

Towards Optimising Certified Programs by Proof Rewriting

Kiran Gopinathan, Ilya Sergey

National University of Singapore

Let's write a program!

Let's write a program!

Q: Free a linked list.

```
void listfree(loc x) {
```

```
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
  
        } else {  
  
  
        }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
  
    }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
  
    }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
  
    }  
}
```



```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
    }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    }  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
}
```

Q: Why is it correct?

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    }  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
}
```

Q: Why is it correct?

Let's write a proof!

Let's write a proof!

Let's write a proof!

$\{\text{lseg}(x, S)\}$ `listfree(x)` $\{\text{emp}\}$

$\{Iseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $Iseg$ (**loc** x , **set** S) {

| $x = 0 \Rightarrow \{ S = \emptyset; emp \}$

| $x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto next * Iseg(next, S_1) \} \}$

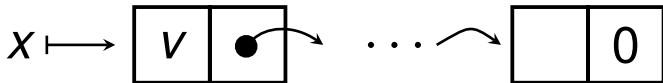
$\{lseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $lseg(\text{loc } x, \text{set } S)\{$

$| x = 0 \Rightarrow \{ S = \emptyset; emp \}$

$| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto next * lseg(next, S_1) \} \}$



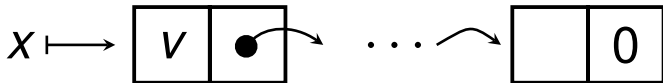
$\{ \text{lseg}(x, S) \} \quad \text{listfree}(x) \quad \{ \text{emp} \}$

predicate $\text{lseg}(\text{loc } x, \text{set } S) \{$

$| x = 0 \Rightarrow \{ S = \emptyset; \text{emp} \}$

$| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto \text{nxt} * \text{lseg}(\text{nxt}, S_1) \} \}$



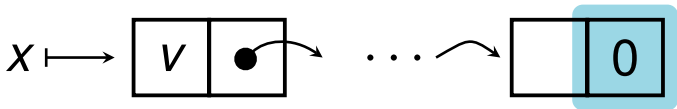
$\{lseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $lseg$ (**loc** x , **set** S) {

$| x = 0 \Rightarrow \{ S = \emptyset; emp \}$

$| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto next * lseg(next, S_1) \} \}$



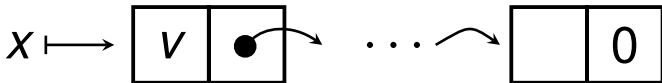
$\{lseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $lseg(\text{loc } x, \text{set } S)\{$

$| x = 0 \Rightarrow \{ S = \emptyset; emp \}$

$| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto next * lseg(next, S_1) \} \}$



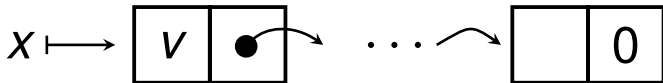
$\{lseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $lseg$ (**loc** x , **set** S) {

| $x = 0 \Rightarrow \{ S = \emptyset; emp \}$

| $x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto next * lseg(next, S_1) \}$ }



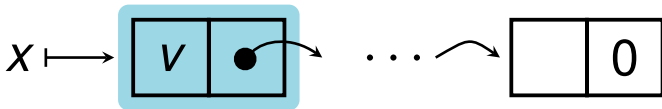
$\{lseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $lseg$ (**loc** x , **set** S) {

| $x = 0 \Rightarrow \{ S = \emptyset; emp \}$

| $x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto nxt * lseg(nxt, S_1) \} \}$



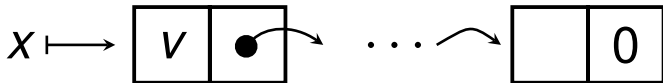
$\{ \text{lseg}(x, S) \} \quad \text{listfree}(x) \quad \{ \text{emp} \}$

predicate $\text{lseg}(\text{loc } x, \text{set } S) \{$

$| x = 0 \Rightarrow \{ S = \emptyset; \text{emp} \}$

$| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto \text{next} * \text{lseg}(\text{next}, S_1) \} \}$



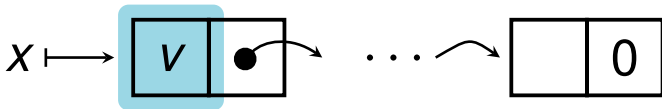
$\{lseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $lseg$ (**loc** x , **set** S) {

| $x = 0 \Rightarrow \{ S = \emptyset; emp \}$

| $x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto next * lseg(next, S_1) \}$ }



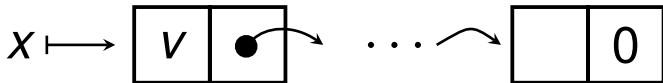
$\{ \text{lseg}(x, S) \} \quad \text{listfree}(x) \quad \{ \text{emp} \}$

predicate $\text{lseg}(\text{loc } x, \text{set } S) \{$

$| x = 0 \Rightarrow \{ S = \emptyset; \text{emp} \}$

$| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto \text{next} * \text{lseg}(\text{next}, S_1) \} \}$



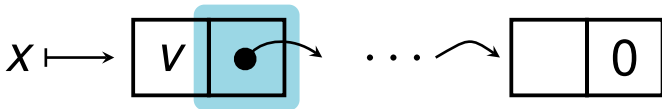
$\{lseg(x, S)\} \quad listfree(x) \quad \{emp\}$

predicate $lseg$ (**loc** x , **set** S) {

| $x = 0 \Rightarrow \{ S = \emptyset; emp \}$

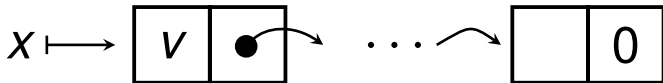
| $x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$

$[x, 2] * x \mapsto v * (x + 1) \mapsto next * lseg(next, S_1) \} \}$



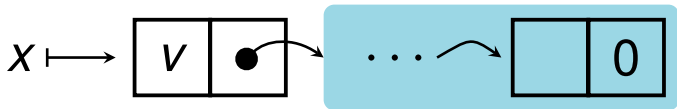
$\{ \text{lseg}(x, S) \} \quad \text{listfree}(x) \quad \{ \text{emp} \}$

predicate $\text{lseg}(\text{loc } x, \text{set } S) \{$
 $| x = 0 \Rightarrow \{ S = \emptyset; \text{emp} \}$
 $| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$
 $[x, 2] * x \mapsto v * (x + 1) \mapsto \text{next} * \text{lseg}(\text{next}, S_1) \} \}$



$\{ \text{lseg}(x, S) \} \quad \text{listfree}(x) \quad \{ \text{emp} \}$

predicate $\text{lseg}(\text{loc } x, \text{set } S) \{$
 $| x = 0 \Rightarrow \{ S = \emptyset; \text{emp} \}$
 $| x \neq 0 \Rightarrow \{ S = \{v\} \cup S_1;$
 $[x, 2] * x \mapsto v * (x + 1) \mapsto \text{next} * \text{lseg}(\text{next}, S_1) \} \}$



{lseg(x, S)} listfree(x) {emp}

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

$\{\text{lseg}(x, S)\}$

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

$\{\text{lseg}(x, S)\}$

