Implementing a Typechecker for an Esoteric Language: Experiences, Challenges, and Lessons

Alex Rice

TYPES 2025, Glasgow

Overview

Exploration of general design principles for writing typecheckers and evaluators.

Overview

Exploration of general design principles for writing typecheckers and evaluators.

What this talk is

✔ An experience report

What this talk isn't

Overview

Exploration of general design principles for writing typecheckers and evaluators.

What this talk is

✔ An experience report

What this talk isn't

❌ A novel framework

Overview

Exploration of general design principles for writing typecheckers and evaluators.

What this talk is

✔ An experience report

✔ Dispelling of folklore

What this talk isn’t

❌ A novel framework

Overview

Exploration of general design principles for writing typecheckers and evaluators.

What this talk is

- ✔️ An experience report
- ✔️ Dispelling of folklore

What this talk isn't

- ❌ A novel framework
- ❌ Language specific

The Language

… has no lambda abstraction.

… has no application.

… has no function types.

… is dependently typed.

… inference in certain places but no unification.

… but does have decidable typechecking.

The Language

… has no lambda abstraction.

… has no application.

… has no function types.

… is dependently typed.

… inference in certain places but no unification.

… but does have decidable typechecking.

Can we leverage the existing literature to write our typechecker?

The Language

… has no lambda abstraction.

… has no application.

… has no function types.

… is dependently typed.

… inference in certain places but no unification.

… but does have decidable typechecking.

Can we leverage the existing literature to write our typechecker?

Yes*!

The Language

… has no lambda abstraction.

… has no application.

… has no function types.

… is dependently typed.

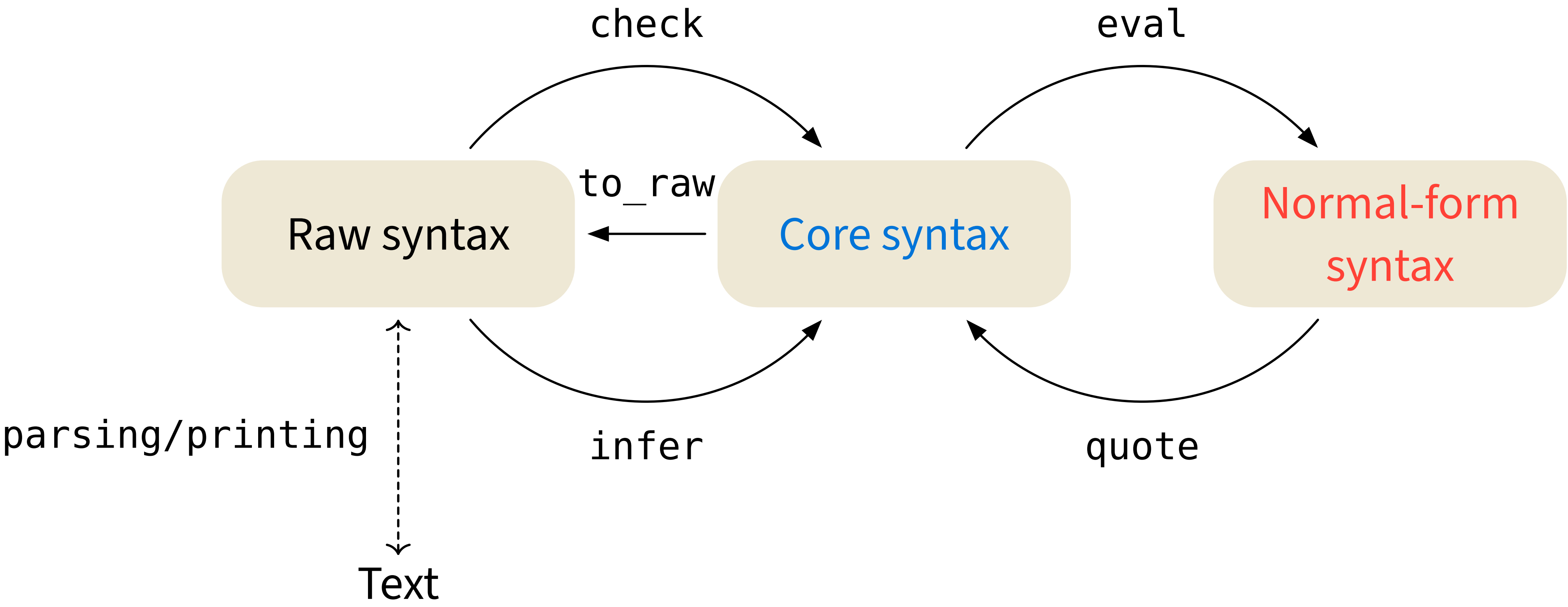
… inference in certain places but no unification.

