Confined Gradual Typing

Esteban Allende, Johan Fabry, Ronald Garcia, **Éric Tanter**

University of Chile   University of British Columbia

OOPSLA 2014

Static

Dynamic

[Siek & Taha]

Gradual Typing

[Siek & Taha]

Gradual Typing

Static Dynamic

equality

$$T_1 = T_2$$

[Siek & Taha]

Gradual Typing

Static Dynamic

equality **consistency**

$T_1 = T_2$ → $T_1 \sim T_2$

[Siek & Taha]

Gradual Typing

Static — Dynamic

equality

$T_1 = T_2$

consistency

$T_1 \sim T_2$

$T \sim Dyn$

$Dyn \sim T$

[Siek & Taha]

Gradual Typing

Static — Dynamic

equality

$T_1 = T_2$

→

consistency

$T_1 \sim T_2$

[Siek & Taha]

Gradual Typing

Static Dynamic

equality

$T_1 = T_2$

definitely go well

consistency

$T_1 \sim T_2$

[Siek & Taha]

Gradual Typing

Static — Dynamic

equality

$T_1 = T_2$

definitely go well

consistency

$T_1 \sim T_2$

might go well

[Siek & Taha]

Gradual Typing

Static — Dynamic

equality

$T_1 = T_2$

definitely go well

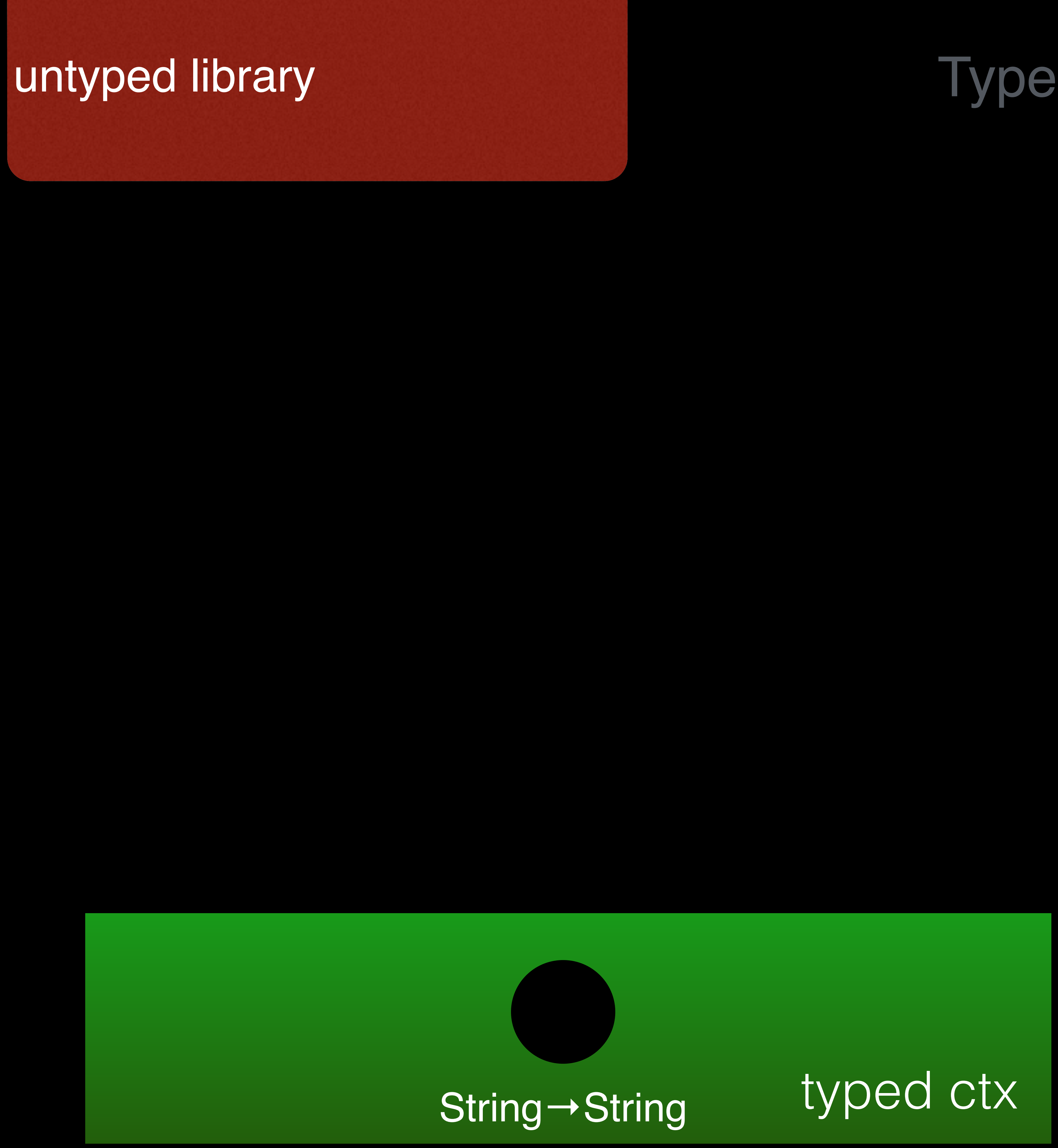
consistency

$T_1 \sim T_2$

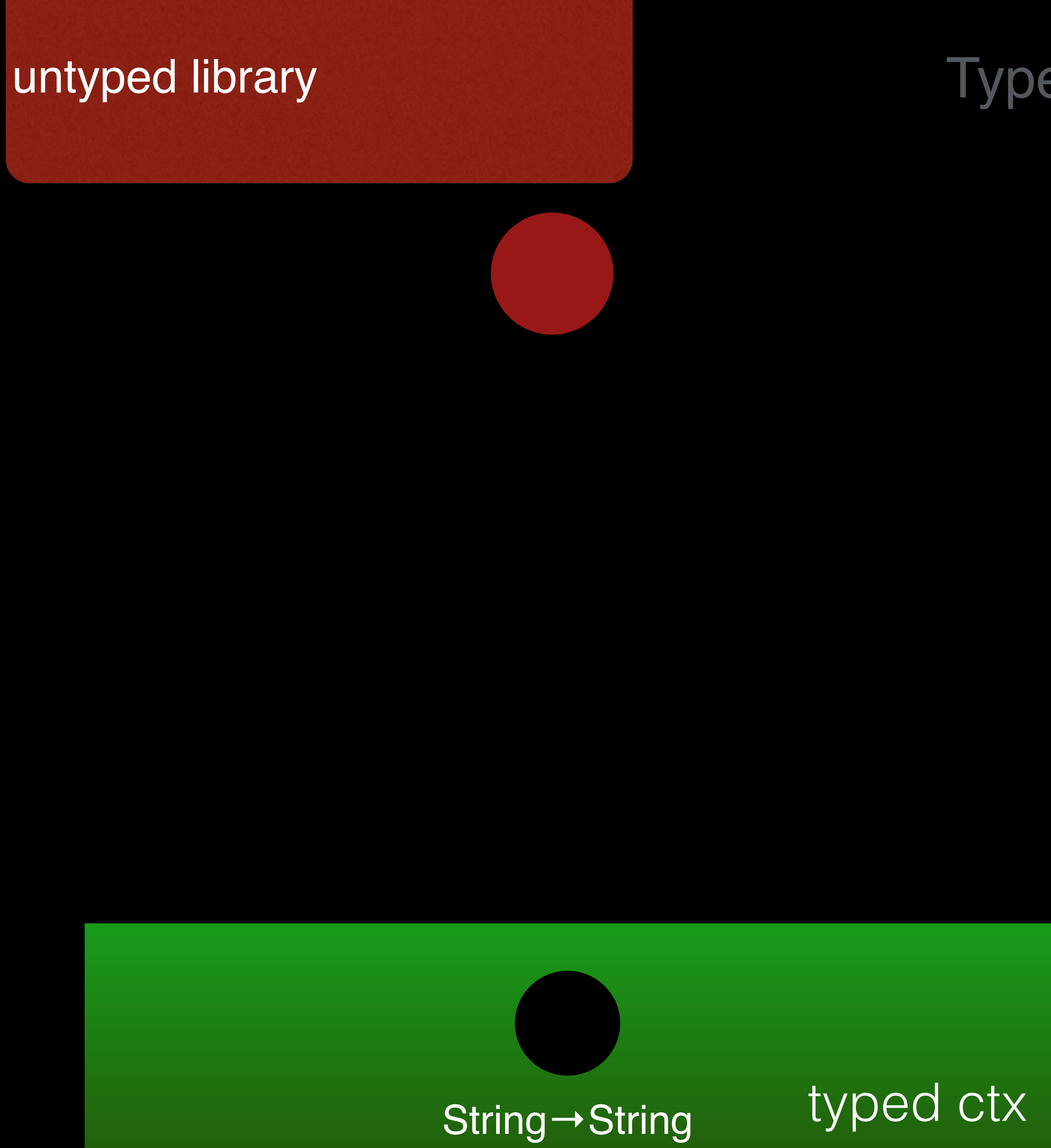
might go well

@runtime: casts

Type Checking
Runtime

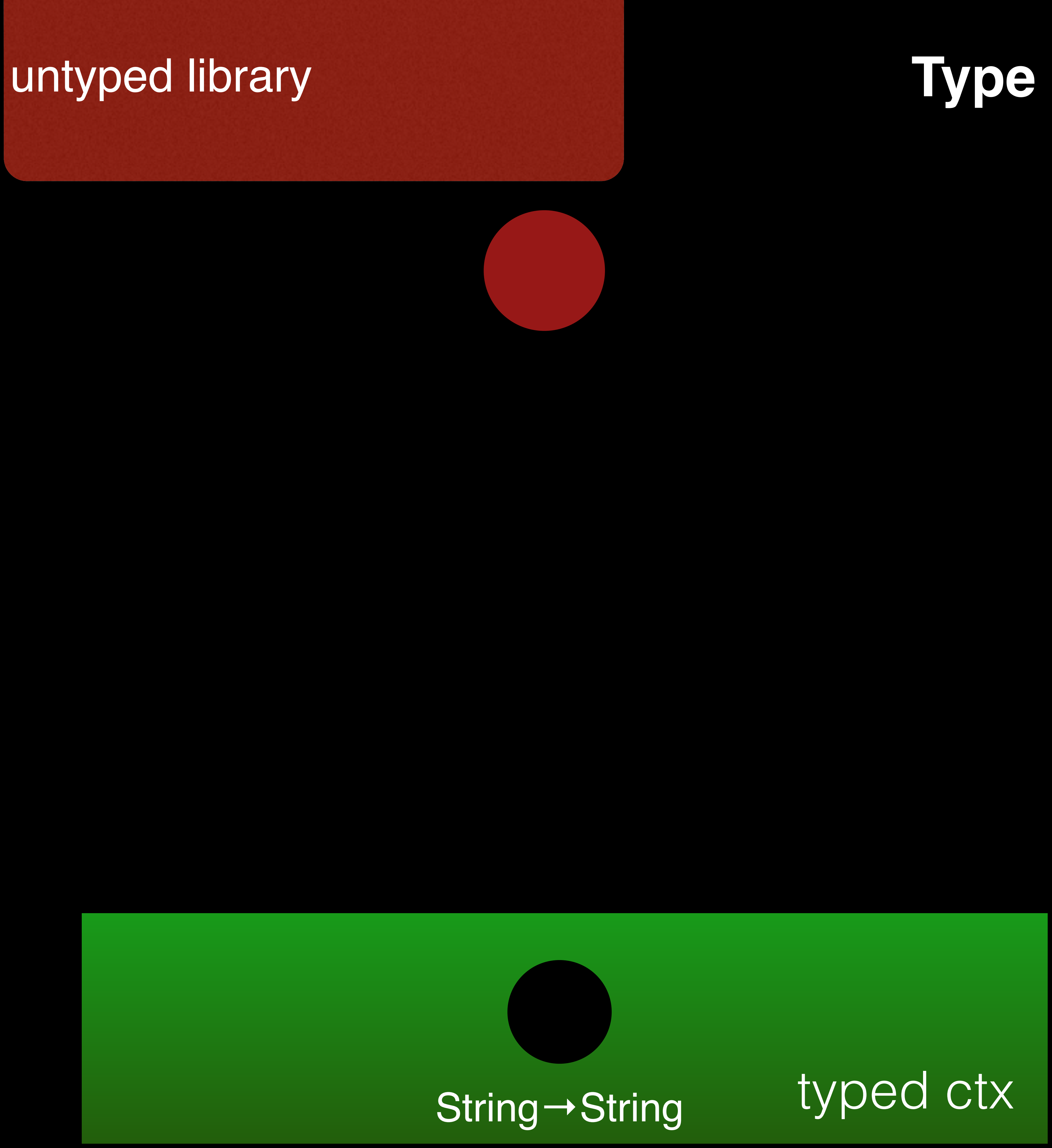


Type Checking
Runtime

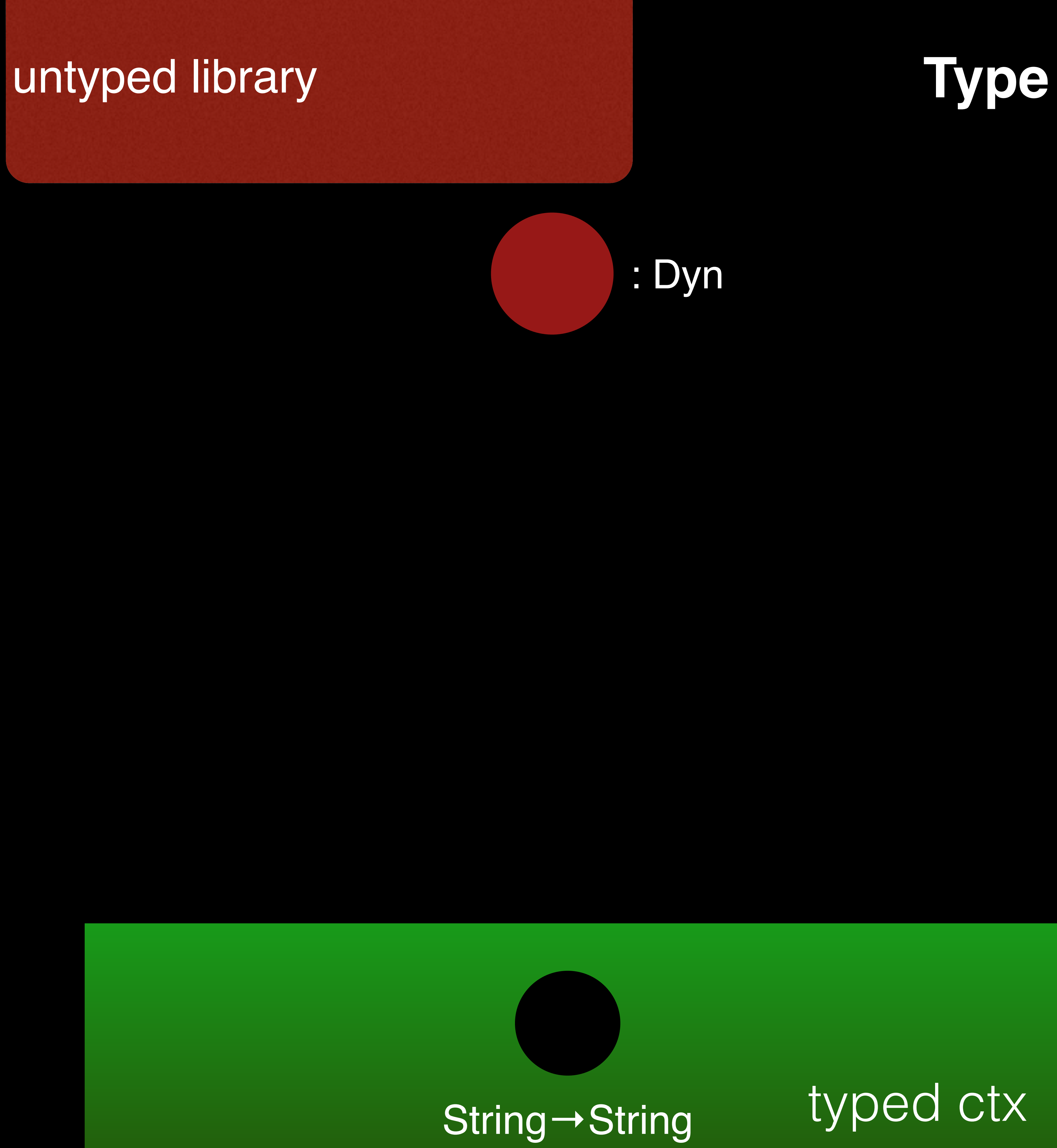


Type Checking

Runtime



Type Checking
Runtime



Type Checking
Runtime

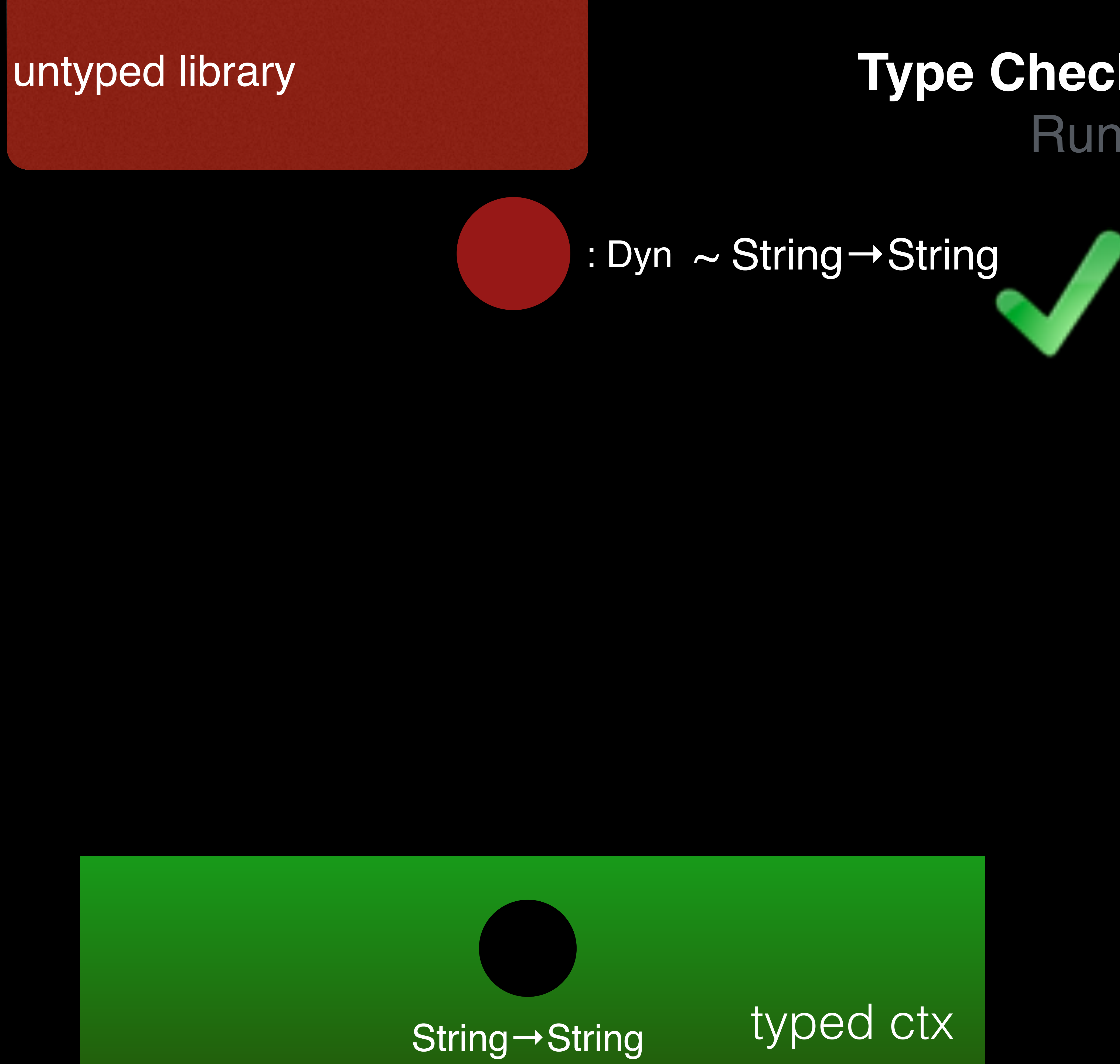
untyped library

: Dyn ~ String→String

String→String

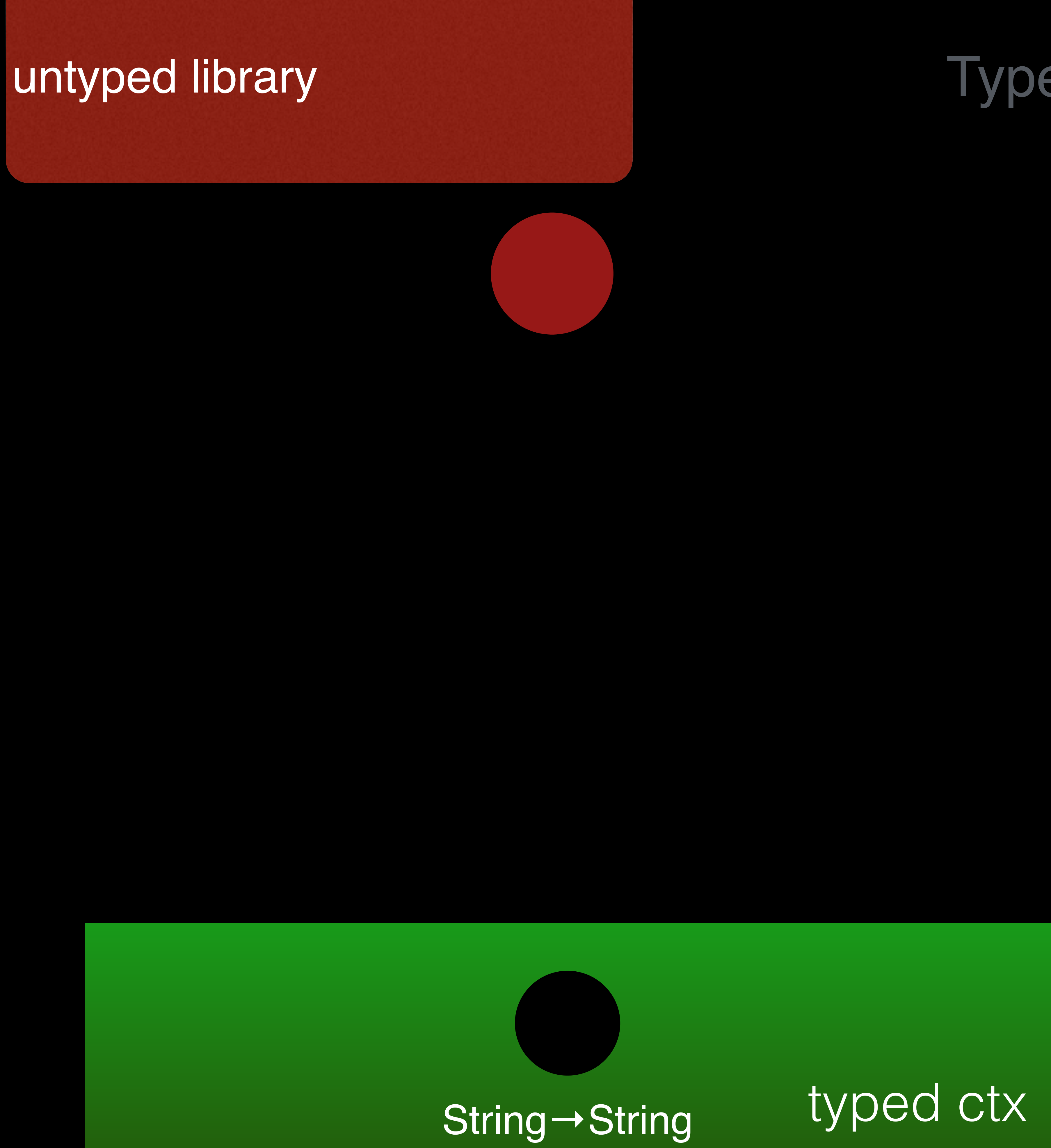
typed ctx

Type Checking