```
void listfree(loc x) {  
    if (x == 0) {                Open(x, lseg)  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

{emp}

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

Open(x, lseg)
– **Emp**

{emp} return {emp}

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

Open(x, lseg)
– **Emp**

$\{[x, 2] * x \mapsto v * (x + 1) \mapsto \text{next} * \text{lseg}(\text{next}, S_1)\}$

```
void listfree(loc x) {  
    if (x == 0) {  
        return;           Open(x, lseg)  
    } else {             - Emp  
        let h = *x;      -  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

$\{[x, 2] * x \mapsto h * (x + 1) \mapsto \text{next} * \text{lseg}(\text{next}, S_1)\}$

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

Open(x, lseg)
– **Emp**
– **Read**(h, x, 0)

$\{[x, 2] * x \mapsto h * (x + 1) \mapsto t * \text{lseg}(t, S_1)\}$

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

Open(x, lseg)
– **Emp**
– **Read**(h, x, 0)
Read(t, x, 1)

$\{[x, 2] * x \mapsto h * (x + 1) \mapsto t\}$

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

Open(x, lseg)
– **Emp**
– **Read**(h, x, 0)
Read(t, x, 1)
Call(listfree, t, lseg(t, S₁))

{emp}

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

Open(x, lseg)
– **Emp**
– **Read**(h, x, 0)
Read(t, x, 1)
Call(listfree, t, lseg(t, S₁))
Free(x)

{emp} return {emp}

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let h = *x;  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

Open(x, lseg)
– **Emp**
– **Read**(h, x, 0)
Read(t, x, 1)
Call(listfree, t, lseg(t, S₁))
Free(x)
Emp

{lseg(x, S)} listfree(x) {emp}