… but does have decidable typechecking.

Can we leverage the existing literature to write our typechecker?

Yes*! We utilise bidirectional typechecking + normalisation by evaluation (NbE).

Anatomy of the Interpreter



Experience: NbE works in non-standard environments

We want:
- A well-specified set of canonical forms.
- To interpret each operation of the language on these canonical forms.

But we started with:
- a confluent terminating rewriting relation,
- but no satisfying definition of “normal form.”

Our normal form syntax doesn’t need to be perfect to see benefits.

The form of the algorithm NbE takes encourages us to be efficient.

Experience: NbE works in non standard environments

My intuition of NbE (from an implementation perspective):

Instead of evaluating term t, instead evaluate t in an environment ρ.

$$\frac{\rho : \mathrm{Env} \qquad\qquad t : \text{Core Term}}{\mathrm{eval}_{\rho}(t) : \text{Normal-form Term}}$$

We evaluate $t[\sigma]$ for a substitution: $\sigma = [n_1/x_1, \ldots, n_k/x_k]$

NbE example

Suppose we want to calculate:

$$\mathrm{eval}_{\rho}(t[\sigma]) \qquad\qquad \text{where } \sigma = [s_1/x_1, \ldots, s_k/x_k])$$

At some point, we should evaluate t, but in what environment?

NbE example

Suppose we want to calculate:

$$\mathrm{eval}_{\rho}(t[\sigma]) \qquad\qquad \text{where } \sigma = [s_1/x_1, \ldots, s_k/x_k])$$

At some point, we should evaluate t, but in what environment?

$$\begin{aligned} t[\sigma][\rho] &= t[s_1/x_1, \ldots, s_k/x_k])[\rho] \\ &= t[s_1[\rho]/x_1, \ldots, s_k[\rho]/x_k] \\ &= t\left[\mathrm{eval}_{\rho}(s_1)/x_1, \ldots, \mathrm{eval}_{\rho}(s_k)/x_k\right] \end{aligned}$$

Therefore: $\mathrm{eval}_{\rho}(t[\sigma]) := \mathrm{eval}_{\mathrm{eval}_{\rho}(\sigma)}(t)$

Lesson: Core syntax can represent “internal” operations

Substitution is used during reduction.

$$\ldots A \ldots \rightsquigarrow \ldots A[\iota] \ldots$$

We already know how to evaluate $A[\iota]$

Lesson: Core syntax can represent “internal” operations

Substitution is used during reduction.

$$\ldots A \ldots \rightsquigarrow \ldots A[\iota] \ldots$$

We already know how to evaluate $A[\iota]$

… except $A[\iota]$ isn’t part of the language’s syntax.

Lesson: Core syntax can represent “internal” operations

Substitution is used during reduction.

$$\ldots A \ldots \rightsquigarrow \ldots A[\iota] \ldots$$

We already know how to evaluate $A[\iota]$

… except $A[\iota]$ isn’t part of the language’s syntax.

Solution: add it anyway.

$$\ldots A \ldots \rightsquigarrow \ldots A[\iota_0] \rightsquigarrow \ldots A[\iota_0][\iota_1] \ldots \rightsquigarrow \ldots A[\iota_0][\iota_1][\iota_2] \ldots$$

Lesson: Variable names are stored in the context/binders

The raw syntax has named variables.

