

# Local temporal reasoning

**Eric Koskinen**

Visiting Assistant Professor

**New York University**

6 May 2014

Joint work with Tachio Terauchi




***Making sure programs operate correctly***

# *Making sure programs operate correctly*

**Safety** Don't crash! (Nothing bad happens.)

SLAM, Blast, Astrée, etc.



so many other  
important properties  
missing!

# *Making sure programs operate correctly*

**Safety** Don't crash! (Prove that the program finishes. (ens.)  
M, Blast, Astrée, etc.

**Liveness** Something good eventually happens.  
Terminator, ARMC, etc.

*“Every time a client connects to the web server, eventually web server gives a response.”*

## Temporal Logic

$G(\neg \text{crash})$

Safety Don't crash! (Nothing bad happens.)

$F(\text{finish})$

Liveness Something good eventually happens.

Terminator, ARMC, etc.

*“Every time a client connects to the web server, eventually web server gives a response.”*

$G(\text{request} \Rightarrow F(\text{response}))$

## Temporal Logic

Safety Don't crash! (Nothing bad happens.)

Liveness Something good eventually happens.

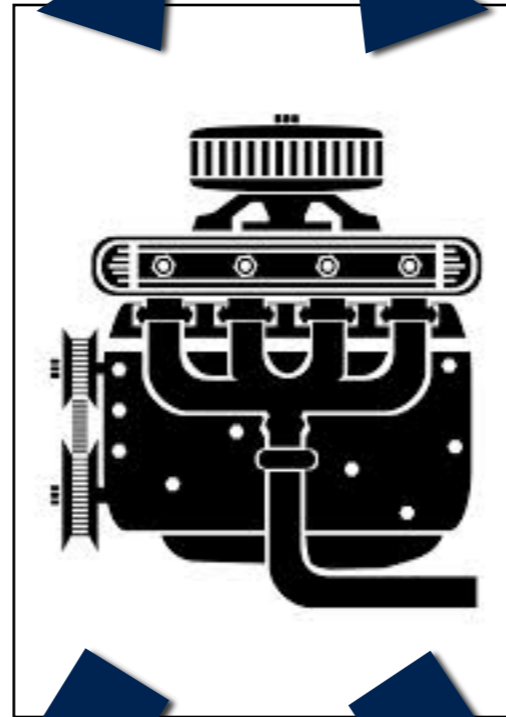
Terminator, ARMC, etc.

*“Every time a client connects to the web server, eventually web server gives a response.”*

Rich properties, but TL tools  
only for hardware verification.

C  
program

Temporal  
Logic  
property



counter-  
example

Proof



# PostgreSQL StreamServer

`int StreamServerPort int family`

```
* StreamServerPort -- open a "listening"
*
* Successfully opened sockets are added to
* at the first position that isn't -1.
*
* RETURNS: STATUS_OK or STATUS_ERROR
*/
```

```
int
StreamServerPort(int family, char *hostName, unsigned short portNumber,
                 char *unixSocketName,
                 int ListenSocket[], int MaxListen)
```

```
void body()
{
```

```
    /* Initialize hint structure */
```

```
#ifdef HAVE_UNIX_SOCKETS
    if (family == AF_UNIX)
    {
```

```
        /* Lock_AF_UNIX will also fill in sock_path. */
        /* if (Lock_AF_UNIX(portNumber, unixSocketName) != STATUS_OK) */
        /*      return STATUS_ERROR; */
        service = sock_path;
```

```
    }
```

```
    else
```

```
#endif /* HAVE_UNIX_SOCKETS */
```

```
    {
        snprintf(1, sizeof(1), "%d", portNumber);
        service = 1;
    }
```

```
    ret = getaddrinfo_all(hostName, service, &hint, &addrs);
```

```
    if (ret || !addrs)
```

```
    {
```

```
        if (hostName) {
```

```
            /* ereport(LOG, *
```

```
    } else
```

$(G \neg \text{error}) \Rightarrow F (\text{added} > 0 \wedge F \text{ret} = \text{OK})$

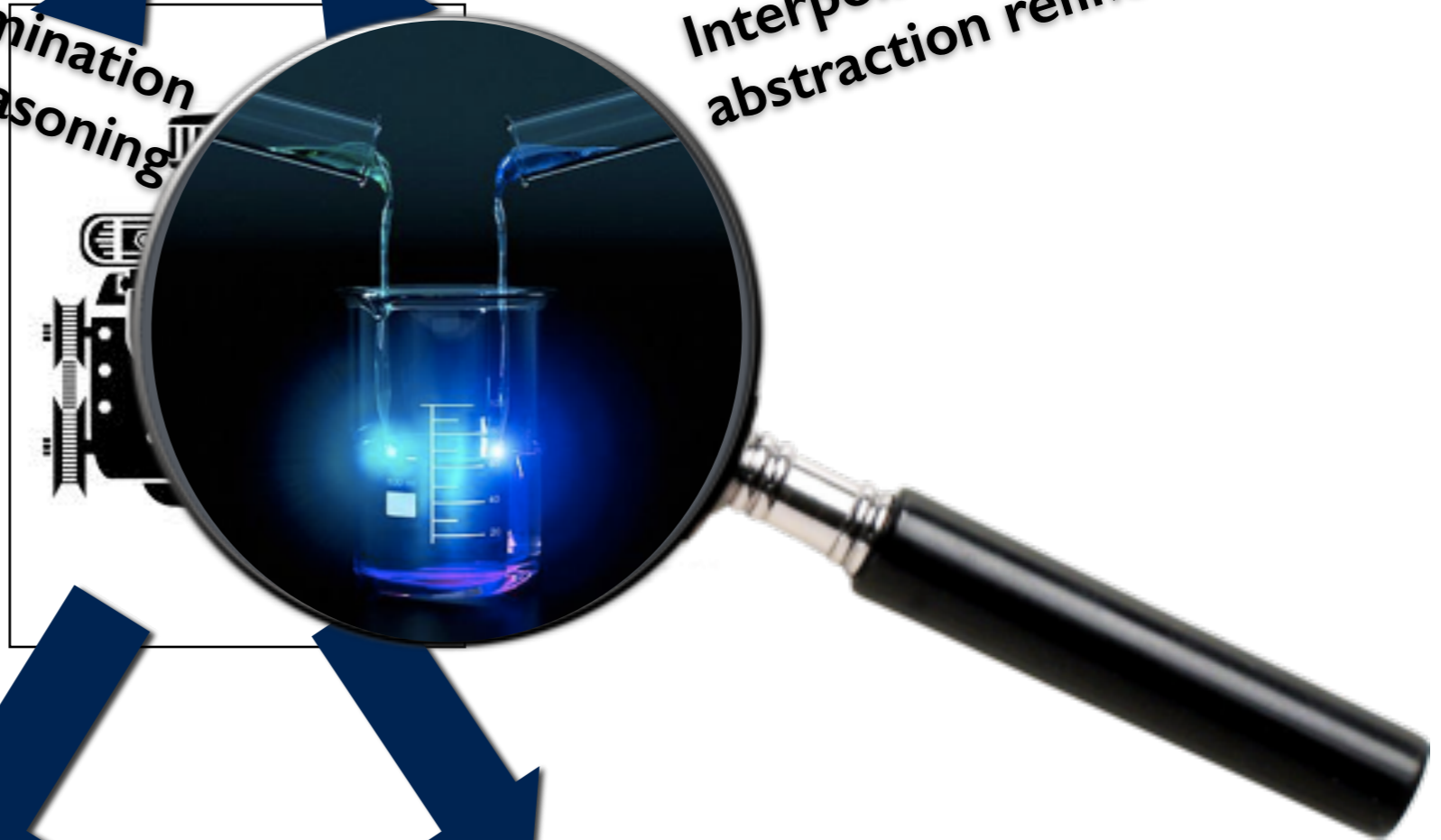
Proving this statically is challenging.  
Reason about termination.  
Reason about reachability.