```
void listfree(loc x) {
```

```
  if (x == 0) {
```

```
    return;
```

{lseg(x, S)} listfree(x) {emp}

```
    let t = *(x + 1);
```

```
    listfree(t);
```

```
    free(x);
```

```
  }
```

```
}
```

Open(x, lseg)

– Emp

– Read(t, x, 0)

Read(t, x, 1)

Call(listfree, t, lseg(t, S₁))

Free(x)

Emp

{lseg(x, S)} listfree(x) {emp}

```
void listfree(loc x) {  
  if (x == 0) {  
    return;  
  } else {  
    let t = *(x + 1);  
    listfree(t);  
    free(x);  
  }  
}
```

Open(x, lseg)
- Emp

{lseg(x, S)} listfree(x) {emp}

- Read(t, x+1)
Read(t, l)
- Free(t, lseg(t, S₁))
Free(x)
Emp

Proved!

$\{\text{lseg}(x, S)\}$ listfree(x) $\{\text{emp}\}$



SuSLiK
Synthesiser



void listfree(loc x)



verify_listfree.v

The only **constant** in life is **change**.

- Heraclitus



code
The only **constant** in ~~life~~ is **change**.

Developers
- ~~Heraclitus~~



code

The only **constant** in ~~life~~ is **change**.

Developers

- ~~Heraclitus~~



```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```

Not stack safe...

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        listfree(t);  
        free(x);  
    }  
}
```



```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        free(x);  
        listfree(t);  
    }  
}
```

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        free(x);  
        listfree(t);  
    }  
}
```

Stack safe!

```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        free(x);  
        listfree(t);  
    }  
}
```

Stack safe!

What about the proof?

What about the proof?

```
void listfree(list *x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        free(x);  
        listfree(t);  
    }  
}
```

Stack safe!

What about the proof?

Problem: Old proof no longer holds

```
void listfree(list x) {  
    if (x == 0) {  
        return;  
    } else {  
        let p = *x;  
        let t = *(x + 1);  
        free(x);  
        listfree(t);  
    }  
}
```

Stack safe!

```
void listfree(list x) {  
  if (x == 0) {  
    return;  
  } else {  
    let p = *x;  
    let t = *(x + 1);  
    free(x);  
    listfree(t);  
  }  
}
```

What about the proof?

Problem: Old proof no longer holds

Idea: *Rewrite* proofs (and programs **together**)

```
void listfree(list x) {  
  if (x == 0) {  
    return;  
  } else {  
    let p = *x;  
    let t = *(x + 1);  
    free(x);  
    listfree(t);  
  }  
}
```

What about the proof?

Problem: Old proof no longer holds

Idea: *Rewrite* proofs (and programs **together**)

 using E-Graphs!

E-graphs over Proofs

E-graphs over Proofs

Open($x, lseg$)

– **Emp**

– **Read**($h, x, 0$)

Read($t, x, 1$)

Call($listfree, t, lseg(t, S_1)$)

Free(x)

Emp

E-graphs over Proofs

Open(x , lseg)

– **Emp**

– **Read**(h , x , 0)

Read(t , x , 1)

Call(listfree, t , lseg(t , S_1))

Free(x)

Emp

E-graphs over Proofs

Open($x, lseg$)

– **Emp**

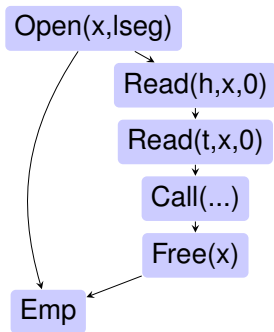
– **Read**($h, x, 0$)

