Do compilers respect programmers?

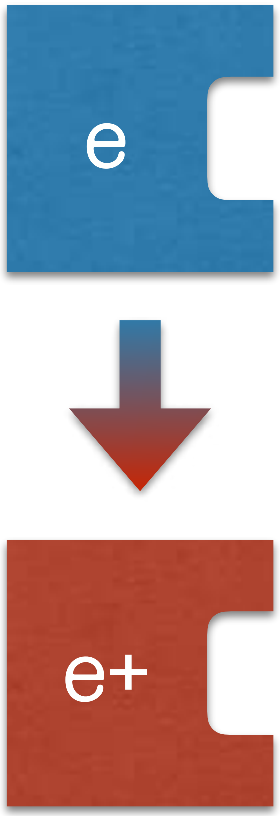
William J. Bowman
https://williamjbowman.com
Northeastern University, CCIS

Do compilers respect programmer *invariants*?

William J. Bowman
https://williamjbowman.com
Northeastern University, CCIS

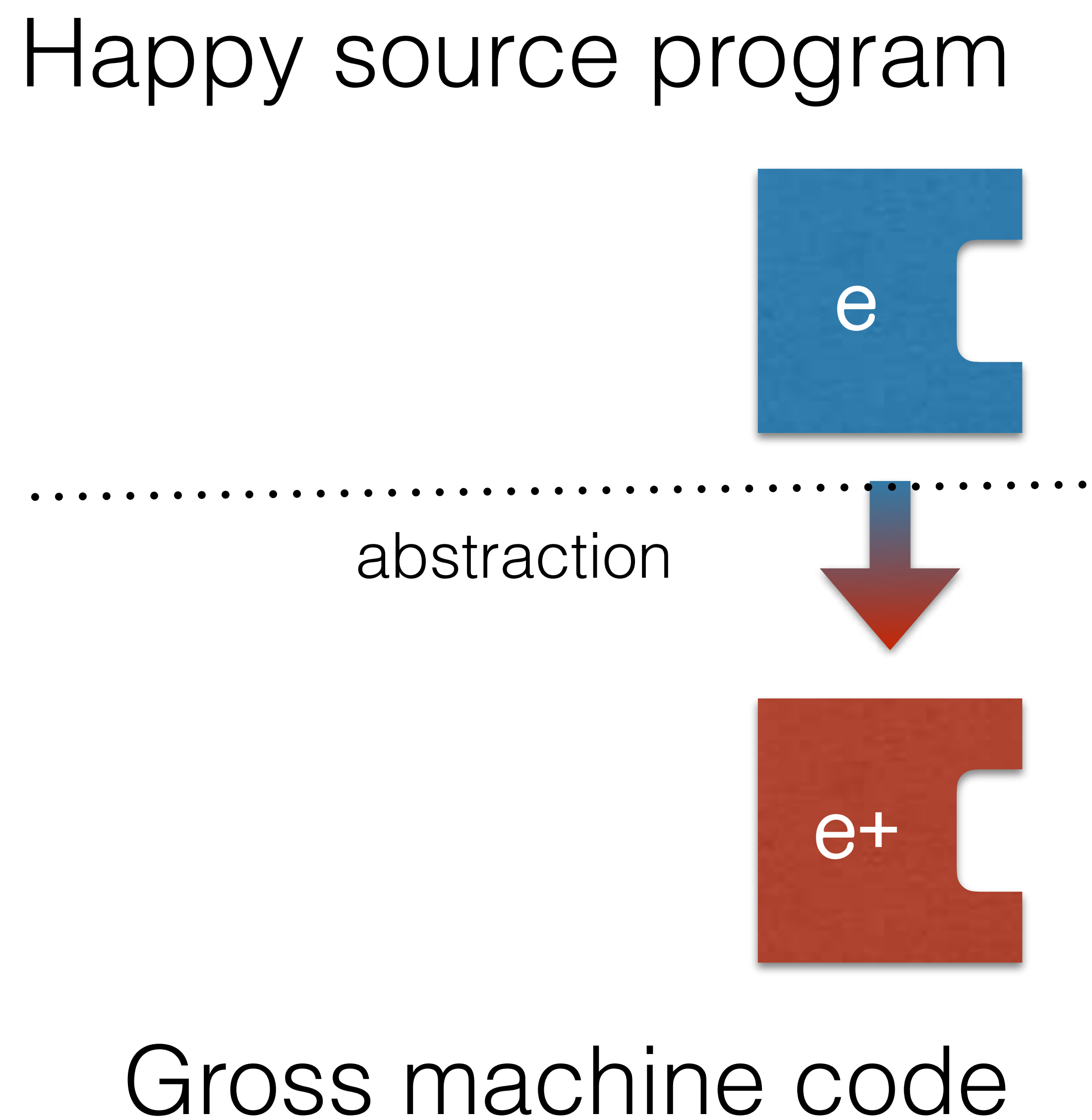
Compilers have one job

Happy source program



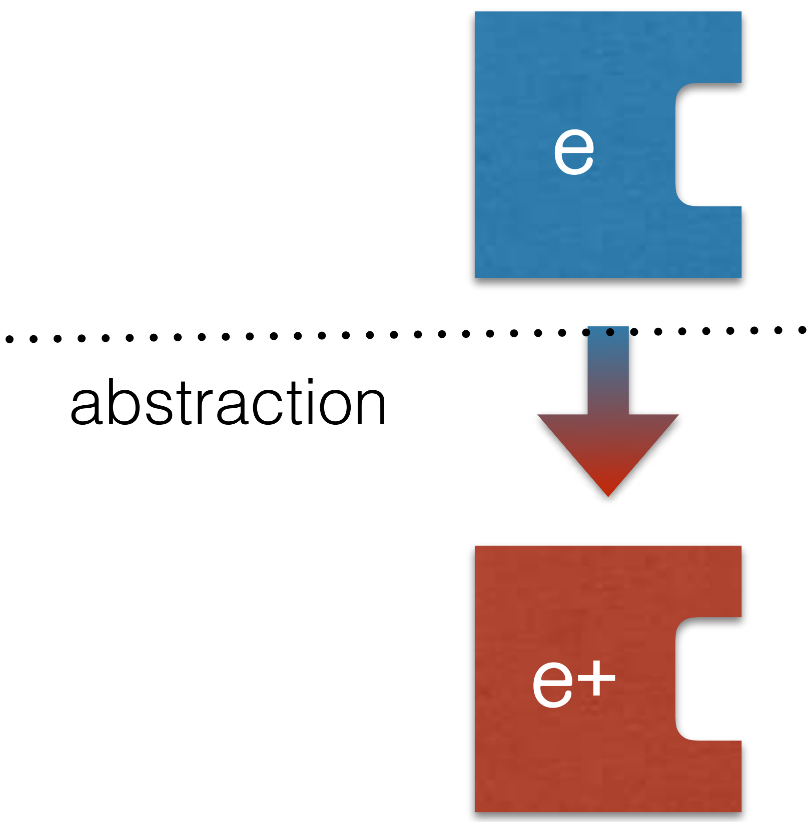
Gross machine code

Compilers have one job



Compilers have one job

Happy source program

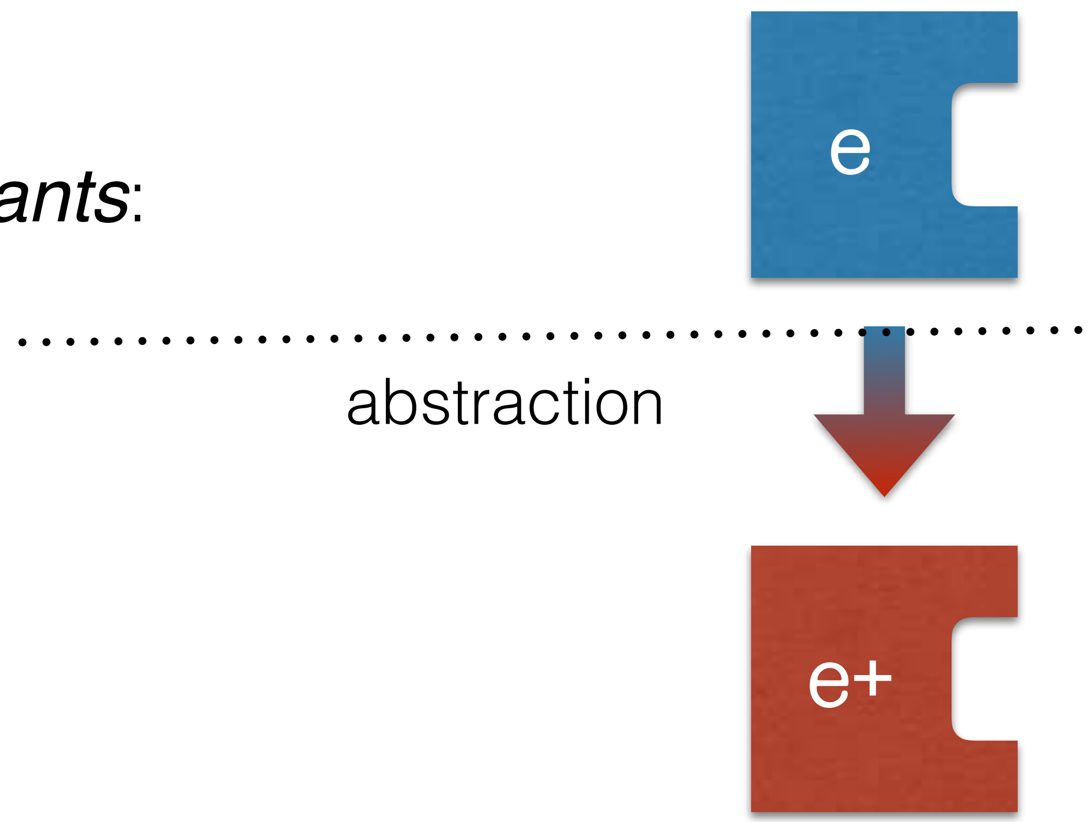


Gross machine code

Abstraction helps programmers design *invariants*: logical assertions that should always hold

Compilers have one job

Happy source program

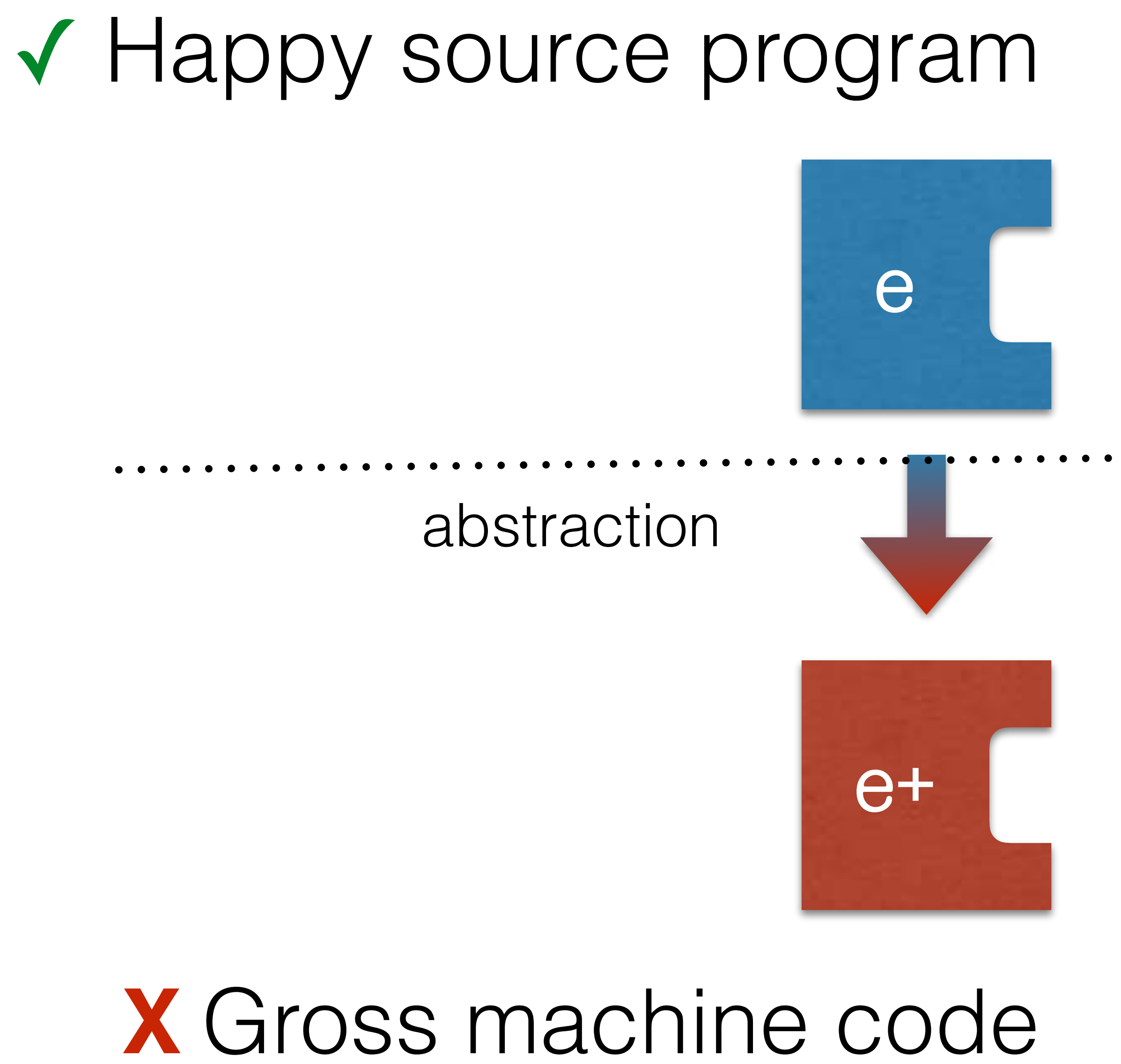


Gross machine code

Abstraction helps programmers design *invariants*: logical assertions that should always hold

- Performance
- Correctness
- Security

and compilers *don't do it*



Three key problems

Often,

1. Programmers can't express invariants

2. Compilers don't respect invariants

3. Linkers don't enforce invariants

Three key problems

Often,

1. Programmers can't express invariants
2. Compilers don't respect invariants
3. Linkers don't enforce invariants

"invariant":
Belief or intent

Three key problems

Often,

1. Programmers $\boxed{\text{can't express invariants}}$
2. Compilers $\boxed{\text{don't respect invariants}}$
3. Linkers don't enforce invariants

"invariant":
Belief or intent

In:
- Optimization
- Code generation

Three key problems

Often,

1. Programmers can't express invariants
2. Compilers don't respect invariants
3. Linkers don't enforce invariants

"invariant":
Belief or intent

In:
- Optimization
- Code generation

On external code,
violating intent

Examples of Problem 1

Often,

1. Programmers can't express invariants

“invariant”:
Belief or intent

2. Compilers don't respect invariants

In:
- Optimization
- Code generation

3. Linkers don't enforce invariants

On external code,
violating intent

Programmers can’t express invariants

Example: Mars Climate Orbiter

“Invariant”	Ground control calculated *Impulse* in ***newton seconds***
Actually	*Impulse* calculated in ***pound-force seconds***
Behavior	

Programmers can't express invariants

Example: Mars Climate Orbiter

"Invariant"	Ground control calculated *Impulse* in ***newton seconds***
Actually	*Impulse* calculated in ***pound-force seconds***
Behavior	Unscheduled disintegration

Programmers can't express invariants

Example: Ariana 501

"Invariant"	Casts from 64-bit floating point to 16-bit integers do not overflow (and are fast)
Actually	One did, involving position and acceleration
Behavior	

Programmers can’t express invariants

Example: Ariana 501

“Invariant”	Casts from 64-bit floating point to 16-bit integers do not overflow (and are fast)
Actually	One did, involving position and acceleration
Behavior	

Programmers can’t express invariants

Solutions:

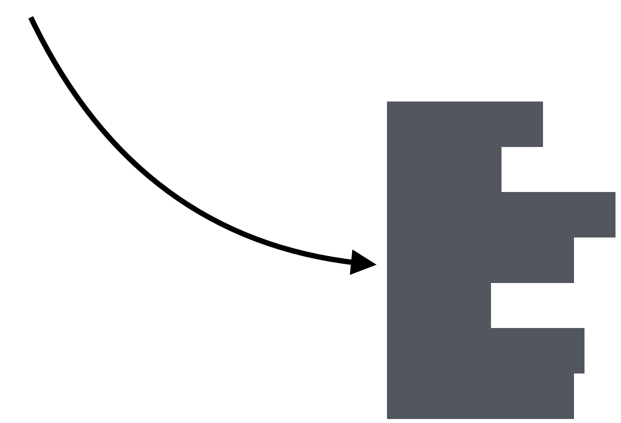
- Types
- Formal methods
- Analyses

Programmers can’t express invariants

Solutions:

- Types
- Formal methods
- Analyses

Possible overflow



Programmers can't express invariants

Solutions:

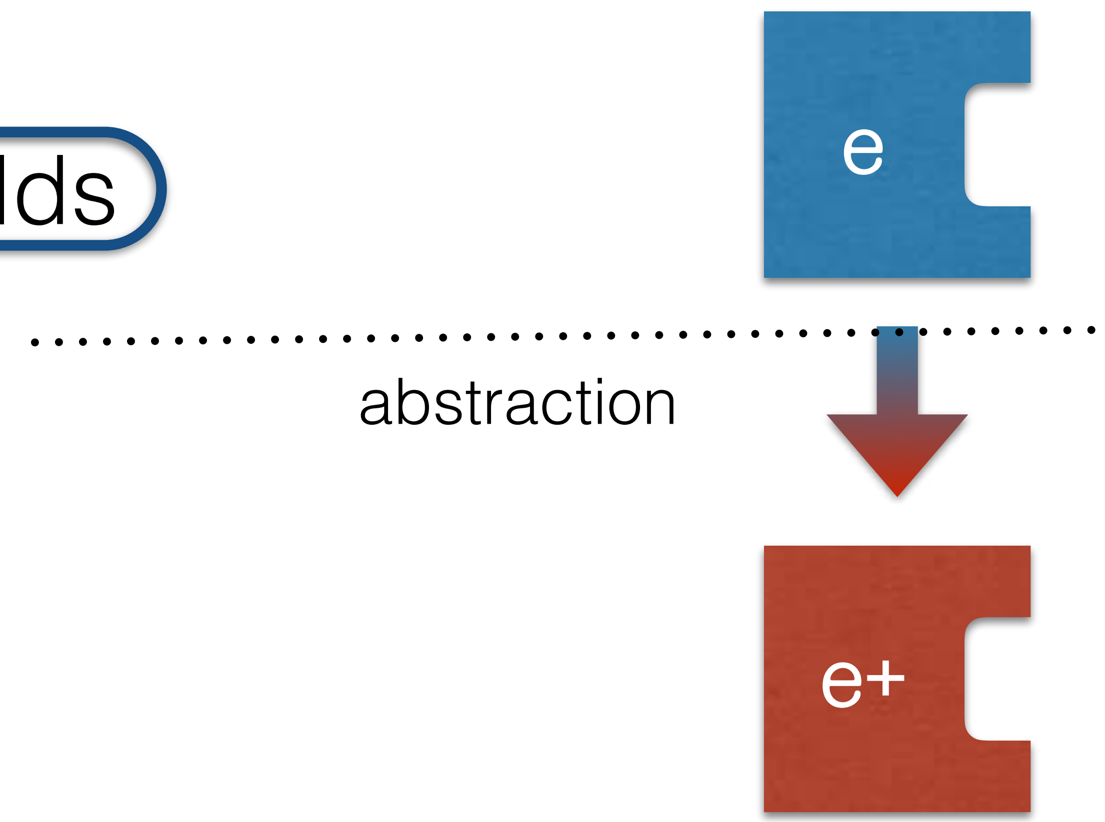
- Types
- Formal methods
- Analyses

```
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```

Supposing we *can* express invariants

✓ Happy source program

Definitely, for sure, absolutely holds



Examples of Problem 2

Often,

1. Programmers can't express invariants

   "invariant":
   Belief or intent

2. Compilers don't respect invariants

   In:
   - Optimization
   - Code generation

3. Linkers don't enforce invariants

   On external code,
   violating intent

Compilers don't respect invariants

Example: scrubbing secret memory

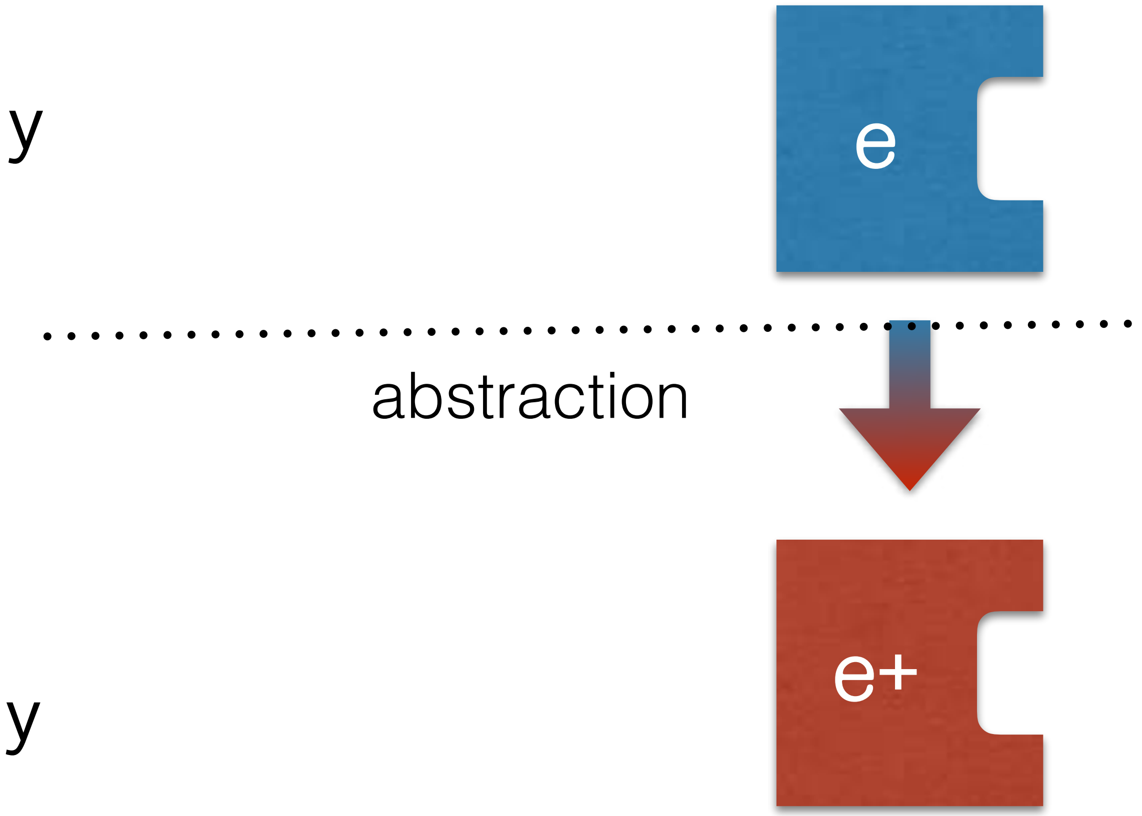
```
crypt(){
  key = 0xC0DE; // read key
  ... // work with the secret key
  key = 0x0;    // scrub memory
}
```

Compilers don't respect invariants

Example: scrubbing secret memory