But in the core and normal-form syntax, we use de Bruijn levels.
- Makes evaluation of variable easy.
- Avoids α-renaming.

Lesson: Variable names are stored in the context/binders

The raw syntax has named variables.

But in the core and normal-form syntax, we use de Bruijn levels.

- Makes evaluation of variable easy.
- Avoids α-renaming.

However, variable names chosen by a programmer are often meaningful.

Storing variable names in the context avoids duplication of information.

Top level symbols form an exception to this rule.

Lesson: Error handling

$$\frac{\text{Type systems can help programmers} \quad \text{Programmers are not perfect}}{\text{Errors can help the programmer}}$$

The user will likely want to know where the error happened.

Attempt 1: Add spans to raw syntax:

```
Term = Var Name Range<int> | ...
```

Lesson: Error handling

$$\frac{\text{Type systems can help programmers} \qquad \text{Programmers are not perfect}}{\text{Errors can help the programmer}}$$

The user will likely want to know where the error happened.

Attempt 1: Add spans to raw syntax:

```
Term = Var Name Range<int> | ...
```

Attempt 2: Add a generic type annotation to raw syntax:

```
Term<S> = Var Name S | ...
Error<S> = UnknownVariable Name S | ...
```

Lesson: Error handling

$$\frac{\text{Type systems can help programmers} \qquad \text{Programmers are not perfect}}{\text{Errors can help the programmer}}$$

The user will likely want to know where the error happened.

Attempt 1: Add spans to raw syntax:

```
Term = Var Name Range<int> | ...
```

Attempt 2: Add a generic type annotation to raw syntax:

```
Term<Range<int>> = Var Name Range<int> | ...
Error<Range<int>> = UnknownVariable Name Range<int> | ...
```

Lesson: Error handling

$$\frac{\text{Type systems can help programmers} \qquad \text{Programmers are not perfect}}{\text{Errors can help the programmer}}$$

The user will likely want to know where the error happened.

Attempt 1: Add spans to raw syntax:

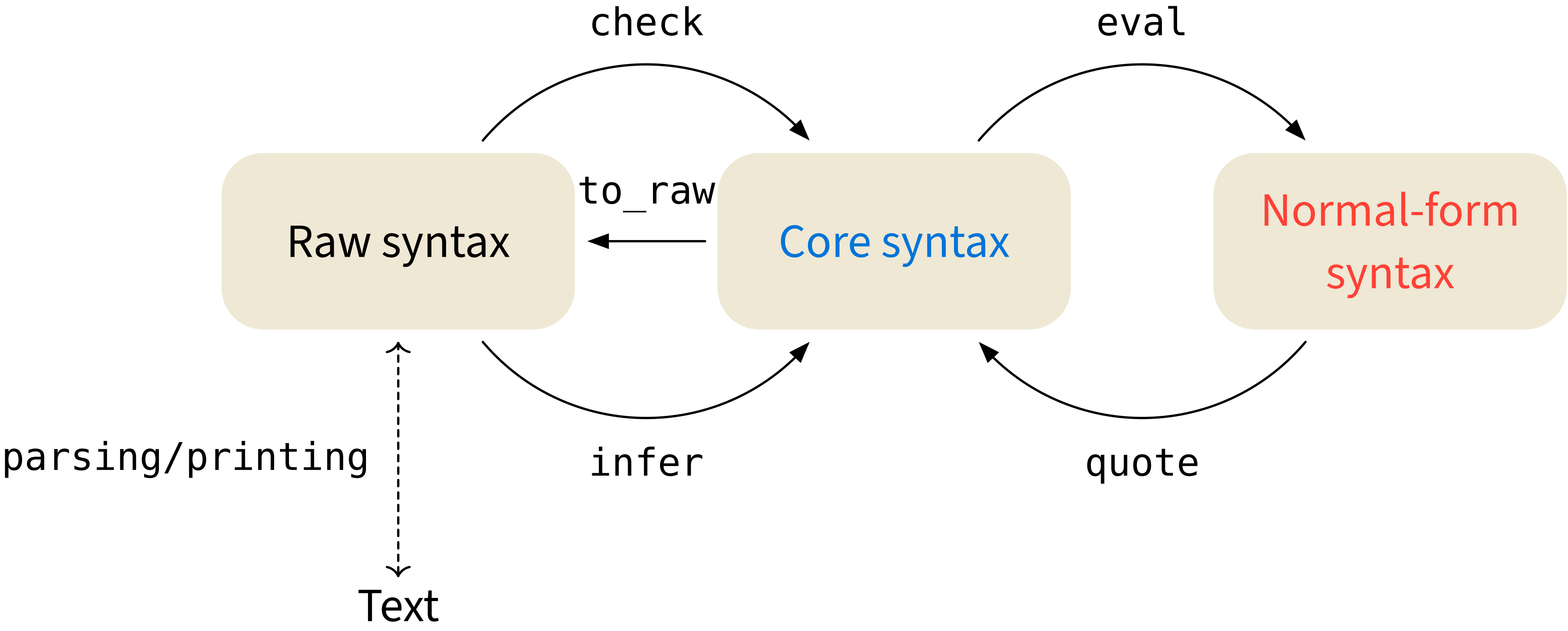
```
Term = Var Name Range<int> | ...
```