Read($t, x, 1$)

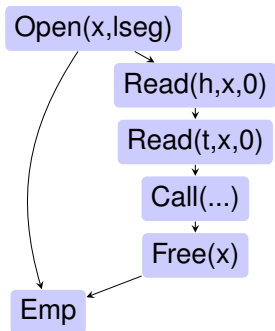
Call($listfree, t, lseg(t, S_1)$)

Free(x)

Emp



E-graphs over Proofs



E-graphs over Proofs

Rewrite Rules

Call(?f, ?H);
Free(?x);
... \Rightarrow **Free(?x);**
Call(?f, ?H);
...

E-graphs over Proofs

Rewrite Rules

Call (?f, ?H);	\Rightarrow	Free (?x);
Free (?x);		Call (?f, ?H);
...		...

Swap **Call** rules followed by a **Free** rule...

E-graphs over Proofs

Rewrite Rules

Call (?f, ?H);	\Rightarrow	Free (?x);
Free (?x);		Call (?f, ?H);
...		...

Not valid
in **general**

Swap **Call** rules followed by a **Free** rule...

E-graphs over Proofs

Rewrite Rules

Call (?f, ?H);	\Rightarrow	Free (?x);
Free (?x);		Call (?f, ?H);
...		...

Swap **Call** rules followed by a **Free** rule...
...if ?x does not occur in ?H

E-graphs over Proofs

Rewrite Rules

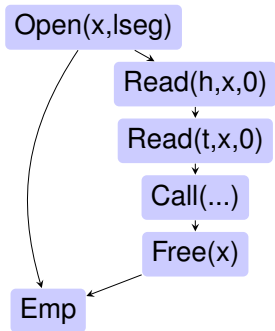
Call (?f, ?H);	\Rightarrow	Free (?x);
Free (?x);		Call (?f, ?H);
...		...

Swap **Call** rules followed by a **Free** rule...

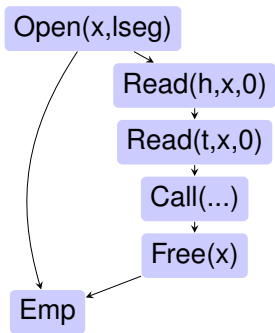
...if ?x does not occur in ?H

Purely syntactic check!