```
crypt(){
  key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;     // scrub memory
}
```

```
crypt(){
  key = 0xC0DE; // read key
  ... // work with the secure key

}
```



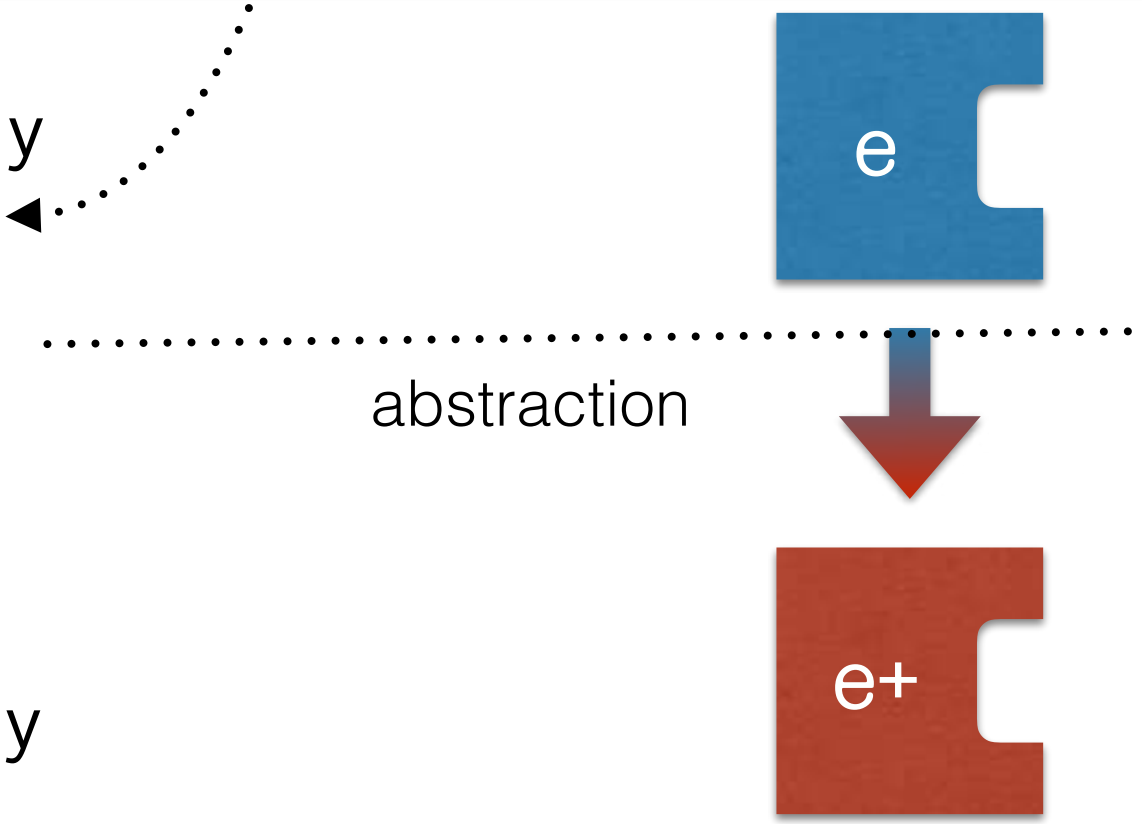
Compilers don’t respect invariants

Example: scrubbing secret memory

Performance invariant says this must go!

```
crypt(){
  key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;     // scrub memory
}
```

```
crypt(){
  key = 0xC0DE; // read key
  ... // work with the secure key

}
```

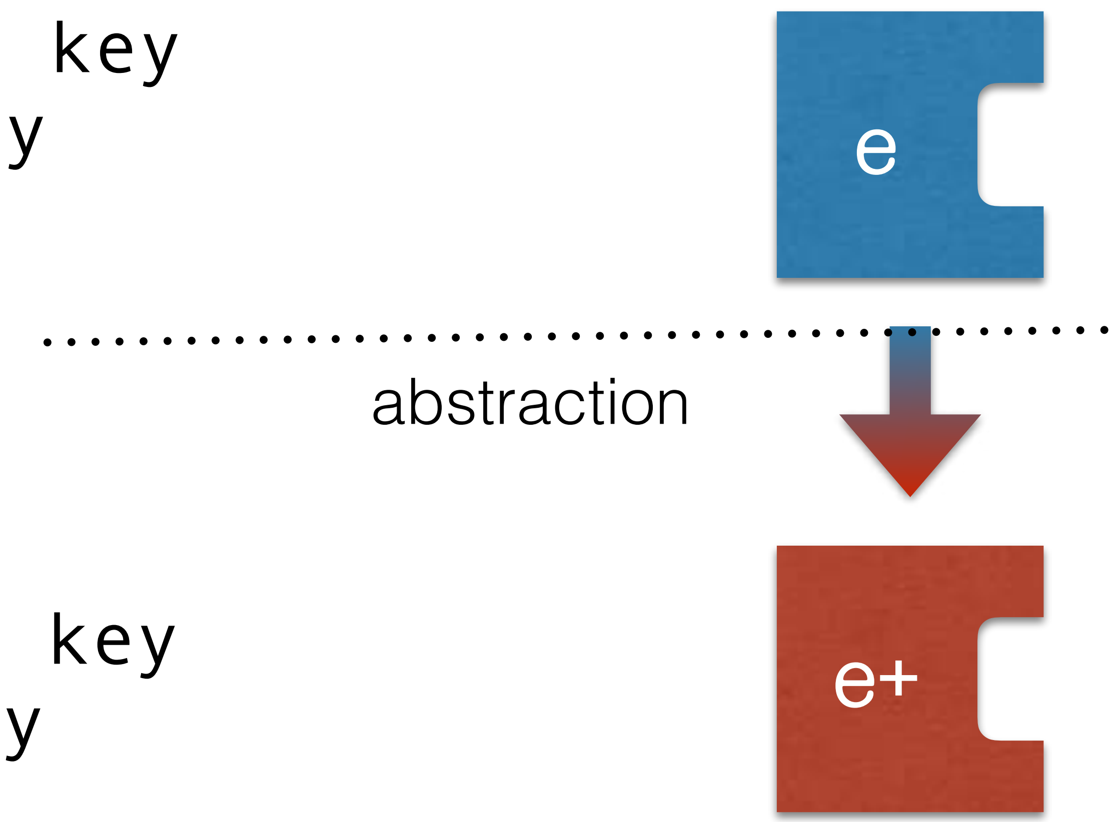


Compilers don't respect invariants

Example: scrubbing secret memory

```
crypt(){
  volatile key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;      // scrub memory
}
```

```
crypt(){
  volatile key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;      // scrub memory
}
```

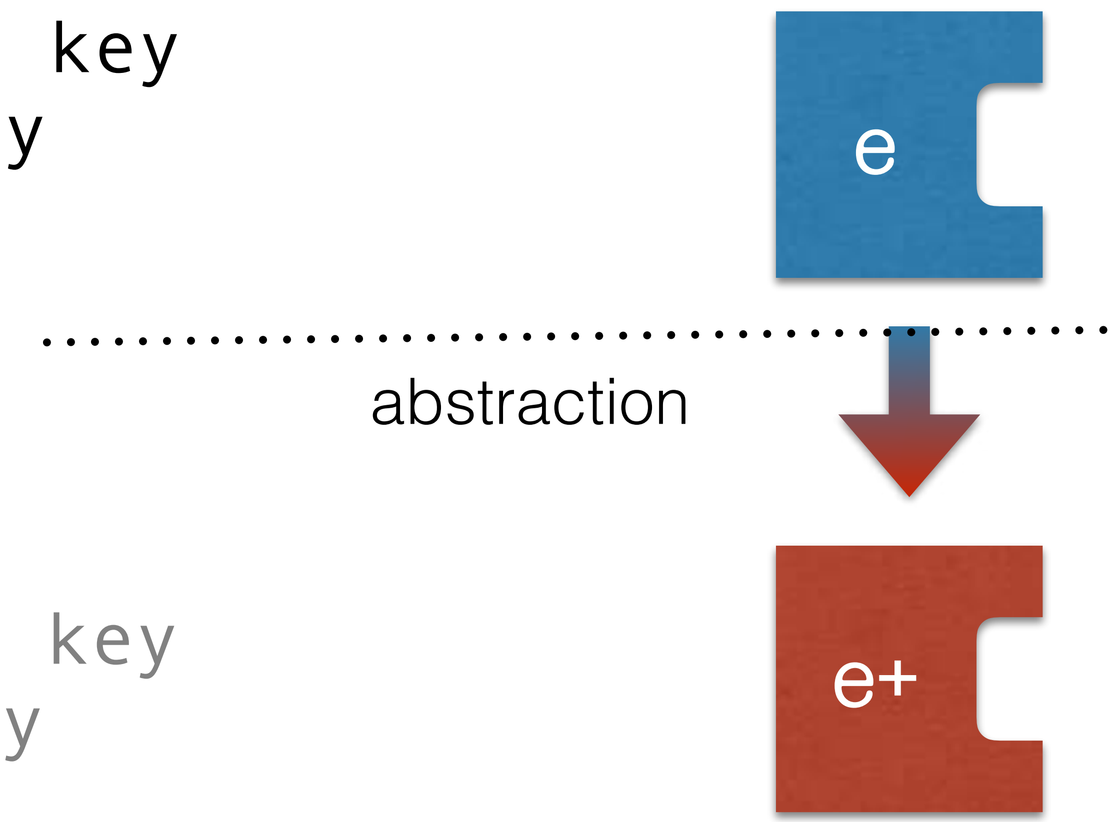


Compilers don’t respect invariants

Example: scrubbing secret memory

```
crypt(){
  volatile key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;     // scrub memory
}
```

```
crypt(){
  volatile key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;     // scrub memory
}
```

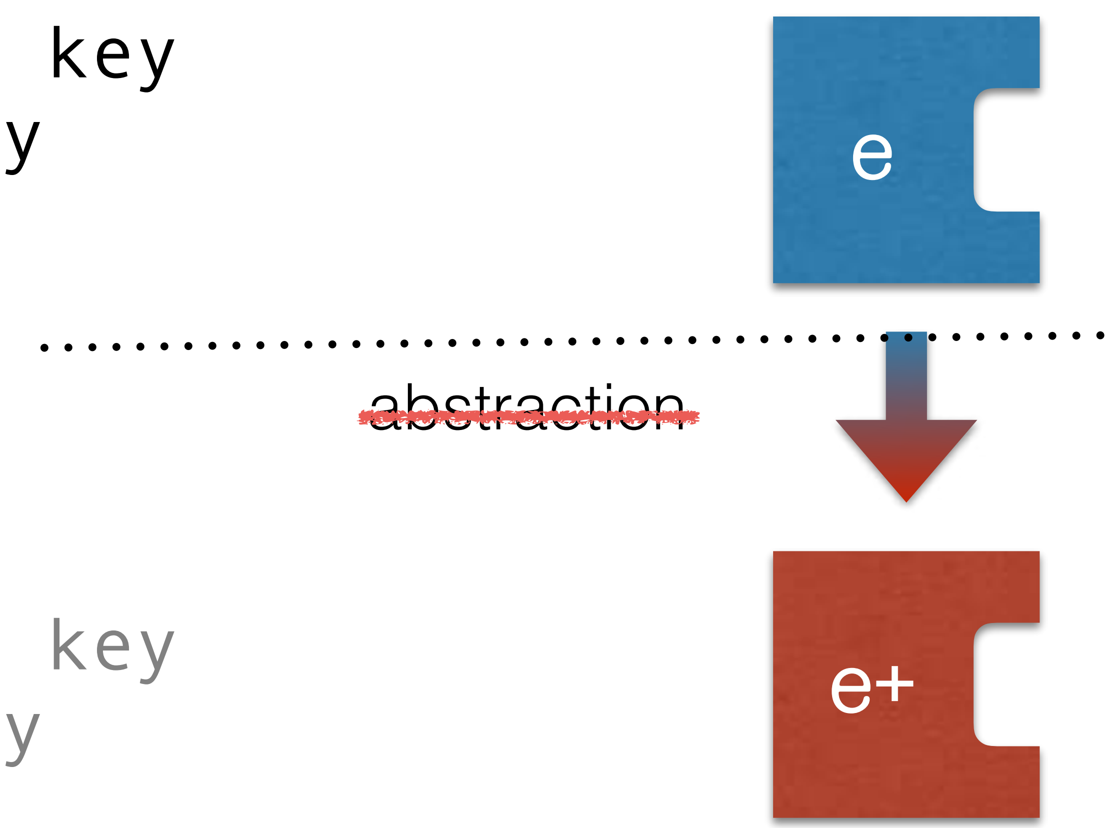


Compilers don’t respect invariants

Example: scrubbing secret memory

“Invariant”	volatile writes aren’t optimized away
Actually	Depends on the compiler
Behavior	Undefined

```
crypt(){
  volatile key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;     // scrub memory
}
```



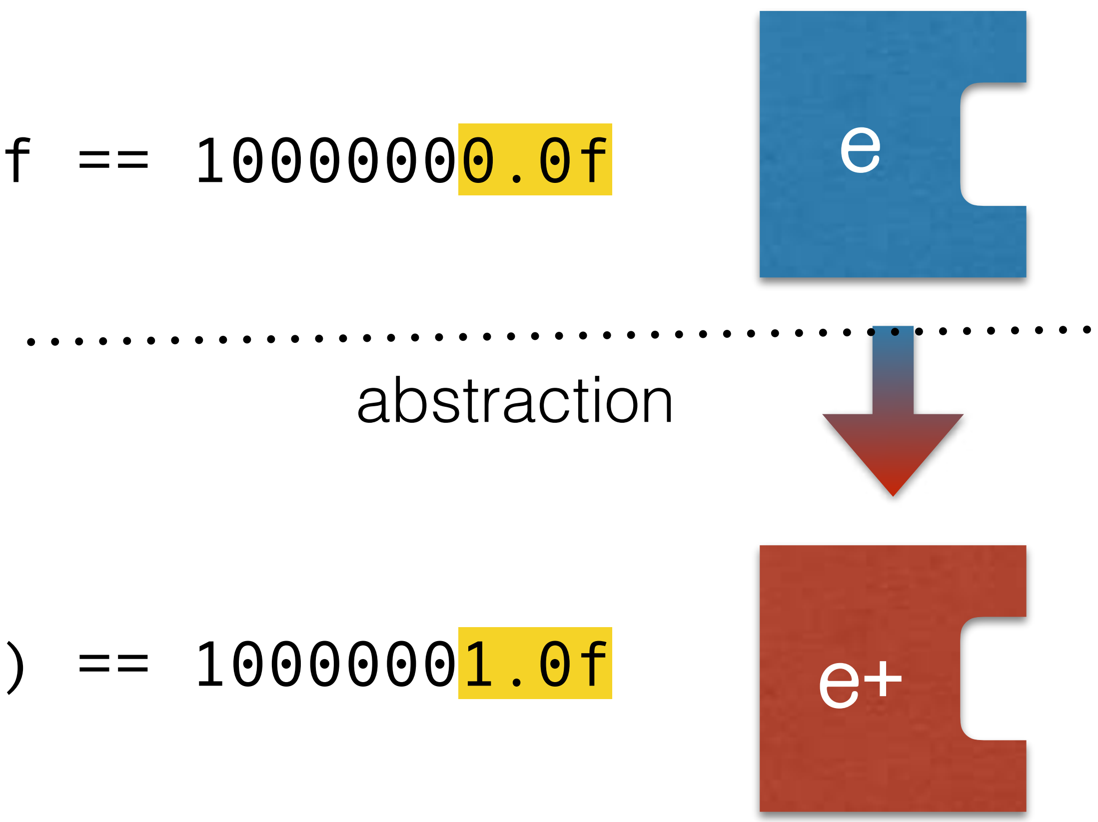

```
crypt(){
  volatile key = 0xC0DE; // read key
  ... // work with the secure key
  key = 0x0;     // scrub memory
}
```

Compilers don't respect invariants

Example: floating point precision

```
(10000001.0f * 10000001.0f) / 10000001.0f == 10000000.0f
```

```
10000001.0f * (10000001.0f / 10000001.0f) == 10000001.0f
```



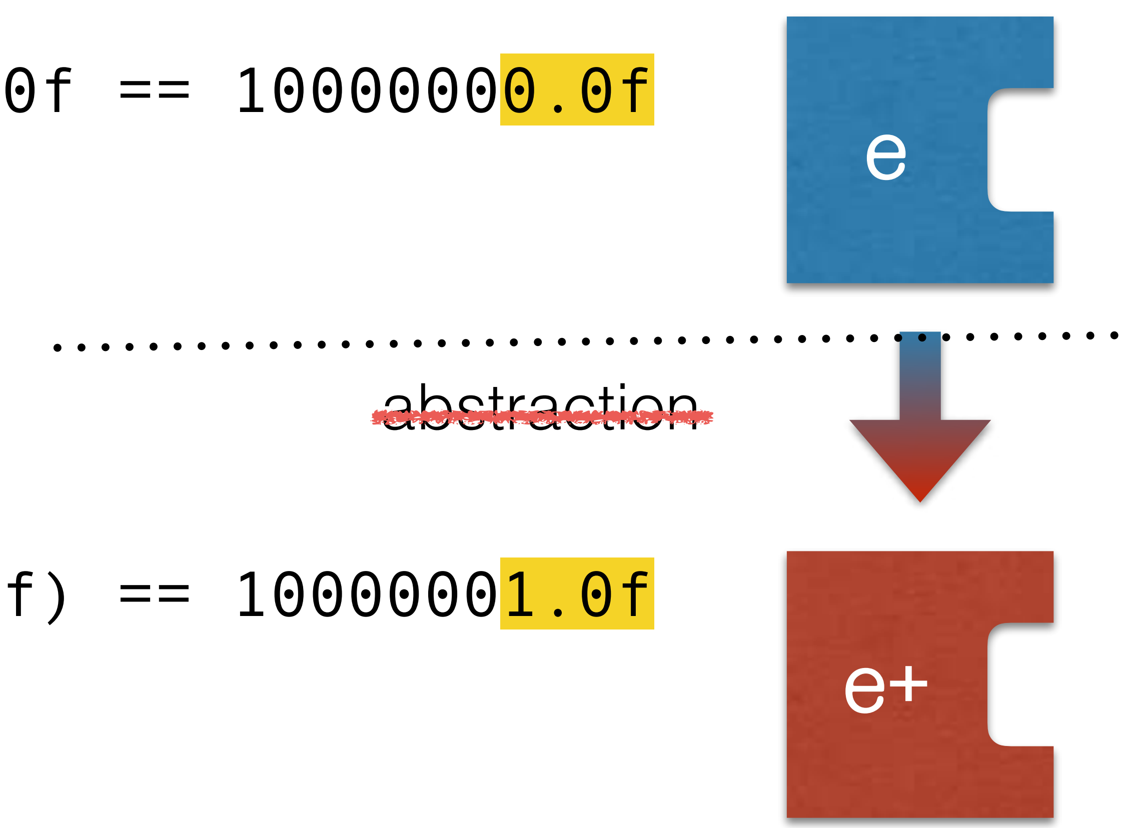
Compilers don’t respect invariants

Example: floating point precision

“Invariant”	Compilation doesn’t change floating point precision
Actually	Depends on the compiler, and the flags
Behavior	Undefined