C  
program

Temporal  
Logic  
property

termination  
reasoning

Interpolation,  
abstraction refinement



counter-  
example

Proof

C program

Temporal Logic property

termination reasoning

Interpolation, abstraction refinement



**Reasoning about Nondeterminism in Programs**

Byron Cook  
Microsoft Research Cambridge & University College London

Eric Koskinen\*  
New York University

**Abstract**  
Branching-time temporal logics (e.g. CTL, CTL\*, modal  $\mu$ -calculus) allow us to ask sophisticated questions about the nondeterminism that appears in systems. Applications of this type of reasoning include planning, games, security analysis, disproving, precondition synthesis, environment synthesis, etc. Unfortunately, existing automatic branching-time verification tools have limitations that have traditionally restricted their applicability (e.g. push-down systems only, universal path quantifiers only, etc.).  
In this paper we introduce an automation strategy that lifts many of these previous restrictions. Our method works reliably for properties with non-trivial mixtures of universal and existential

Here we are using two temporal operators:  
\*  $A[a \ W \ b]$  specifies that  $a$  and  $b$  are temporally sequenced in all executions through the system: either  $a$  might happen forever.

method of proving temporal properties. We observe that, temporal reasoning can be essential for reasoning about backtracking, eventuality checking, etc. When naturally performed by analysis tools (e.g. termination arguments (e.g. [3, 6, 17]), we can implement tem-

PLDI'13

**Temporal property verification as a program analysis task**

Byron Cook<sup>1</sup>, Eric Koskinen<sup>2</sup>, and Moshe Vardi<sup>3</sup>

<sup>1</sup>Microsoft Research and Queen Mary University of London  
<sup>2</sup>University of Cambridge  
<sup>3</sup>Rice University

We describe a reduction from temporal property verification to program analysis. We produce an encoding which, with recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (e.g. backtracking, eventuality checking, tree counters for branching-time properties, abstraction refinement, etc.). Examples drawn from the PostgreSQL database server, Apache httpd, and Windows OS kernel, we demonstrate the practical value of our work.

**Abstract**  
Temporal property verification is a program analysis task. We describe a reduction from temporal property verification to program analysis. We produce an encoding which, with recursion and nondeterminism, enables off-the-shelf program analysis tools to naturally perform the reasoning necessary for proving temporal properties (e.g. backtracking, eventuality checking, tree counters for branching-time properties, abstraction refinement, etc.). Examples drawn from the PostgreSQL database server, Apache httpd, and Windows OS kernel, we demonstrate the practical value of our work.

CAV'11 Award

**Making Prophecies with Decision Predicates**

Eric Koskinen  
University of Cambridge  
ejk39@cam.ac.uk

temporal properties expressed in CTL without fairness can be proved in a purely syntax-directed manner using state-based reasoning techniques, whereas LTL requires deeper reasoning about whole sets of traces and the subtle relationships between families of them.  
In this paper we aim to make an LTL prover for infinite-state programs with performance closer to what one would expect from a CTL prover. We use the observation that VCTL without fairness can be a useful abstraction of LTL. The problem with this strategy is that the pieces don't always fit together: there are cases when, due to some instances of nondeterminism in the transition system, VCTL alone is not powerful enough to prove an LTL property.  
In these cases our LTL prover works around the problem using something we call *decision predicates*, which are used to characterize and treat such instances of nondeterminism. A decision predicate is represented as a pair of first-order logic formulae  $(a, b)$ , where the formula  $a$  defines the decision predicate's presupposition (i.e. when the decision is made), and  $b$  characterizes the binary choice made when this presupposition holds. Any transition from state  $s$  to state  $s'$  in the system that meets the constraint  $a(s) \wedge b(s')$  is distinguished by the decision predicate  $(a, b)$  from  $a(s) \wedge \neg b(s')$ .  
We use decision predicates as the basis of a partial symbolic determinization procedure: for each predicate we introduce a new prophecy variable [3] to predict the future outcome of the decision. After partially determinizing with respect to these prophecy variables, we find that CTL proof methods succeed, thus allowing us to prove LTL properties with CTL proof techniques in cases where

2.4 [Software Engineering: Model checking; Correctness of Systems]; Reliability of Programs; Specification of Programs; F.3.2 [Logic Programming Languages]

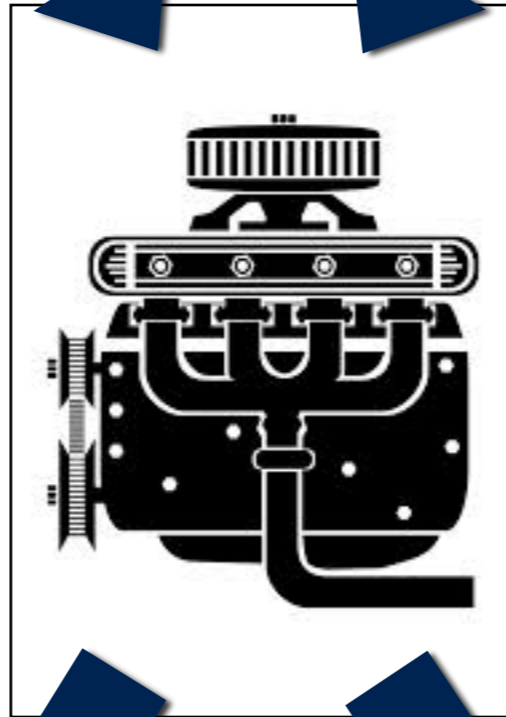
verification, termination, server, and the Windows OS kernel.