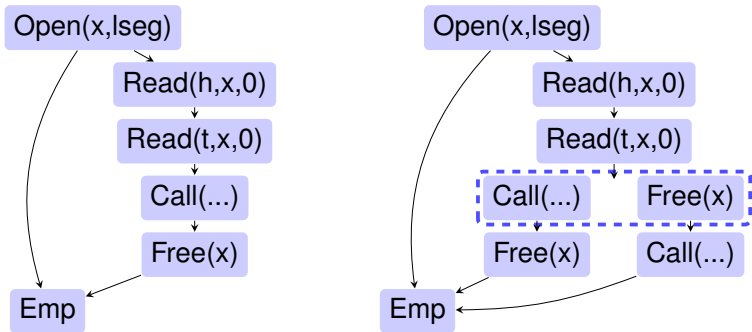
E-graphs over Proofs



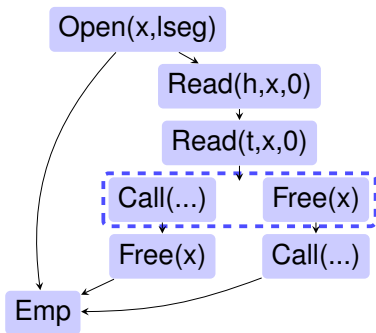
E-graphs over Proofs



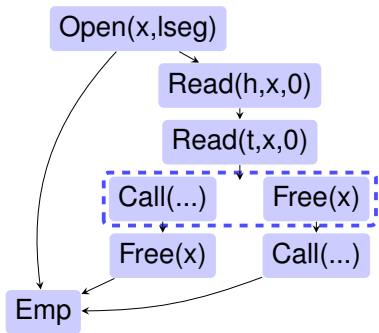
E-graphs over Proofs



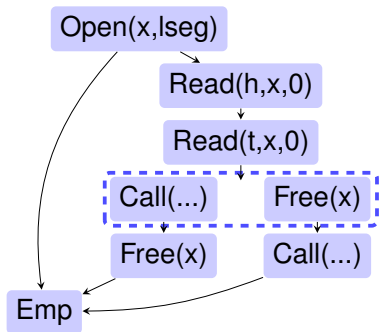
E-graphs over Proofs



E-graphs over Proofs

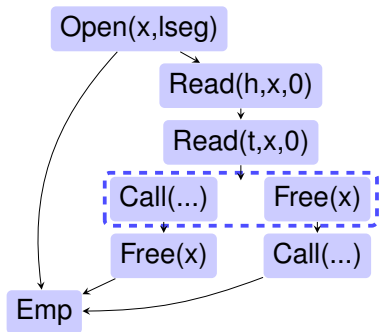


E-graphs over Proofs



```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        free(x);  
        listfree(t);  
    }  
}
```

E-graphs over Proofs



```
void listfree(loc x) {  
    if (x == 0) {  
        return;  
    } else {  
        let h = *x;  
        let t = *(x + 1);  
        free(x);  
        listfree(t);  
    }  
}
```

Done!

Challenges

Challenges

What causes problems?

Challenges




What causes problems?

...when proofs diverge from programs

Challenges

- Branch equivalence checking.
- Transposing through branches.
- Logically redundant code elimination.

Challenges

-  Branch equivalence checking.
-  Transposing through branches.
-  Logically redundant code elimination.

Challenges

```
if (C) {  
    P  
} else {  
    P  
}
```

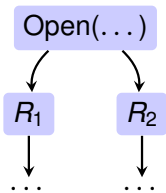
Challenges

```
if (C) {  
    P  
} else {  
    P  
}
```

P

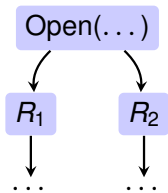
Challenges

```
if (C) {  
    P  
} else {  
    P  
}
```



Challenges

```
if (C) {  
    P  
} else {  
    P  
}
```






*May not be
syntactically
equivalent*



Challenges

- Branch equivalence checking.
- Transposing through branches.
- Logically redundant code elimination.

Challenges

-  Branch equivalence checking.
-  **Transposing through branches.**
-  Logically redundant code elimination.

Challenges

```
if (C) {  
    let y = *x;  
    P  
} else {  
    let y = *x;  
    Q  
}
```

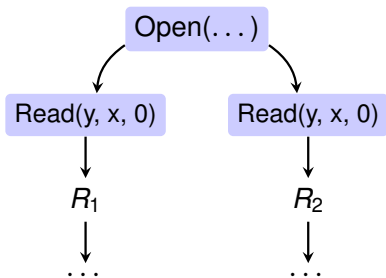
Challenges

```
if (C) {  
  let y = *x;  
  P  
} else {  
  let y = *x;  
  Q  
}
```

```
let y = *x;  
if (C) {  
  P  
} else {  
  Q  
}
```

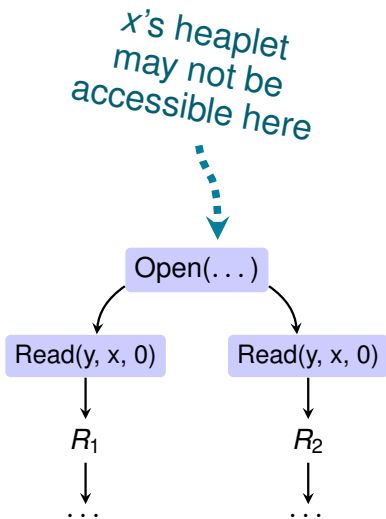
Challenges

```
if (C) {  
  let y = *x;  
  P  
} else {  
  let y = *x;  
  Q  
}
```



Challenges

```
if (C) {  
  let y = *x;  
  P  
} else {  
  let y = *x;  
  Q  
}
```



Challenges

- Branch equivalence checking.
- Transposing through branches.
- Logically redundant code elimination.