```
(10000001.0f * 10000001.0f) / 10000001.0f == 10000000.0f
```

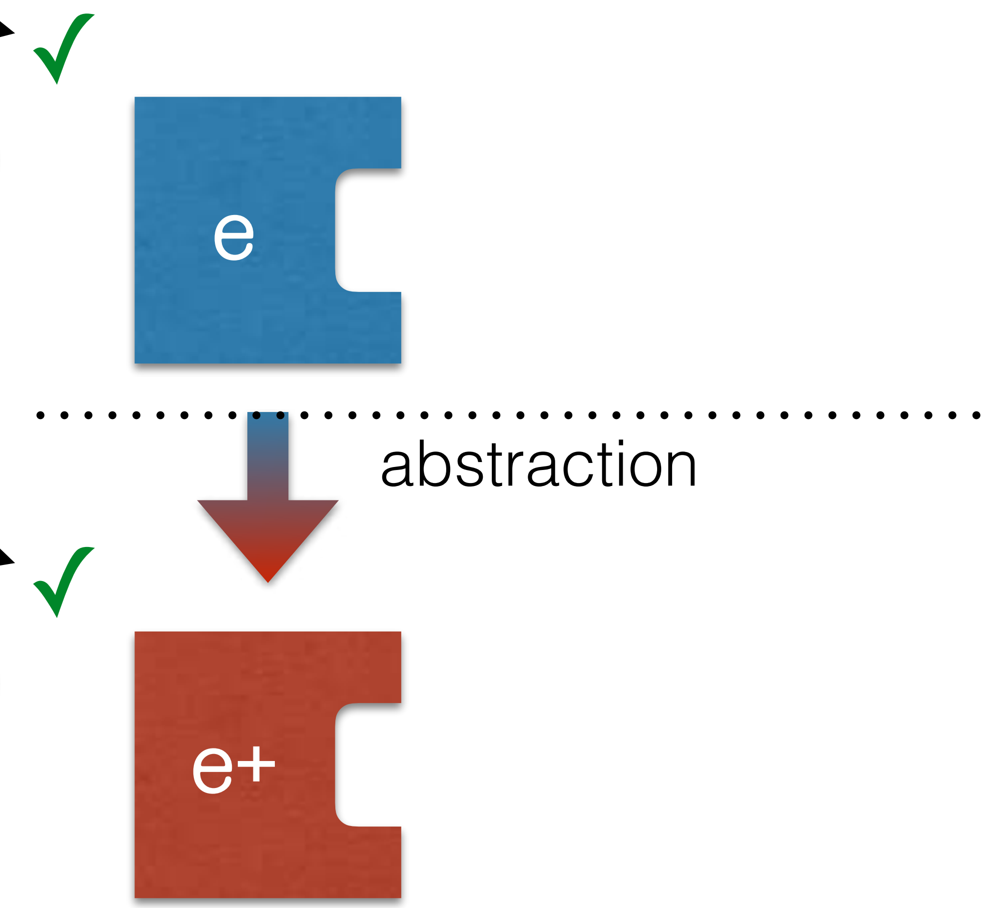
```
10000001.0f * (10000001.0f / 10000001.0f) == 10000001.0f
```



Supposing we *can preserve* invariants

Definitely, for sure, absolutely holds

Definitely, for sure, still holds



Examples of Problem 3

Often,

1. Programmers can't express invariants

   “invariant”:
   Belief or intent

2. Compilers don't respect invariants

   In:
   - Optimization
   - Code generation

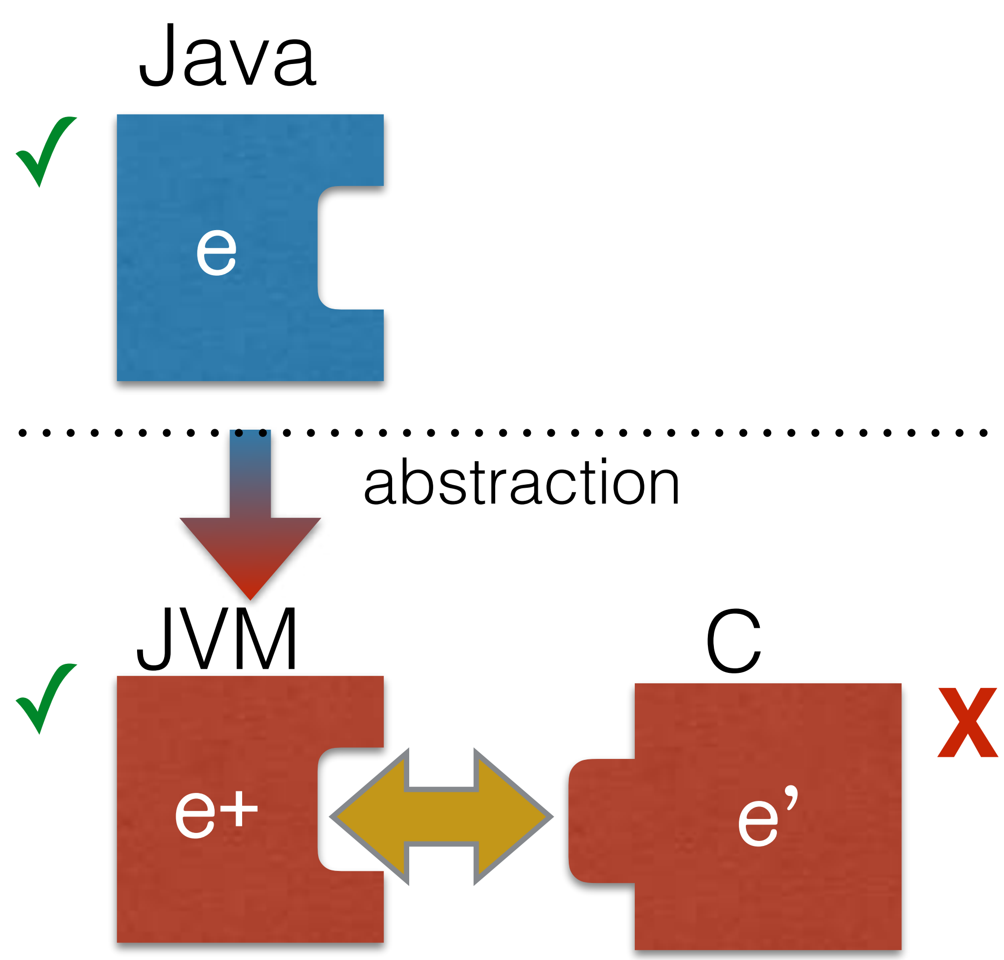
3. Linkers don't enforce invariants

   On external code,
   violating intent

Linkers don't enforce invariants

Example: Java isn't memory safe

"Invariant"	All Java programs are memory safe
Actually	… unless you use the JNI to link with native code
Behavior	Undefined



Linkers don't enforce invariants

Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

Linkers don’t enforce invariants

Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml
[1]    43185 segmentation fault (core dumped)
ocaml verified.ml
```

Linkers don't enforce invariants

Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml
[1]    43185 segmentation fault (core dumped)
ocaml verified.ml
```

Linkers don't enforce invariants

Example: Coq program goes wrong

Coq: proof assistant for writing high-assurance software with machine-checked proofs.

```
> coqc verified.v
```

1 line file

```
> link verified.ml unverified.ml

> ocaml verified.ml
[1]    43185 segmentation fault (core dumped)
ocaml verified.ml
```

Linkers don't enforce invariants

Example: Coq program goes wrong

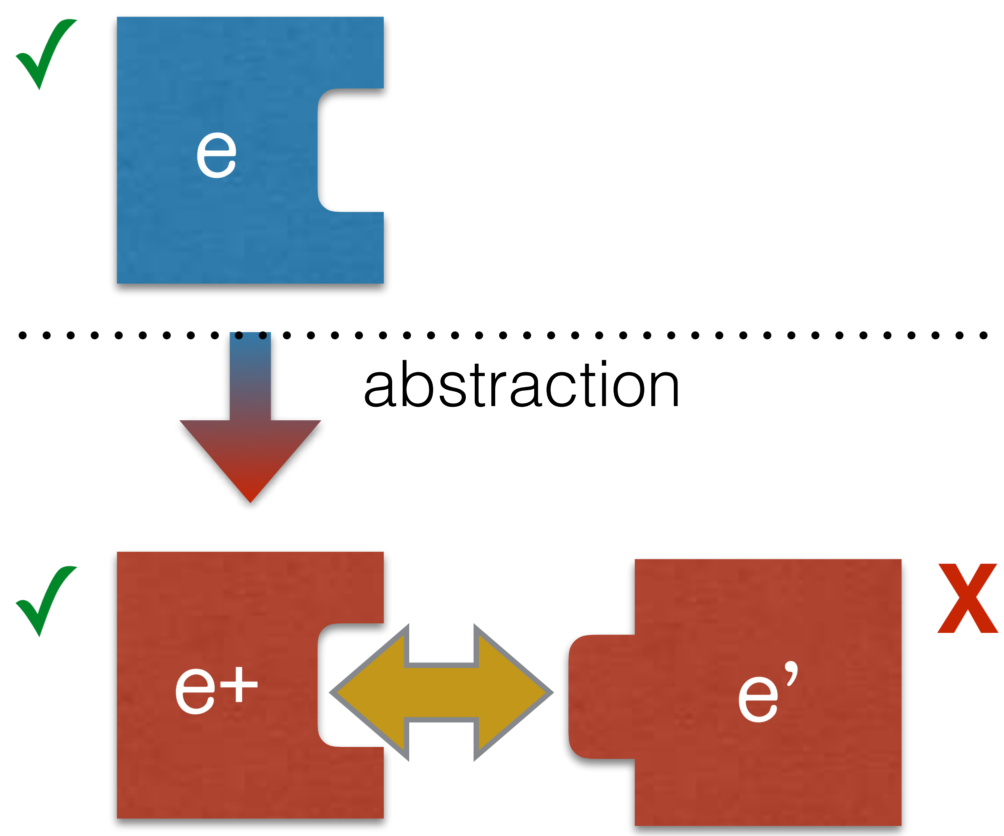
Coq: proof assistant for writing high-assurance software with machine-checked proofs.