Attempt 2: Add a generic type annotation to raw syntax:

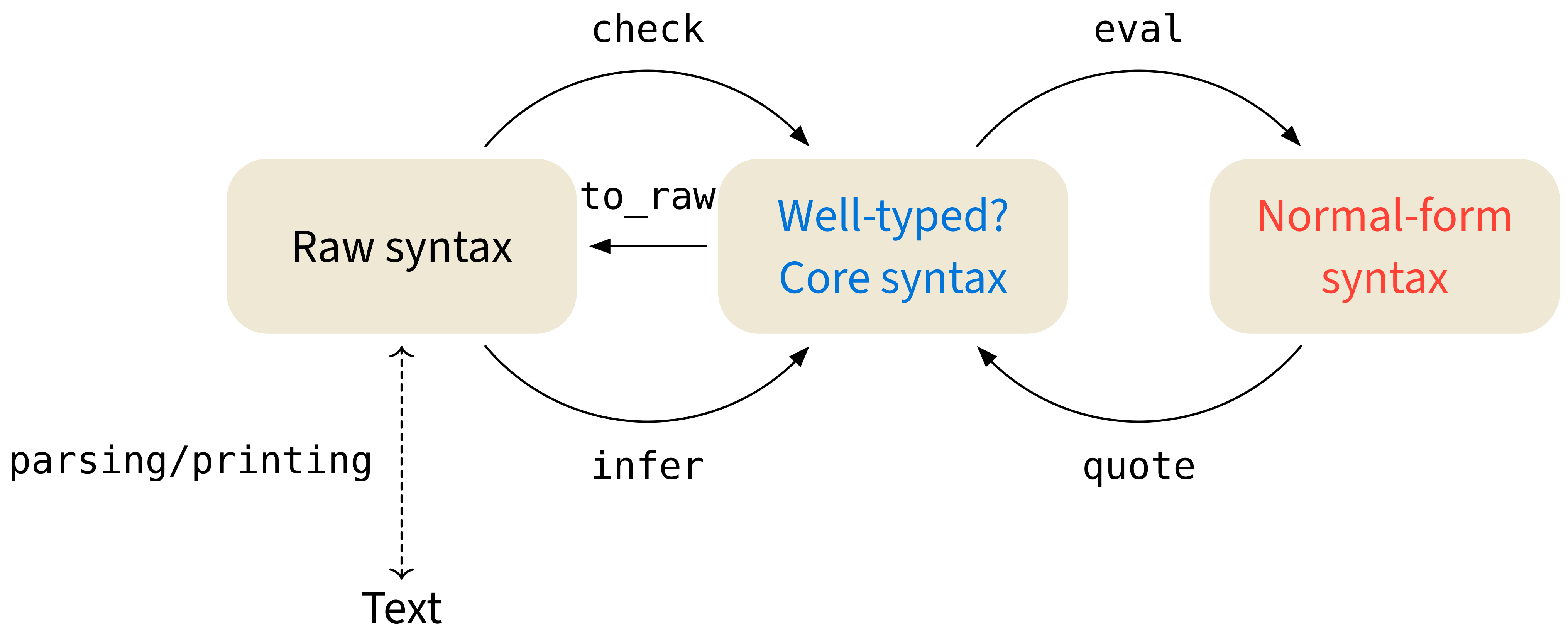
```
Term<Unit> = Var Name Unit | ...
Error<Unit> = UnknownVariable Name Unit | ...
```