Challenges

- Branch equivalence checking.
- Transposing through branches.
- Logically redundant code elimination.

Challenges

```
let v = *rx;  
let l = *(rx + 1);  
let r = *(rx + 2);  
*(rx + 2) = l;  
*(rx + 1) = lx;  
*(x + 2) = r;  
*(x + 1) = rx;  
*rx = vx;  
*x = v;
```

Challenges

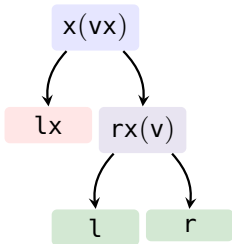
```
let v = *rx;  
let l = *(rx + 1);  
let r = *(rx + 2);  
*(rx + 2) = l;  
*(rx + 1) = lx;  
*(x + 2) = r;  
*(x + 1) = rx;  
*rx = vx;  
*x = v;
```

*Found in real
synthesized code*



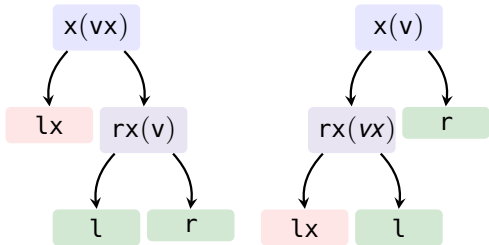
Challenges

```
let v = *rx;  
let l = *(rx + 1);  
let r = *(rx + 2);  
*(rx + 2) = l;  
*(rx + 1) = lx;  
*(x + 2) = r;  
*(x + 1) = rx;  
*rx = vx;  
*x = v;
```



Challenges

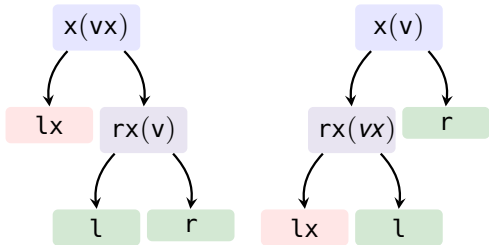
```
let v = *rx;  
let l = *(rx + 1);  
let r = *(rx + 2);  
*(rx + 2) = l;  
*(rx + 1) = lx;  
*(x + 2) = r;  
*(x + 1) = rx;  
*rx = vx;  
*x = v;
```



Challenges

How can this be justified?

```
let v = *rx;  
let l = *(rx + 1);  
let r = *(rx + 2);  
*(rx + 2) = l;  
*(rx + 1) = lx;  
*(x + 2) = r;  
*(x + 1) = rx;  
*rx = vx;  
*x = v;
```



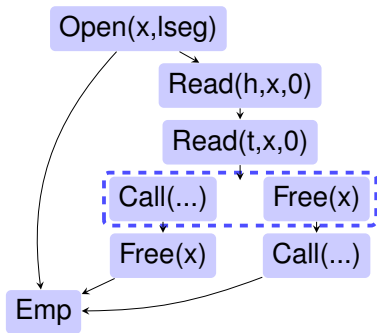
Future work

- Support other proof rewrites
- Better support of proof footprints non-det.
- Handle other languages (higher-order?)