Runtime



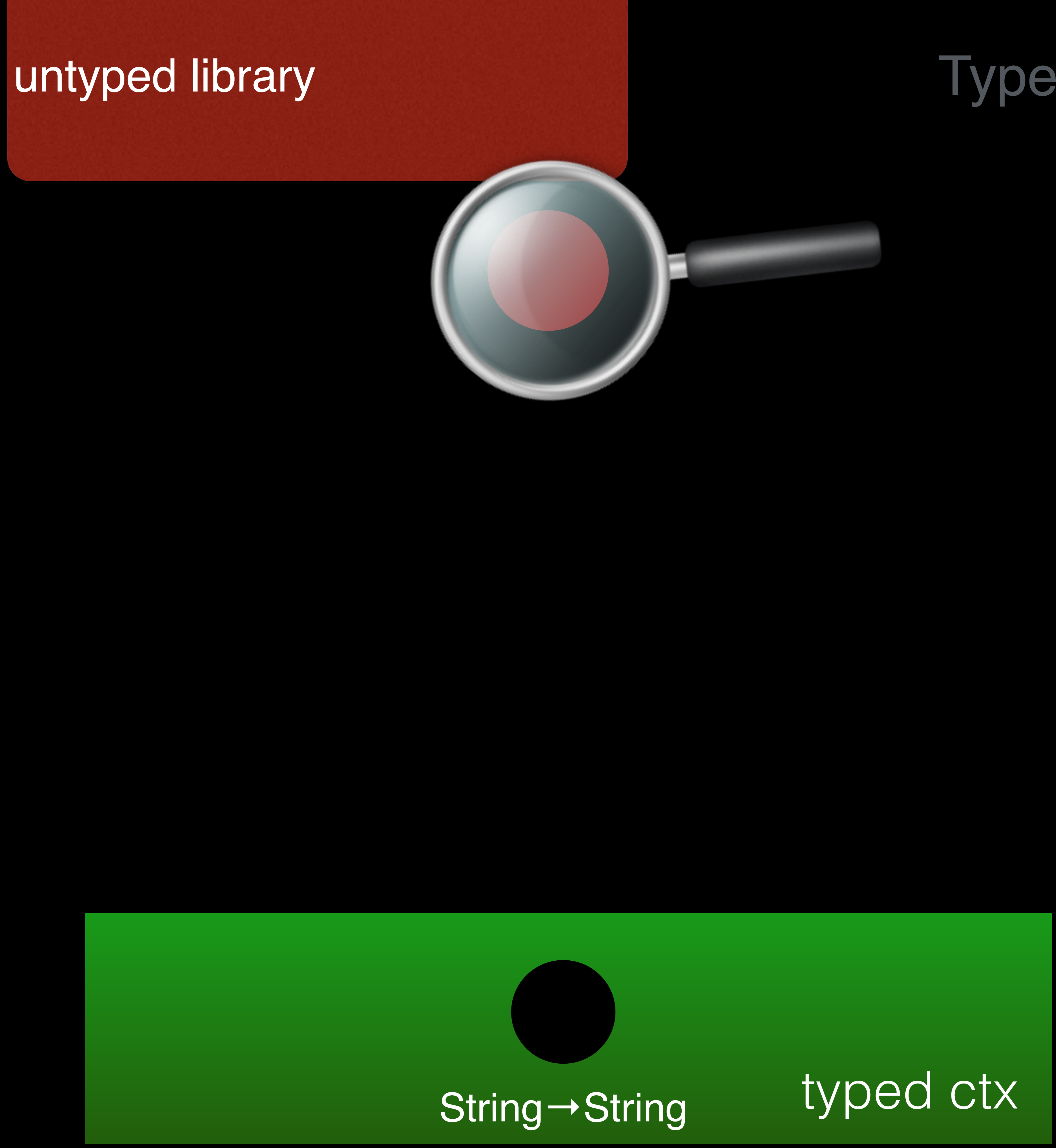
Type Checking

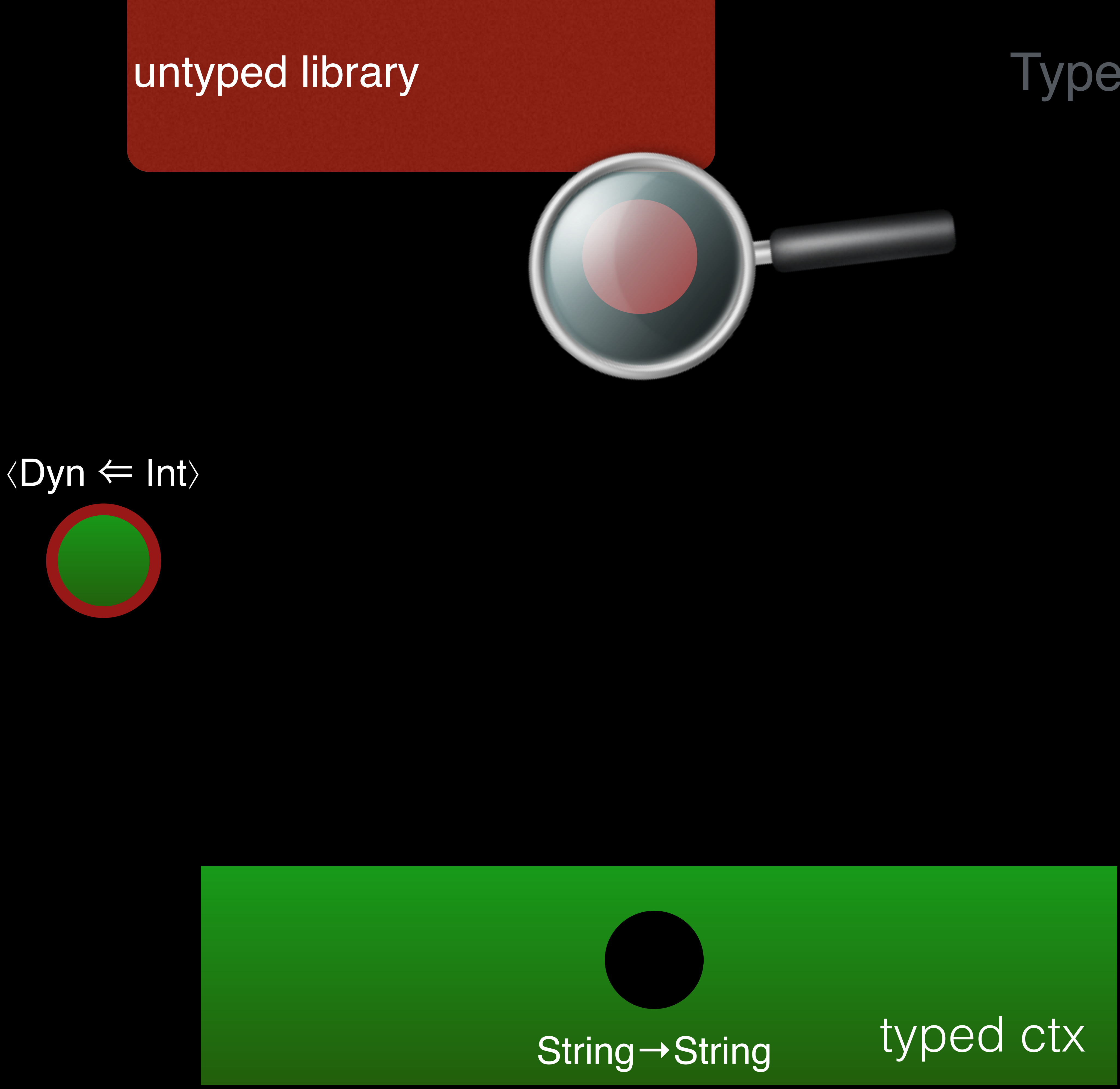
Runtime



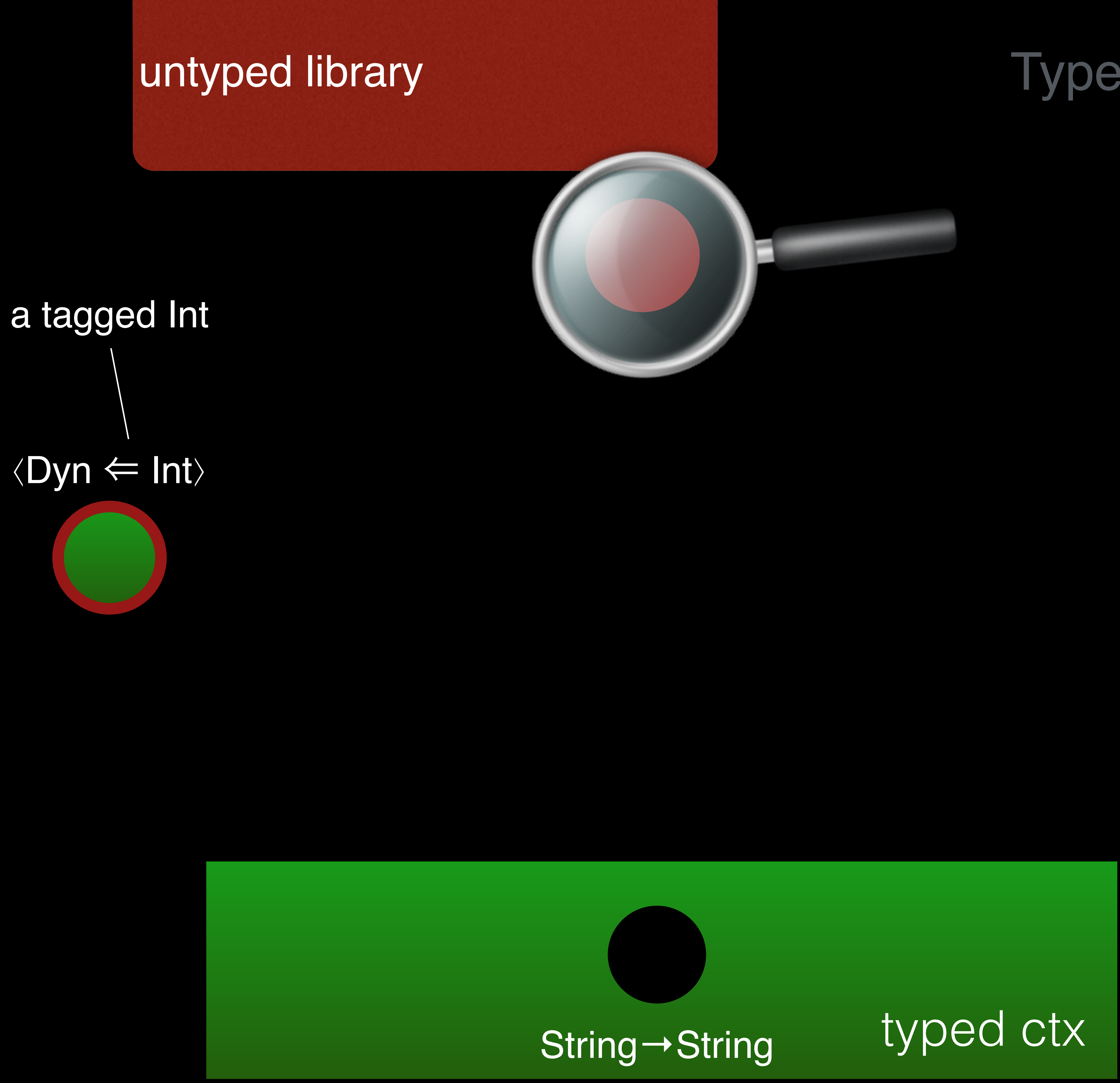
Type Checking

Runtime



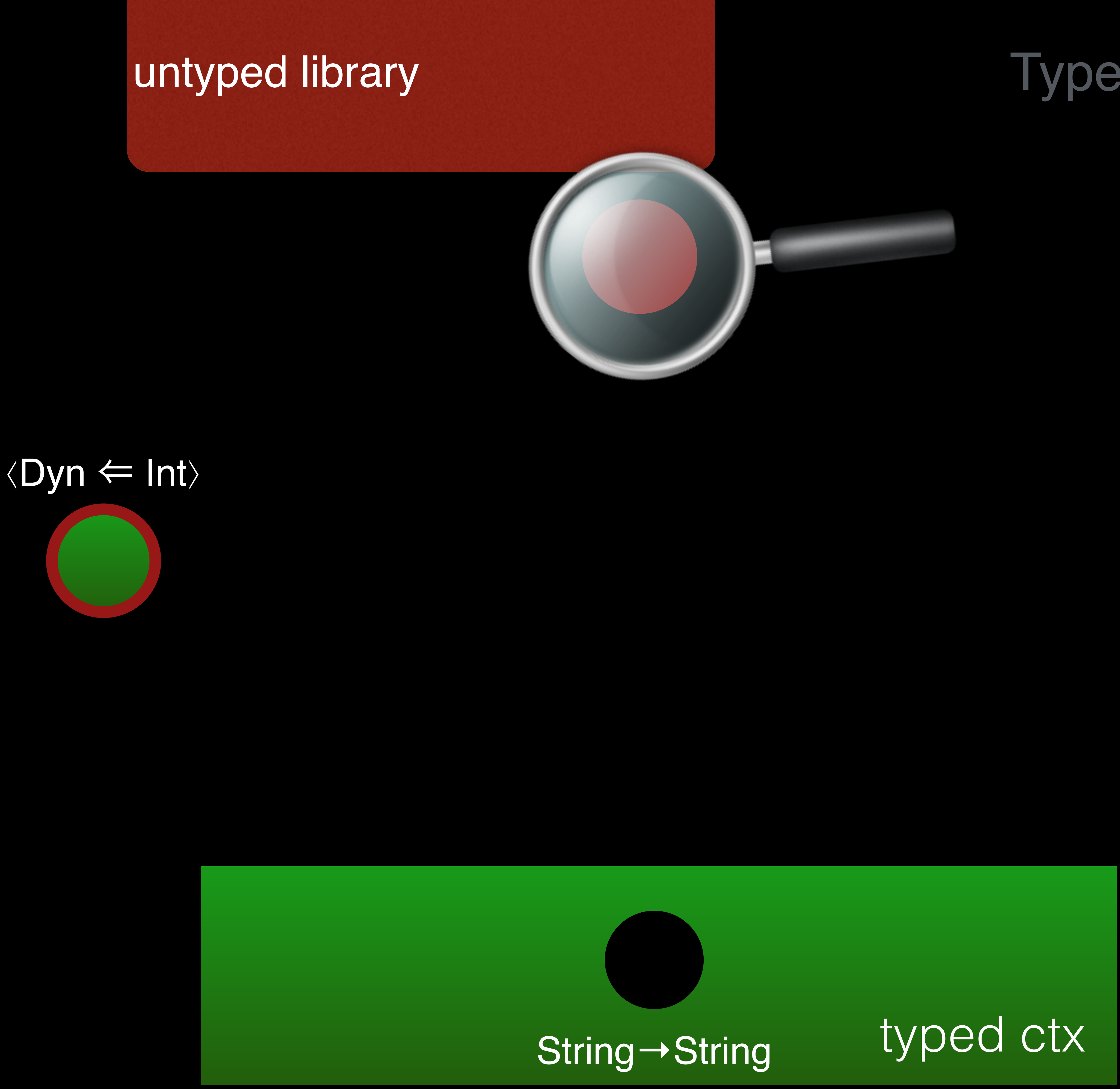

untyped library
⟨Dyn ⇐ Int⟩
String→String
typed ctx

Type Checking
Runtime

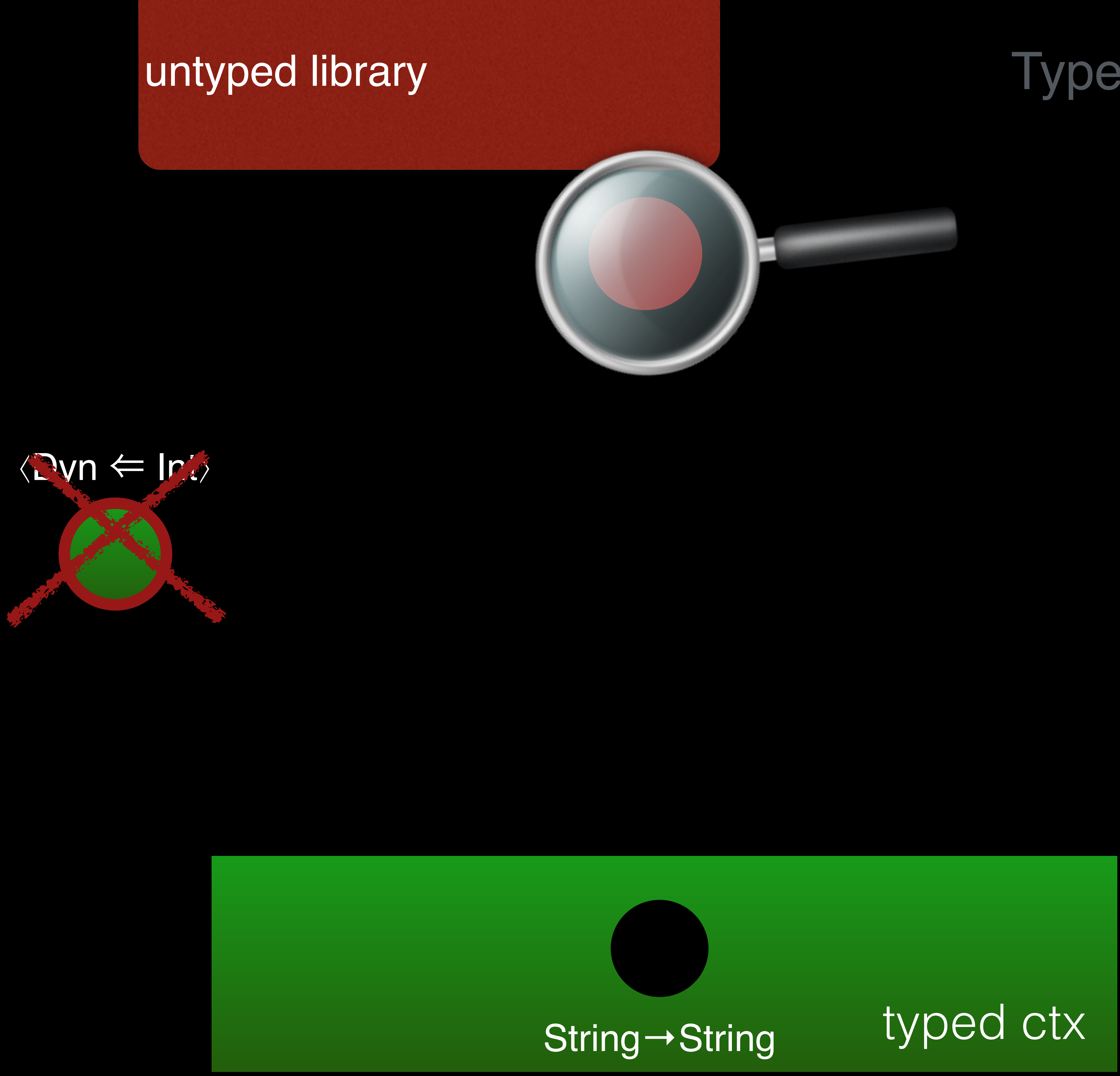


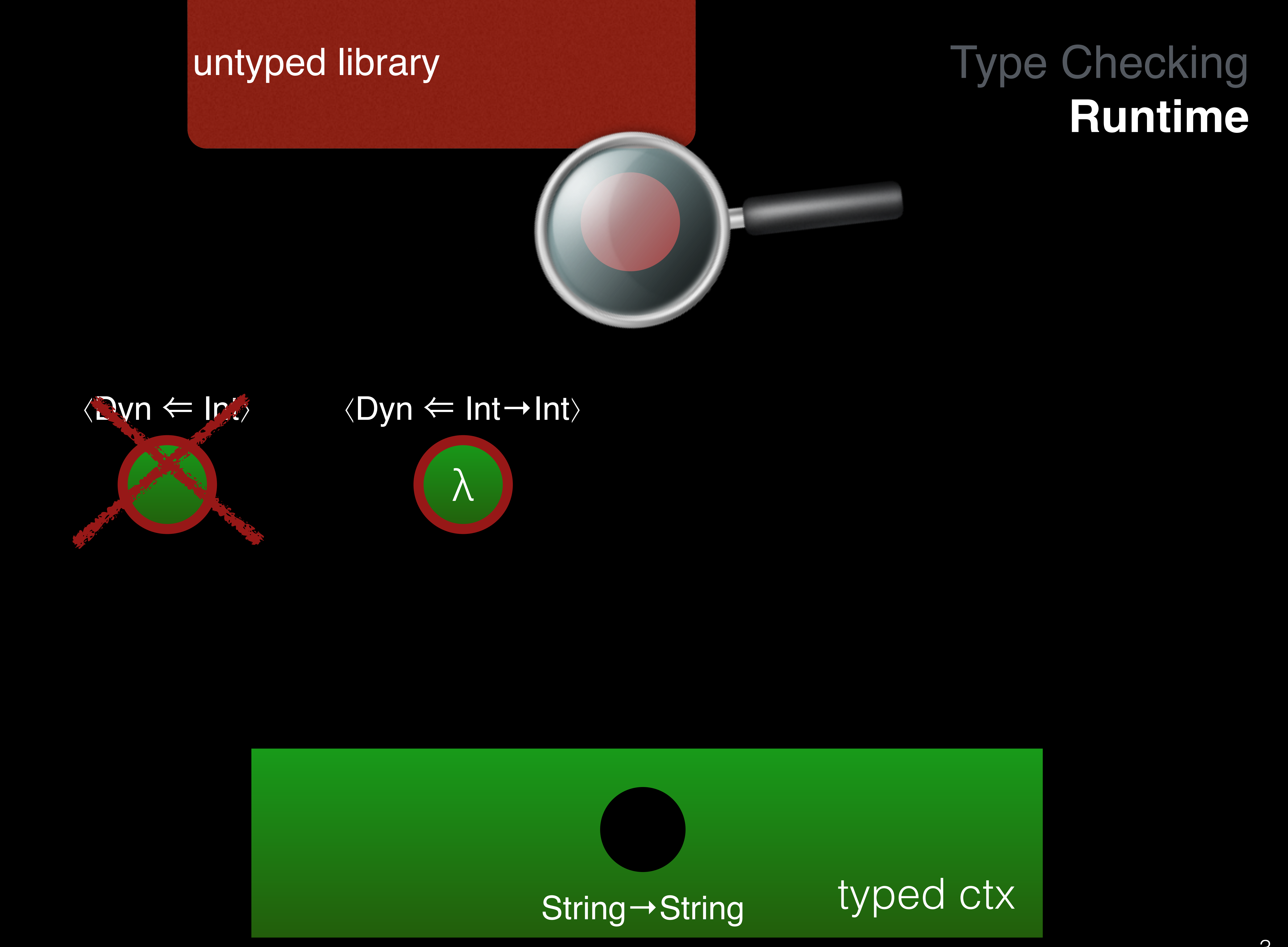
Type Checking

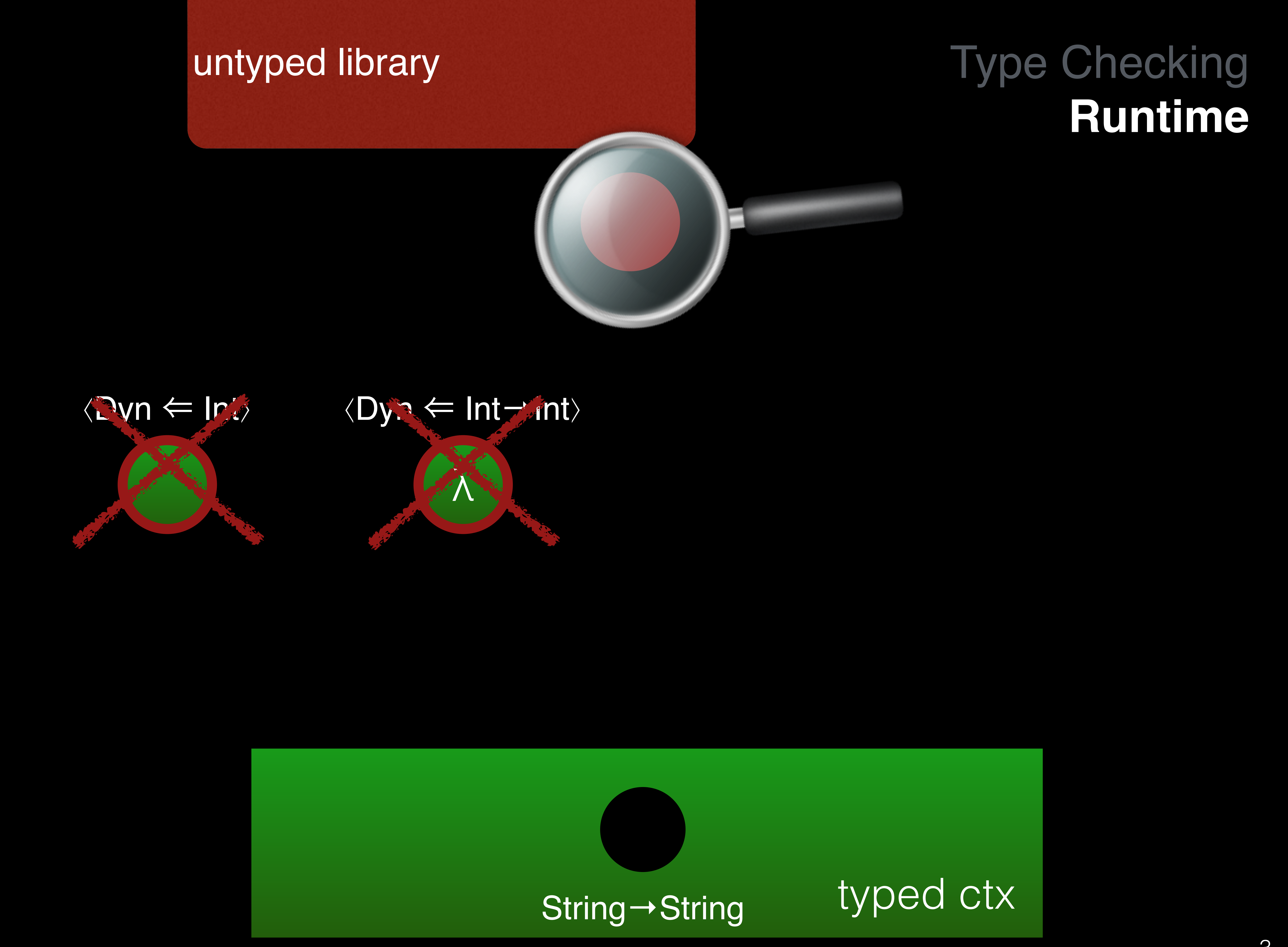
Runtime



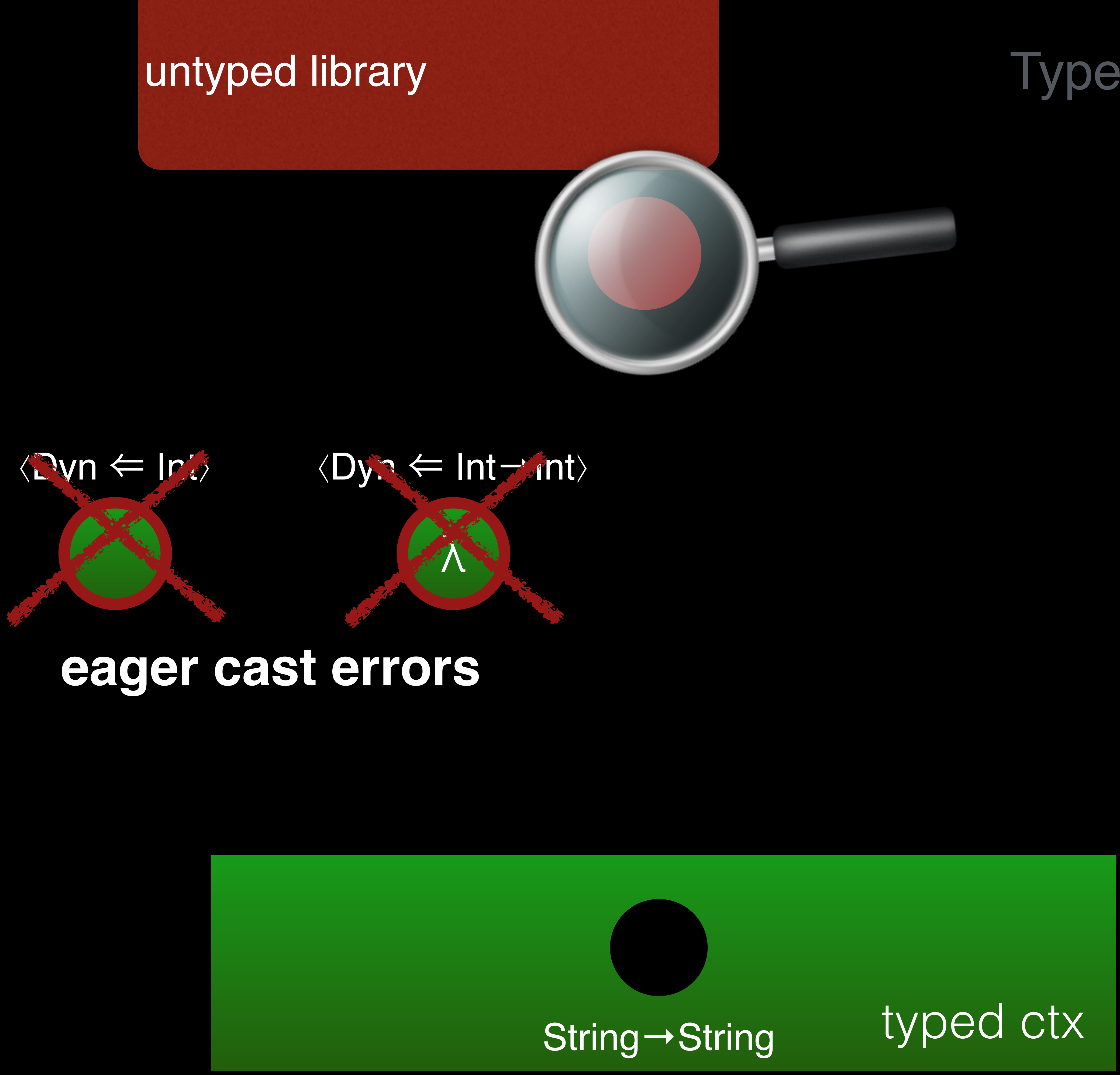
Type Checking
Runtime

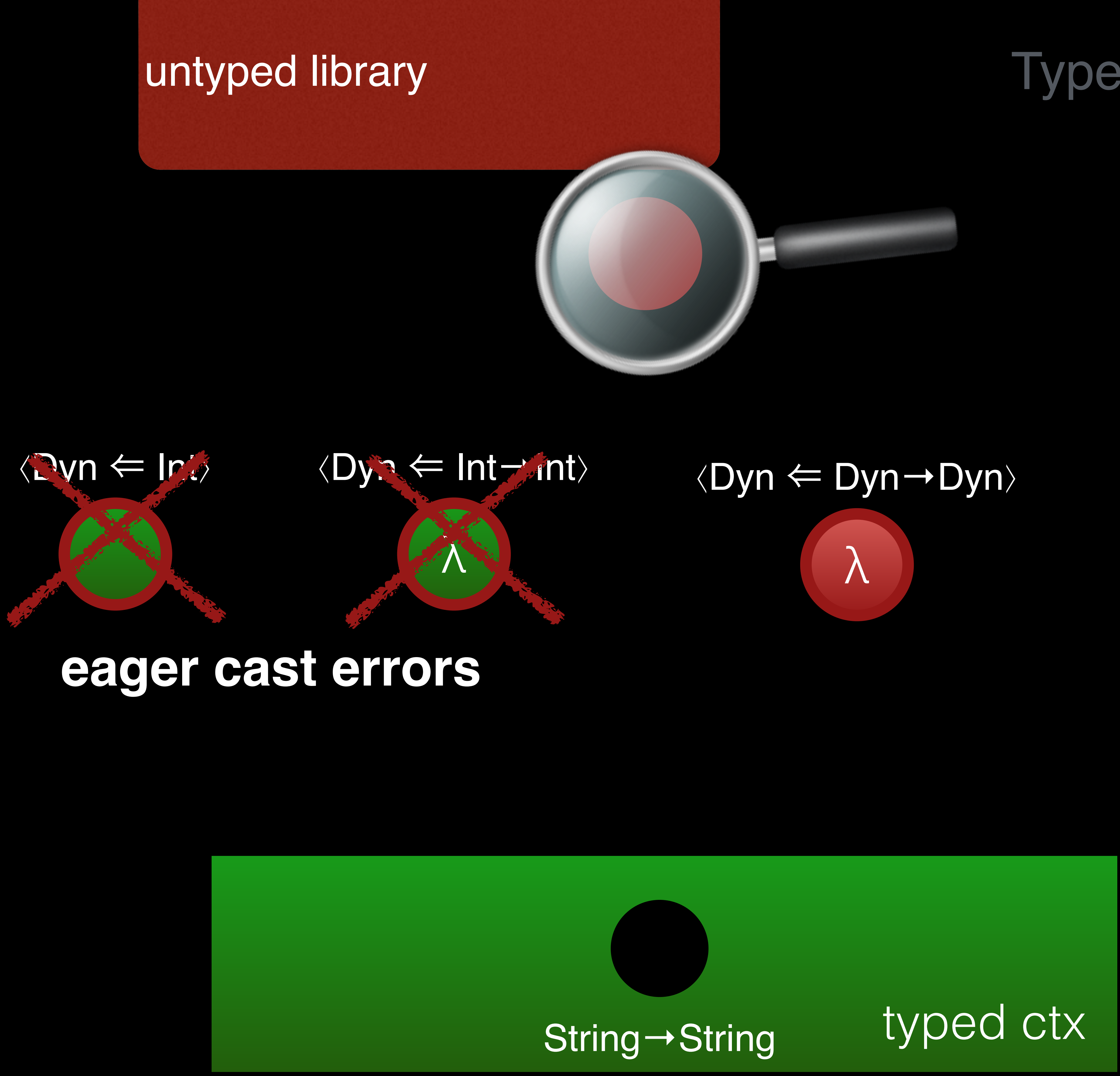



untyped library
Type Checking
Runtime
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
String→String
typed ctx

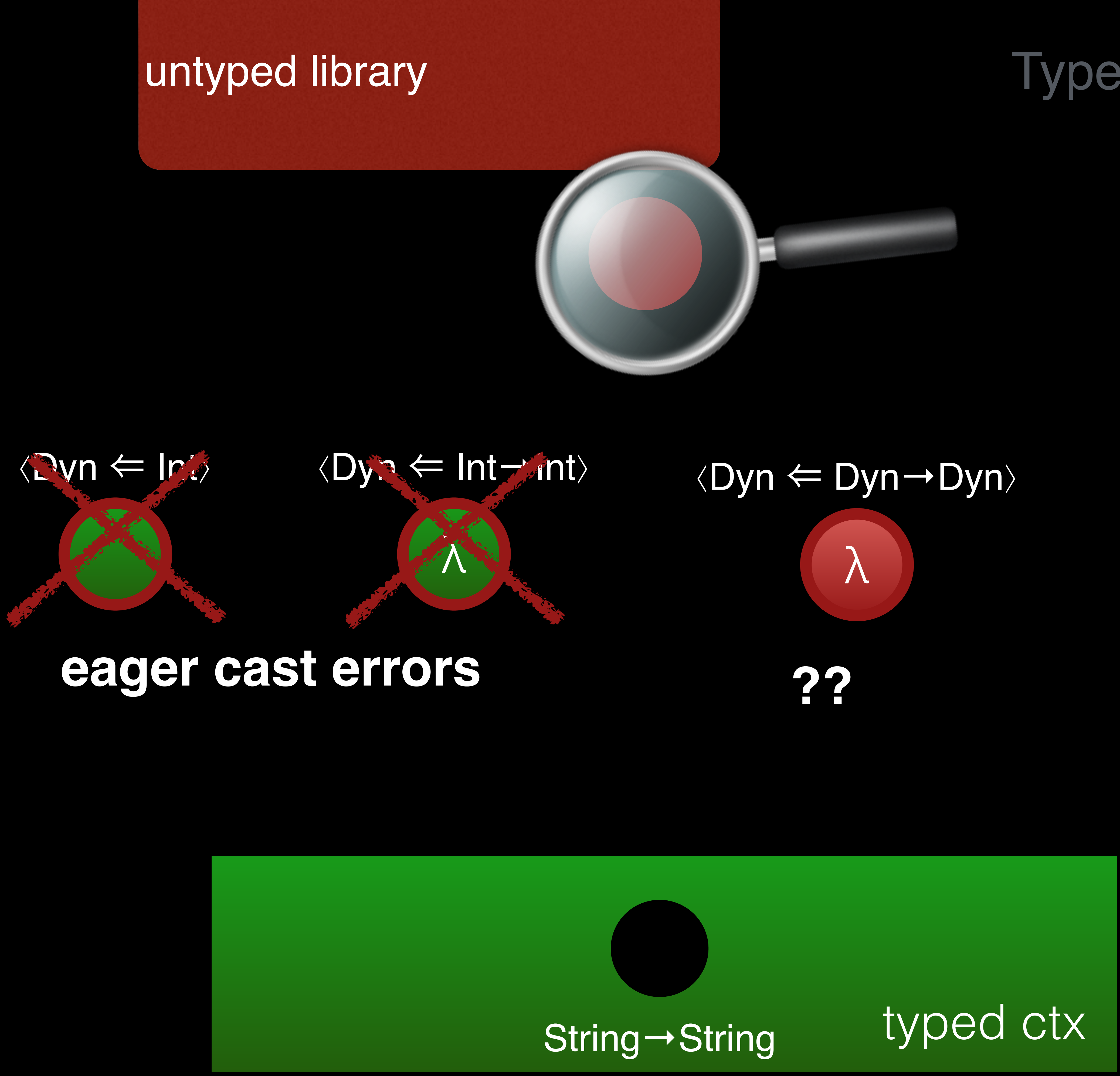