POPL'11

Apache,  
PostgreSQL,  
Windows OS code

C  
program

Temporal  
Logic  
property



counter-  
example

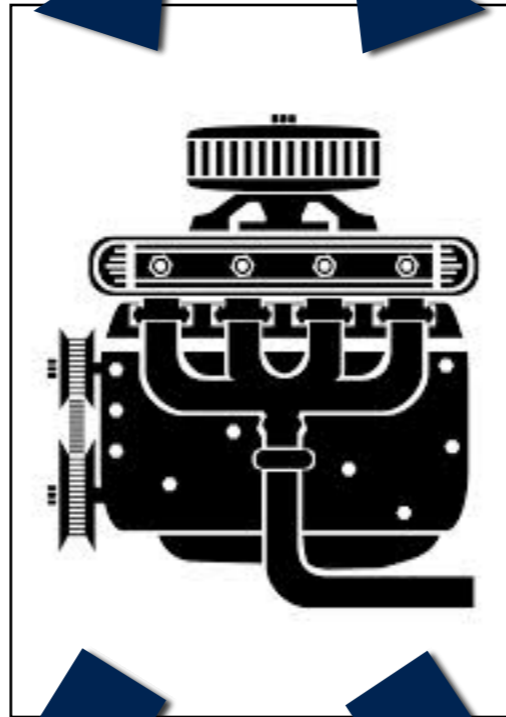
Proof



*Higher-order  
programs*

Java, C#,  
Scala, ML

Temporal  
Logic  
property



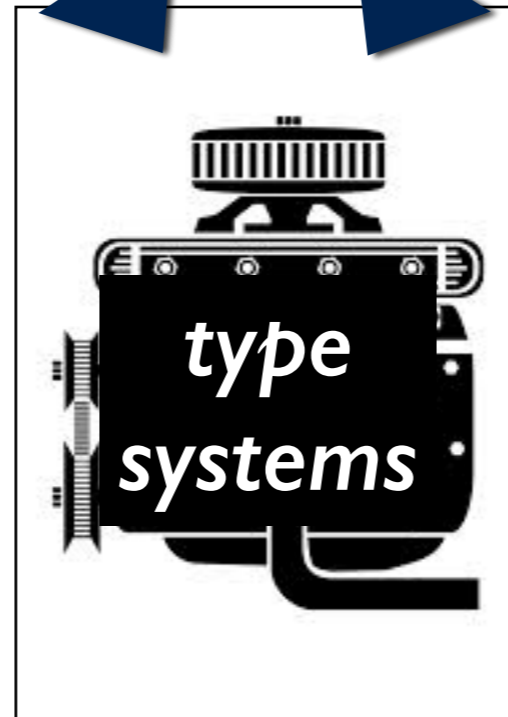
counter-  
example

Proof

*Higher-order  
programs*

Java, C#,  
Scala, ML

Temporal  
Logic  
property



```
let rec halt _ =  
  halt ()  
and shrink f =  
  if ( f() = 0 ) then  
    halt ()  
  else  
    shrink (λ_. f() - 1)  
  
and main() =  
  let t = *+ in  
    shrink (λ_. t)
```



Events

```
let rec halt _ = ev[halt];  
  halt ()  
and shrink f = ev[shrink];  
  if ( f() = 0 ) then  
    halt ()  
  else  
    shrink (λ_. f() - 1)  
  
and main() = ev[main];  
  let t = *+ in  
    shrink (λ_. t)
```

**main X (shrink U halt)**

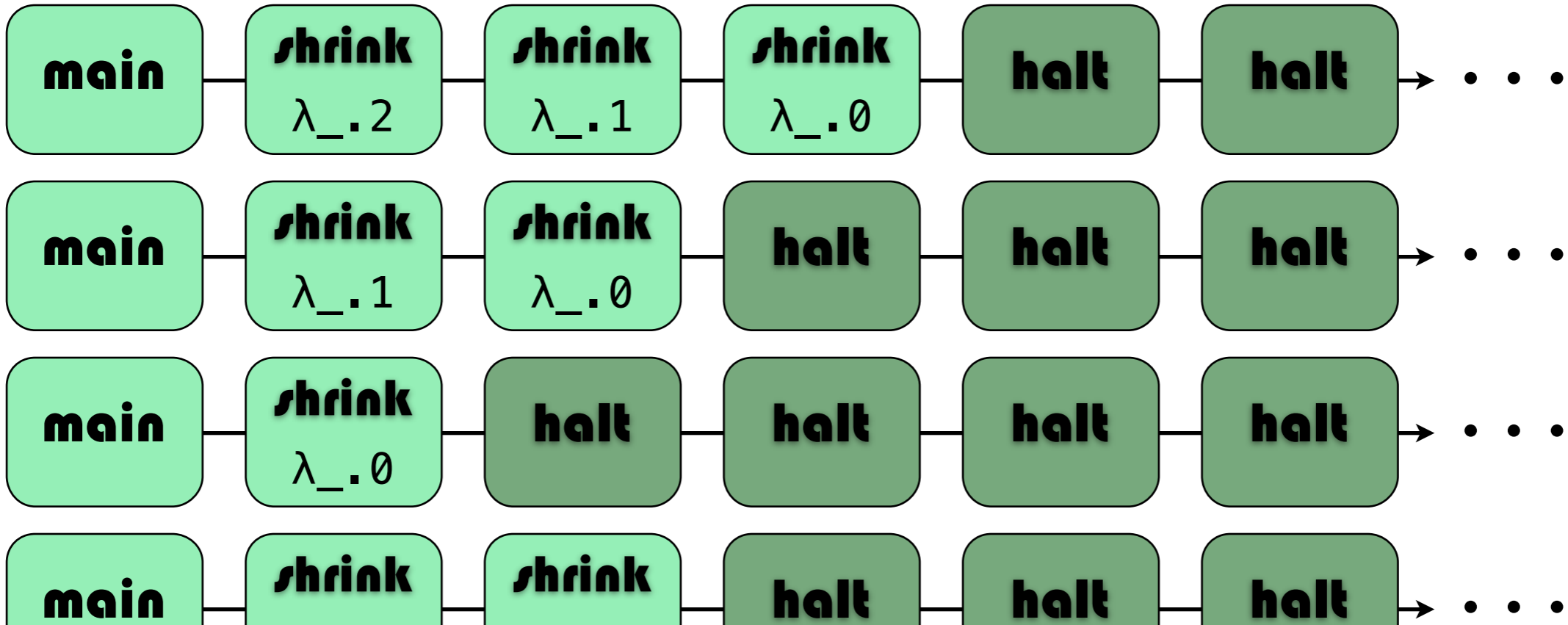
```

let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)

and main() =
  let t = *+ in
    shrink (λ_. t)