```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml
[1]    43185 segmentation fault (core dumped)
ocaml verified.ml
```

Linkers don’t enforce invariants

Example: Coq program goes wrong

“Invariant”	Coq program verified to be free of out-of-bounds dereferences
Actually	… unless you link with anything
Behavior	Jumps to arbitrary location in memory



What is the behavior when an “invariant” isn’t?

What is the behavior when an “invariant” isn’t?

Bugs

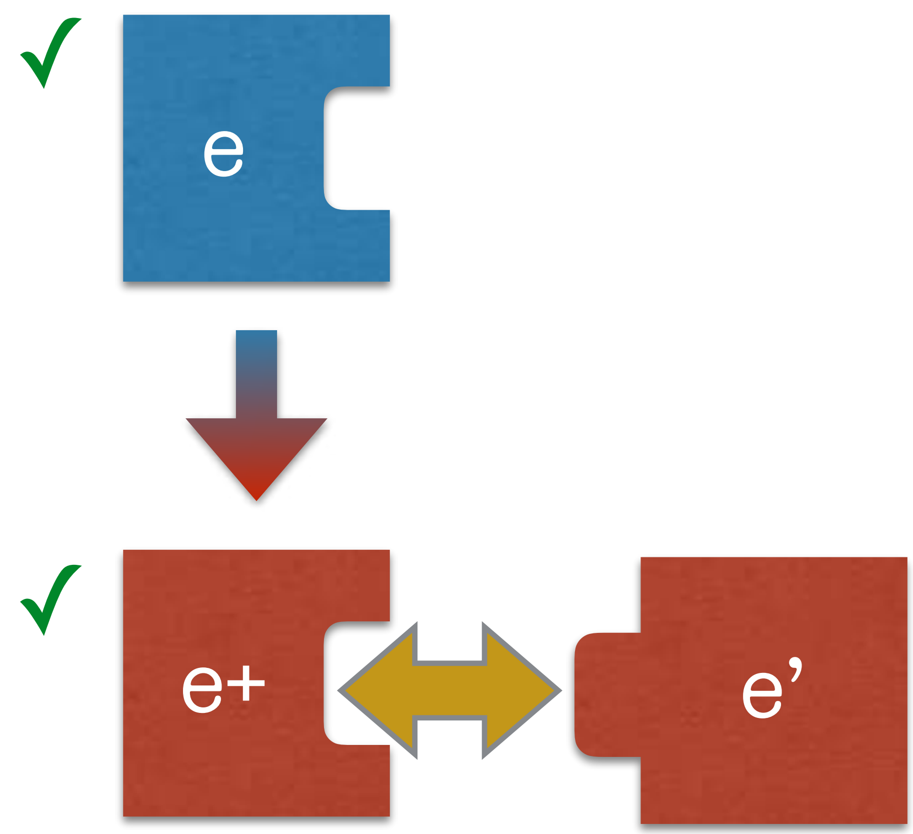
Security Attacks

Undefined Behavior

Goal of my work

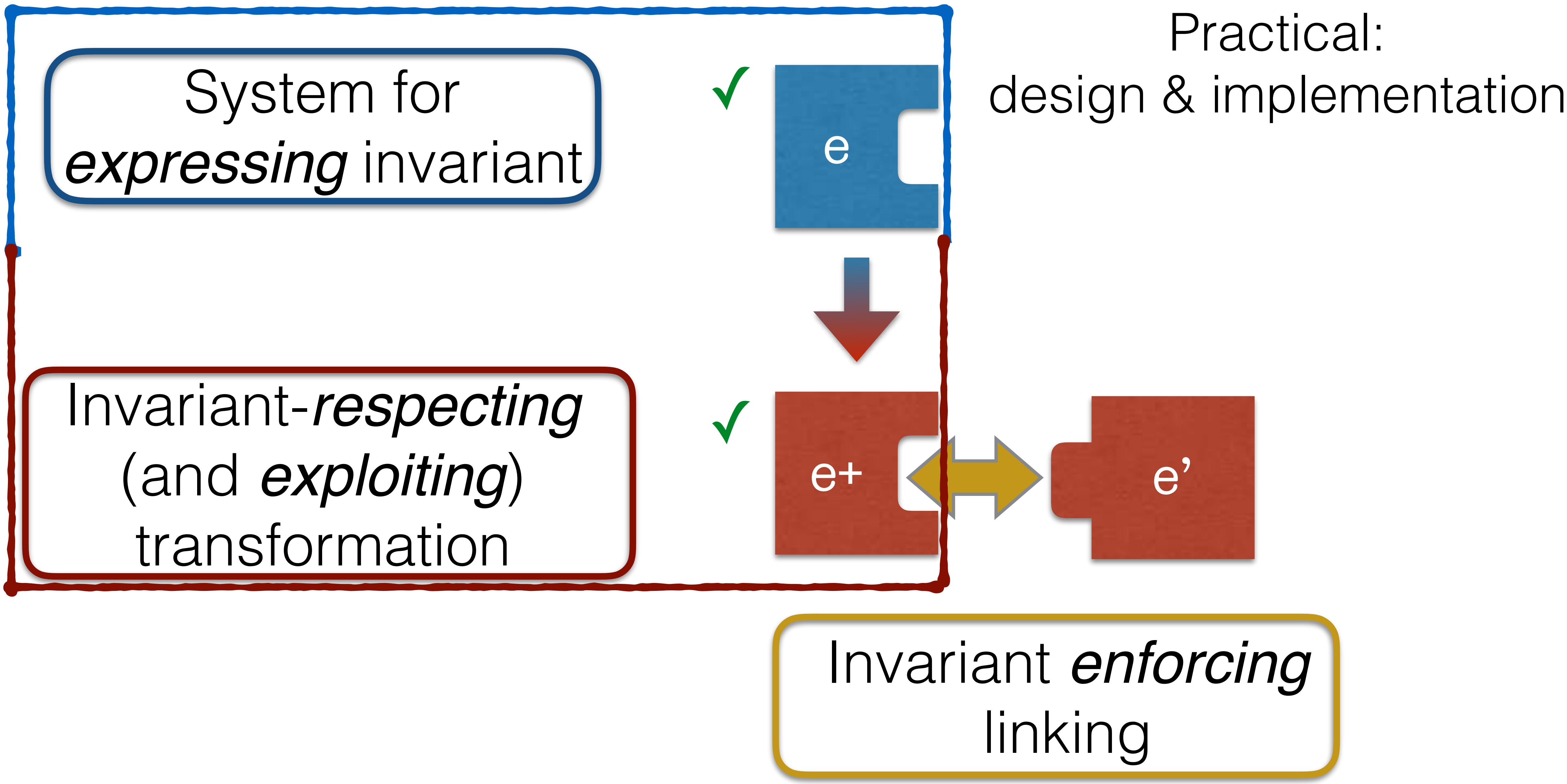
System for *expressing* invariant

Invariant-*respecting* (and *exploiting*) transformation

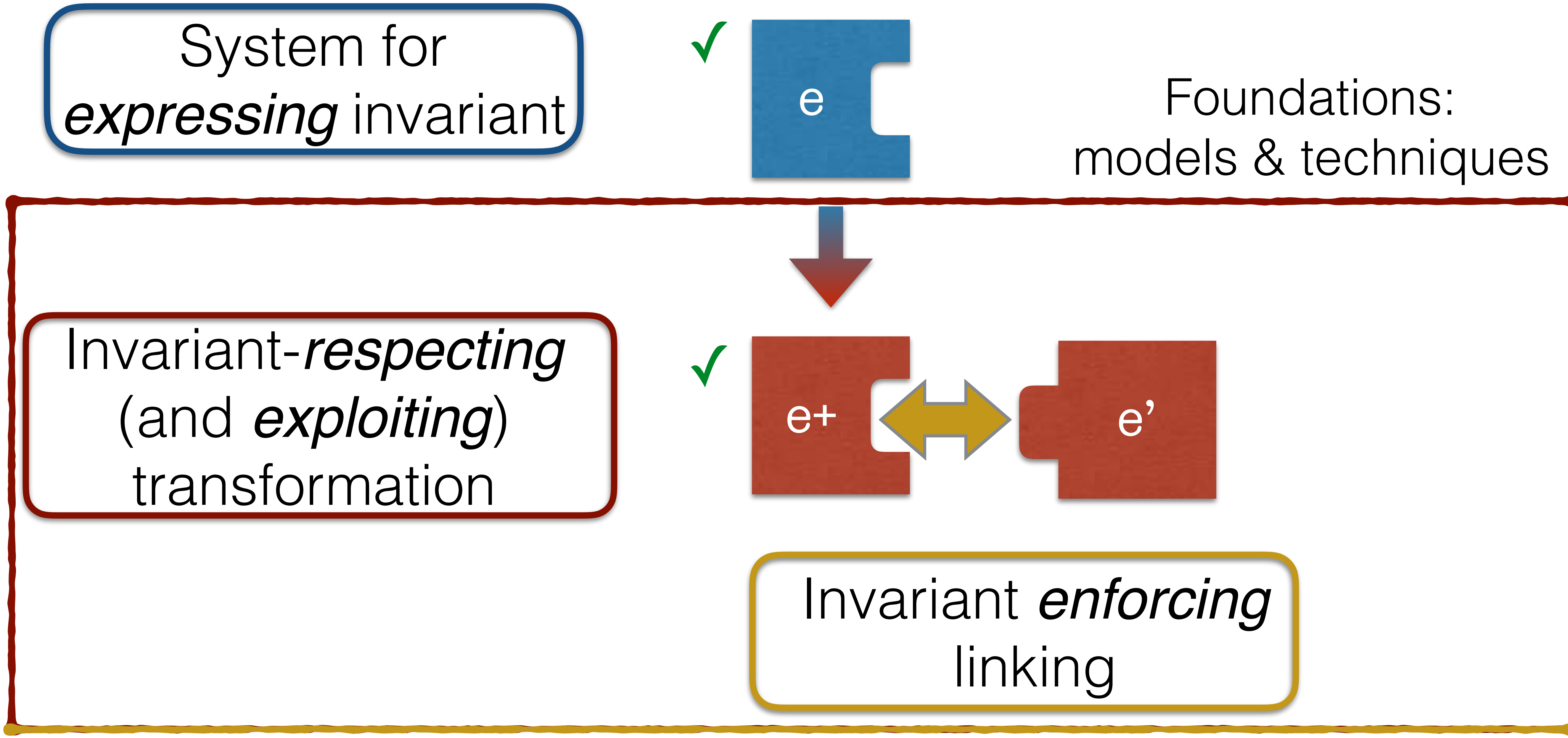


Invariant *enforcing* linking

My work so far



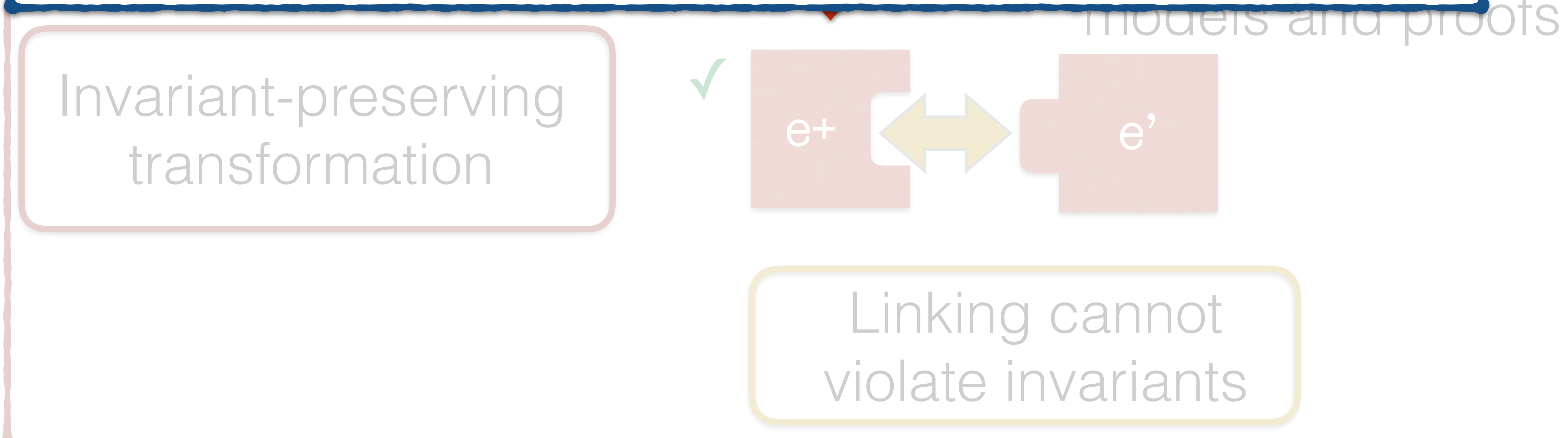
My work so far



Building systems to express and exploit invariants:

How do we express and exploit domain-specific performance invariants?

Bowman, Miller, St-Amour, Dybvig. *Profile-Guided Meta-Programming*. PLDI 2015.



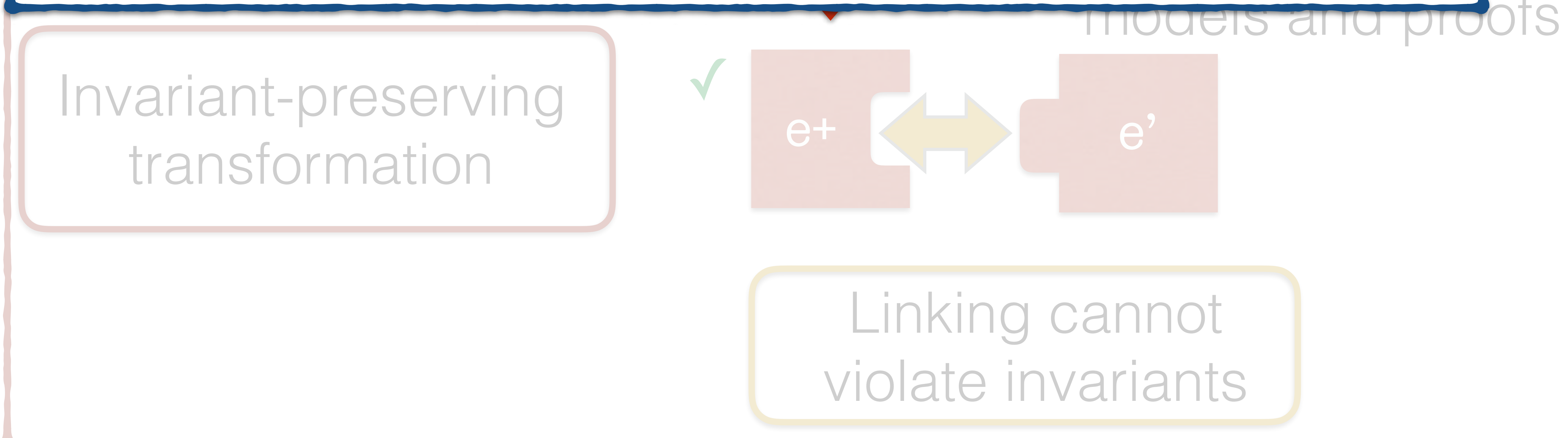
Building systems to express and exploit invariants:

How do we express and exploit domain-specific performance invariants?

Bowman, Miller, St-Amour, Dybvig. *Profile-Guided Meta-Programming*. PLDI 2015.

How do we more easily express correctness/safety invariants and proofs?

Bowman. *Growing a Proof Assistant.* HOPE 2016.



System for expressing invariant

√

e

Practical: design and impl.

Foundations for preserving and enforcing invariants:

System for expressing invariant

✓ e

Practical: design and impl.

Foundations for preserving and enforcing invariants:

How do we preserve and enforce ***language-level invariants (language specs)***?

Bowman, Ahmed. *Noninterference for free.* ICFP 2015.

New, **Bowman**, Ahmed. *Fully Abstract Compilation via Universal Embedding*. ICFP 2016.

System for expressing invariant

✓ e

Practical: design and impl.

Foundations for preserving and enforcing invariants:

How do we preserve and enforce ***language-level invariants (language specs)***?

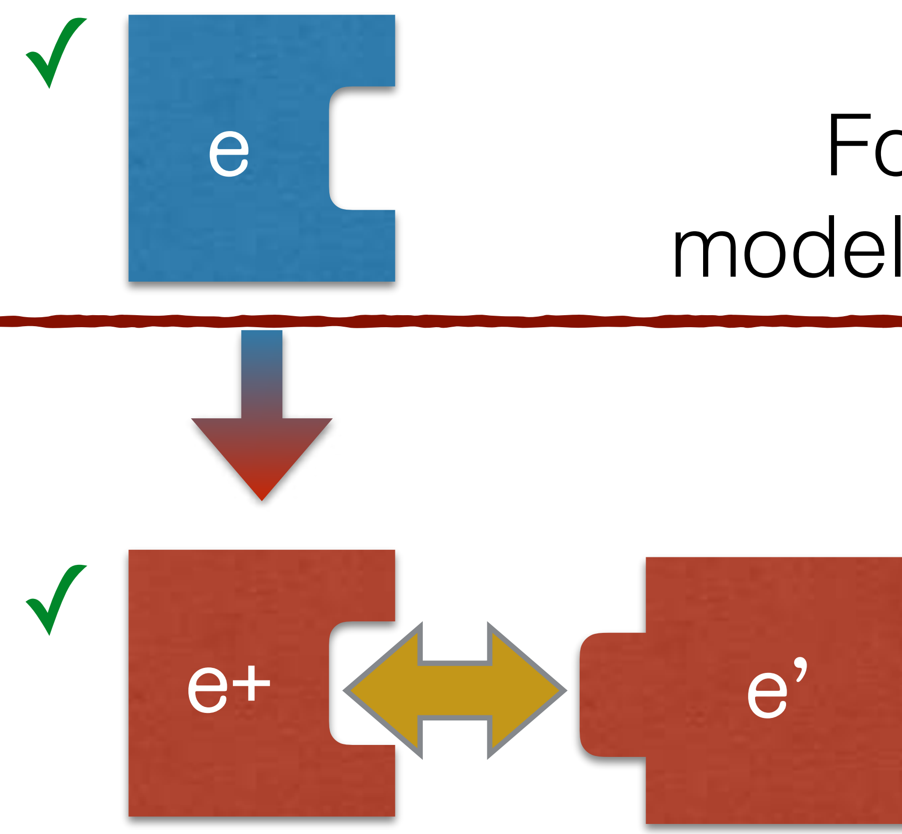
Bowman, Ahmed. *Noninterference for free.* ICFP 2015.
New, **Bowman**, Ahmed. *Fully Abstract Compilation via Universal Embedding*. ICFP 2016.

How do we preserve and enforce ***invariants encoded in dependent types***?

Bowman, Cong, Rioux, Ahmed. *Type-Preserving CPS … is Not Not Possible.* POPL 2018.
Bowman, Ahmed. *Typed Closure Conversion for the Calculus of Constr.* PLDI 2018.

Preserving and Enforcing Invariants

System for *expressing* invariant



Foundations:
models & techniques

Invariant-*respecting* (and *exploiting*) transformation

Invariant *enforcing* linking

Preserving Dependent Types

Dependent types

Coq

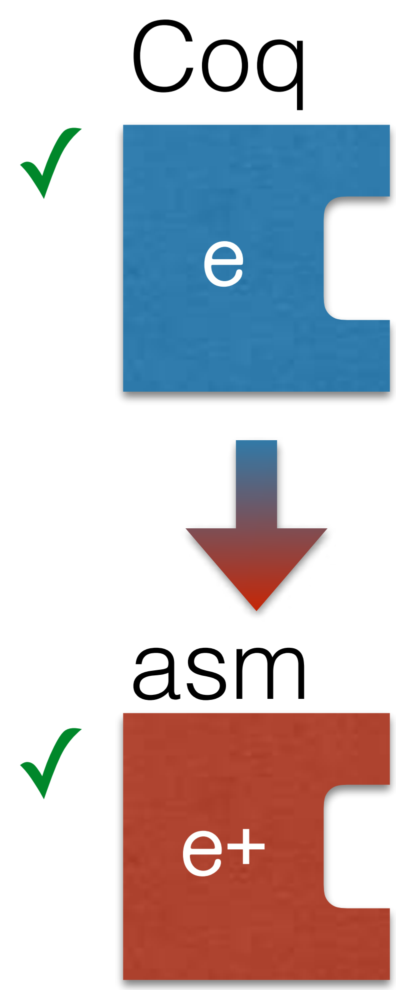
✓

Invariants and proofs encoded in *dependent types*

Preserving Dependent Types

Dependent types

Type-preserving compiler



Invariants and proofs encoded in *dependent types*

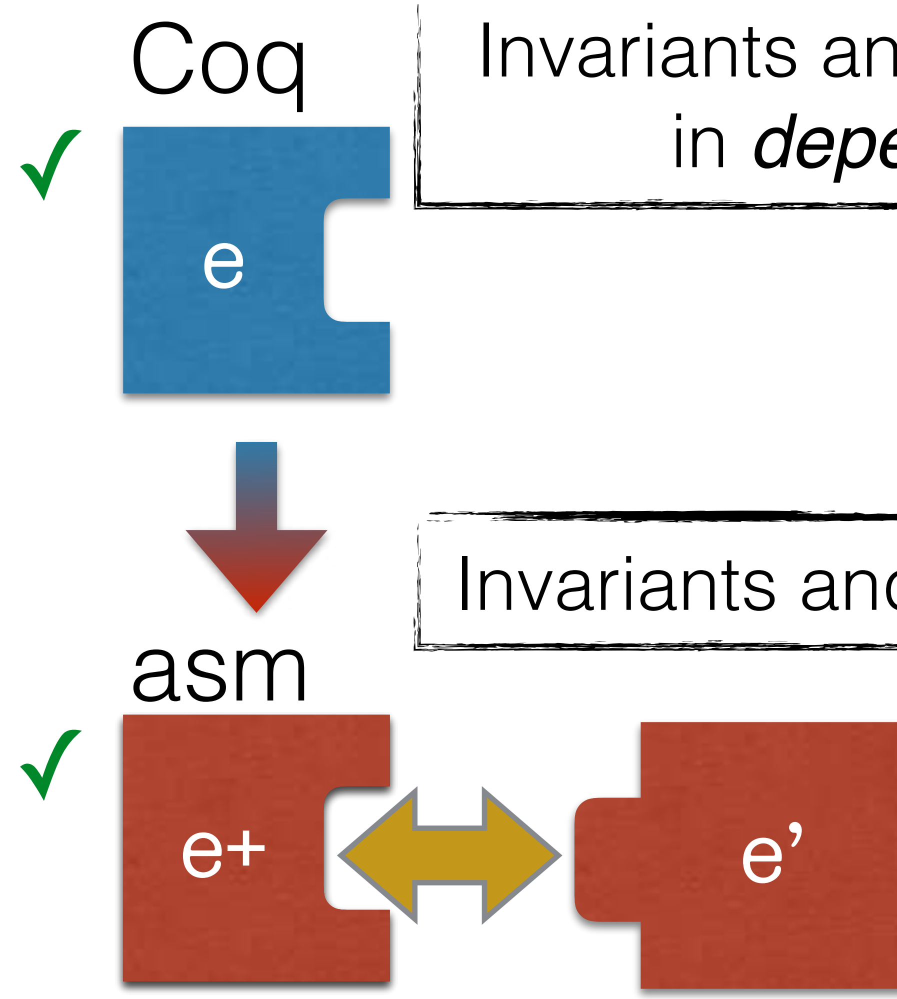
Invariants and proofs preserved

Preserving Dependent Types

Dependent types

Type-preserving compiler

Safety and correctness invariant *enforced* at link time.



Invariants and proofs encoded in ***dependent types***

Invariants and proofs preserved

Type-checking at link time

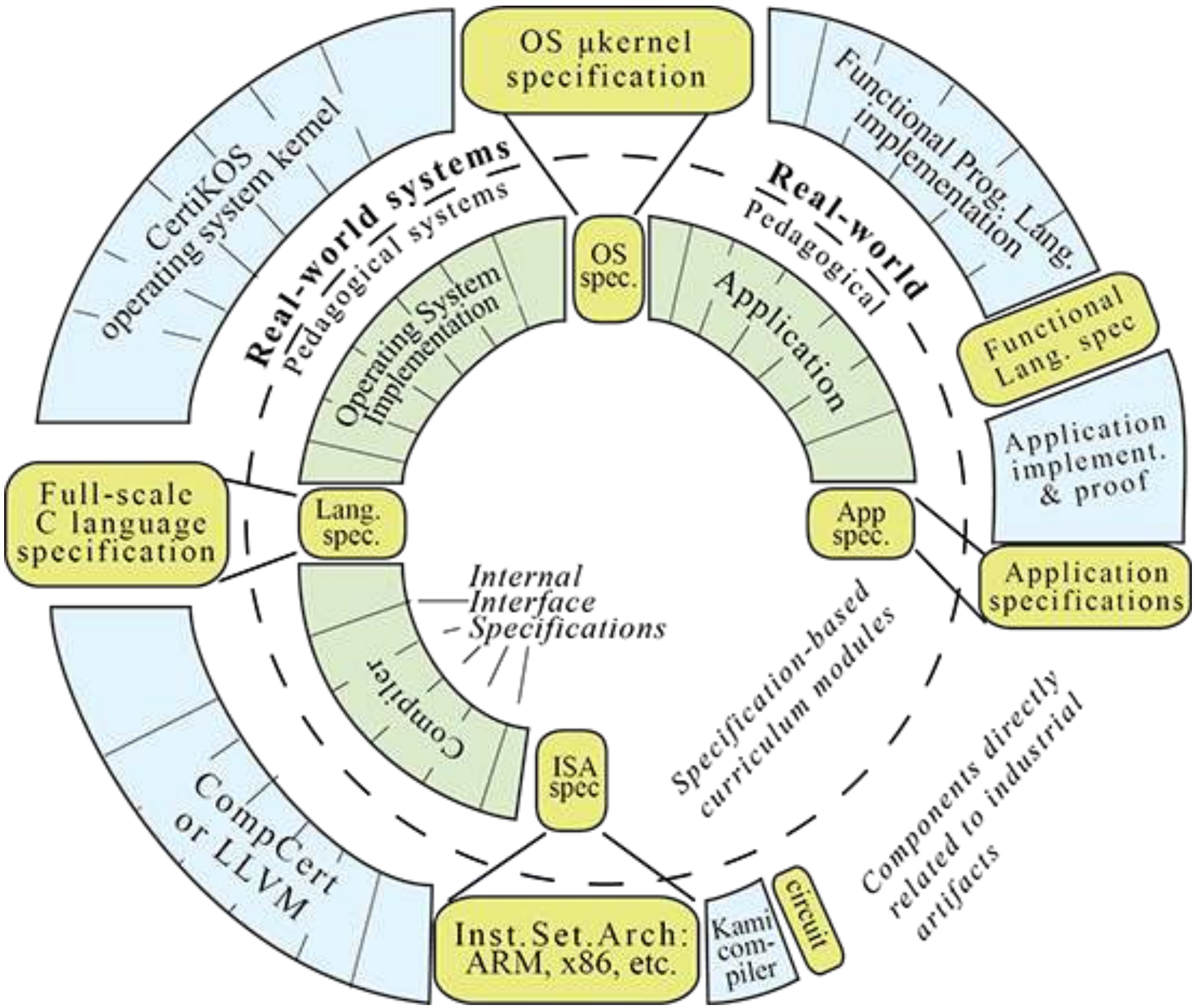
Why Coq and Dependent Types

Verified in Coq!

- CompCert
- CertiKOS
- Vellvm
- RustBelt
- CertiCrypt

…

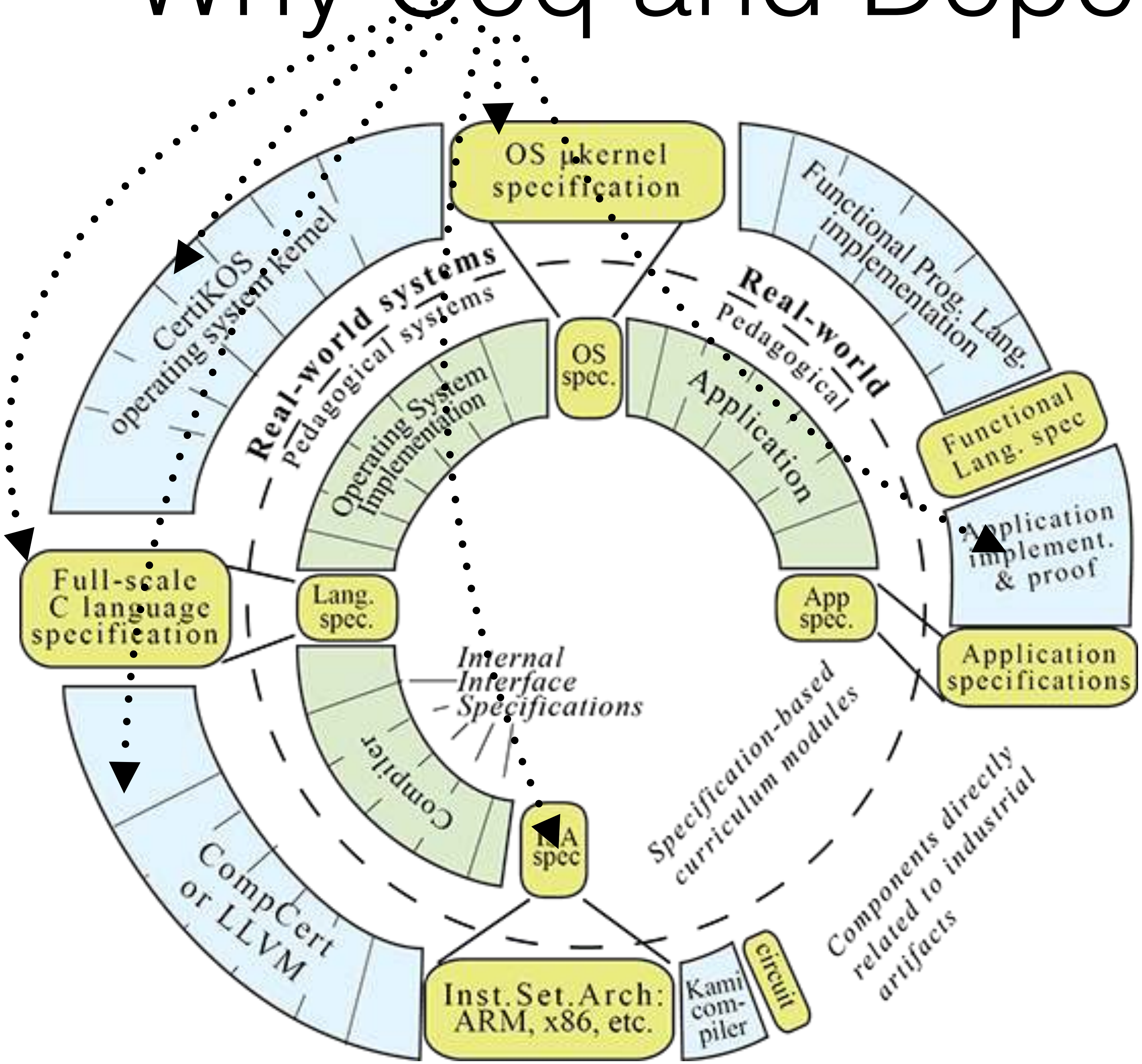
Why Coq and Dependent Types



deep spec

An NSF Expedition

Why Coq and Dependent Types



deep spec

An NSF Expedition

… which heavily relies on Coq

Why Coq and Dependent Types

Hope verified Coq programs can't go wrong…

1 line file

```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml
[1]    43185 segmentation fault (core dumped) ocaml verified.ml
```

Encoding invariants in types

Simple types, simple invariants

```
5 : Int

/ : Int -> Int -> Int

10 / 5 : Int
```

Encoding invariants in types

More type system features, the more we can encode.

```
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```

Encoding invariants in types

More type system features, the more we can encode.

```
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```

Encoding invariants in types

More type system features, the more we can encode.

```
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```

Encoding invariants in types

More type system features, the more we can encode.

```
newtype Newton_Seconds = Int
newtype Pound_Force_Seconds = Int

impulse_calc : () -> Pound_Force_Seconds

autopilot : Newton_Seconds -> ()

autopilot(impulse_calc()) -- Type error!
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
0 : Int

/ : y:Int -> x:{Int | x != 0} -> Int

10 / [5, 5 != 0] : Int
10 / [0, 0 != 0] -- Type error!
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
0 : Int

/ : y:Int -> x:{Int | x != 0} -> Int

10 / [5, 5 != 0] : Int
10 / [0, 0 != 0] -- Type error!
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
0 : Int

/ : y:Int -> x:{Int | x != 0} -> Int

10 / [5, 5 != 0] : Int
10 / [0, 0 != 0] -- Type error!
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
crypt : () -> () -- No guarantees

crypt : () -> {key : *Int | *key = 0x0}
-- Returns proof invariant was enforced
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
c_compiler : C_AST -> x86_AST
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
          c_interp(x) = x86_interp(c_compile(x))
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
            c_interp(x) = x86_interp(c_compile(x))
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
            c_interp(x) = x86_interp(c_compile(x))
```

Encoding invariants in types

Dependent types = Types *depend on* terms

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
            c_interp(x) = x86_interp(c_compile(x))
```

Checking dependent types

- Write programs that express proofs

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
          c_interp(x) = x86_interp(c_compile(x))
```

Checking dependent types

- Write programs that express proofs
- Refer to code in types

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
          c_interp(x) = x86_interp(c_compile(x))
```

Checking dependent types

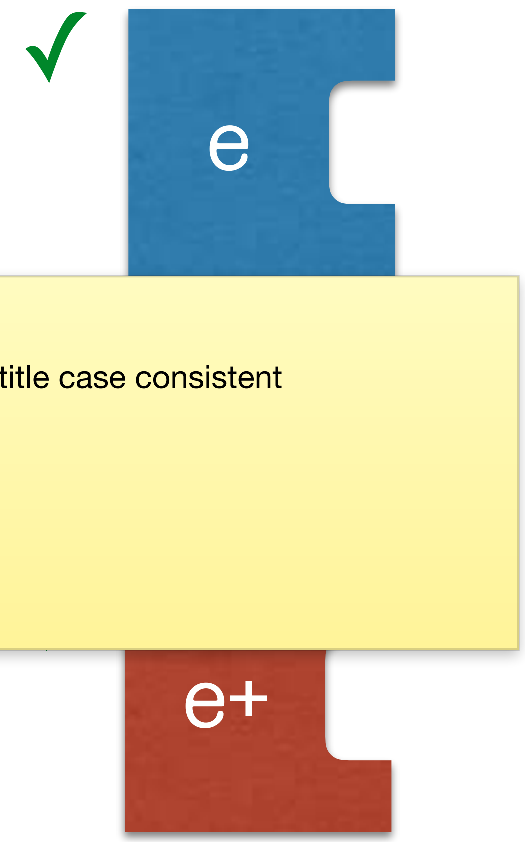
- Write programs that express proofs
- Refer to code in types
- Decide equality between programs