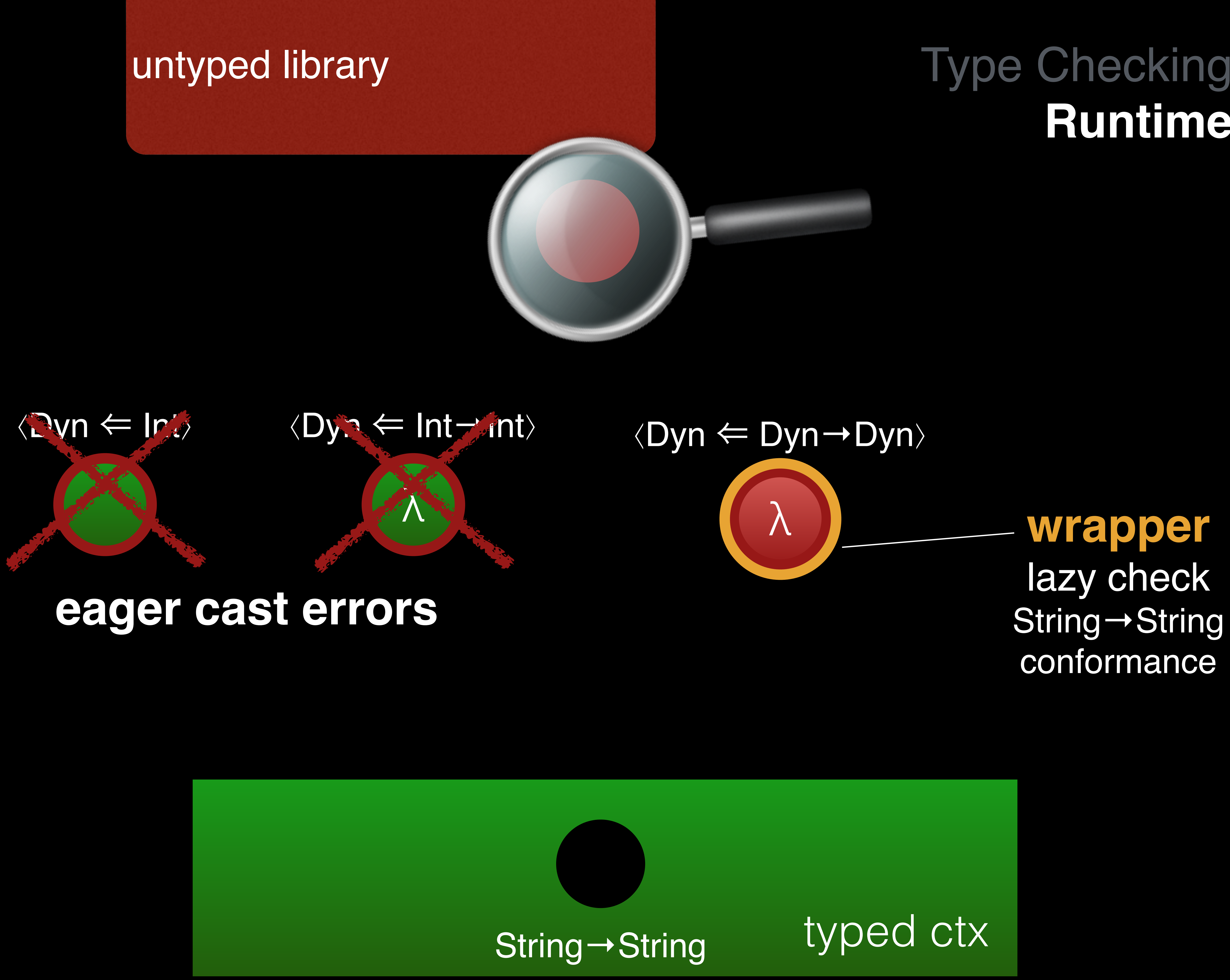
untyped library
Type Checking
Runtime
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
String→String
typed ctx


untyped library
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
eager cast errors
String→String
typed ctx


untyped library
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
⟨Dyn ⇐ Dyn→Dyn⟩
λ
eager cast errors
String→String
typed ctx

Runtime


untyped library
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
⟨Dyn ⇐ Dyn→Dyn⟩
λ
eager cast errors
??
String→String
typed ctx


untyped library
Type Checking
Runtime
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
⟨Dyn ⇐ Dyn→Dyn⟩
λ
wrapper
lazy check
String→String
conformance
eager cast errors
String→String
typed ctx

