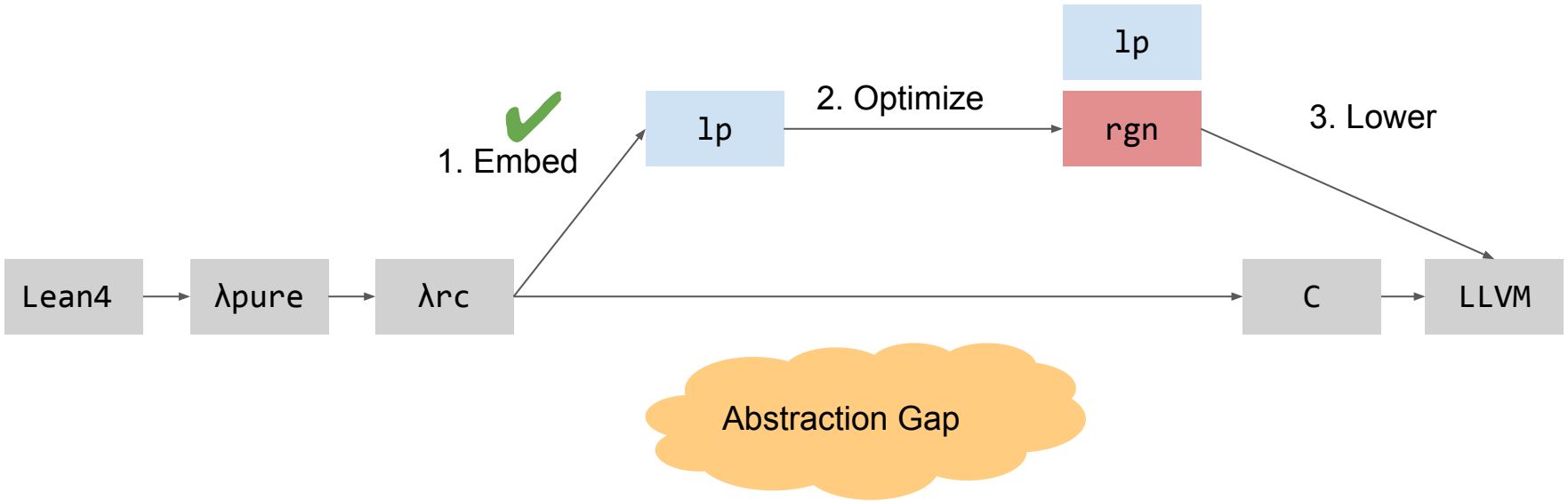
Reference Counting

```
lp.inc: !lp.t -> ()
lp.inc %<v>

lp.dec: !lp.t -> ()
lp.dec %<v>
```

Join Points

```
lp.joinpoint : ()
lp.joinpoint @<label> {
    // after-jump
  }, {
    // pre-jump
  }

lp.jump : (!lp.t)* -> ()
lp.jump @<label> %<arg>*
```

A Feature Complete Embedding into SSA+Regions



Optimizing `match` via Classical SSA Transforms

```
def out :=
  match True with
  | True  → 1
  | False → 2
```

⬇

```
func out() {
  lp.switch true
    true  → { return 1 }
    false → { return 2 }
}
```

⬇?

```
func out() {
   return 1
}
```

```
select true %l, %r → %l
```

```
%l = 1
%r = 2
%z = select true, %l, %r
return %z
```

➡

```
%l = 1
%r = 2
%z = %l
return %z
```

➡

```
%l = 1
return %l
```

➡

```
func out() {
  %l = rgn.val {
         return 1
       }
  %r = rgn.val {
         return 2
       }
  %z = select true, %l, %r
  rgn.run %z
}
```

➡

```
func out() {
  %l = rgn.val {
         return 1
       }
  %r = rgn.val {
         return 2
       }
  %z = %l
  rgn.run %z
}
```

➡

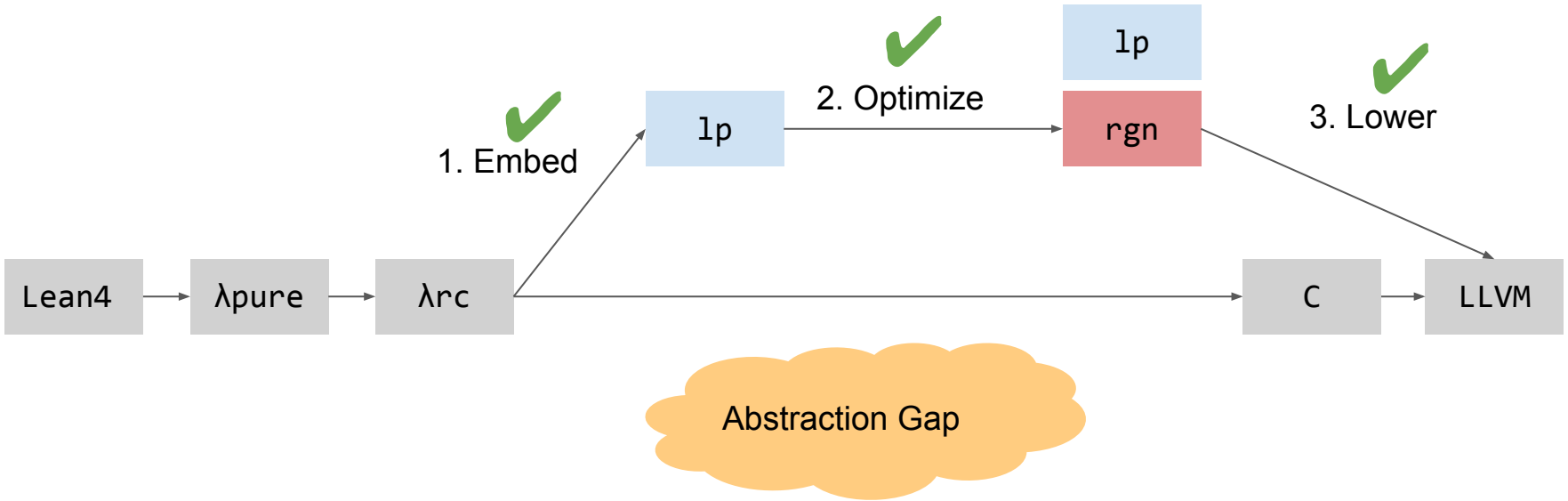
```
func out() {
  %l = rgn.val {
         return 1
       }
  rgn.run %l
}
```