```
c_compiler : C_AST -> x86_AST

c_interp : C_AST -> Int
x86_interp : x86_AST -> Int

correct : x:C_AST ->
            c_interp(x) = x86_interp(c_compile(x))
```

A type-preserving compiler

Dependent types

Type-preserving compiler



Make title case consistent

A type-preserving compiler

CPS: Make control flow explicit

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code

Morrisett, Walker, Crary, Glew 1998

A type-preserving compiler

Theorem. (Type Preservation)

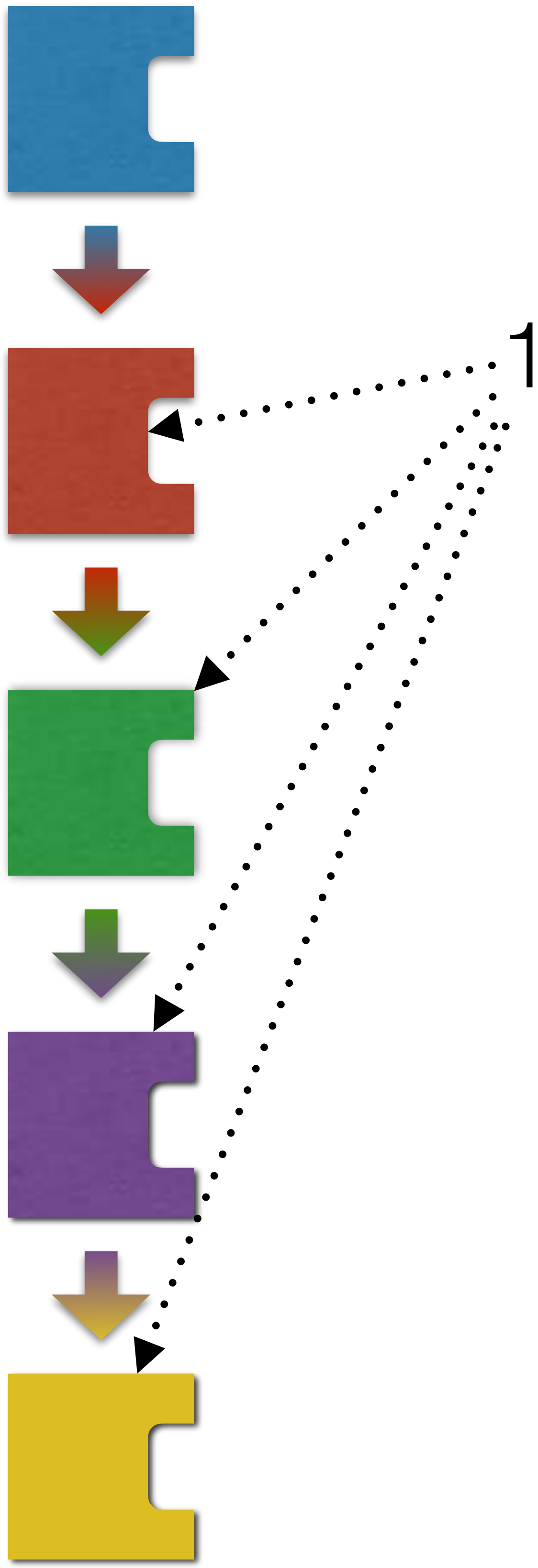
If $e : A$

then (translates to)

$e+ : A+$

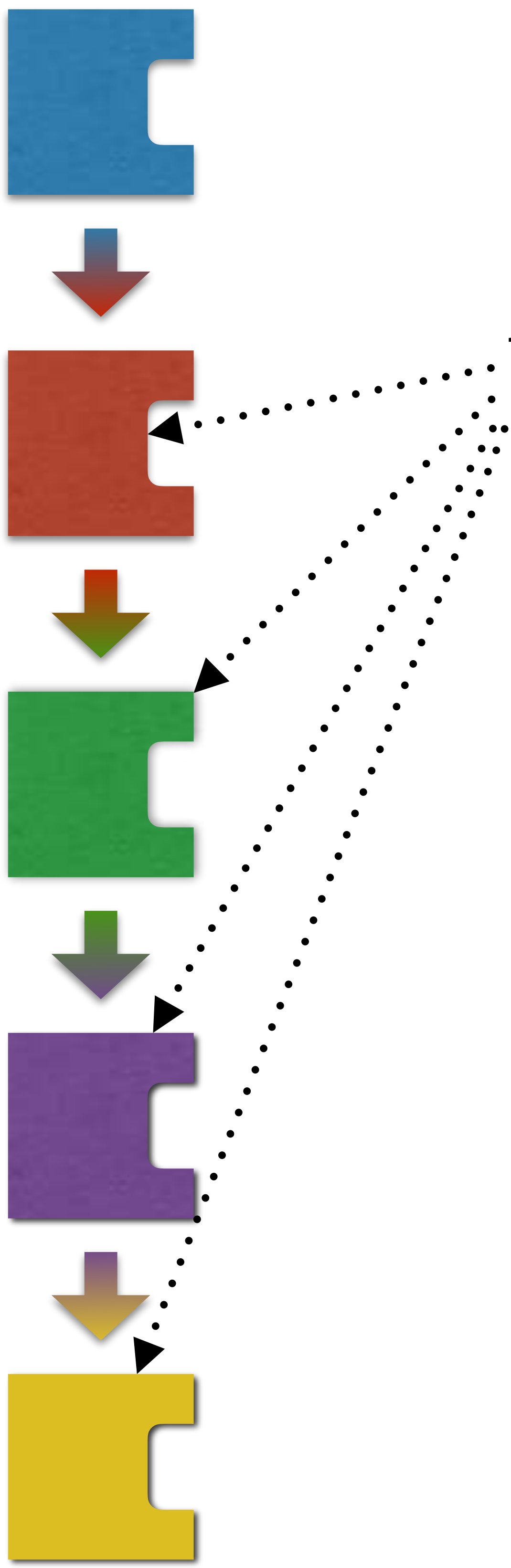
⋮

A type-preserving compiler



1. Design typed intermediate language
   Prove soundness, decidability, etc

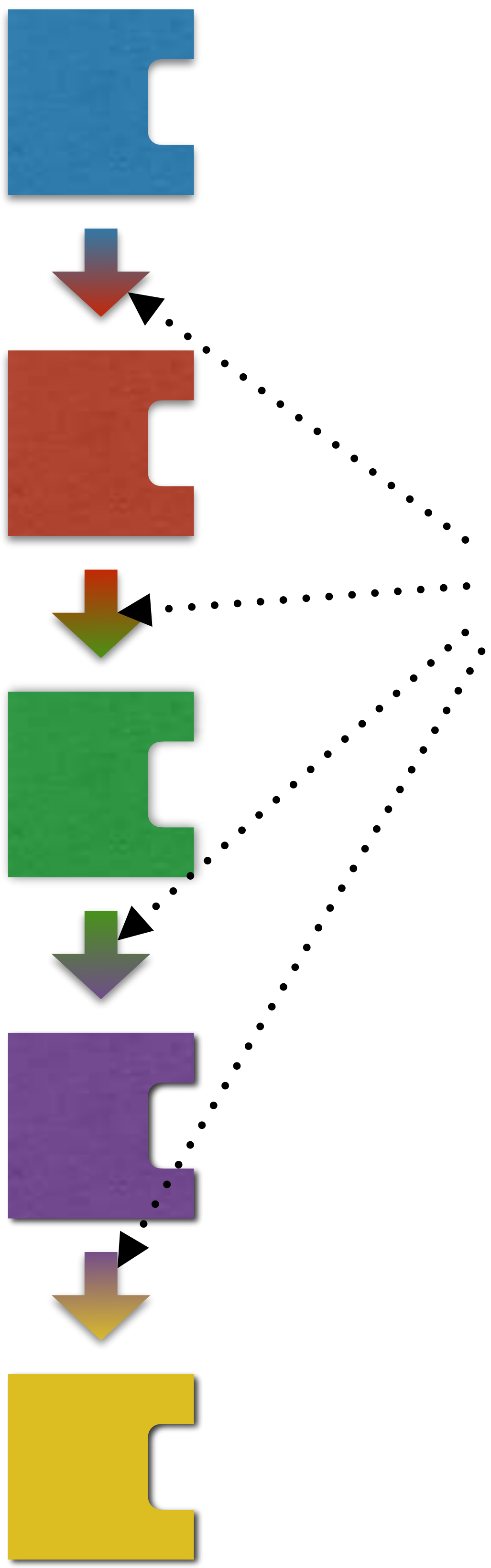
A type-preserving compiler



Hard to design for low-level languages
With ***dep. types,*** hard to prove sound

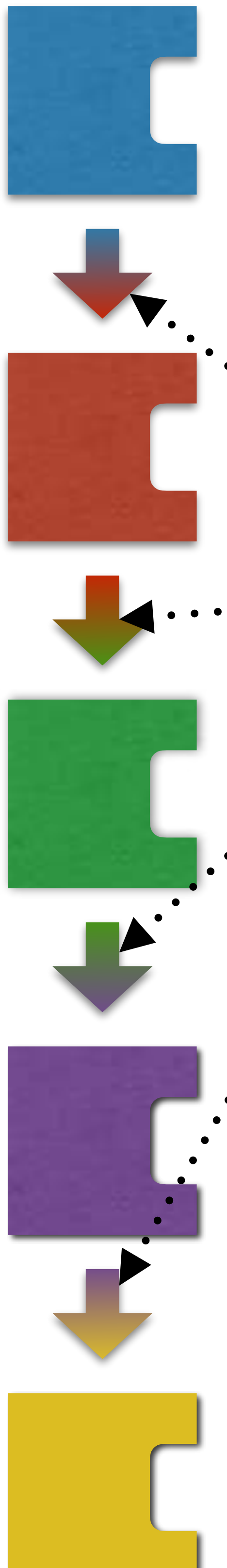
1. Design typed intermediate language
   Prove soundness, decidability, etc

A type-preserving compiler



1. Design typed intermediate language
   Prove soundness, decidability, etc
2. Adapt standard translation
   Prove correctness, preservation, etc

A type-preserving compiler



With dep. types, hard to transform *proofs*

1. Design typed intermediate language
   Prove soundness, decidability, etc
2. Adapt standard translation
   Prove correctness, preservation, etc

Theorem. (Type Preservation)

Contains proofs

e : A

then ↓ ↓ translates to

e+ : A+

Brief history of preserving *simple* types

CPS: Make control flow explicit

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code

Brief history of preserving *simple* types

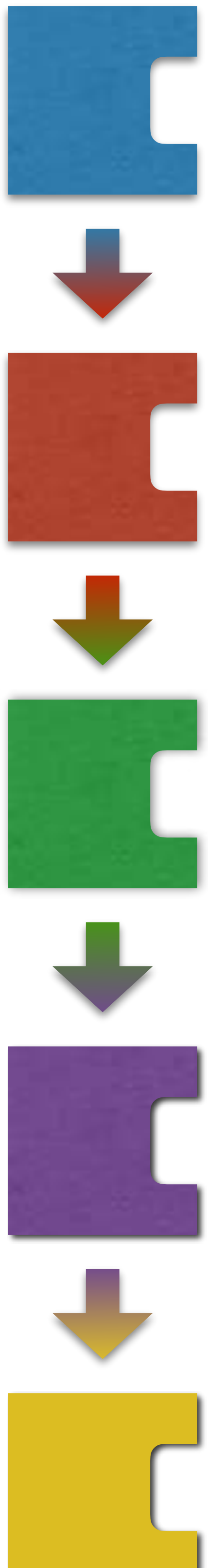
CPS: Make control flow explicit
1985

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code

Brief history of preserving *simple* types



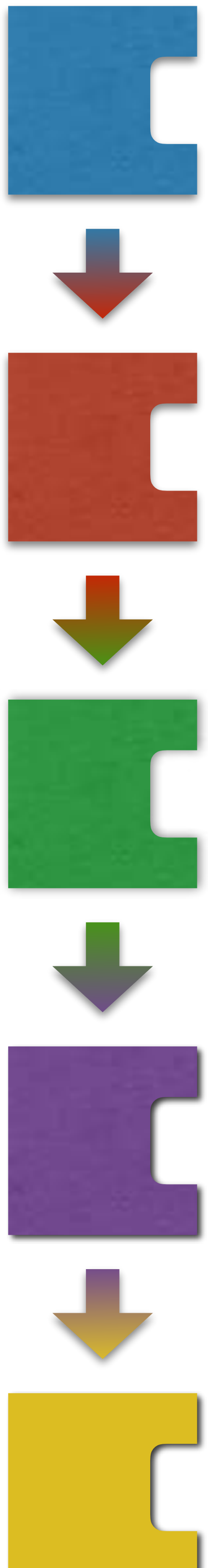
CPS: Make control flow explicit
1985

CC: Make data flow explicit
1996

Alloc: Make allocation explicit

Code gen: Generate assembly code

Brief history of preserving *simple* types



CPS: Make control flow explicit
1985

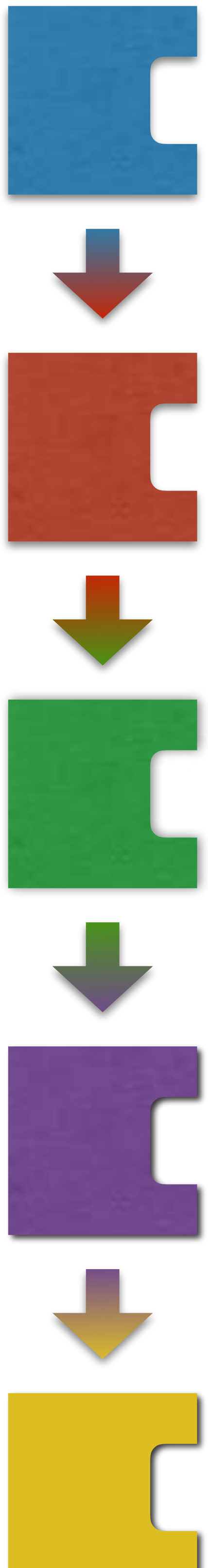
CC: Make data flow explicit
1996

Alloc: Make allocation explicit
1998

Code gen: Generate assembly code
1998

Brief history of preserving *simple* types

Applications in *secure* compilation



CPS: Make control flow explicit

1985 2015

CC: Make data flow explicit

1996 2016

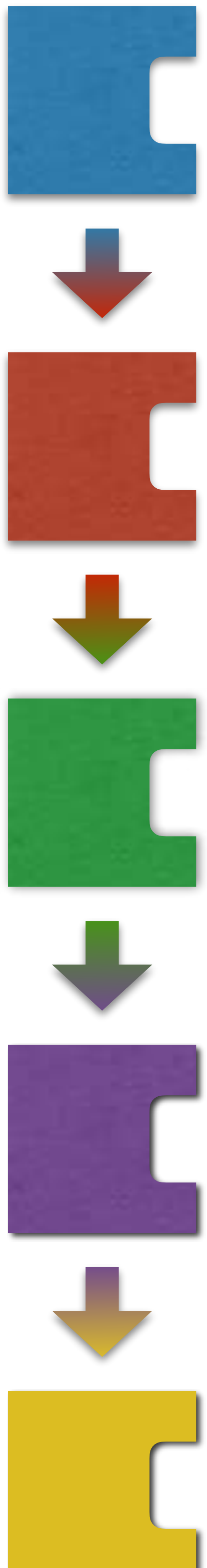
Alloc: Make allocation explicit

1998 On-going

Code gen: Generate assembly code

1998 On-going

Brief history of preserving *dependent* types



CPS: Make control flow explicit
1999, 2002

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code

Brief history of preserving *dependent* types

CPS: Make control flow explicit

1999, 2002 ◂······························ Impossibility result.

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code

Brief history of preserving *dependent* types

CPS: Make control flow explicit

1999, 2002

POPL 2018

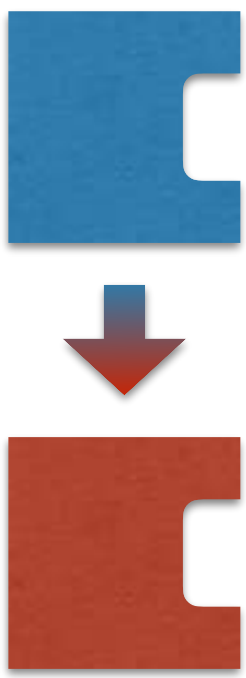
Bowman, Cong, Rioux, Ahmed

CC: Make data flow explicit

Alloc: Make allocation explicit

Code gen: Generate assembly code

CPS for Dependent Types



POPL 2018

Bowman, Cong, Rioux, Ahmed

CPS: Continuation-Passing Style

Makes control flow explicit in syntax.

Equivalent to:

- CFGs
- SSA

CPS: Continuation-Passing Style

```
function f(x) { … }
f(e)
~>
```

CPS: Continuation-Passing Style

Function f is now a labeled block

```
function f(x) { … }
f(e)
~>
f : …+
l1: k = l2; e+
l2: x = r; goto f
```

CPS: Continuation-Passing Style

```
function f(x) { … }
f(e)
~>
f : …+
l1: k = l2; e+
l2: x = r; goto f
```

+ means, recursively translate

CPS: Continuation-Passing Style

```
function f(x) { … }
f(e)
~>
f : …+
l1: k = l2; e+
l2: x = r; goto f
```

Setup calling conventions:
When **e+** is finished, store result in **r**, **goto k**

CPS: Continuation-Passing Style

```
function f(x) { … }
f(e)
~>
f : …+
l1: k = l2; e+
l2: x = r; goto f
```

```
ln: …
    r = v
    goto k
```

Setup calling conventions:
When **e+** is finished, store result in **r**, **goto k**

CPS: Continuation-Passing Style

```
function f(x) { … }
f(e)
~>
f : …+
l1: k = l2; e+
l2: x = r; goto f
```

CPS: Continuation-Passing Style

A few more technical details

```
function f(x) { … }
f(e)
~>
f    : …+
main: k = halt; goto l1
l1   : k1 = k; k = l2; e+
l2   : k = k1; x = r; goto f;
```

In **main**, initialize **k** to special **halt**

CPS: Continuation-Passing Style

A few more technical details

```
function f(x) { … }
f(e)
~>
f   : …+
main: k = halt; goto l1
l1  : k1 = k; k = l2; e+
l2  : k = k1; x = r; goto f;
```

Save/restore "continuation"
around translated subexprs

CPS: Continuation-Passing Style

A few more technical details

```
function f(x) { … }
f(e)
~>
f    : …+
main: k = halt; goto l1
l1   : k1 = k; k = l2; e+
l2   : k = k1; x = r; goto f;
```

CPS: Continuation-Passing Style

```
5
~>
main: k = halt; goto l1
l1  : r = 5; goto k
```

CPS: Continuation-Passing Style

```
5
~>
main: k = halt; goto l1
l1  : r = 5; goto k
```

CPS: Continuation-Passing Style

```
5
~>
main: k = halt; goto l1
l1  : r = 5; goto k
```

CPS: Continuation-Passing Style

```
…
x
~>
main: k = halt; goto l1
l1  : k1 = k; k = l2; …+
l2  : k = k1; r = x; goto k
```

CPS: Continuation-Passing Style

```
…
x
~>
main: k = halt; goto l1
l1  : k1 = k; k = l2; …+
l2  : k = k1; r = x; goto k
```

CPS: Continuation-Passing Style

```
...
x
~>
main: k = halt; goto l1
l1  : k1 = k; k = l2; ...+
l2  : k = k1; r = x; goto k
```

CPS: Continuation-Passing Style

```
…
x
~>
main: k = halt; goto l1
l1  : k1 = k; k = l2; …+
l2  : k = k1; r = x; goto k
```

CPS: Continuation-Passing Style

From now on, ignoring `main` and save/restore

```
[e1, e2]
~>
l1: k = l2; e1+
l2: x1 = r; k = l3; e2+
l3: x2 = r; r = [x1, x2]
```

CPS: Continuation-Passing Style

From now on, ignoring `main` and save/restore

```
[e1, e2]
~>
l1: k = l2; e1+
l2: x1 = r; k = l3; e2+
l3: x2 = r; r = [x1, x2]
```

CPS: Continuation-Passing Style

From now on, ignoring `main` and save/restore

```
[e1, e2]
~>
l1: k = l2; e1+
l2: x1 = r; k = l3; e2+
l3: x2 = r; r = [x1, x2]
```

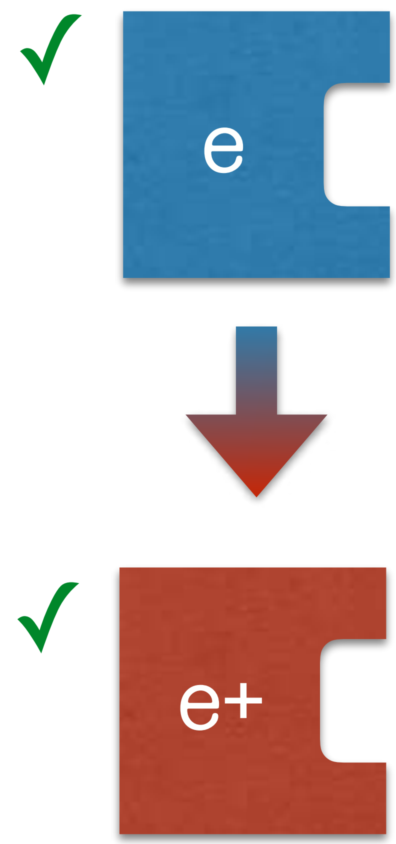
CPS: Continuation-Passing Style

From now on, ignoring `main` and save/restore