untyped library

~~⟨Dyn ⇐ Int⟩~~

~~⟨Dyn ⇐ Int→Int⟩~~

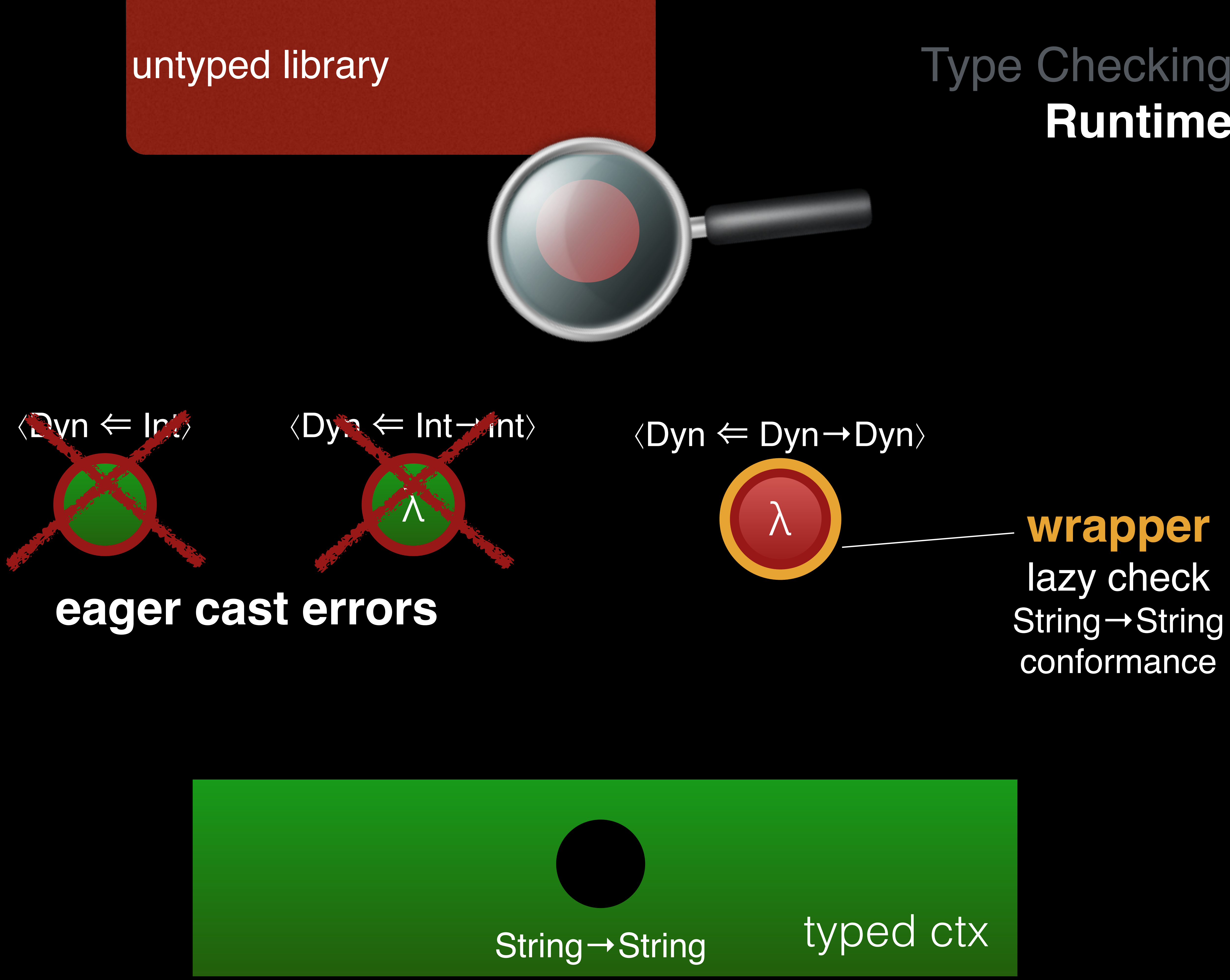
⟨Dyn ⇐ Dyn→Dyn⟩

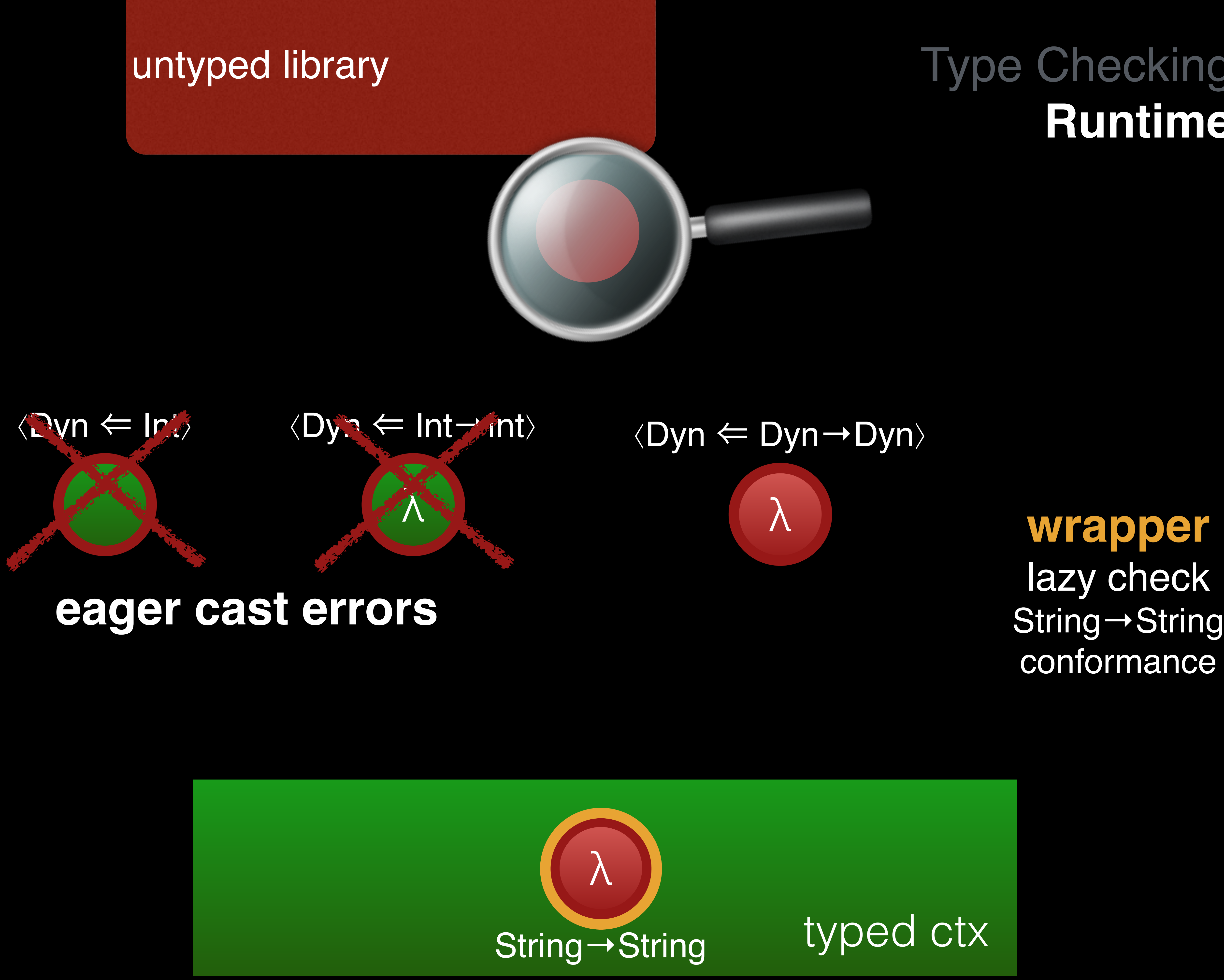
eager cast

λx:String.
⟨String ⇐ Dyn⟩ (λ (⟨Dyn ⇐ String⟩ x))

String→String

typed ctx

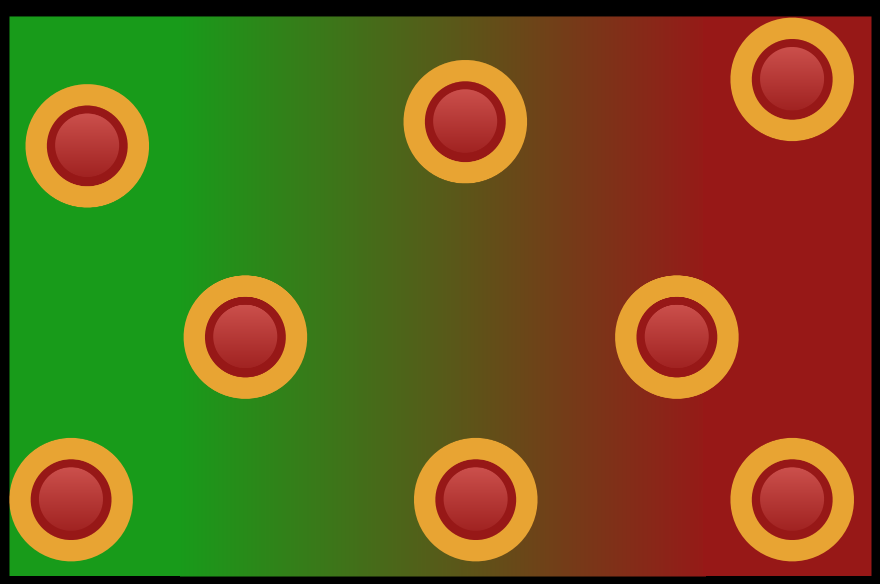

untyped library
Type Checking
Runtime
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
⟨Dyn ⇐ Dyn→Dyn⟩
λ
wrapper
lazy check
String→String
conformance
eager cast errors
String→String
typed ctx


untyped library
Type Checking
Runtime
⟨Dyn ⇐ Int⟩
⟨Dyn ⇐ Int→Int⟩
λ
⟨Dyn ⇐ Dyn→Dyn⟩
λ
eager cast errors
wrapper
lazy check
String→String
conformance
λ
String→String
typed ctx

issues with higher-order wrappers

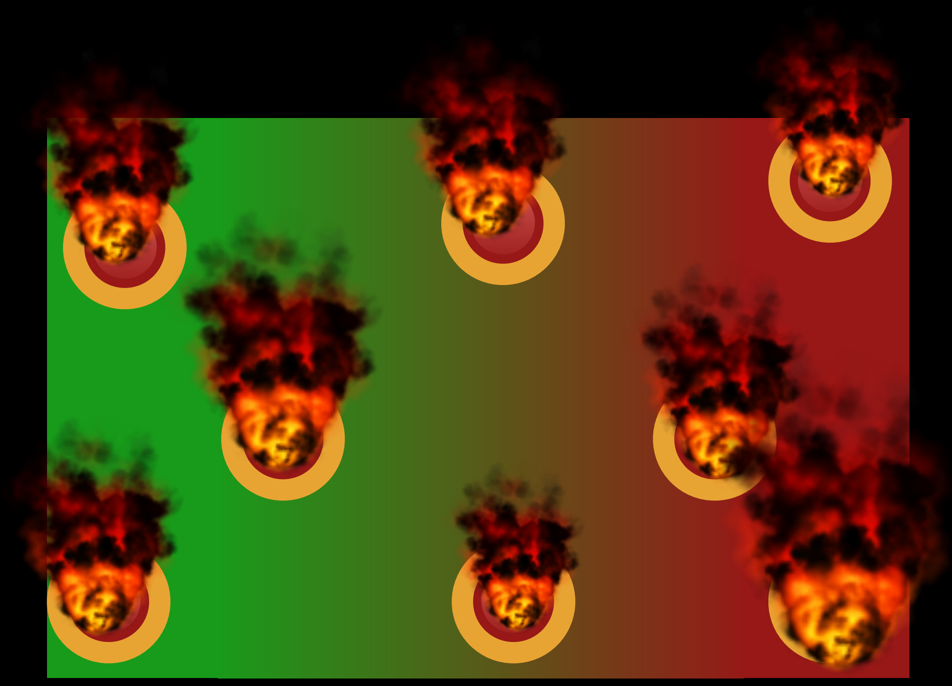
issues with higher-order wrappers

reliability



issues with higher-order wrappers

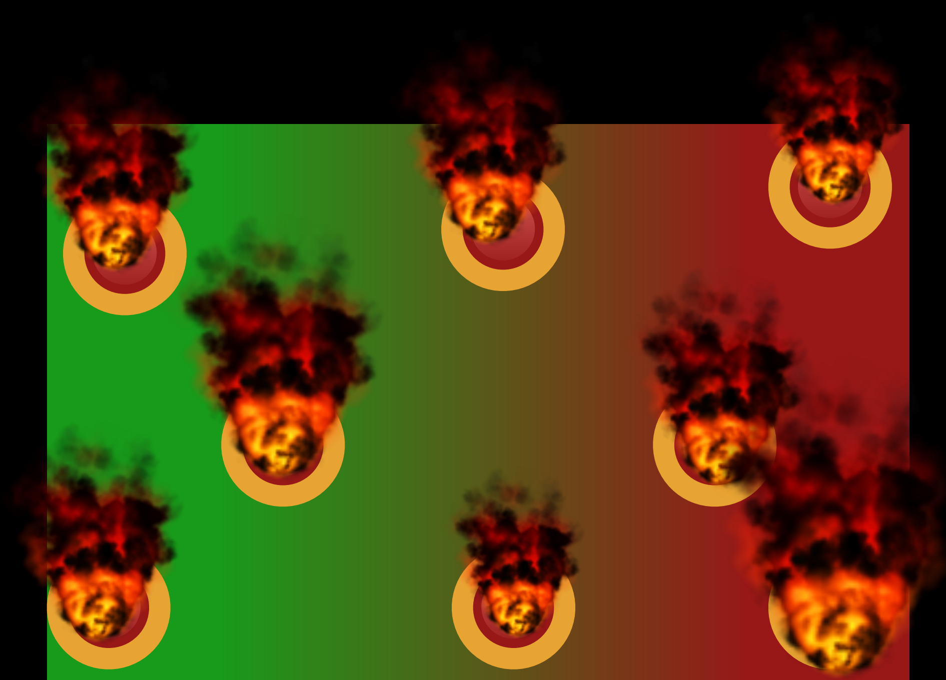
reliability



lazy cast errors
can happen anywhere

issues with higher-order wrappers

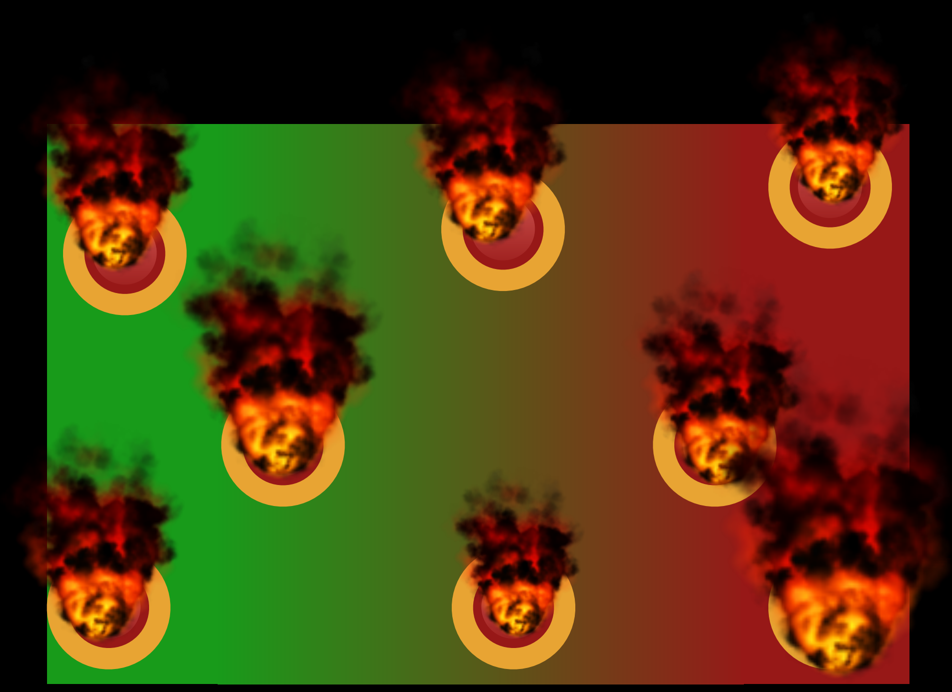
reliability **performance**



lazy cast errors
can happen anywhere

issues with higher-order wrappers

reliability



lazy cast errors
can happen anywhere

performance

space and
time issues

programming with gradual types

programming with gradual types

unpredictable

casts are
introduced
implicitly

programming with gradual types

unpredictable

casts are introduced *implicitly*

fragile

a missing type annotation can have a big impact

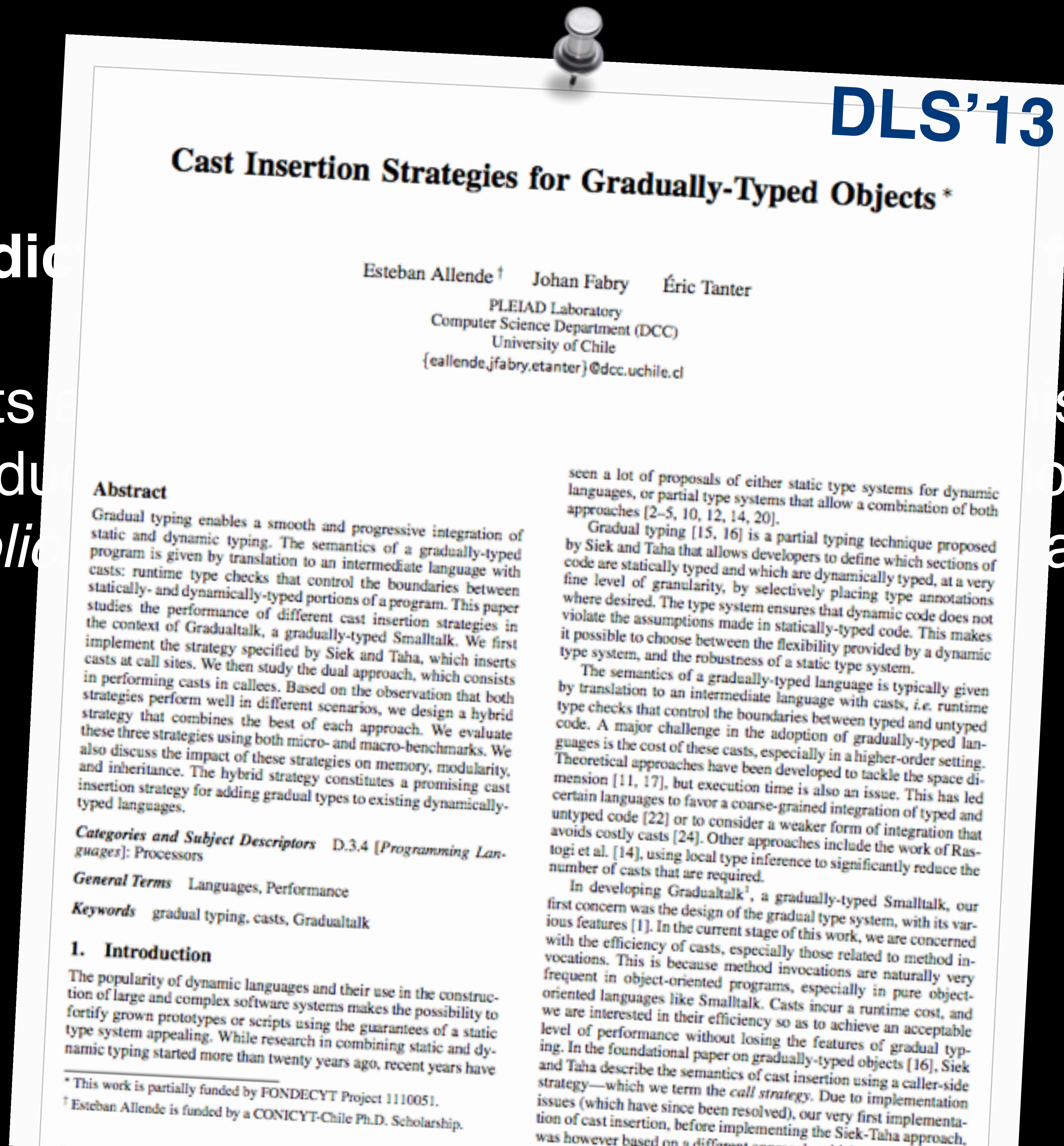
programming with gradual types

unpredic

casts
introdu
implic

ragile

ssing type
otation can
a big impact



DLS'13

Cast Insertion Strategies for Gradually-Typed Objects *

Esteban Allende † Johan Fabry Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
{eallende,jfabry,etanter}@dcc.uchile.cl

Abstract

Gradual typing enables a smooth and progressive integration of static and dynamic typing. The semantics of a gradually-typed program is given by translation to an intermediate language with casts: runtime type checks that control the boundaries between statically- and dynamically-typed portions of a program. This paper studies the performance of different cast insertion strategies in the context of Gradualtalk, a gradually-typed Smalltalk. We first implement the strategy specified by Siek and Taha, which inserts casts at call sites. We then study the dual approach, which consists in performing casts in callees. Based on the observation that both strategies perform well in different scenarios, we design a hybrid strategy that combines the best of each approach. We evaluate these three strategies using both micro- and macro-benchmarks. We also discuss the impact of these strategies on memory, modularity, and inheritance. The hybrid strategy constitutes a promising cast insertion strategy for adding gradual types to existing dynamically-typed languages.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors

General Terms Languages, Performance

Keywords gradual typing, casts, Gradualtalk

1. Introduction

The popularity of dynamic languages and their use in the construction of large and complex software systems makes the possibility to fortify grown prototypes or scripts using the guarantees of a static type system appealing. While research in combining static and dynamic typing started more than twenty years ago, recent years have seen a lot of proposals of either static type systems for dynamic languages, or partial type systems that allow a combination of both approaches [2–5, 10, 12, 14, 20].