```

**main X (shrink U halt)**



Infinite  
state space

```
let rec halt _ =  
  halt ()  
and shrink f =  
  if ( f() = 0 ) then  
    halt ()  
  else  
    shrink (λ_. f() - 1)  
  
and main() =  
  let t = *+ in  
    shrink (λ_. t)
```

Reachability

Higher  
Order

Non-  
determinism

**main X (shrink U halt)**

Termination

***No previous technique can prove this property.***

*Previously:*

- Expressive logics, but finite data [K/O:LICS'09]
- Infinite data, but just safety [Terauchi:POPL'10]
- Expressive logics, but first-order programs [CK:PLDI'13]

shrink ( $\lambda_.$  t)

(**shrink** **U** halt)

Terminates  
or diverges?

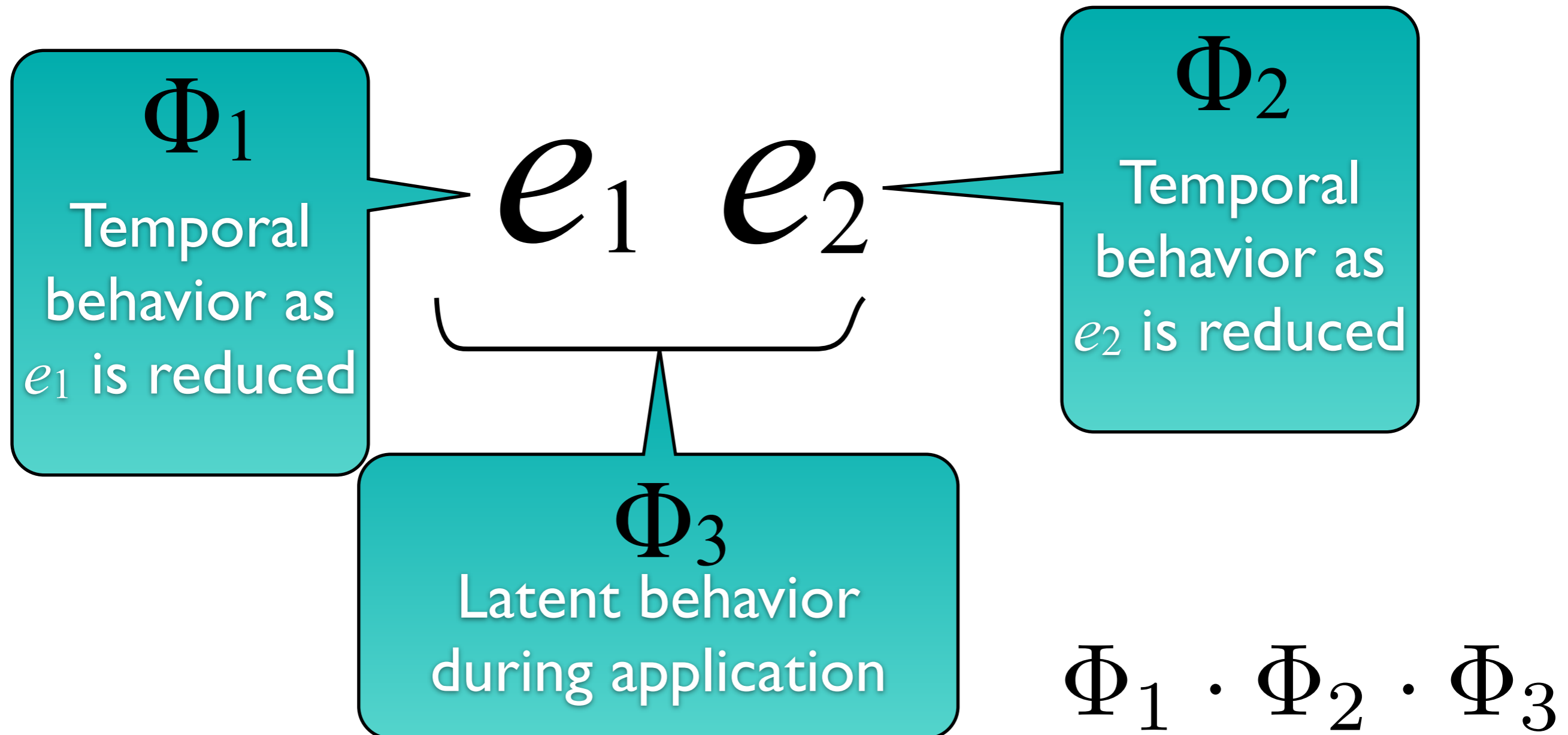
$e_1$   $e_2$

Terminates  
or diverges?

Terminates  
or diverges?