```
[e1, e2]
~>
l1: k = l2; e1+
l2: x1 = r; k = l3; e2+
l3: x2 = r; r = [x1, x2]
```

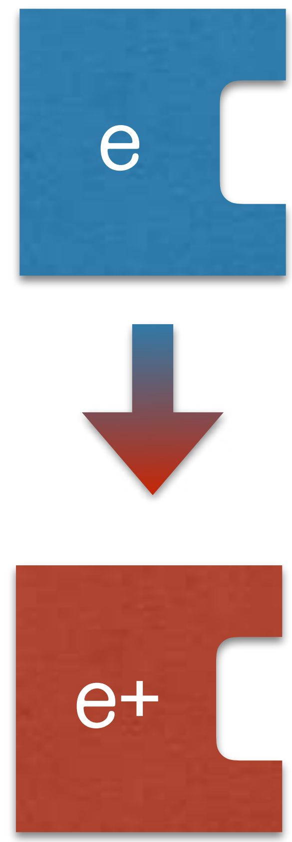
Why would CPS be hard?

Invariants about programs
with call/return



Why would CPS be hard?

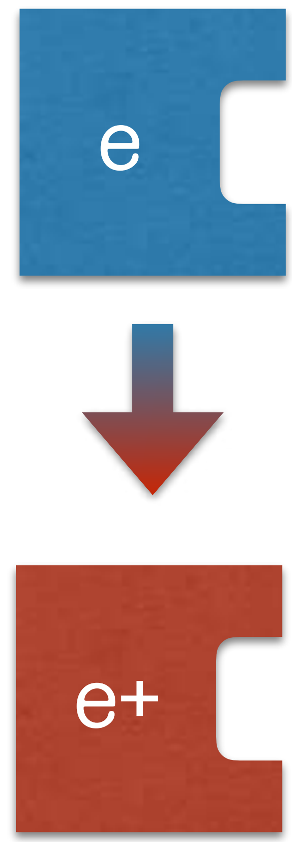
Invariants about programs ✓
with call/return



Invariants about programs ✓
with **goto**

Why would CPS be hard?

Invariants about programs with call/return ✓



Invariants about programs with **goto** ✓

Must design type system to reason about **goto**

Goal: *Type-preserving* CPS translation

Theorem. (Type Preservation)

If $e : A$

then translates to

$e+ : A+$

Goal: *Type-preserving* CPS translation

Problem: *Σ types* (strong dependent pairs)

$$\frac{\texttt{p : \{x : A | B\}}}{\texttt{snd(p) : B[x} \mapsto \texttt{fst(p)]}}$$

Barthe and Uustalu 2002

`p` is pair of an `A` and a `B`

$$\frac{\texttt{p} : \{\texttt{x} : \texttt{A} \mid \texttt{B}\}}{\texttt{snd(p)} : \texttt{B[x} \mapsto \texttt{fst(p)]}}$$

p is pair of an **A** and a **B**

$$\frac{\texttt{p} \;:\; \{\texttt{x} : \texttt{A} \mid \texttt{B}\}}{\texttt{snd(p)} \;:\; \texttt{B}[\texttt{x} \mapsto \texttt{fst(p)}]}$$

where
B (a type) can refer to
x (a program var. that stands for **fst(p)**)

$\texttt{p}$ is pair of an $\texttt{A}$ and a $\texttt{B}$

$$\frac{\texttt{p} \ : \ \{\texttt{x} \ : \ \texttt{A} \ | \ \texttt{B}\}}{\texttt{snd(p)} \ : \ \texttt{B[x} \mapsto \texttt{fst(p)]}}$$

where

$\texttt{B}$ (a type) can refer to

$\texttt{x}$ (a program var. that stands for $\texttt{fst(p)}$)

Goal: *Type-preserving* CPS translation

$$\frac{\texttt{p} : \{\texttt{x} : \texttt{A} \mid \texttt{B}\}}{\texttt{snd(p)} : \texttt{B[x} \mapsto \texttt{fst(p)]}}$$

Goal: *Type-preserving* CPS translation

$$\frac{\texttt{p} : \{\texttt{x} : \texttt{A} \mid \texttt{B}\}}{\texttt{snd(p)} : \texttt{B}[\texttt{x} \mapsto \texttt{fst(p)}]}$$

```
l1: k = l2; p+
l2: x = r; r = snd(x)
```

Goal: *Type-preserving* CPS translation

$$\frac{\texttt{p : \{x : A | B\}}}{\texttt{snd(p) : B[x} \mapsto \texttt{fst(p)]}}$$

Need:

l1: k = l2; p+

l2: x = r; r = snd(x) : (B[x ↦ fst(p)])+

Goal: *Type-preserving* CPS translation

$$\frac{\texttt{p : \{x : A | B\}}}{\texttt{snd(p) : B[x} \mapsto \texttt{fst(p)]}}$$

Need:

```
l1: k = l2; p+
l2: x = r; r = snd(x) : (B[x ↦ fst(p)])+
```

Have:

```
l1: k = l2; p+
l2: x = r; r = snd(x) : B+[x ↦ fst(x)]
```

Goal: *Type-preserving* CPS translation

Suffices:

`(fst(p))+ = fst(x)`

$$\frac{\texttt{p : \{x : A | B\}}}{\texttt{snd(p) : B[x} \mapsto \texttt{fst(p)]}}$$

Need:

```
l1: k = l2; p+
l2: x = r; r = snd(x) : (B[x ↦ fst(p)])+
```

Have:

```
l1: k = l2; p+
l2: x = r; r = snd(x) : B+[x ↦ fst(x)]
```

Need:

```
l1: k = l2; p+
l2: x = r; r = snd(x) : (B[x ↦ fst(p)])+
```

Have:

```
l1: k = l2; p+
l2: x = r; r = snd(x) : B+[x ↦ fst(x)]
```

Suffices:

```
(fst(p))+ = fst(x)
```

Suffices:

```
(fst(p))+
```

```
(fst(p))+ = fst(x)
```

```
l1: k = l2; p+
l2: y = r; r = fst(y)
```

Suffices:

```
(fst(p))+                                    (fst(p))+ = fst(x)

l1: k = l2; p+                               ln: ...
l2: y = r; r = fst(y)                             r = v
                                                  goto k
```

Suffices:

`(fst(p))+ = fst(x)`

`(fst(p))+`

```
l1: k = l2; p+
l2: y = r; r = fst(y)
```

```
ln: …
    r = v
    goto k
```

Suffices:

`x = value-of(p+)`

```
(fst(p))+
```

Suffices:

```
(fst(p))+ = fst(x)
```

```
l1: k = l2; p+
l2: y = r; r = fst(y)
```

```
ln: ...
    r = v
    goto k
```

Suffices:

```
x = value-of(p+)
```

But remember, this is *intuitive;*
no external code/separate compilation

Barthe and Uustalu 2002

`(fst(p))+`

Suffices:

`(fst(p))+ = fst(x)`

```
l1: k = l2; p+
l2: y = r; r = fst(y)
```

```
ln: ...
    r = v
    goto k
```

Suffices:

`x = value-of(p+)`

But remember, this is *intuitive;*
no external code/separate compilation

Impossibility result.

In *standard* CPS, can prove

`x != value-of(p+)`

Barthe and Uustalu 2002

```
(fst(p))+
```

```
l1: k = l2; p+
l2: y = r; r = fst(y)
```

Impossibility Proof.

If … can use arbitrary `goto`,
this program can do anything.

If we try to admit this equality, type system must be inconsistent.

Suffices:

```
(fst(p))+ = fst(x)
```

```
ln: …
    r = v
    goto k
```

Suffices:

```
x = value-of(p+)
```

Barthe and Uustalu 2002

In implicit invariant
to the rescue

Suffices:

`(fst(p))+`

`(fst(p))+ = fst(x)`

```
l1: k = l2; p+
l2: y = r
```

```
ln: ...
    r = v
    goto k
```

Theorem was misunderstood as:
In *every* CPS translation …
Hence this is impossible.

`x = value-of(p+)`

But remember, this is *intuitive;*
no external code/separate compilation

Impossibility result.

In *standard* CPS, can prove

`x != value-of(p+)`

Barthe and Uustalu 2002

Suffices:

```
(fst(p))+ = fst(x)
```

```
(fst(p))+
```

```
l1: k = l2; p+
l2: y = r; r = fst(y)
```

```
ln: ...
    r = v
    goto k
```

Suffices:

```
x = value-of(p+)
```

Impossibility Proof.

If … can use arbitrary `goto`, this program can do anything.

If we try to admit this equality, type system must be inconsistent.

Barthe and Uustalu 2002

Solution

Suffices:

`x = value-of(p+)`

Non-standard, but
same control-flow properties

$$\frac{\texttt{p+ : Definitely jumps to l2} \qquad \texttt{x = value-of(p+)} \vdash \texttt{e : B}}{\begin{array}{l}\texttt{l1: k = l2; p+} \\ \texttt{l2: x = r; r = e : B}\end{array}}$$

Solution

Suffices:

`x = value-of(p+)`

If we can prove this invariant

$$\frac{\texttt{p+ : Definitely jumps to l2} \qquad \texttt{x = value-of(p+)} \vdash \texttt{e : B}}{\begin{array}{l}\texttt{l1: k = l2; p+} \\ \texttt{l2: x = r; r = e : B}\end{array}}$$

Solution

Suffices:

`x = value-of(p+)`

If we can prove this invariant

$$\frac{\texttt{p+ : Definitely jumps to l2} \qquad \texttt{x = value-of(p+)} \vdash \texttt{e : B}}{\texttt{l1: k = l2; p+} \qquad \texttt{l2: x = r; r = e : B}}$$

Then it's safe to assume this equality

Solution

Suffices:

`x = value-of(p+)`

`p+ : Definitely jumps to l2`

`x = va` … $\vdash$ `e : B`

`l1: k =`

`l2: x = r; r = e : B`

Pop up type around here

Develop proof system for this
Develop value interpretation of code with `goto`
Prove type system consistent

Closure Conversion for Dependent Types

PLDI 2018

Bowman, Ahmed

Closure Conversion for Dependent Types

```
...
function f(x) {
    return function (y) {
        return x + y;
    }
}
f(a)(b)
```

Turn higher-order functions

Closure Conversion for Dependent Types

```
...
function f(x) {
    return function (y) {
        return x + y;
    }
}
f(a)(b)
```

Turn higher-order functions

Closure Conversion for Dependent Types

```
...
function f(x) {
    return function (y) {
        return x + y;
    }
}
f(a)(b)
```

```
...
function f(x) {
    return [x, g];
}

function g(x, y) {
    return x + y;
}
c = f(a)
c[2](c[1], b)
```

Turn higher-order functions

into

"objects" (closures)

Closure Conversion for Dependent Types

```
...
function f(x) {
    return function (y) {
        return x + y;
    }
}
f(a)(b)
```

```
...
function f(x) {
    return [x, g];
}

function g(x, y) {
    return x + y;
}
c = f(a)
c[2](c[1], b)
```

Turn higher-order functions

into

“objects” (closures)

Closure Conversion for Dependent Types

```
...
function f(x) {
    return function (y) {
        return x + y;
    }
}
f(a)(b)
```

↓

```
...
function f(x) {
    return [x, g];
}

function g(x, y) {
    return x + y;
}
c = f(a)
c[2](c[1], b)
```

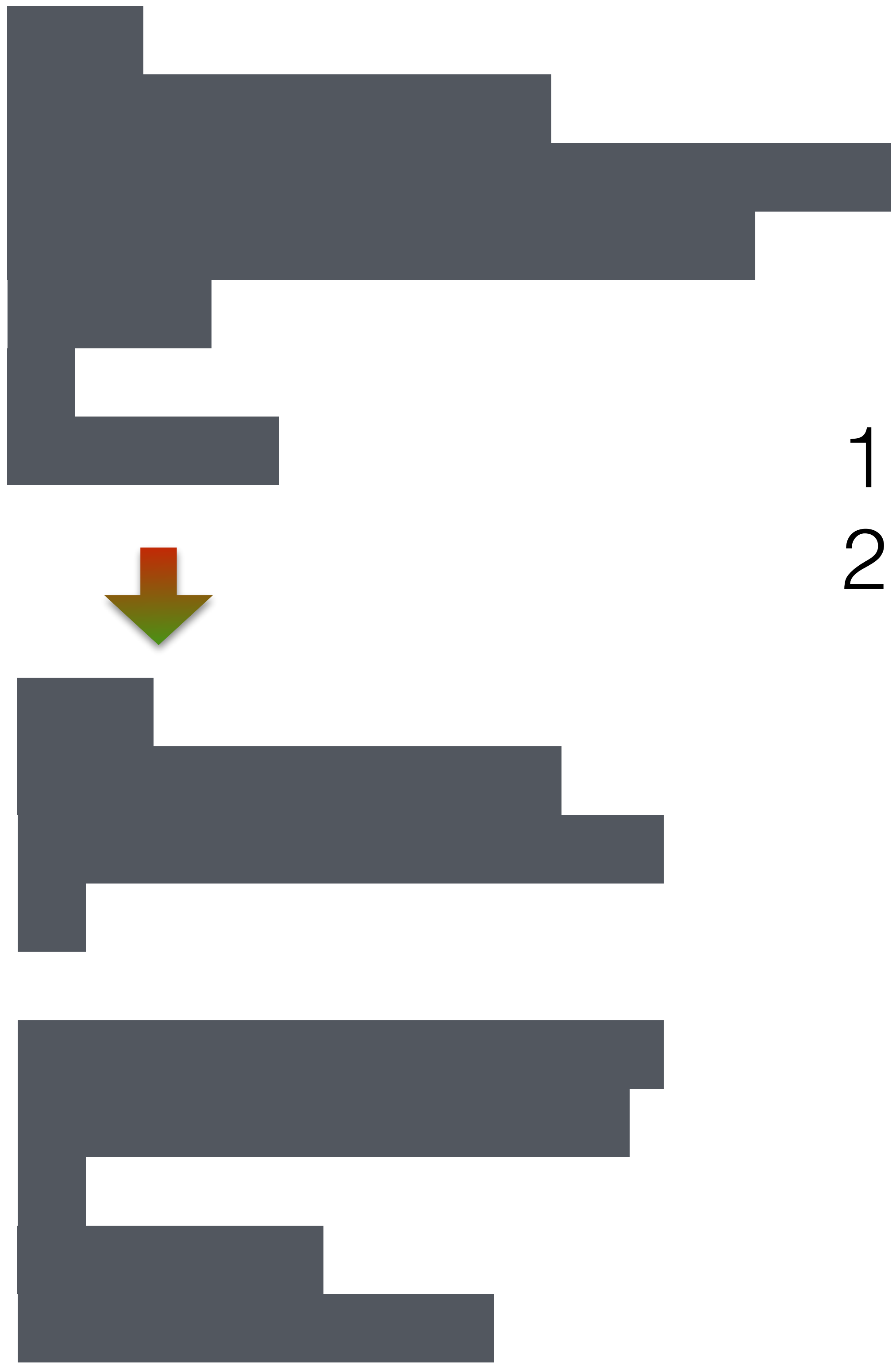
Turn higher-order functions

into

“objects” (closures)

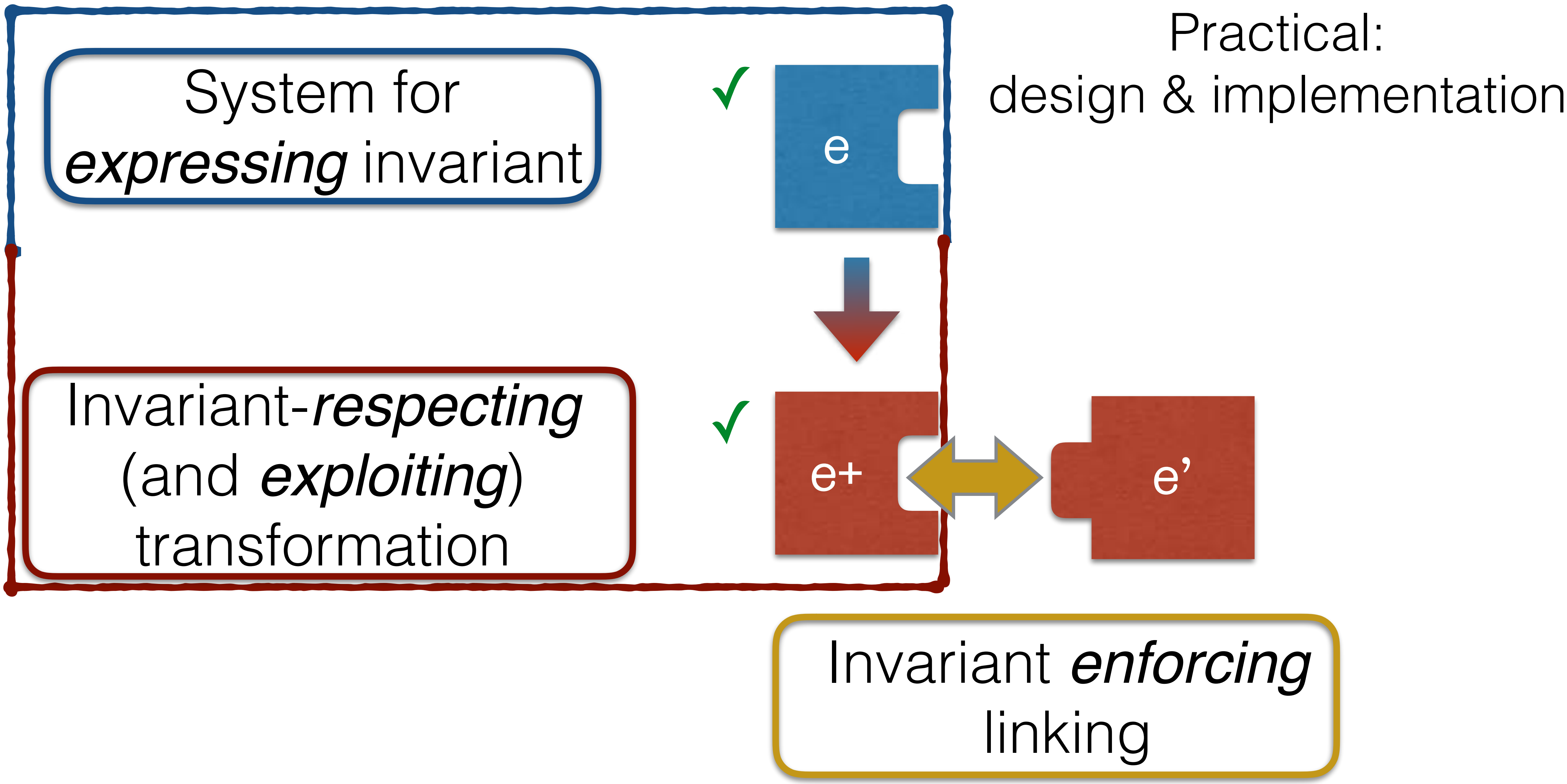
1. Decide closure equivalence
2. Transform local proofs into global

X Translation for Dependent Types



1. Decide X equivalence
2. Transform implicit invariants into explicit invariants

Expressing and Exploiting Invariants



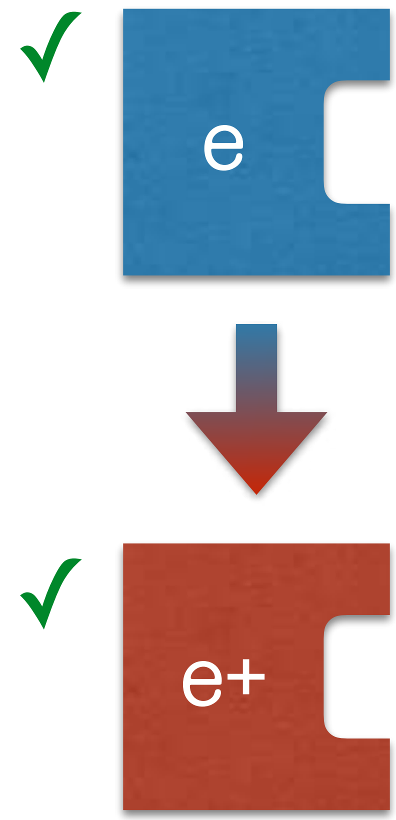
Profile-Guided Optimization for DSLs

Domain-specific languages

Profile-Guided Optimization for DSLs

Domain-specific languages

Optimizing (invariant exploiting) compiler



Expressing Invariants in DSLs

Suppose you have a pattern when working in some domain

```
if f(e) {
    ...
} else if g(e) {
    ...
} else if h(e) {
    ...
}
```

Expressing Invariants in DSLs

Maybe you have a lot of such patterns

```
if f(e) {
    ...
} else if g(e) {
    ...
} else if h(e) {
    ...
}
```

```
function (k) {
    ... (function (x) {
        ... x k
        })
}
```

```
for (i = 0; i < n; i++){
    f(a[i])
}
```

Expressing Invariants in DSLs

You could create abstractions in a general-purpose lang.