Gradual typing [15, 16] is a partial typing technique proposed by Siek and Taha that allows developers to define which sections of code are statically typed and which are dynamically typed, at a very fine level of granularity, by selectively placing type annotations where desired. The type system ensures that dynamic code does not violate the assumptions made in statically-typed code. This makes it possible to choose between the flexibility provided by a dynamic type system, and the robustness of a static type system.

The semantics of a gradually-typed language is typically given by translation to an intermediate language with casts, *i.e.* runtime type checks that control the boundaries between typed and untyped code. A major challenge in the adoption of gradually-typed languages is the cost of these casts, especially in a higher-order setting. Theoretical approaches have been developed to tackle the space dimension [11, 17], but execution time is also an issue. This has led certain languages to favor a coarse-grained integration of typed and untyped code [22] or to consider a weaker form of integration that avoids costly casts [24]. Other approaches include the work of Rastogi et al. [14], using local type inference to significantly reduce the number of casts that are required.

In developing Gradualtalk[1], a gradually-typed Smalltalk, our first concern was the design of the gradual type system, with its various features [1]. In the current stage of this work, we are concerned with the efficiency of casts, especially those related to method invocations. This is because method invocations are naturally very frequent in object-oriented programs, especially in pure object-oriented languages like Smalltalk. Casts incur a runtime cost, and we are interested in their efficiency so as to achieve an acceptable level of performance without losing the features of gradual typing. In the foundational paper on gradually-typed objects [16], Siek and Taha describe the semantics of cast insertion using a caller-side strategy—which we term the *call strategy*. Due to implementation issues (which have since been resolved), our very first implementation of cast insertion, before implementing the Siek-Taha approach, was however based on a different approach...

* This work is partially funded by FONDECYT Project 1110051.
† Esteban Allende is funded by a CONICYT-Chile Ph.D. Scholarship.

programming with gradual types

unpredictable

casts are introduced *implicitly*

fragile

a missing type annotation can have a big impact

programming with gradual types

unpredictable

casts are introduced *implicitly*

fragile

a missing type annotation can have a big impact

cannot "seal" a typed module to protect it from cast errors and costly wrappers

tackling higher-order wrappers

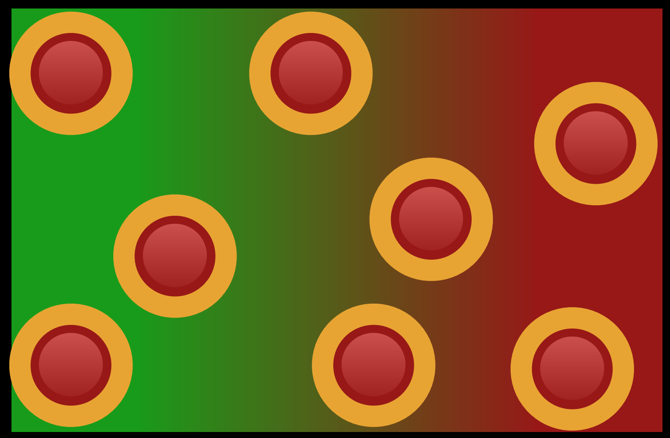
tackling higher-order wrappers

space efficiency: coercions [Hermann+], threesomes [Siek+]

tackling higher-order wrappers

space efficiency: coercions [Hermann+], threesomes [Siek+]

eliminate some wrappers
[Rastogi+]

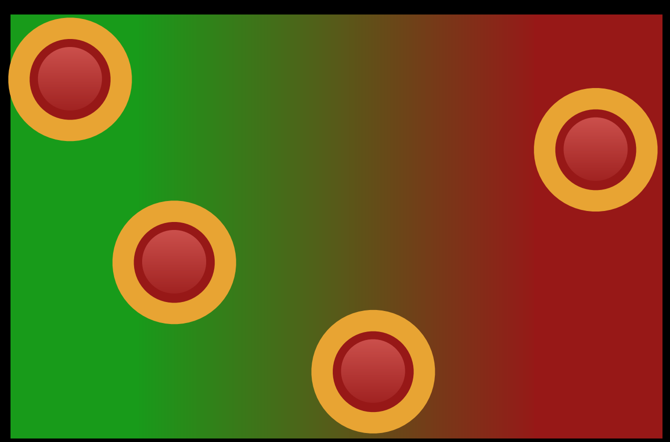


tackling higher-order wrappers

space efficiency: coercions [Hermann+], threesomes [Siek+]

eliminate some wrappers

[Rastogi+]

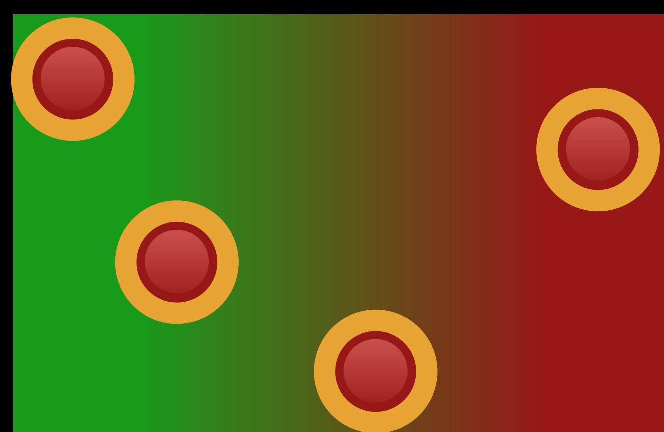


type-based static analysis

tackling higher-order wrappers

space efficiency: coercions [Hermann+], threesomes [Siek+]

eliminate some wrappers
[Rastogi+]



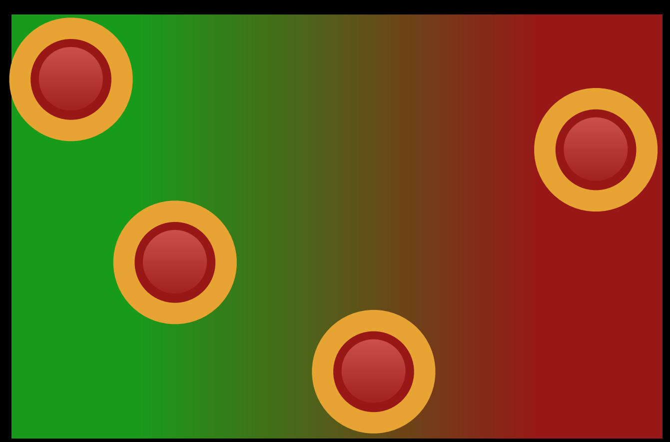
type-based static analysis

reduce (?) the need for wrappers
[Tobin-Hochstadt+]

tackling higher-order wrappers

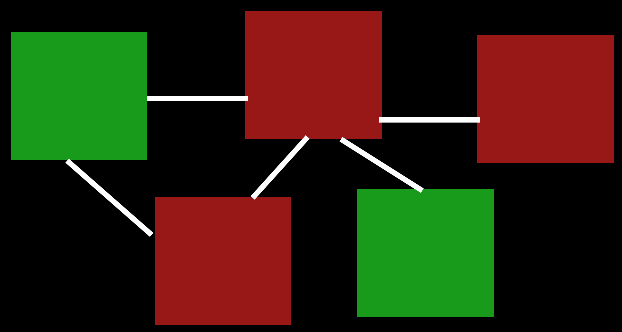
space efficiency: coercions [Hermann+], threesomes [Siek+]

eliminate some wrappers
[Rastogi+]



type-based static analysis

reduce (?) the need for wrappers
[Tobin-Hochstadt+]

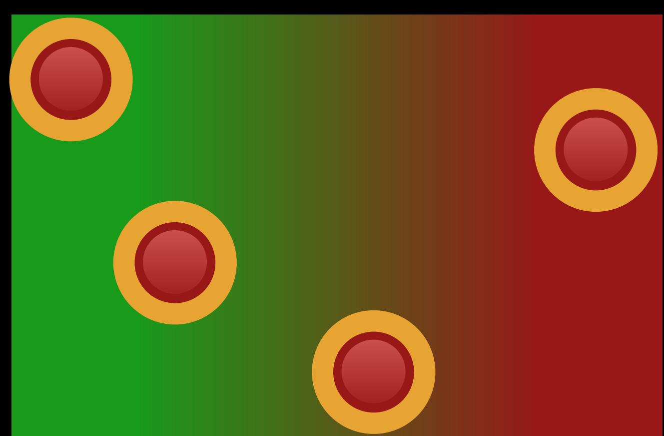


coarse-grained gradual typing

tackling higher-order wrappers

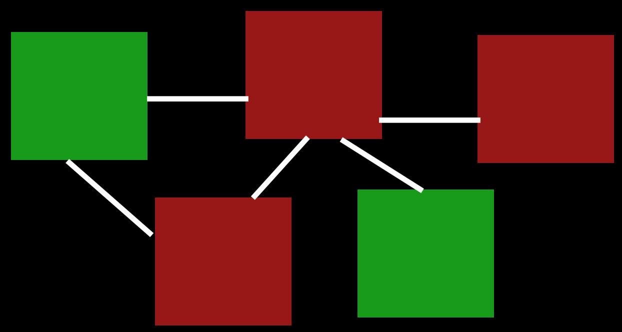
space efficiency: coercions [Hermann+], threesomes [Siek+]

eliminate some wrappers
[Rastogi+]



type-based static analysis

reduce (?) the need for wrappers
[Tobin-Hochstadt+]



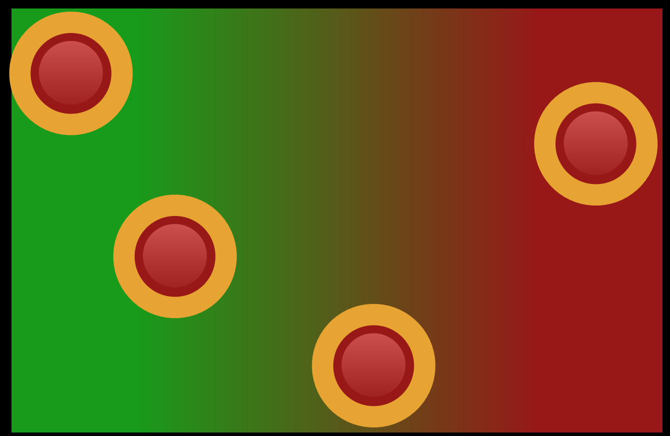
coarse-grained gradual typing

forbid implicit wrappers
[Swamy+]

tackling higher-order wrappers

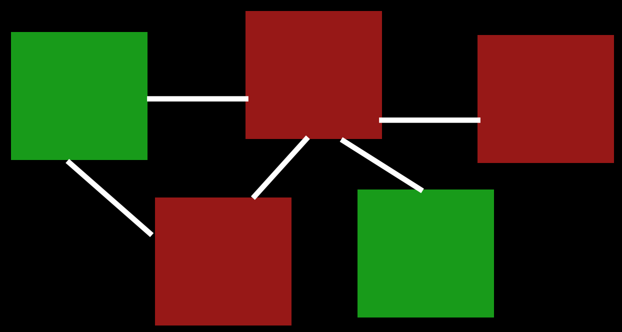
space efficiency: coercions [Hermann+], threesomes [Siek+]

eliminate some wrappers
[Rastogi+]



type-based static analysis

reduce (?) the need for wrappers
[Tobin-Hochstadt+]



coarse-grained gradual typing

forbid implicit wrappers
[Swamy+]

ban wrappers
[Wrigstad+]

Confined Gradual Typing

Confined Gradual Typing

Gradual Typing without Losing Control

providing explicit means to

trade some flexibility

increase predictability, reliability, performance

Confined Gradual Typing

Confined Gradual Typing

type qualifiers to control the **flow** of values

at the typed-untyped boundary

Confined Gradual Typing

type qualifiers to control the **flow** of values

at the typed-untyped boundary

$\uparrow T$

protects the ***future*** flow

Confined Gradual Typing

type qualifiers to control the **flow** of values

at the typed-untyped boundary

$\uparrow T$

protects the ***future*** flow

$\downarrow T$

constrains the ***past*** flow

Confined Gradual Typing

strict

relaxed

comes in two flavors!

Strict CGT

Strict CGT

$\uparrow T$

cannot flow into untyped

Strict CGT

$\uparrow$T

cannot flow into untyped

```
foo(f : ↑T) =
    let g : Dyn = ...
     ... g(f) ...
```

Strict CGT

↑T

cannot flow into untyped

```
foo(f : ↑T) =
    let g : Dyn = …
     … g(f) …
```