# Decompose in two ways

I. Divide up program into expressions



# Decompose in two ways

I. Divide up program into expressions

*Characterize temporal behavior of exprs. via type-and-effect:*

$$\Gamma \vdash e_1 e_2 : \tau \ \& \ \Phi$$

Typing  
environment

Dependent  
Type

Temporal  
Effect

# Decompose in two ways

I. Divide up program into expressions

*Characterize temporal behavior of exprs. via type-and-effect:*

$$\Gamma \vdash e_1 e_2 : \tau \ \& \ \Phi$$

Typing  
environment

Dependent  
Type

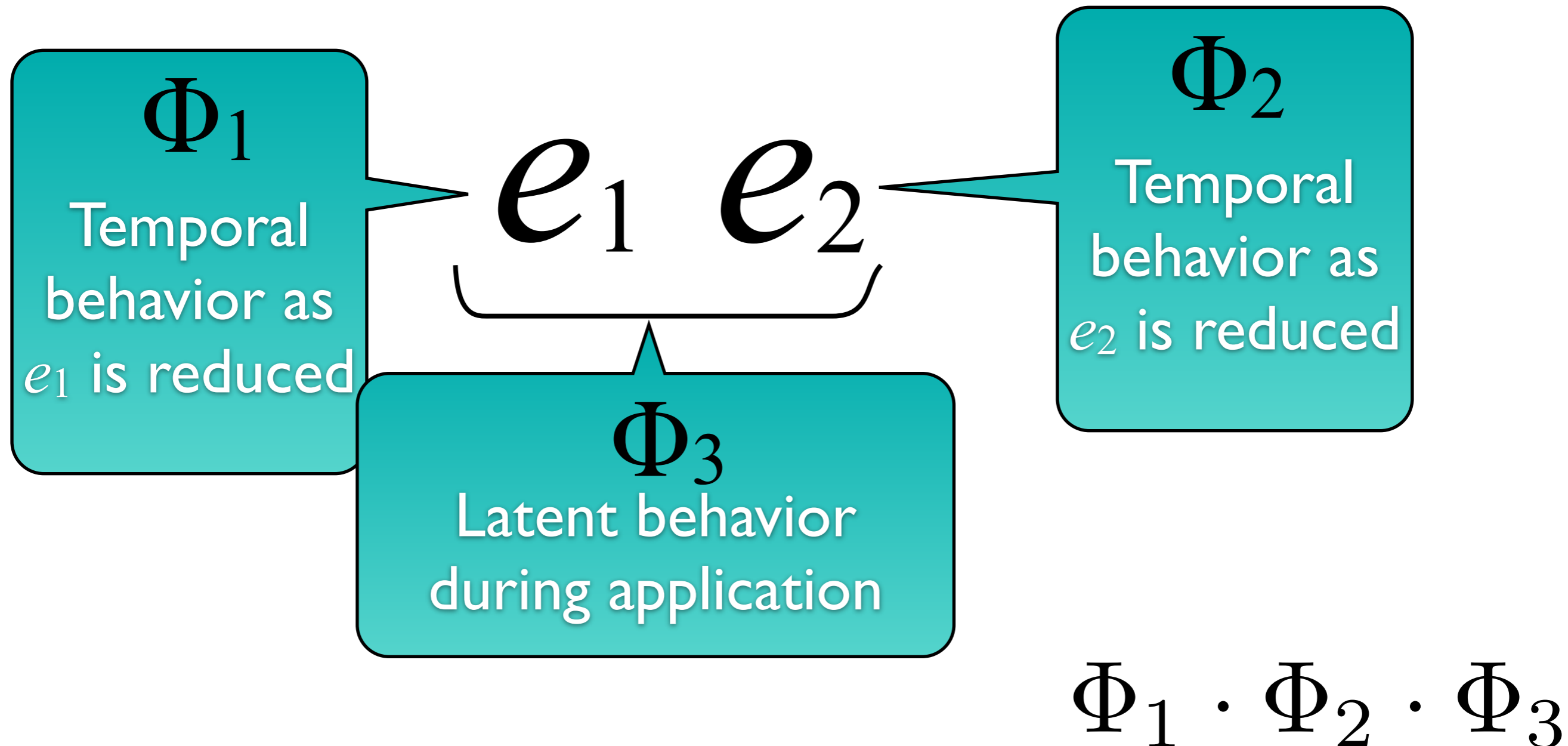
Temporal  
Effect



# Decompose in two ways

1. Divide up program into expressions

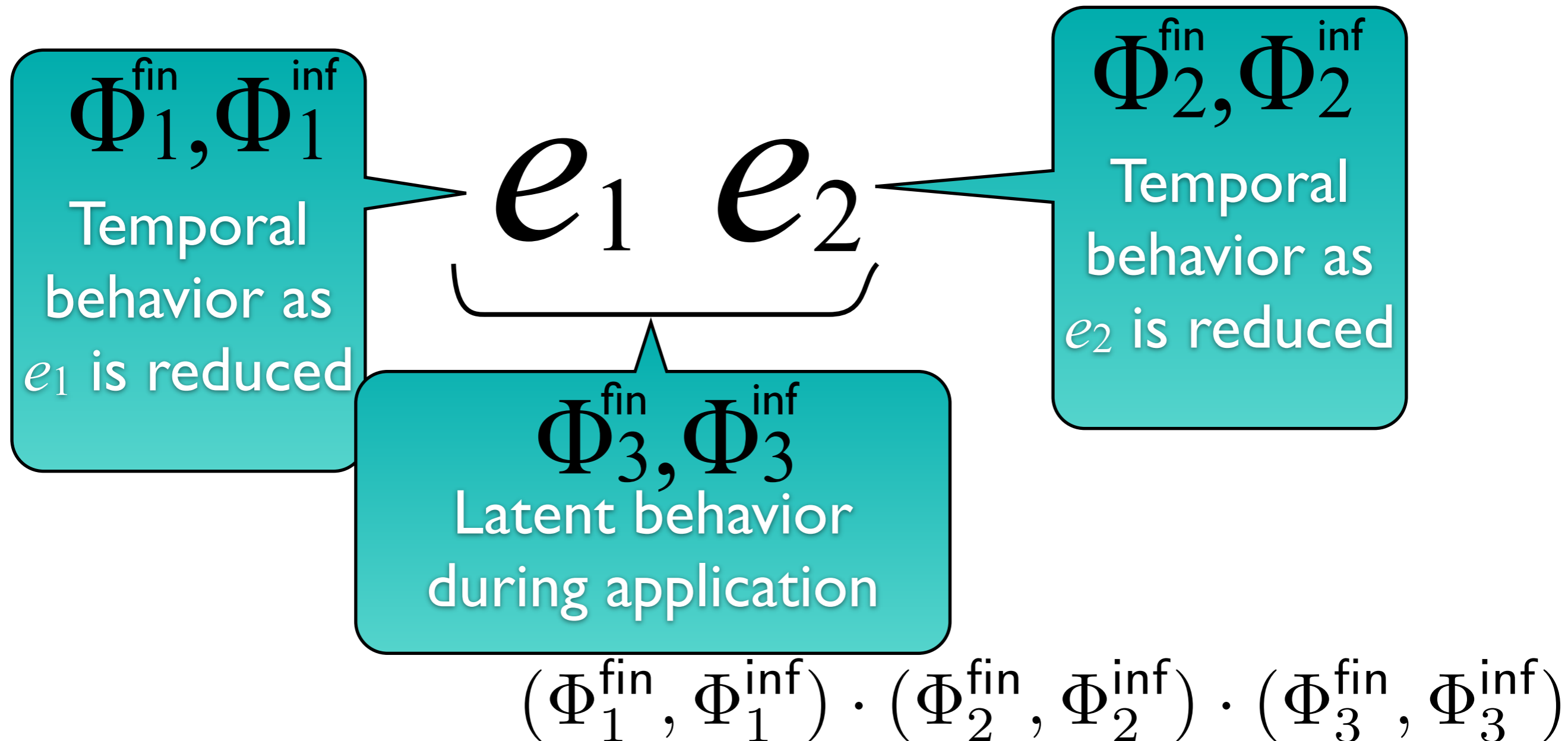
2. Track behavior of *finite* traces separate from *infinite* traces