```
function discriminate(e) {
                if f(e) {
                    ...
                } else if g(e) {
                    ...
                } else if h(e) {
                    ...
                }
 }
```

Expressing Invariants in DSLs

But maybe in your domain, you know more

```
function discriminate(e) {
                if f(e) {
                    ...
                } else if g(e) {
                    ...
                } else if h(e) {
                    ...
                }
 }
```

e must have certain structure

f(e), **g(e)**, **h(e)** mutually exclusive

Expressing Invariants in DSLs

So, you make a new DSL to enforce those invariants

```
route e {
    ...;
    ...;
    ...;
}
```

checks that **`e`** is valid at compile-time

knows mutually exclusive

Expressing Invariants in DSLs

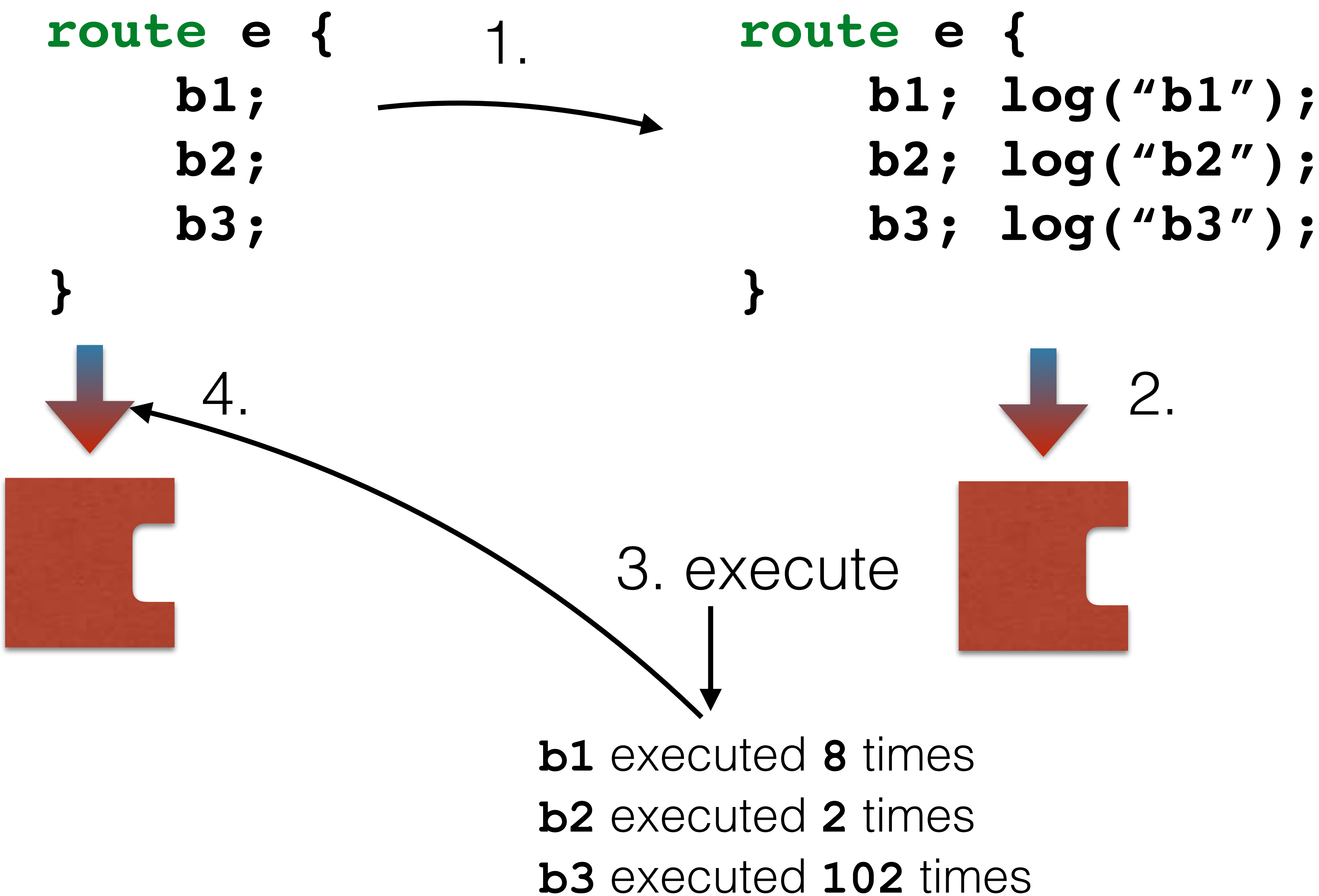
Compile to some general-purpose language

```
route e {
    ...;
    ...;
    ...;
}
```

```
if f(e) {
    ...
} else if g(e) {
    ...
} else if h(e) {
    ...
}
```

And invariants are lost, can't be used for optimization

Profile-Guided Optimization



Profile-Guided Optimization

```
route e {
    b1;
    b2;
    b3;
}
```

1.

```
route e {
    b1; log("b1");
    b2; log("b2");
    b3; log("b3");
}
```

But! PGO can't use domain invariants
(… unless you implement the compiler toolchain yourself)

3. execute

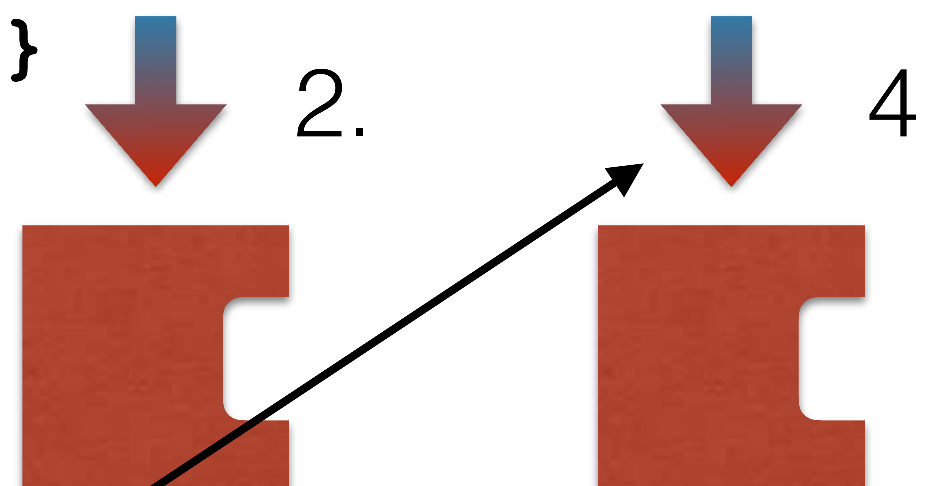
b1 executed **8** times
b2 executed **2** times
b3 executed **102** times

Profile-Guided Optimization

```
route e {
    b1;
    b2;
    b3;
}
```

1.

```
if f(e) {
    b1;
} else if g(e) {
    b2;
} else if h(e) {
    b3;
}
```



3. execute

b1 executed **8** times
b2 executed **2** times
b3 executed **102** times

Profile-Guided Meta-Programming

PLDI 2015

Bowman, Miller, St-Amour, Dybvig.

Profile-Guided Meta-Programming

```
route e {
    b1;
    b2;
    b3;
}
```

```
if f(e) {
    b1;
} else if g(e) {
    b2;
} else if h(e) {
    b3;
}
```

Meta-programming (macros)
for writing DSLs

Profile-Guided Meta-Programming

```
route e {
    b1;
    b2;
    b3;
}
```

```
if f(e) {
    b1;
} else if g(e) {
    b2;
} else if h(e) {
    b3;
}
```

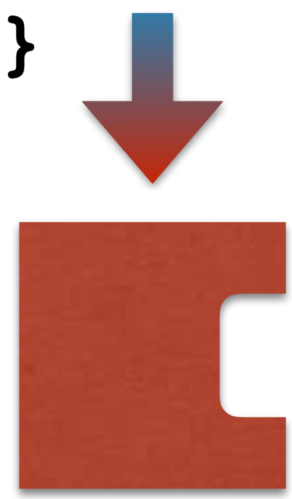
Method for mapping profile back to DSL

b1 executed **8** times
b2 executed **2** times
b3 executed **102** times

Profile-Guided Meta-Programming

```
route e {
    b1;
    b2;
    b3;
}
```

```
if f(e) {
    b1;
} else if g(e) {
    b2;
} else if h(e) {
    b3;
}
```



Method for mapping profile back to DSL

Multiple profiles may belong to *same* DSL expr.
Some profiles may belong to *no* DSL expr.

Profile-Guided Meta-Programming

```
route e {
    b1;
    b2;
    b3;
}
```

```
if f(e) {
    b1;
} else if g(e) {
    b2;
} else if h(e) {
    b3;
}
```

Expand, compile and profile

b1 executed 8 times

b2 executed 2 times

b3 executed 102 times

Profile-Guided Meta-Programming

```
route e {
    b1;
    b2;
    b3;
}
```

Re-expand, using profile to optimize

```
if h(e) {
    b3;
} else if f(e) {
    b1;
} else if g(e) {
    b2;
}
```

b1 executed 8 times

b2 executed 2 times

b3 executed 102 times

Profile-Guided Meta-Programming

PLDI 2015
Bowman, Miller, St-Amour, Dybvig.

Implemented for:

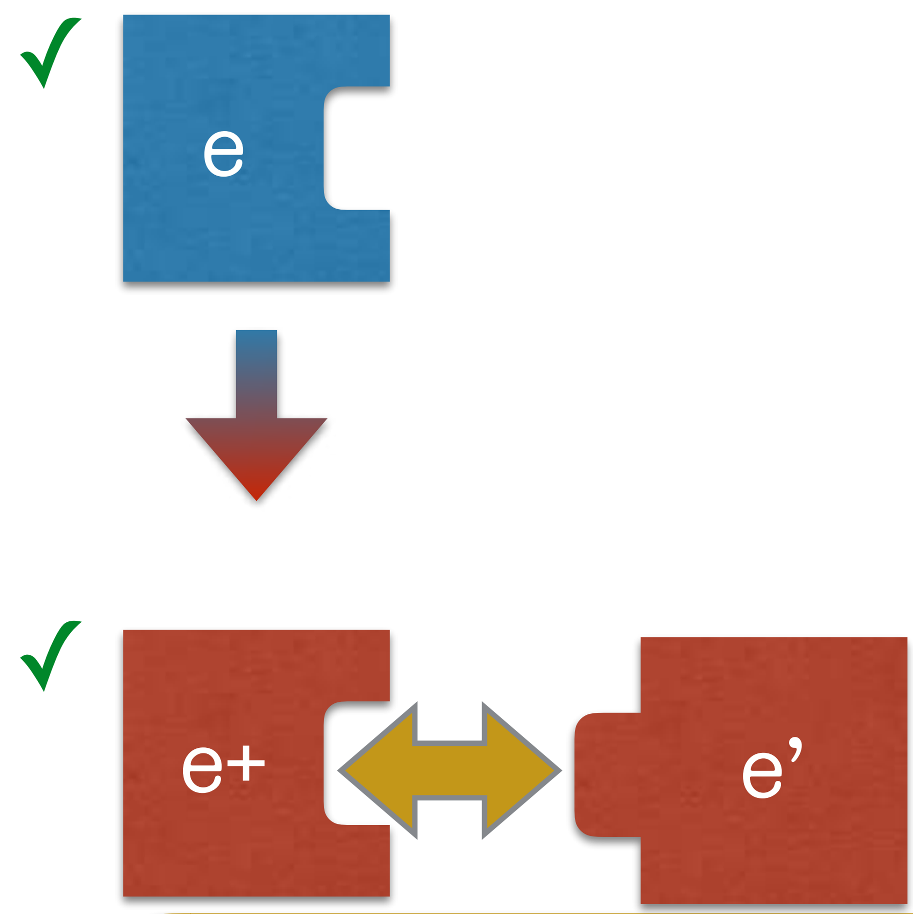
- Cisco's Chez Scheme
- Racket

API, and design sketch for:

- Template Haskell
- MetaOCaml
- Scala

In future

System for *expressing* invariant

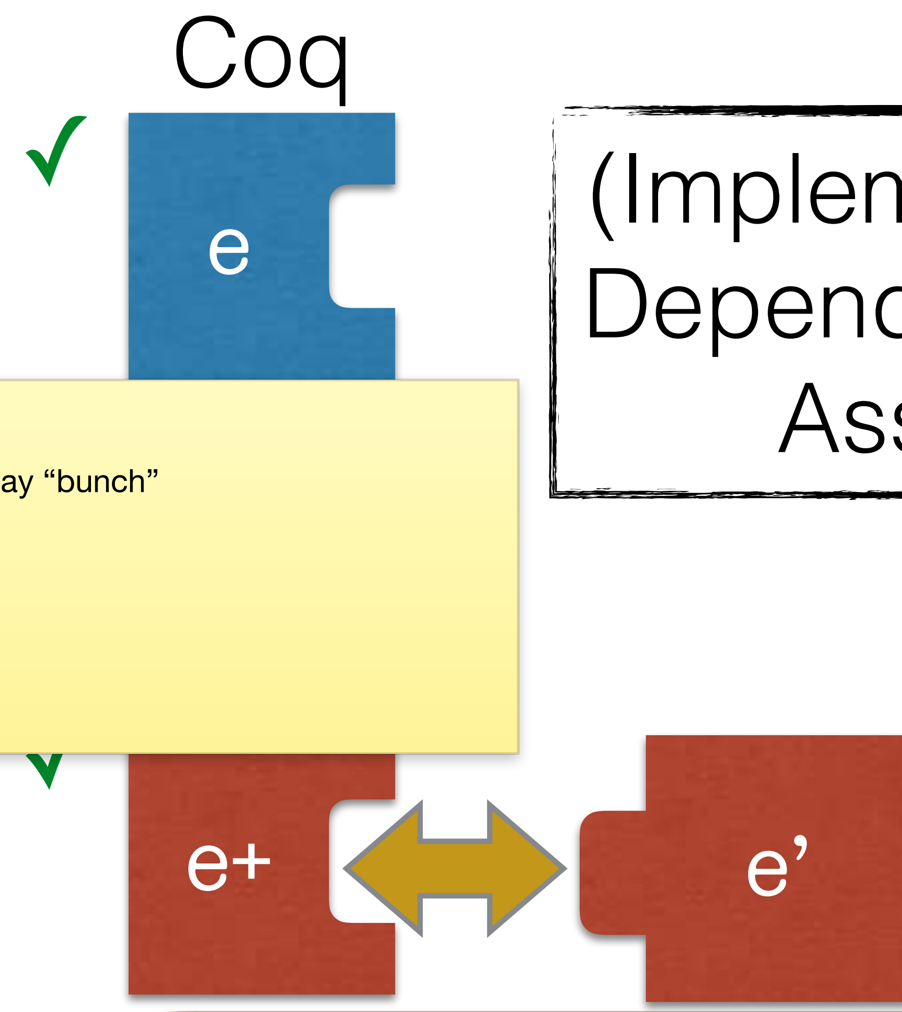


Invariant-*respecting* (and *exploiting*) transformation

Invariant *enforcing* linking

In future

Dependent Types



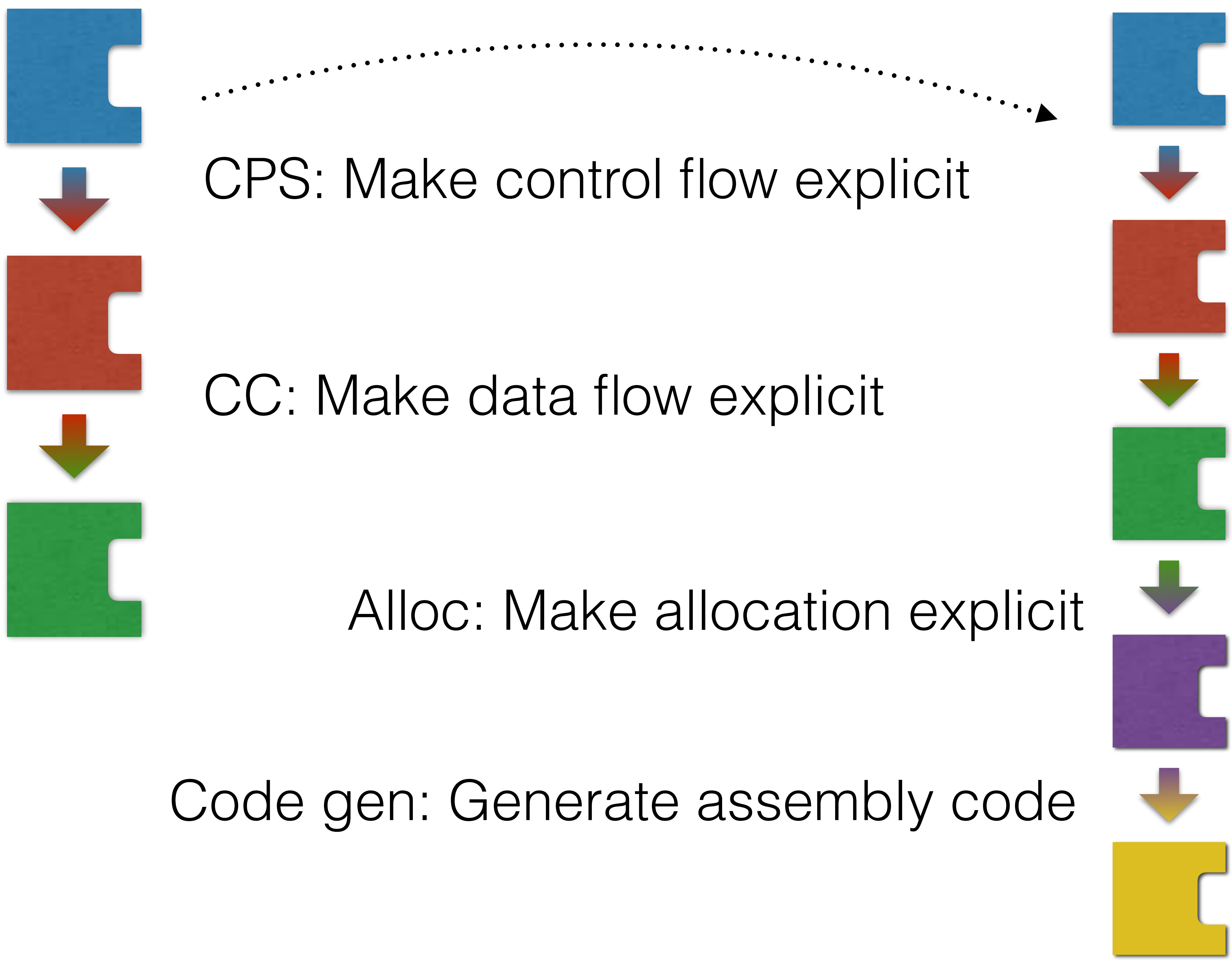
(Implement) Coq to Dependently Typed Assembly!

dont say “bunch”

Type-preserving compiler

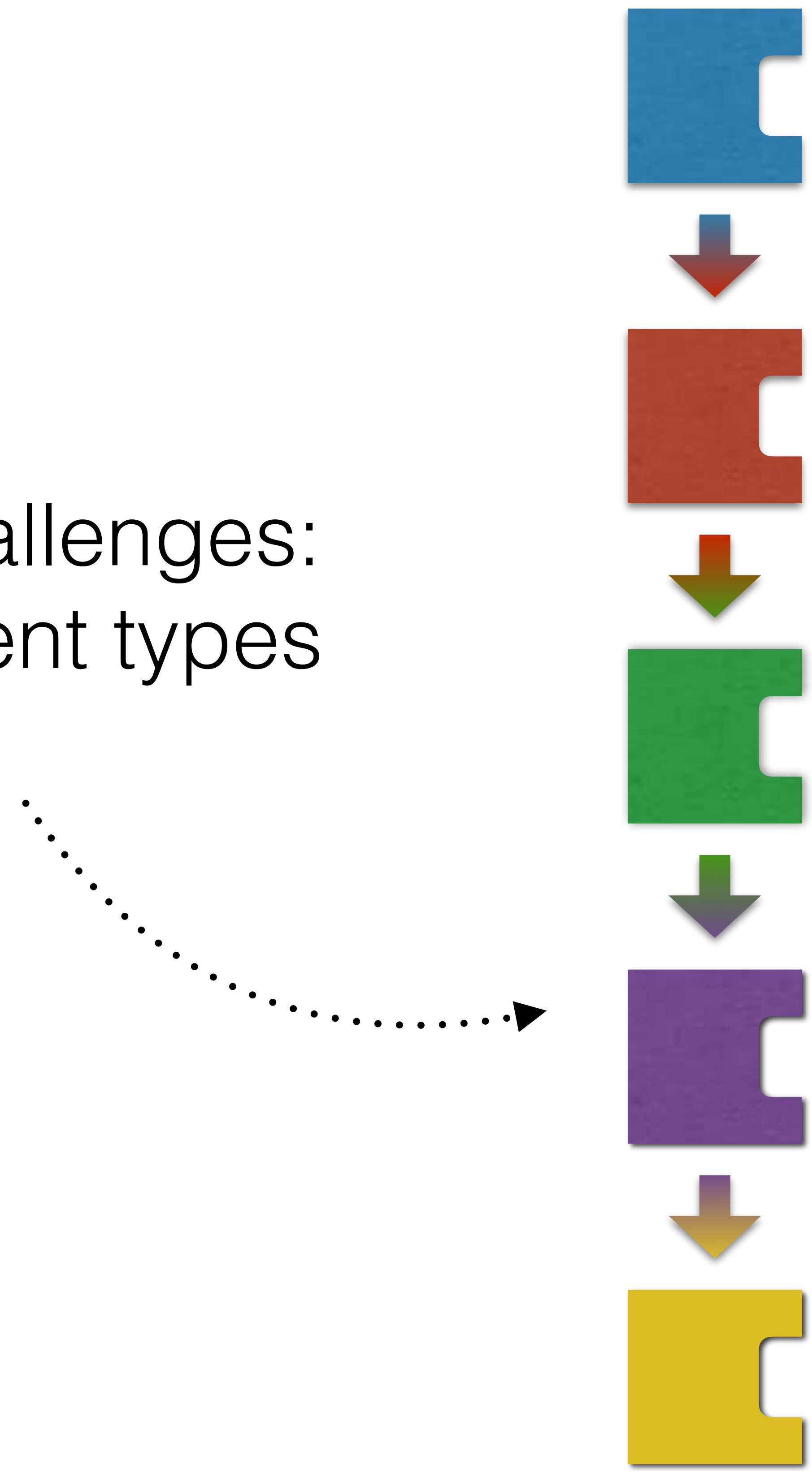
Type-checking at link time

Coq to Asm


CPS: Make control flow explicit
CC: Make data flow explicit
Alloc: Make allocation explicit
Code gen: Generate assembly code

Coq to Asm

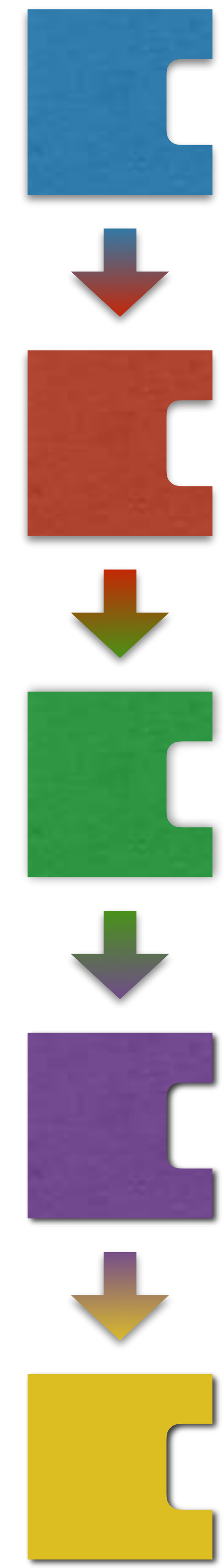
Foundational challenges:
linear + dependent types



Coq to Asm

Practical challenges

- General purpose
- Low-level optimizations
- Interoperability



Coq to Asm

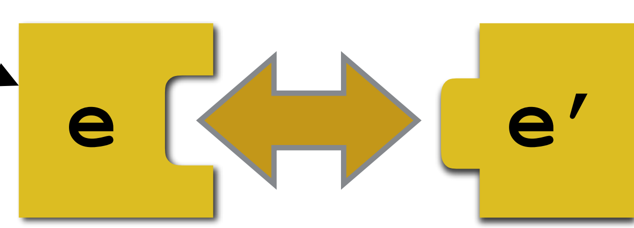
```
> coqc verified.v