⬇

```
rgn.run (rgn.val %l)
    → inline %l
```

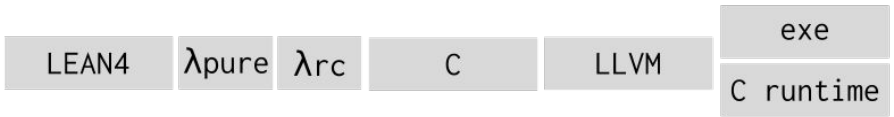
```
func out() {
  return 1
}
```

End To End Pipeline for Functional IRs in SSA



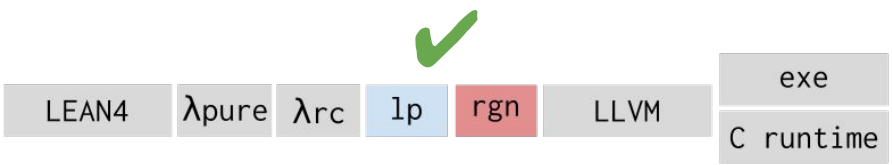
Full Correctness On the Lean4 Test Suite

Test suite: 648/648 [Golden tests]



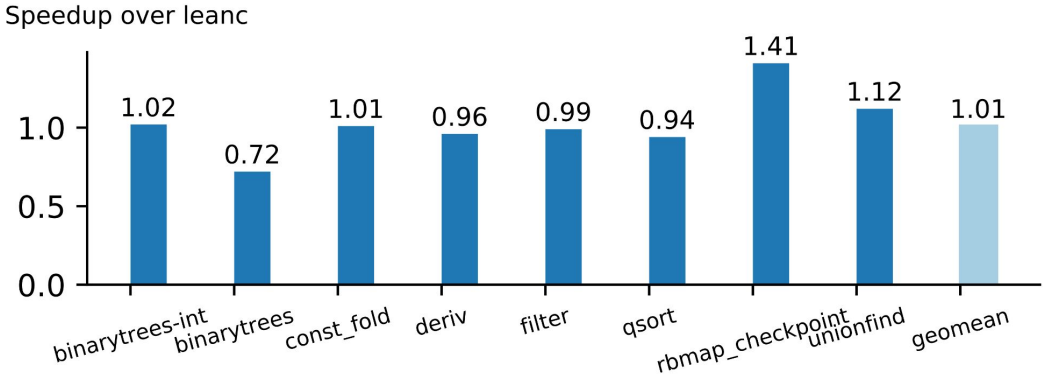
Full Correctness On the Lean4 Test Suite

Test suite: 648/648 [Golden tests]



More Semantic Guarantees!

Performance Parity with Existing Backend



Lean4 Benchmark Suite

`binarytrees`	functional binary tree lookup, insert, and delete.
`binarytrees-int`	functional int binary tree lookup, insert, and delete.
`const_fold`	constant folding on expression language.
`deriv`	symbolic derivative on expression trees.
`filter`	filtering values from a linked list.
`qsort`	real in-place quicksort using Lean4 arrays.
`rbmap_checkpoint`	red-black tree insertion and lookup
`unionfind`	Tarjan’s union-find algorithm.

Why lz

```
def out :=
  match True with
  | True  → 1
  | False → 2
```

⬇

```
func out() {
  lp.switch true
    true → { return 1 }
    false → { return 2 }
}
```

⬇

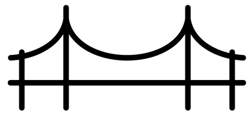
```
func out() {
  %1 = rgn.val { return 1 }
  %r = rgn.val { return 2 }
  %out = select true, %1, %r
  rgn.run %out
}
```

⬇

```
select true %1, %r → %1
rgn.run (rgn.val %r) → inline %r
```

⬇

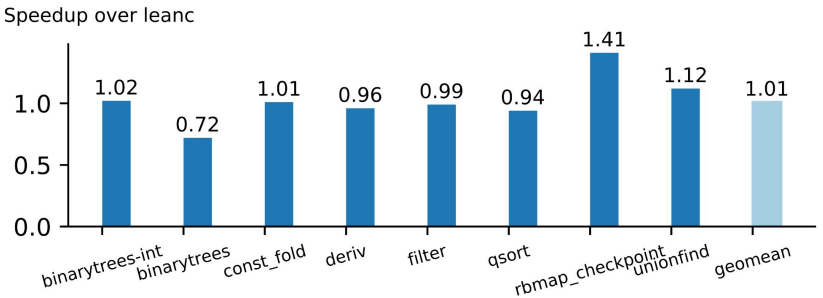
```
func out() { return 1 }
```

Functional IR  SSA + Regions

lp & rgn

Test suite: 648/648 Golden Tests

Performance Parity



Research → Production: Real World Concerns

MLIR

Lean4: early adopter

Lean4, C Bindings, C++ defined IR

LLVM

Other Users in Production (GHC)

Lean4, C Bindings