# Decompose in two ways

1. Divide up program into expressions

2. Track behavior of *finite* traces separate from *infinite* traces



# Decompose in two ways

1. Divide up program into expressions

2. Track behavior of *finite* traces separate from *infinite* traces

$$\Gamma \vdash e : \tau \ \& \ (\Phi^{\text{fin}}, \Phi^{\text{inf}})$$

```
let rec halt _ =  
  halt ()  
and shrink f =  
  if ( f() = 0 ) then  
    halt ()  
  else  
    shrink (λ_. f() - 1)  
  
and main() =  
  let t = *+ in  
    shrink (λ_. t)
```

**main X (shrink U halt)**

```

let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)

and main() =
  let t = *+ in
    shrink (λ_. t)

```

**(shrink U halt)**

Latent Effect

*Safety*: the latent behavior when shrink is applied

$$\Gamma \vdash \text{shrink} : (\text{unit} \rightarrow \text{int}) \xrightarrow{\text{shrink } \forall \text{halt}} \text{unit} \&(\varepsilon, \varepsilon)$$

```

let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)

and main() =
  let t = *+ in
    shrink (λ_. t)

```

refinement types

“all traces  
exit shrink”

(shrink **U** halt)

*Safety*: the latent behavior when shrink is applied

$$\Gamma \vdash \text{shrink} : (\text{unit} \rightarrow \text{int}) \xrightarrow{\text{shrink} \vee \text{halt}} \text{unit} \& (\delta, \varepsilon)$$

*Liveness*: the conditions under which shrink terminates

$$\Gamma, f : \text{unit} \rightarrow \{i \mid i \geq 0\} \vdash \text{shrink } f : \text{unit} \& (\top, \text{F} \neg \text{shrink})$$

```

let rec halt _ =
  halt ()
and shrink f =
  if ( f() = 0 ) then
    halt ()
  else
    shrink (λ_. f() - 1)

and main() =
  let t = *+ in
    shrink (λ_. t)

```

**(shrink U halt)**

*Safety*

...

**App**

⋮

*Liveness*

**Comb**

⋮

**Sub**

$\Gamma, t : \{i \mid i \geq 0\} \vdash \text{shrink } (\lambda_. t) : \text{unit} \ \& \ (\text{shrink } \mathbf{U} \ \text{halt})$

# *Liveness in a type system ???*

*Safety*

...

**App**

⋮

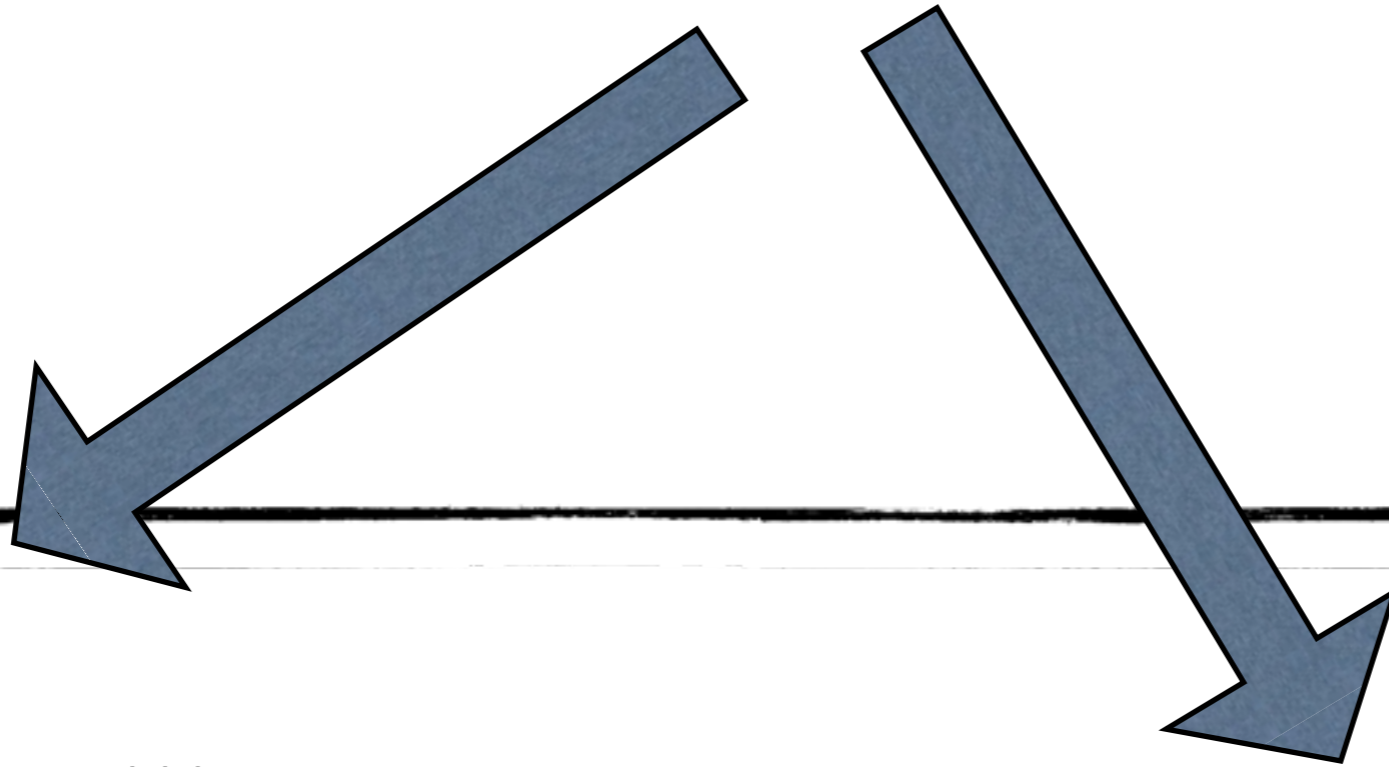
*Liveness*

**Comb**

⋮



*Where do these pieces come from?*



*Safety*

...