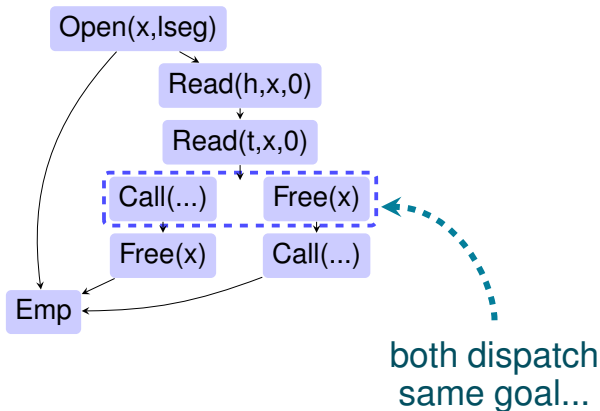
The End

E-classes over proofs

E-classes over proofs

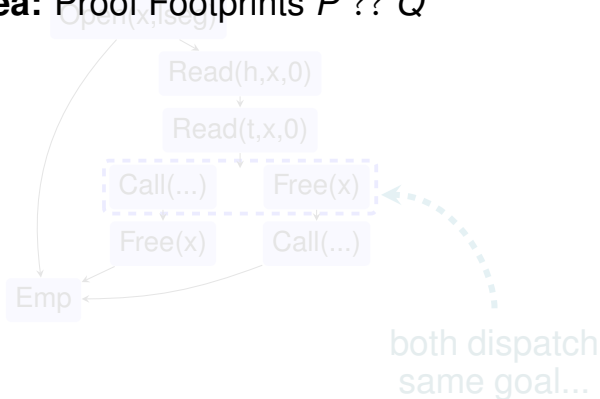


E-classes over proofs



E-classes over proofs

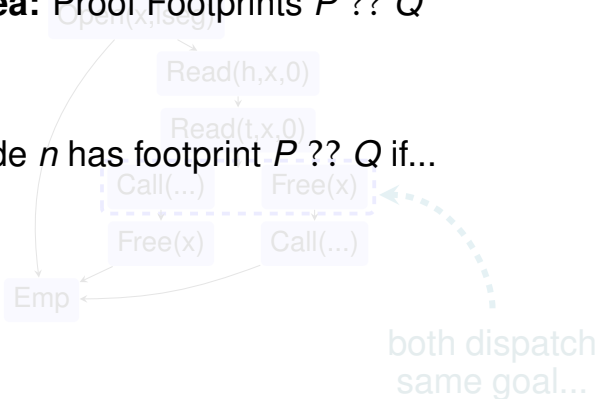
Idea: Proof Footprints P ?? Q



E-classes over proofs

Idea: Proof Footprints P ?? Q

Node n has footprint P ?? Q if...

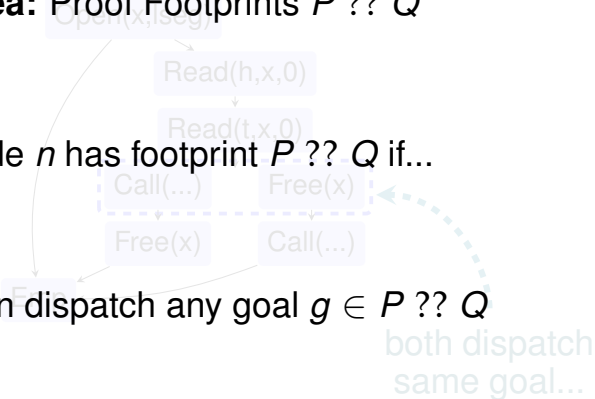


E-classes over proofs

Idea: Proof Footprints $P ?? Q$

Node n has footprint $P ?? Q$ if...

... n can dispatch any goal $g \in P ?? Q$



E-classes over proofs

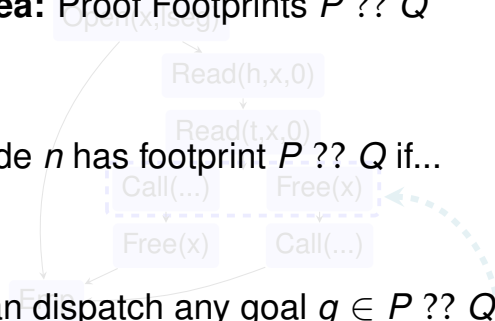
Idea: Proof Footprints P ?? Q

Node n has footprint P ?? Q if...

... n can dispatch any goal $g \in P$?? Q

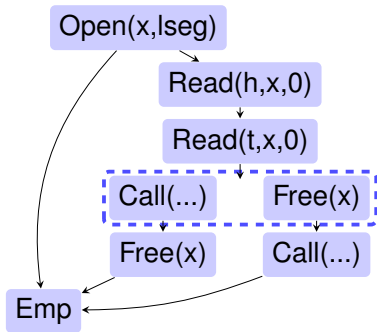
*captures
superset of goals
that can be dispatched*