> link verified.ml unverified.ml

> ocaml verified.ml
[1]    43185 segmentation fault (core dumped)
ocaml verified.ml
```

Coq to Asm

~~> coqc verified.v~~

~~> **link verified.ml unverified.ml**~~

~~> ocaml verified.ml~~
~~[1]   43185 **segmentation fault** (core dumped) ocaml verified.ml~~

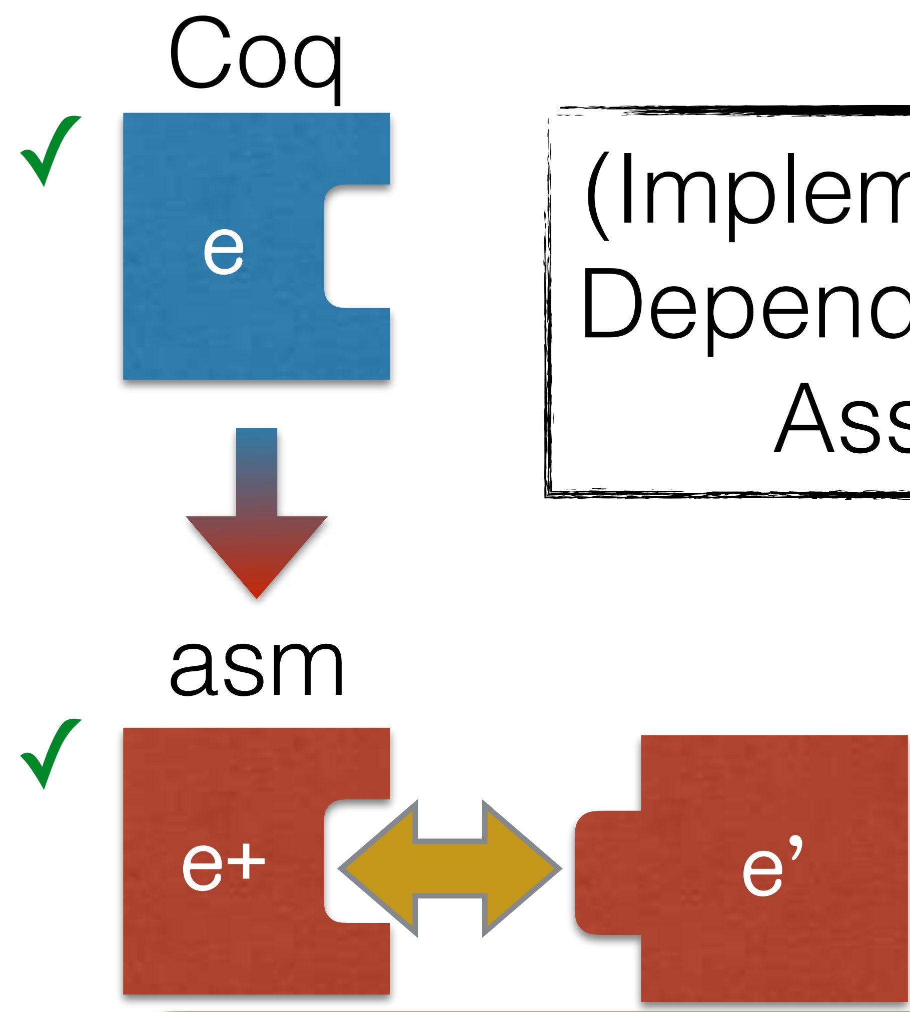


Type error! Can't prove **`e′`** safe

OR Types turned into dynamic checks

In future

Dependent Types

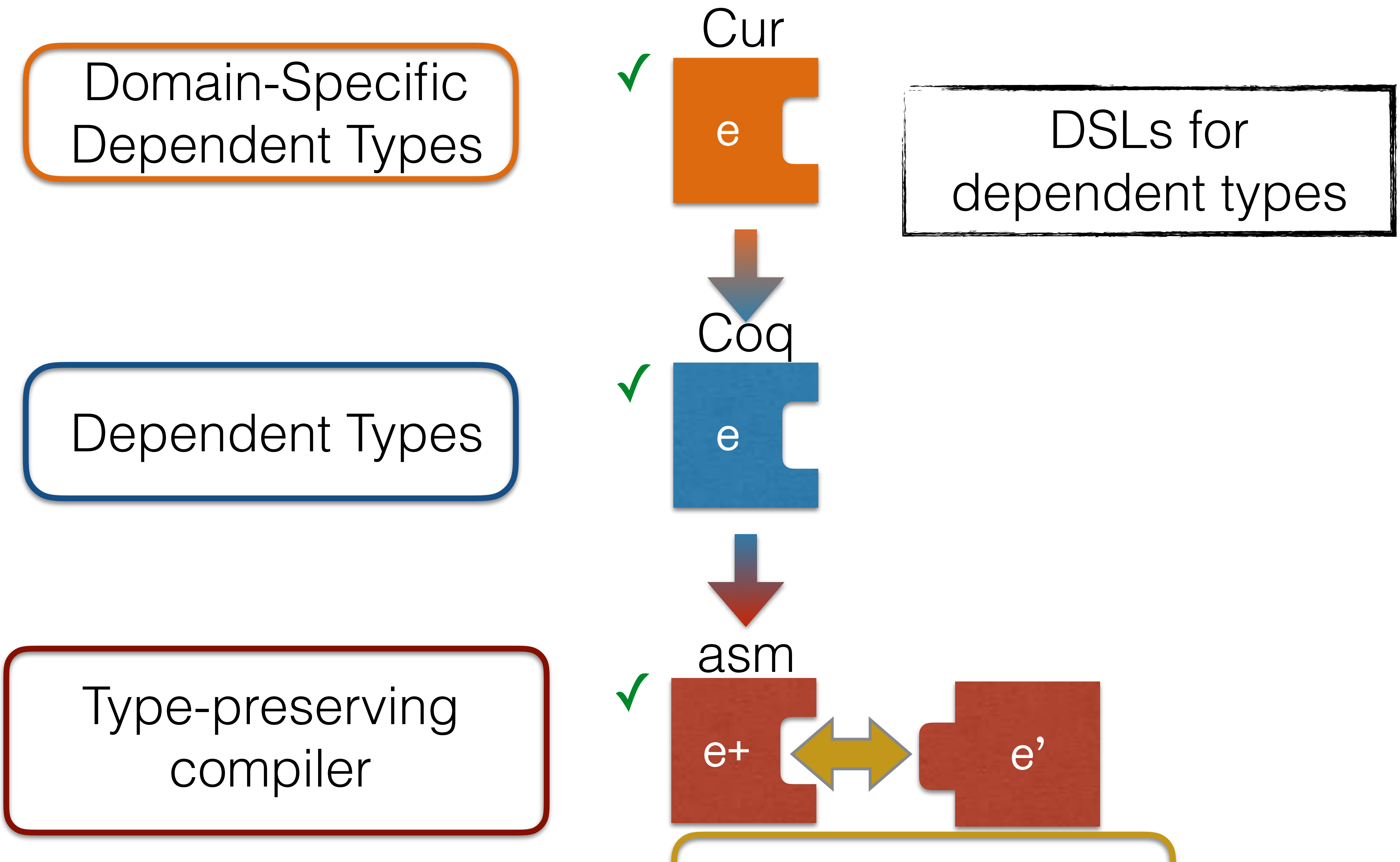


(Implement) Coq to Dependently Typed Assembly!

Type-preserving compiler

Type-checking at link time

In future

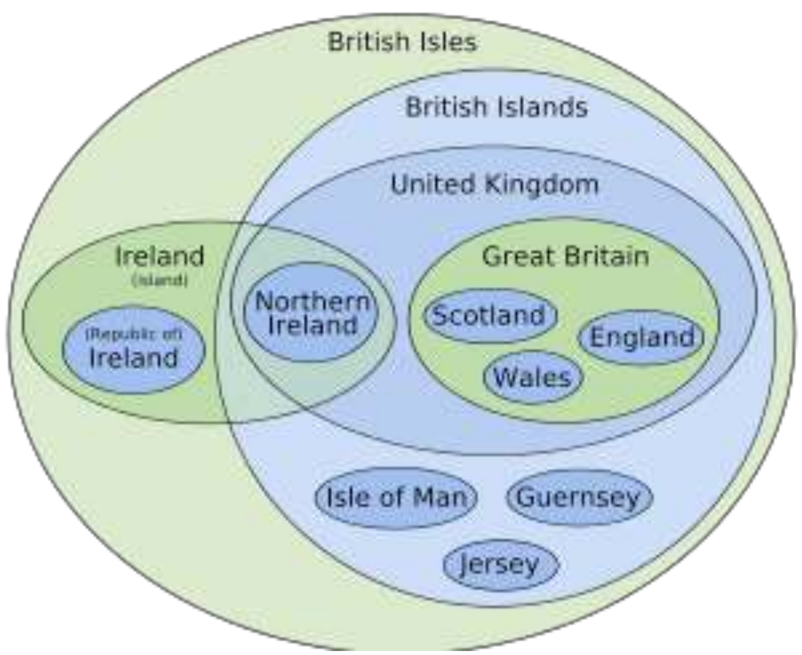


Domain-Specific Dependent Types
Dependent Types
Type-preserving compiler
Cur
e
Coq
e
asm
e+
e’
DSLs for dependent types

How do you write sets in your domain?

`F = {n is an integer | 0 ≤ n ≤ 19}`

`<expr> ::= <term>|<expr><addop><term>`



How do you write sets in Coq

```
<expr> ::= <term>|<expr><addop><term>
```

```
Inductive expr : Set :=
| term_expr : term -> expr.
| add_expr : expr -> term -> expr.
```

How do you write sets in Cur

```
<expr> ::= <term>|<expr><addop><term>
```

```
Inductive expr : Set :=
| term_expr : term -> expr.
| add_expr : expr -> term -> expr.
```

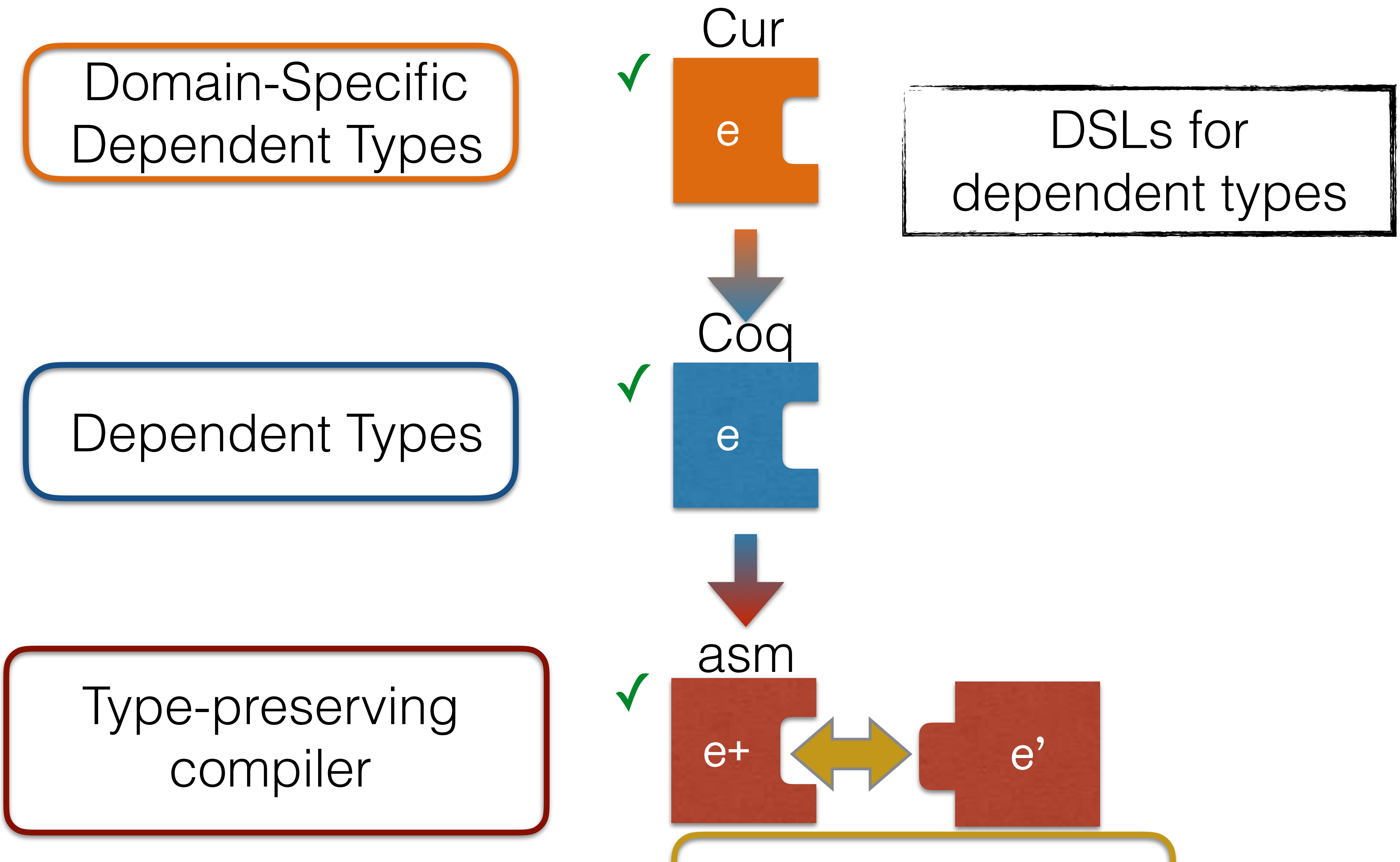
How do you write sets in Cur

```
<expr> ::= <term>|<expr><addop><term>
```

```
F = {n is an integer | 0 ≤ n ≤ 1
```

```
Inductive [redacted] : Set :=
| [redacted] : [redacted] .
| [redacted] : [redacted] .
```

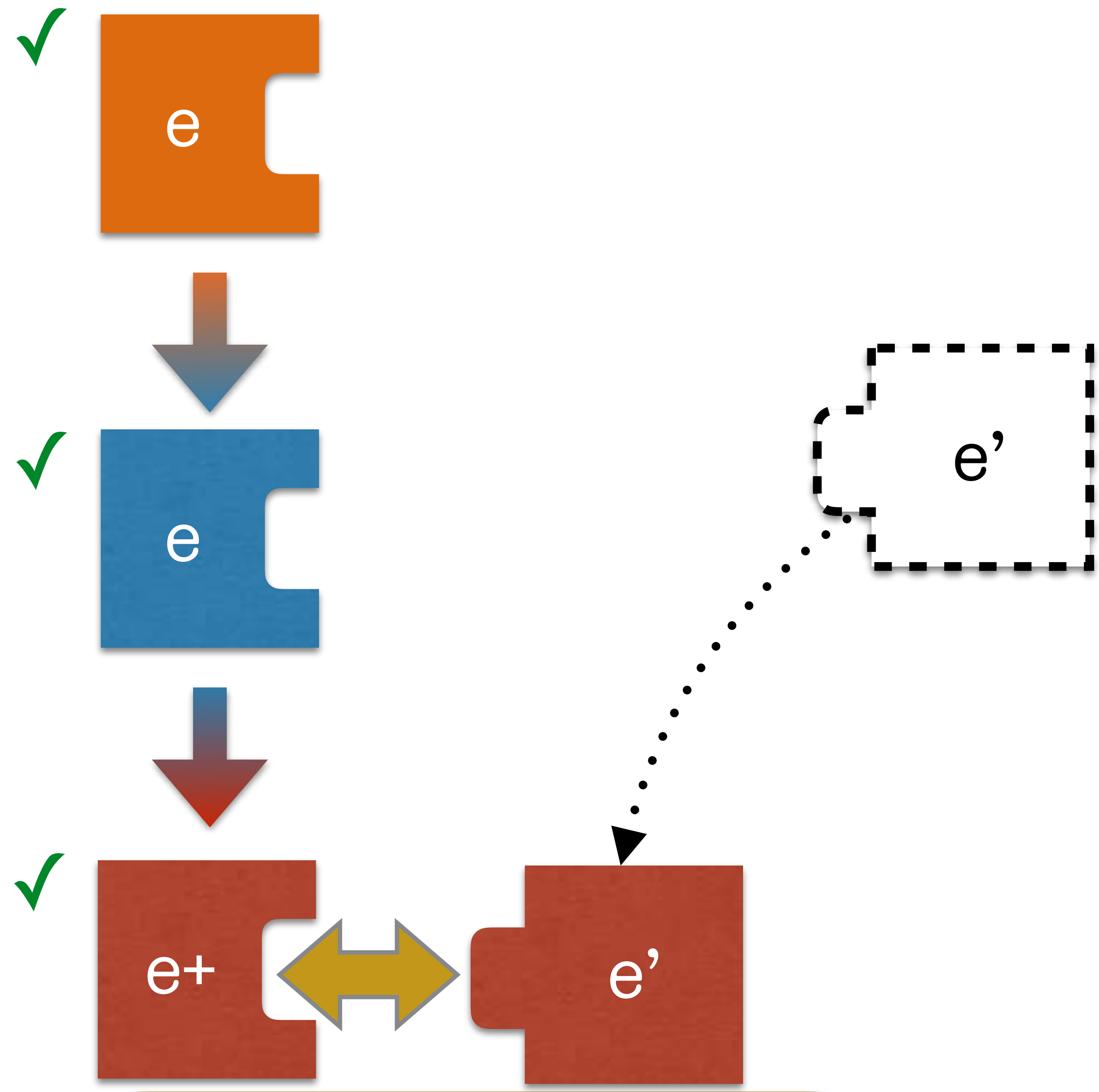
In future


Domain-Specific Dependent Types
Cur
e
DSLs for dependent types
Coq
Dependent Types
e
Type-preserving compiler
asm
e+
e’

Long term vision

Expressing and *preserving* (domain-specific) invariants

Invariant-*respecting* (and *exploiting*) compilation

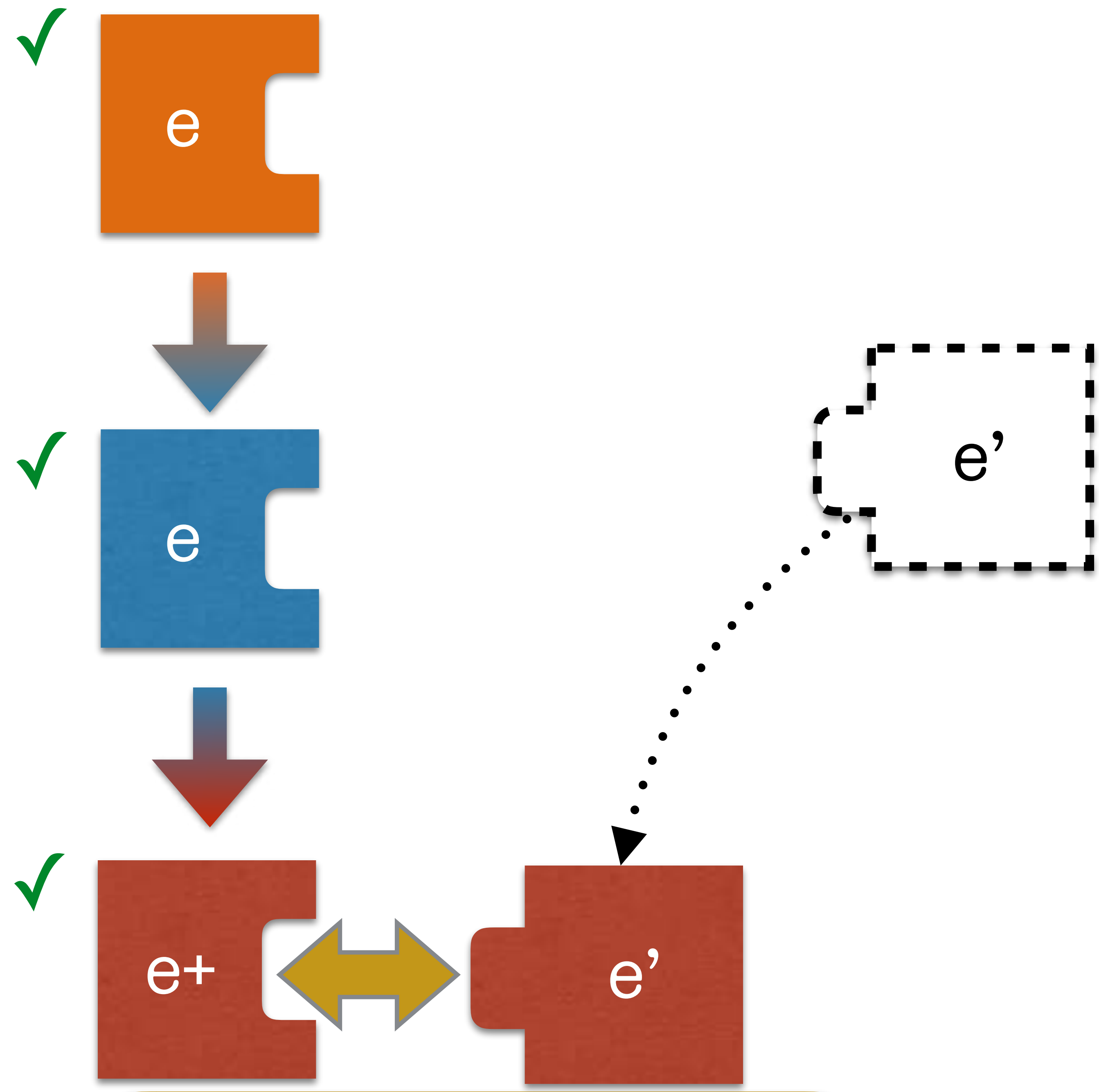


Invariant *enforcing* linking

Do compilers respect programmers?

Not yet, but here's a recipe

Expressing and *preserving* (domain-specific) invariants



Invariant-*respecting* (and *exploiting*) compilation

Invariant *enforcing* linking