Challenge: I am not a perfect programmer

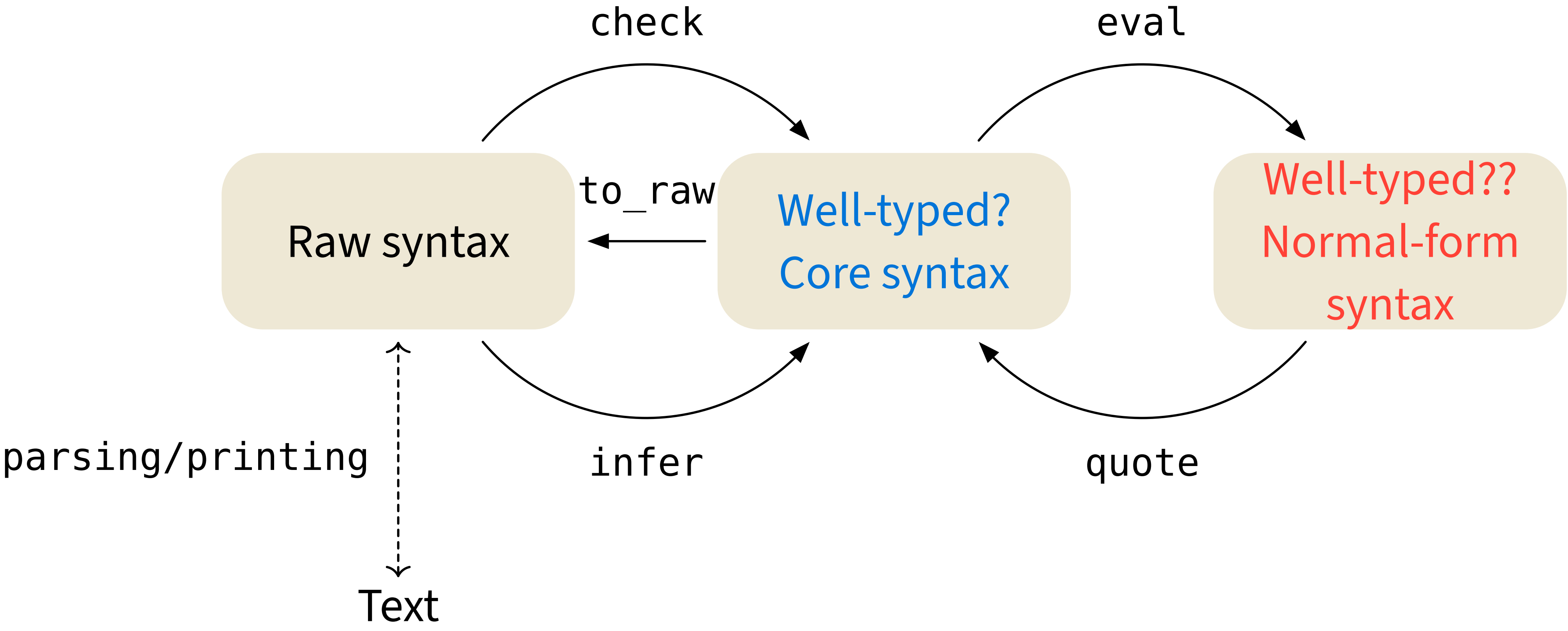
Challenge: I am not a perfect programmer