*both dispatch
same goal...*



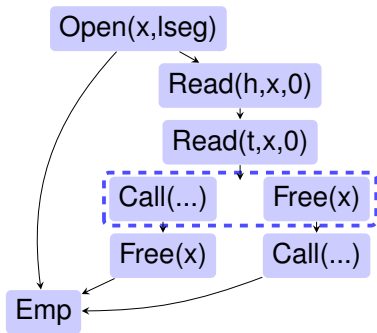
E-classes over proofs

Proof transformers



E-classes over proofs

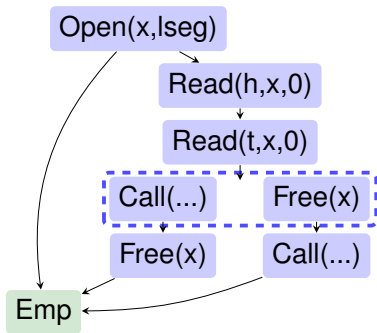
Proof transformers



Calculate proof footprints bottom-up.

E-classes over proofs

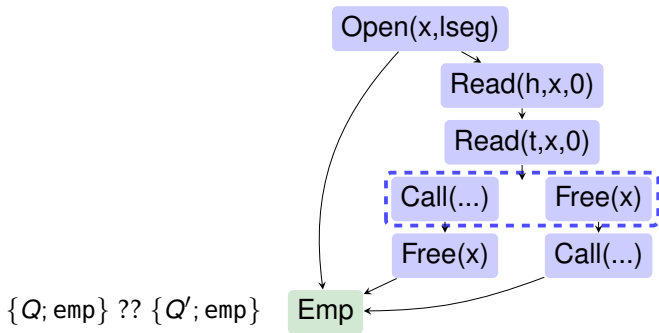
Proof transformers



Calculate proof footprints bottom-up.

E-classes over proofs

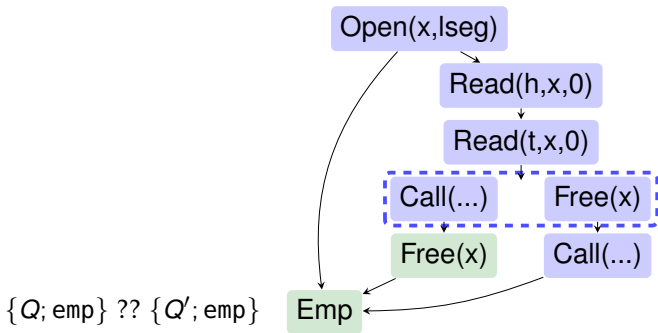
Proof transformers



Calculate proof footprints bottom-up.

E-classes over proofs

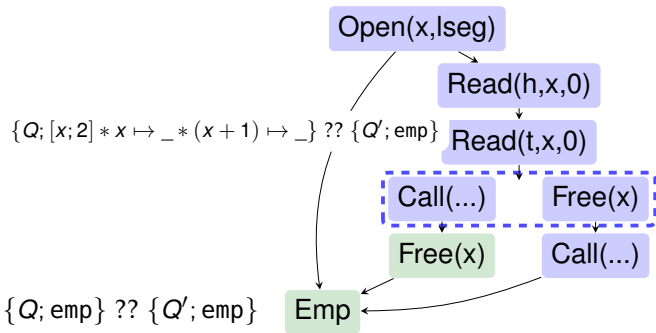
Proof transformers



“Invert” execution of rules

E-classes over proofs

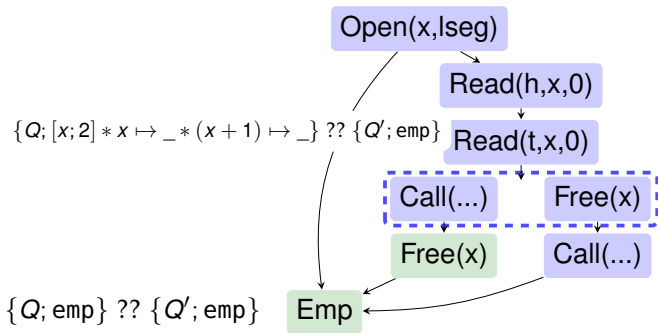
Proof transformers



“Invert” execution of rules

E-classes over proofs

Proof transformers



Provides **deeper** analysis of proofs...