**App**

⋮

*Liveness*

**Comb**

⋮

The type system itself  
(solution to fixpoint eqn)

*Under what conditions  
does shrink terminate?*

Adapt work on H.O.  
termination

*Safety*

...

**App**

⋮

*Liveness*

**Comb**

⋮



**Advanced  
type systems**

*Safety*

...

**App**

⋮

*Liveness*

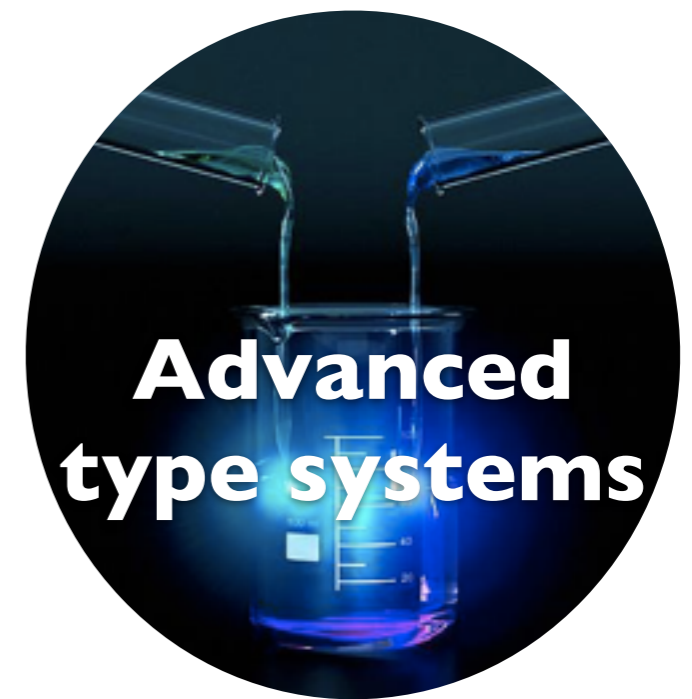
*LICS'14*

**Comb**

⋮

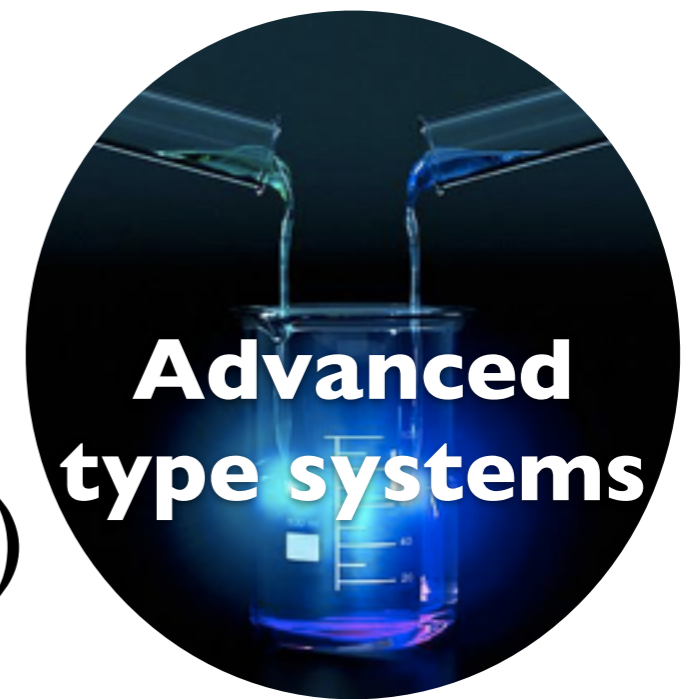
# Why is this important?

- First technique for temporal properties of higher-order, infinite-data programs
- Instantiation to wide variety of spec. logics,  
Instantiation to type environments,  
Instantiation to oracles