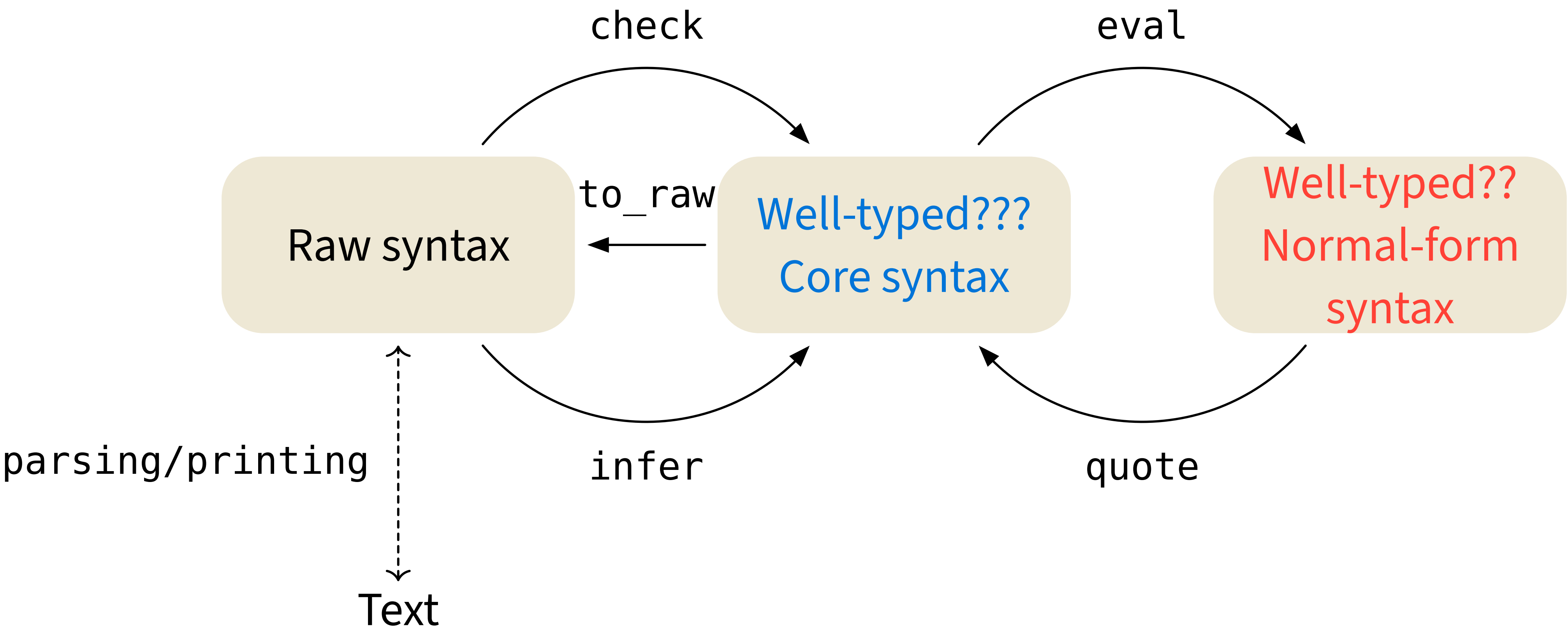
Challenge: I am not a perfect programmer



Challenge: I am not a perfect programmer



Challenge: I am not a perfect programmer



Challenge: I am not a perfect programmer

The ability to verify/re-typecheck terms would help debug errors.

Can we pass through the same typechecking function again?

Re-typechecking involves passing through raw syntax.

Can we independently verify core/normal-form syntax?

Any typechecking depends on evaluation.

Can we at least nicely print core/normal-form syntax?

Not automatically.

Challenge: I am not a perfect programmer

Scoped syntax?

Challenge: I am not a perfect programmer

Scoped syntax?

Smaller Kernel?

Challenge: I am not a perfect programmer

Scoped syntax?

Smaller Kernel?