~~… g(f) …~~ **type error**

Strict CGT

$\uparrow T$

cannot flow into untyped

$\downarrow T$

has never flowed through untyped

foo(f : $\uparrow T$) =
    let g : Dyn = …
        ~~… g(f) …~~ **type error**

Strict CGT

↑T

cannot flow into untyped

foo(f : ↑T) =
  let g : Dyn = …
  ~~… g(f) …~~ **type error**

↓T

has never flowed through untyped

foo(f : ↓T) =
  … f() …

Strict CGT

↑T

cannot flow into untyped

foo(f : ↑T) =
    let g : Dyn = …
    ~~… g(f) …~~ **type error**

↓T

has never flowed through untyped

foo(f : ↓T) =
    … f() …
    … h(f) …

Strict CGT

↑T	↓T
cannot flow into untyped	**has never flowed through untyped**

foo(f : ↑T) =
  let g : Dyn = …
  ~~… g(f) …~~ **type error**

foo(f : ↓T) =
  … f() …
  … h(f) …

fully static, but restrictive

Relaxed CGT

if what matters most is the **performance** guarantee

we can allow *some* boundary crossing

Relaxed CGT

if what matters most is the **performance** guarantee

we can allow *some* boundary crossing

$\uparrow T$

cannot be wrapped

Relaxed CGT

if what matters most is the **performance** guarantee

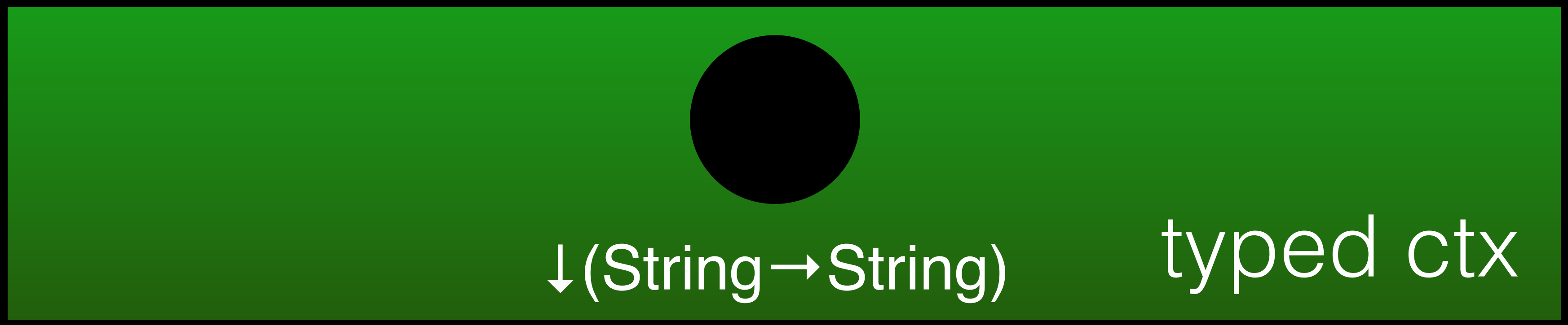
we can allow *some* boundary crossing

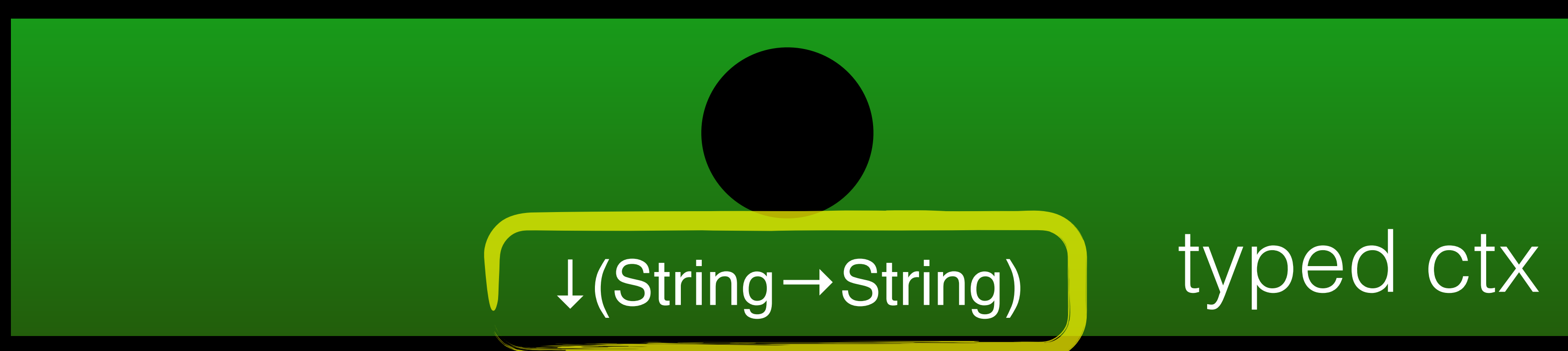
$\uparrow T$

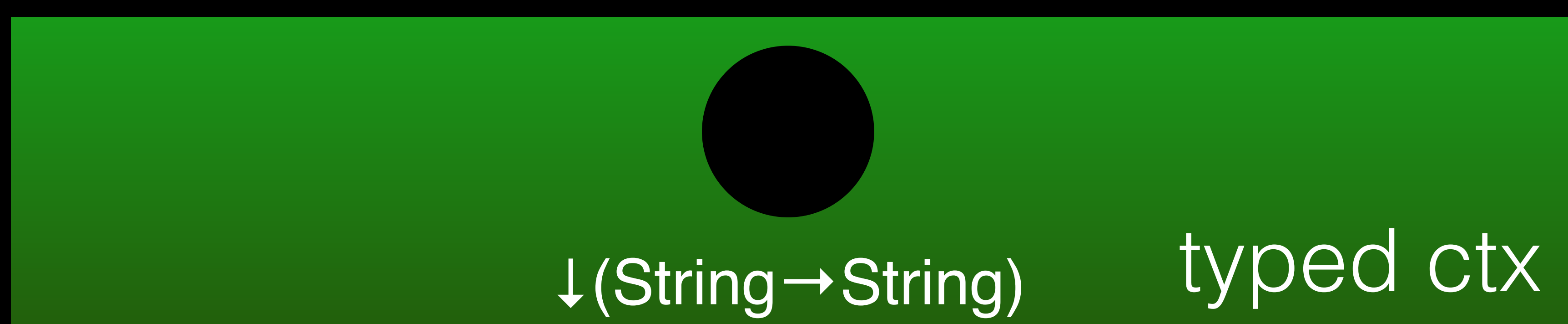
cannot be wrapped

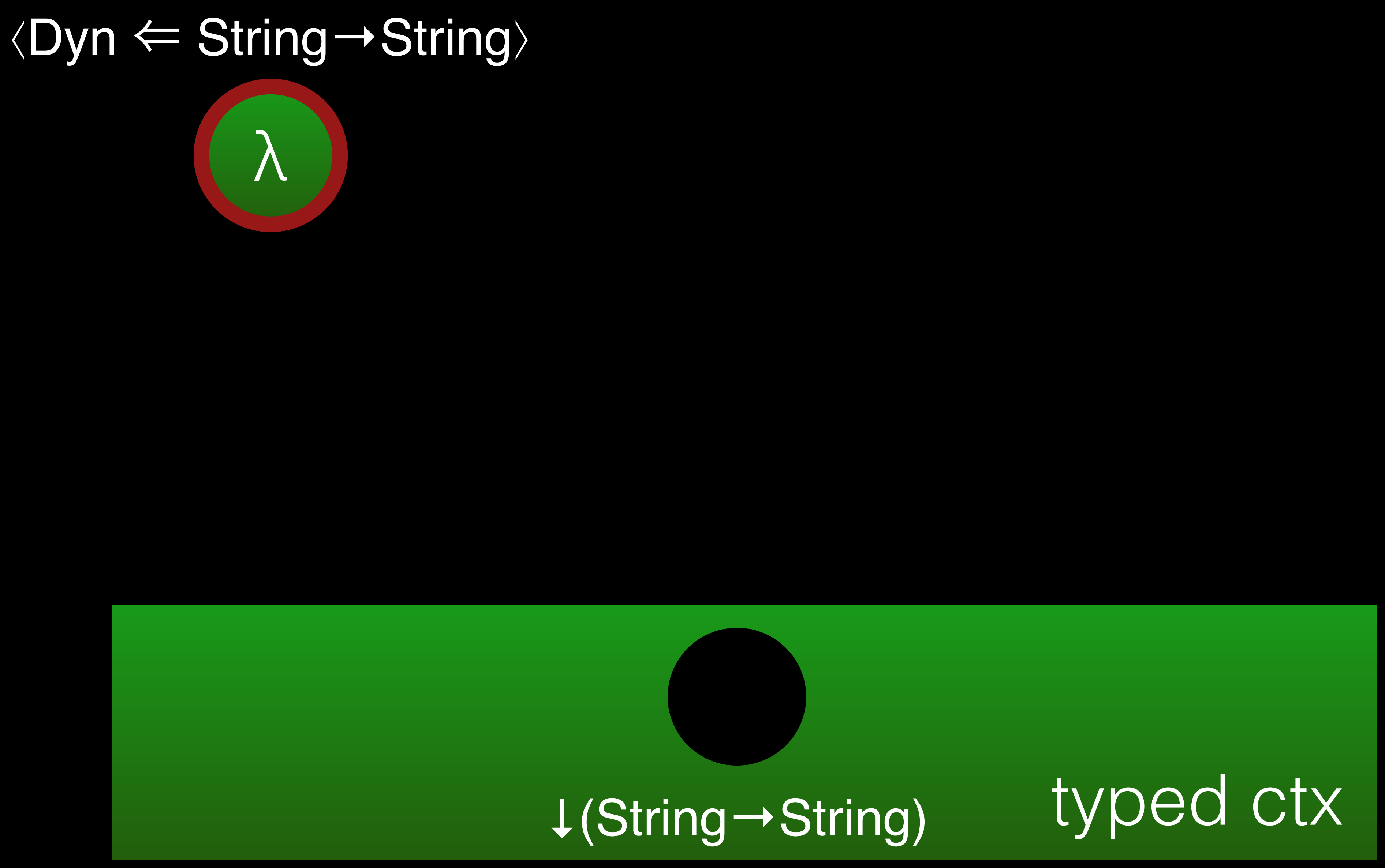
$\downarrow T$

has not been wrapped

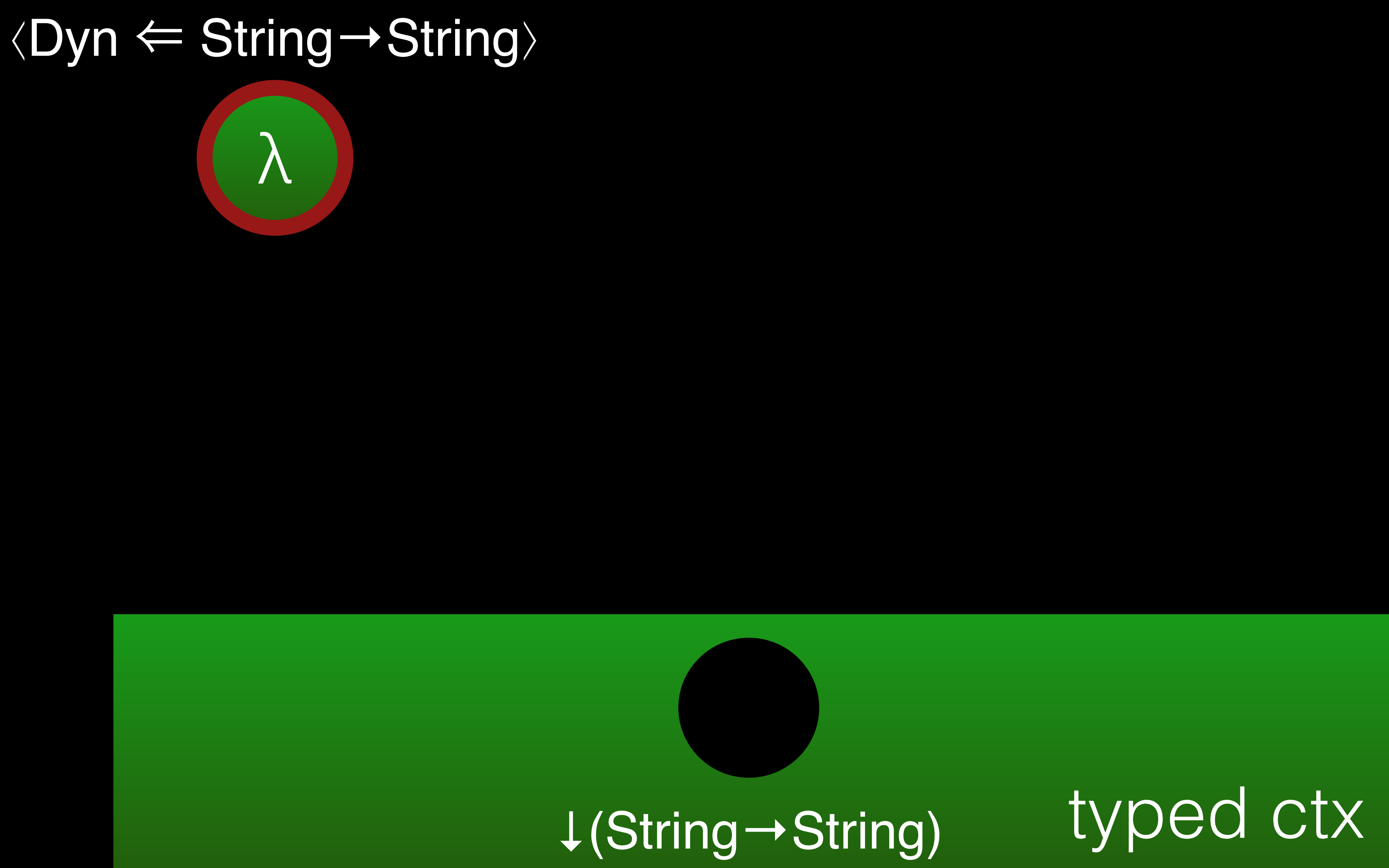

↓(String→String)
typed ctx

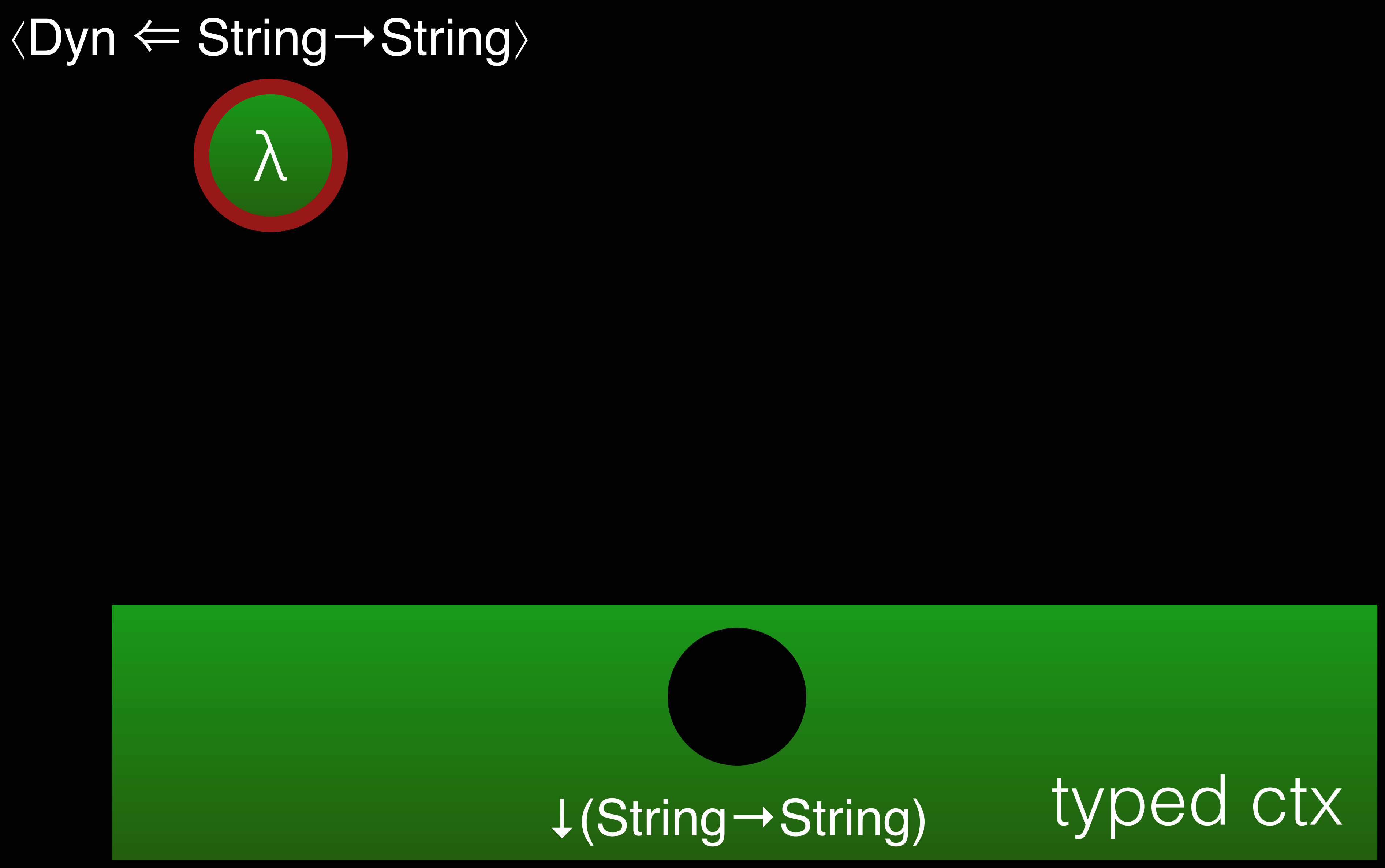

↓(String→String)
typed ctx

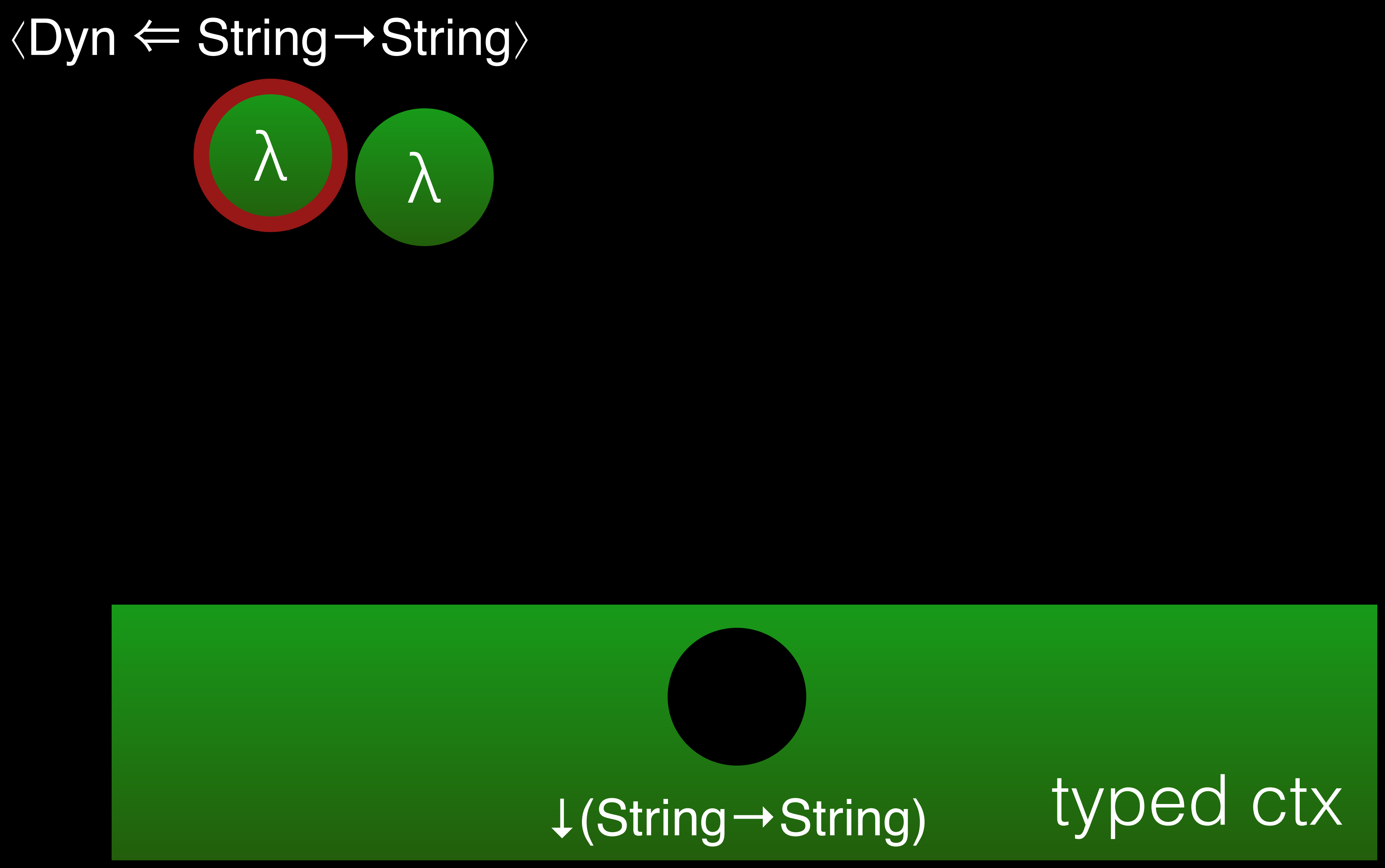

↓(String→String)
typed ctx

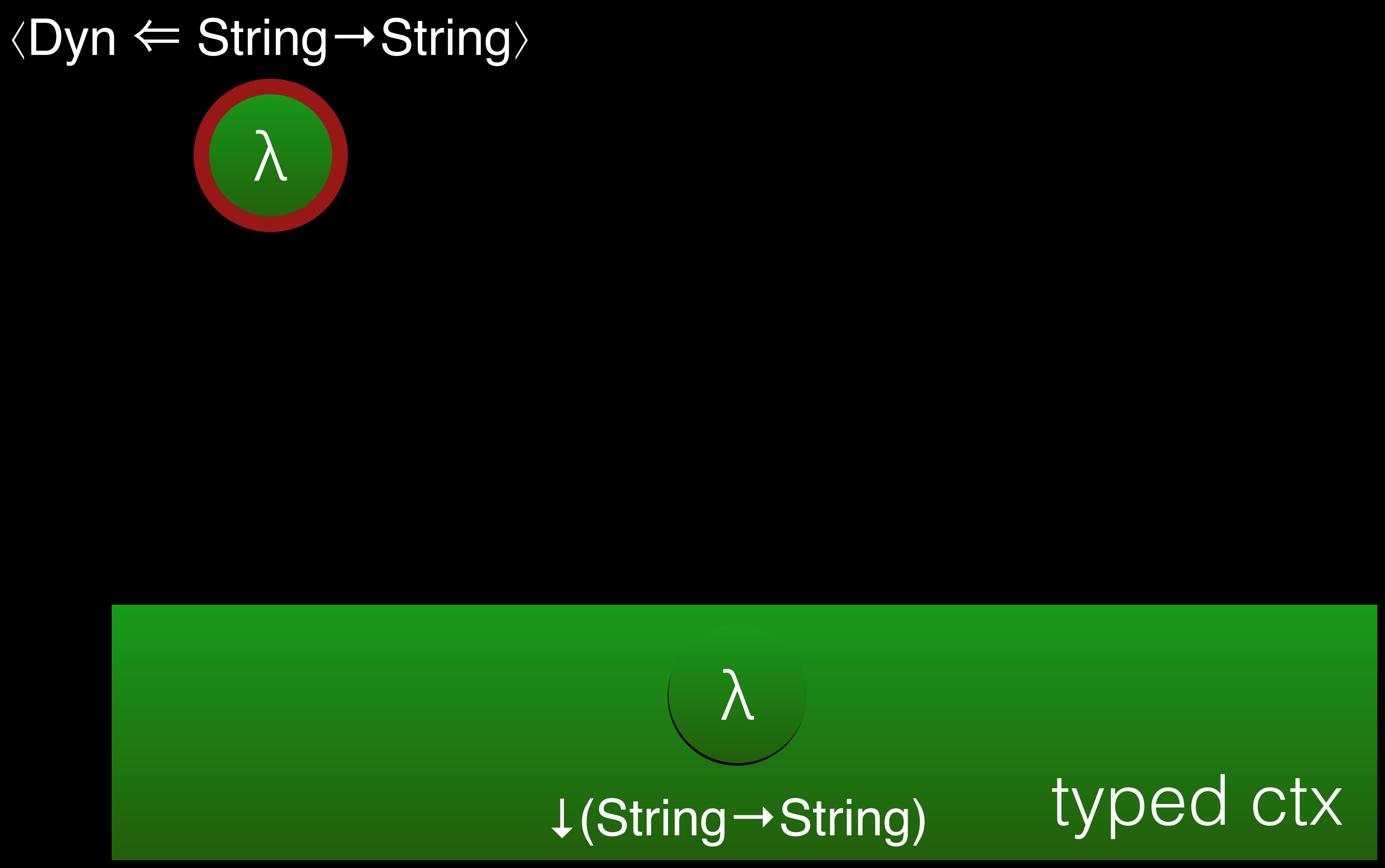

⟨Dyn ⇐ String→String⟩
λ
↓(String→String)
typed ctx

(would have been rejected by Strict CGT)