# Why is this important?

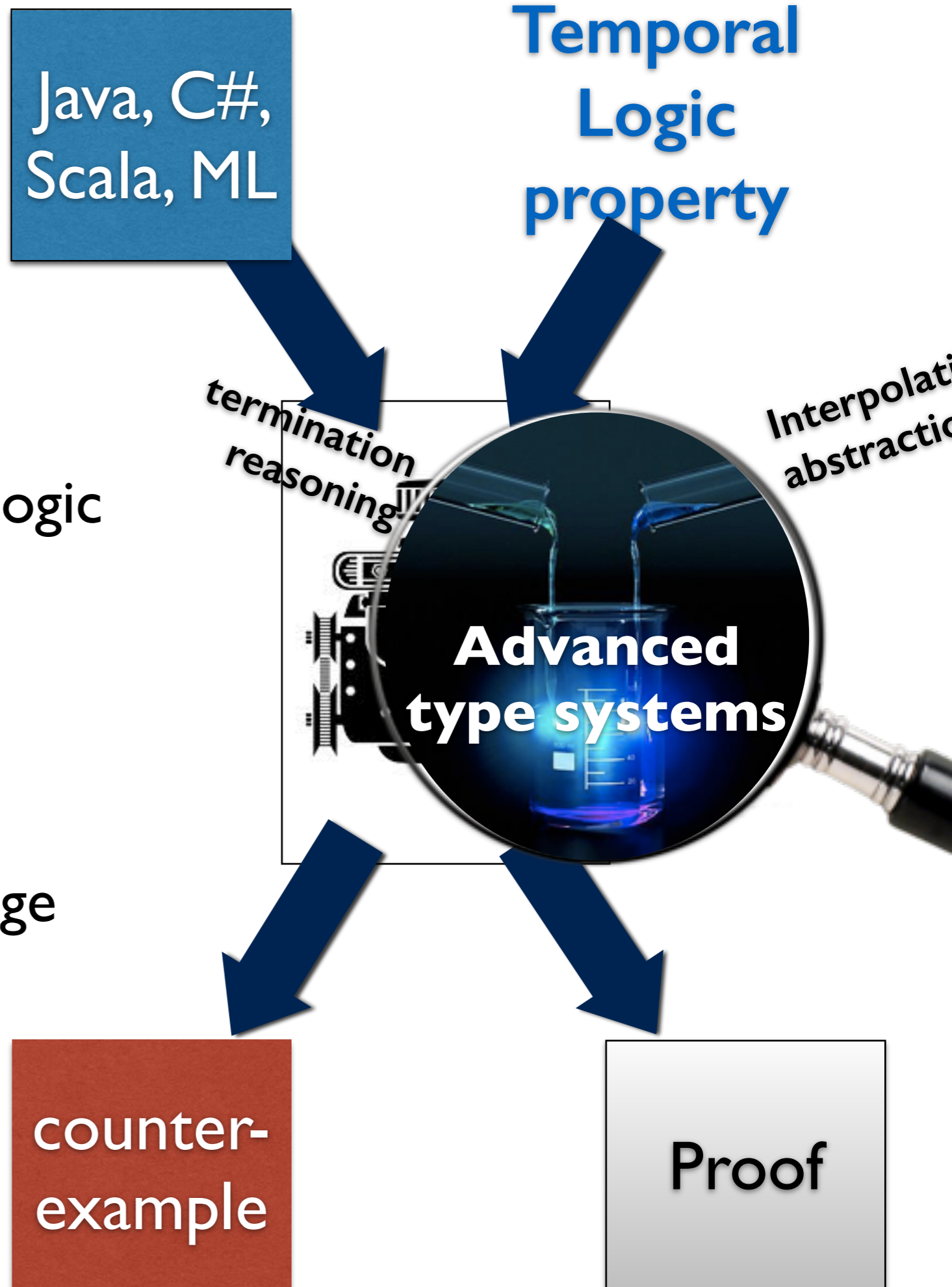
- Compositional
- Does not require input program be in continuation-passing style (CPS)
- First-order interprocedural programs



# Next Steps

## Innovations Needed!

- Type systems
- Formal methods, temporal logic
- Abstraction refinement
- Algorithms
- Scalable program analysis
- Systems, experiments
- Temporal Logic of Knowledge



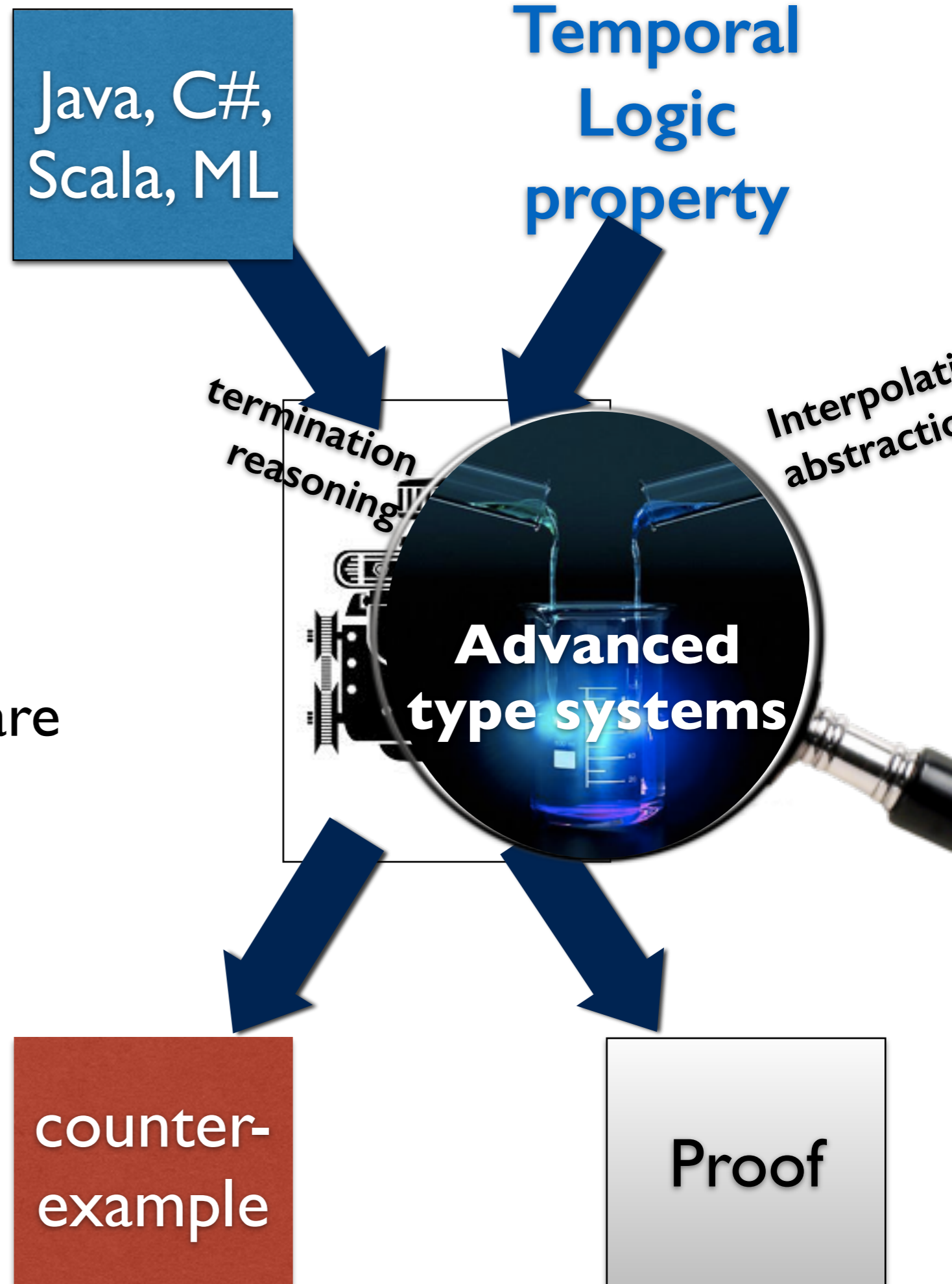
# Conclusion

## Temporal Logic

- Safety properties
- Liveness properties
- Mixtures
- Worked well for hardware
- Need techniques for software

## 2025 Languages

- Java, C#, Scala, ML



***Thank you!***