“Pretty” debug printing?

Challenge: I am not a perfect programmer

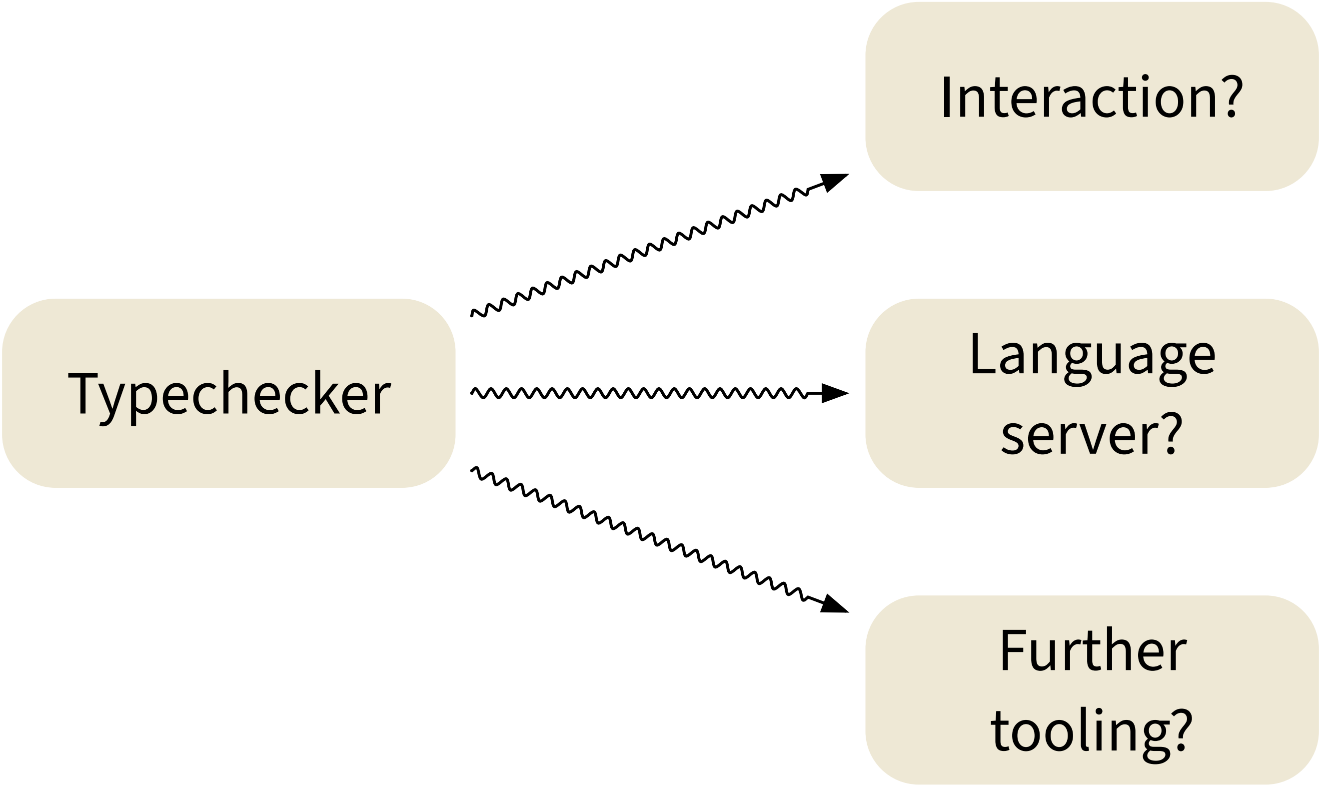
Scoped syntax?

Smaller Kernel?

“Pretty” debug printing?

Formalise everything?

Challenge/Propaganda: Beyond a typechecker



Conclusions

- I created a typechecker for a non-standard language.
- Even with the language being non-standard, it was still possible to adapt common techniques.
- Despite this, it is easy to make mistakes.
- It's difficult to retroactively add tooling to a language.
- Tell me why this set up is wrong!