⟨Dyn ⇐ String→String⟩
λ
↓(String→String)
typed ctx


⟨Dyn ⇐ String→String⟩
λ
λ
↓(String→String)
typed ctx


⟨Dyn ⇐ String→String⟩
λ
λ
↓(String→String)
typed ctx

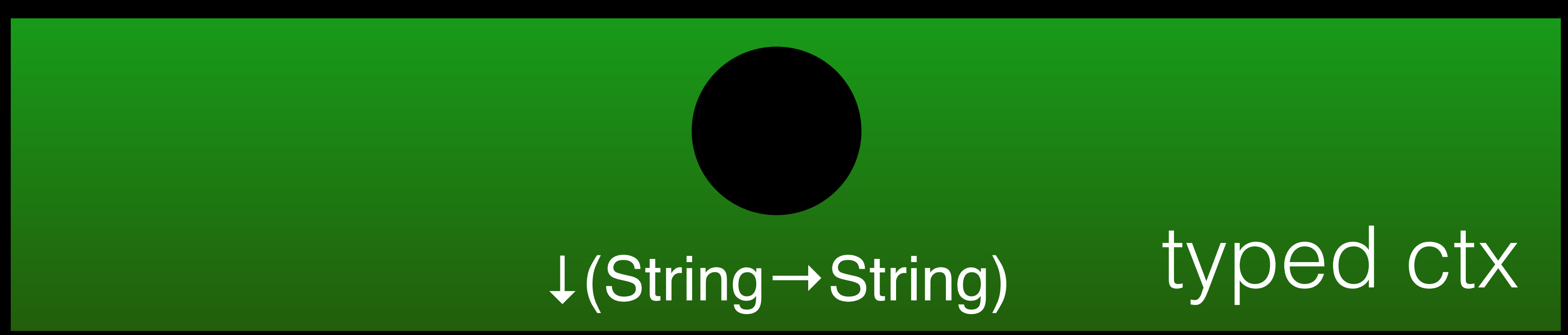
$\langle \mathrm{Dyn} \Leftarrow \mathrm{String} \rightarrow \mathrm{String} \rangle$

cannot fail, no extra cost

$\langle \text{Dyn} \Leftarrow \text{String} \to \text{String} \rangle$

$\langle \text{Dyn} \Leftarrow \text{Dyn} \to \text{Dyn} \rangle$

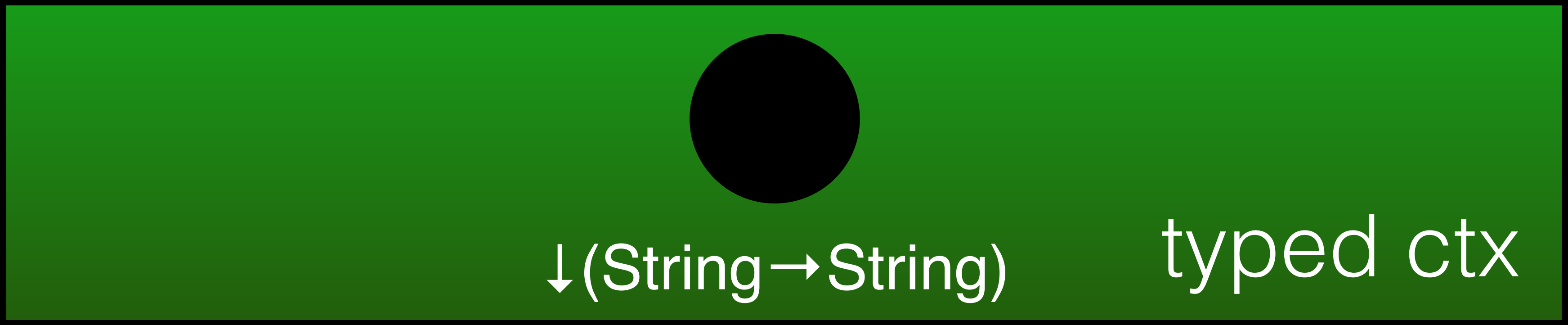
cannot fail, no extra cost



$\langle \text{Dyn} \Leftarrow \text{String} \to \text{String} \rangle$

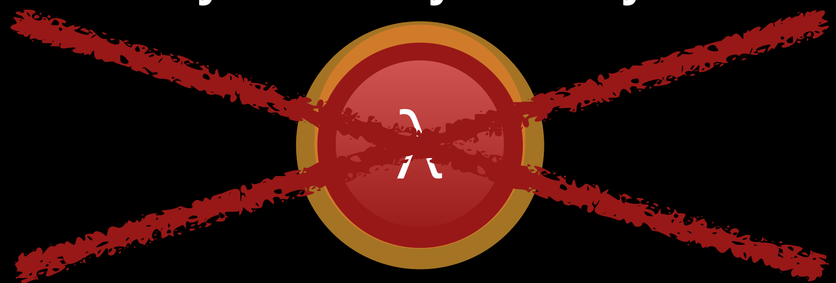
$\langle \text{Dyn} \Leftarrow \text{Dyn} \to \text{Dyn} \rangle$

cannot fail, no extra cost



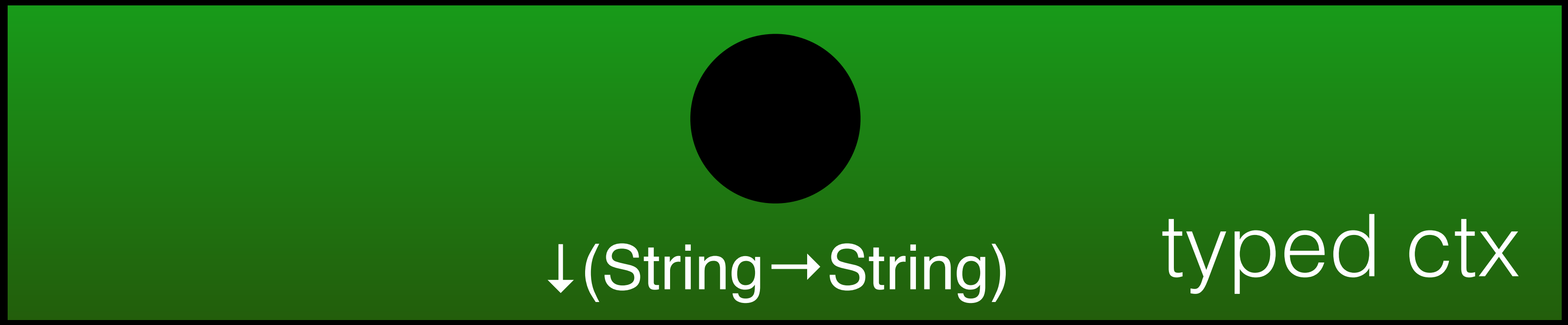
⟨Dyn ⇐ String→String⟩

⟨Dyn ⇐ Dyn→Dyn⟩



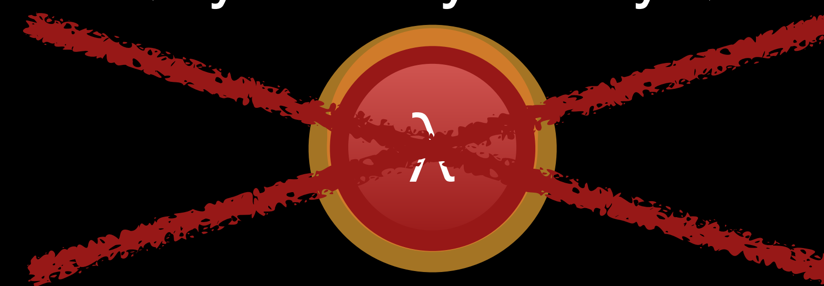
wrapping error @runtime

cannot fail, no extra cost



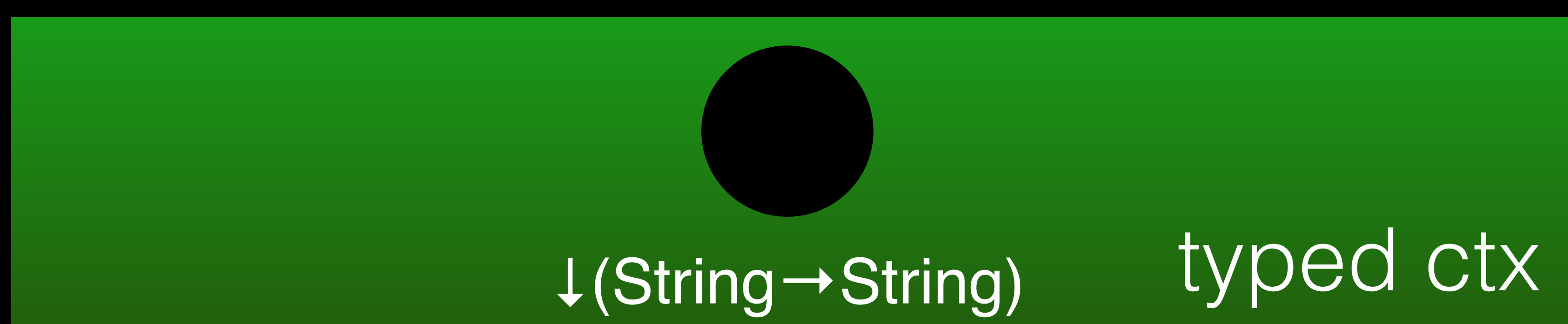
$\langle \mathrm{Dyn} \Leftarrow \mathrm{String} \to \mathrm{String} \rangle$

$\langle \mathrm{Dyn} \Leftarrow \mathrm{Dyn} \to \mathrm{Dyn} \rangle$



wrapping error @runtime

cannot fail, no extra cost



new kind of (eager) runtime errors at the boundary

Directed consistency $\rightsquigarrow$

Directed consistency $\rightsquigarrow$

$$\frac{T_1 \rightsquigarrow T_2}{T_1 \rightsquigarrow \uparrow T_2}$$

Directed consistency $\rightsquigarrow$

$$\frac{T_1 \rightsquigarrow T_2}{T_1 \rightsquigarrow \uparrow T_2}$$

can **impose** restriction on **future**
*but cannot **lose** it*

Directed consistency $\leadsto$

$$\frac{T_1 \leadsto T_2}{T_1 \leadsto \uparrow T_2}$$

can **impose** restriction on **future**
*but cannot **lose** it*

$$\frac{T_1 \leadsto T_2}{\downarrow T_1 \leadsto T_2}$$

Directed consistency $\rightsquigarrow$

$$\frac{T_1 \rightsquigarrow T_2}{T_1 \rightsquigarrow \uparrow T_2}$$

can **impose** restriction on **future**
*but cannot **lose** it*

$$\frac{T_1 \rightsquigarrow T_2}{\downarrow T_1 \rightsquigarrow T_2}$$

can **lose** guarantee on the **past**
*but cannot **forge** it*

Directed consistency $\leadsto$

$$\frac{T_1 \leadsto T_2}{T_1 \leadsto \uparrow T_2}$$

can **impose** restriction on **future**
*but cannot **lose** it*

$$\frac{T_1 \leadsto T_2}{\downarrow T_1 \leadsto T_2}$$

can **lose** guarantee on the **past**
*but cannot **forge** it*

$$\frac{\uparrow T \leadsto \text{Dyn} \quad \text{Dyn} \leadsto \downarrow T}{}$$

Strict CGT

Relaxed CGT

Directed consistency $\leadsto$

$$\frac{T_1 \leadsto T_2}{T_1 \leadsto \uparrow T_2}$$

can **impose** restriction on **future**
*but cannot **lose** it*

$$\frac{T_1 \leadsto T_2}{\downarrow T_1 \leadsto T_2}$$

can **lose** guarantee on the **past**
*but cannot **forge** it*

	$\uparrow T \leadsto Dyn$	$Dyn \leadsto \downarrow T$
Strict CGT	✗	✗
Relaxed CGT		

Directed consistency $\rightsquigarrow$

$$\frac{T_1 \rightsquigarrow T_2}{T_1 \rightsquigarrow \uparrow T_2}$$

can **impose** restriction on **future**
*but cannot **lose** it*

$$\frac{T_1 \rightsquigarrow T_2}{\downarrow T_1 \rightsquigarrow T_2}$$

can **lose** guarantee on the **past**
*but cannot **forge** it*

	$\uparrow T \rightsquigarrow Dyn$	$Dyn \rightsquigarrow \downarrow T$
Strict CGT	✗	✗
Relaxed CGT	✓	✓

Directed consistency $\leadsto$

$$\frac{T_1 \leadsto T_2}{T_1 \leadsto \uparrow T_2}$$

can **impose** restriction on **future**
*but cannot **lose** it*

$$\frac{T_1 \leadsto T_2}{\downarrow T_1 \leadsto T_2}$$

can **lose** guarantee on the **past**
*but cannot **forge** it*

	$\uparrow T \leadsto Dyn$	$Dyn \leadsto \downarrow T$
Strict CGT	✗	✗
Relaxed CGT	✓	✓

runtime semantics rejects wrappers from $\uparrow T$ or to $\downarrow T$

Correctness of qualifiers

Correctness of qualifiers

Strict CGT: taint tracking semantics [Grossman+]

Correctness of qualifiers

Strict CGT: taint tracking semantics [Grossman+]

a value of type ↓T *is untainted*

a value of type ↑T *("untaintable") is not tagged*

Correctness of qualifiers

Strict CGT: taint tracking semantics [Grossman+]

a value of type ↓T *is untainted*

a value of type ↑T *("untaintable") is not tagged*

Relaxed CGT

Correctness of qualifiers

Strict CGT: taint tracking semantics [Grossman+]

a value of type ↓T is untainted

a value of type ↑T ("untaintable") is not tagged

Relaxed CGT

no function wrapper has ↑T as source type

or ↓T as target type

Experiments in the paper

Implemented in Gradualtalk, a gradually-typed Smalltalk

Benchmarks confirm the performance costs/benefits

Usage Scenarios

Usage Scenarios

post-hoc

- add type qualifiers to track “leaks”
- leave them in place to prevent future issues

Usage Scenarios

post-hoc

- add type qualifiers to track “leaks”
- leave them in place to prevent future issues

interface provider

- add qualifiers to interface of critical components
- eg. GUI callbacks (perfs), core system components (reliability)

Usage Scenarios

post-hoc

- add type qualifiers to track “leaks”
- leave them in place to prevent future issues

interface provider

- add qualifiers to interface of critical components
- eg. GUI callbacks (perfs), core system components (reliability)

interface client

- annotate callbacks passed to a critical, typed, 3rd-party library

Perspectives

Perspectives

Language design

Perspectives

Language design

- combine both variants

Perspectives

Language design

- combine both variants
- dual semantics: use qualifiers to *allow* boundary crossing

Perspectives

Language design

- combine both variants
- dual semantics: use qualifiers to *allow* boundary crossing
- inference of qualifiers

Perspectives

Language design

- combine both variants
- dual semantics: use qualifiers to *allow* boundary crossing
- inference of qualifiers

More practical experience (other languages)

Confined Gradual Typing

Gradual Typing without Losing Control

- providing explicit means to
  - trade some flexibility
  - increase predictability, reliability, performance

Confined Gradual Typing

Gradual Typing without Losing Control

- providing explicit means to
  - trade some flexibility
  - increase predictability, reliability, performance

↓ ¿ questions ? ↑