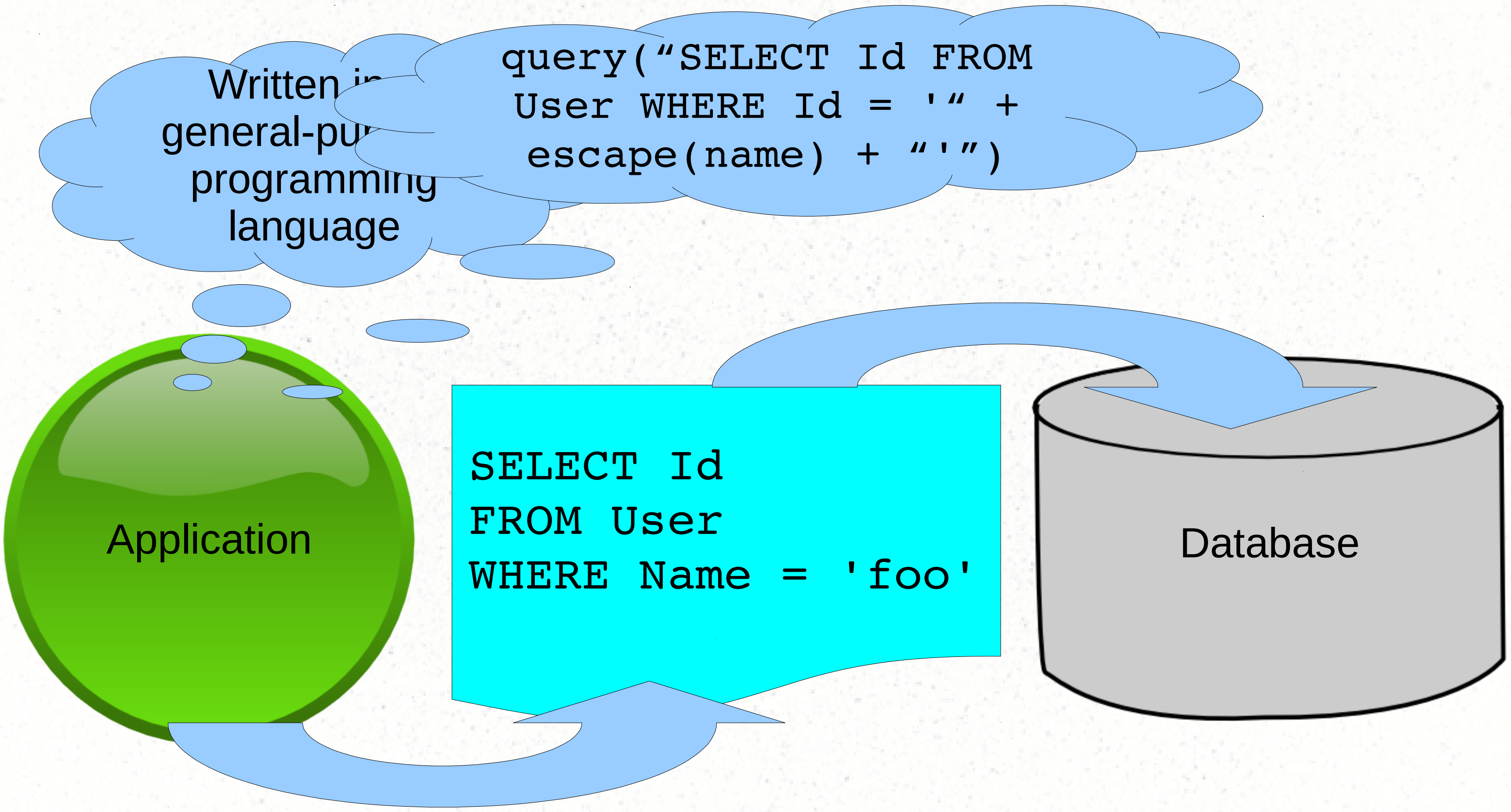
Safe Database Abstractions with Type-Level Record Computation

Adam Chlipala

RADICAL 2010

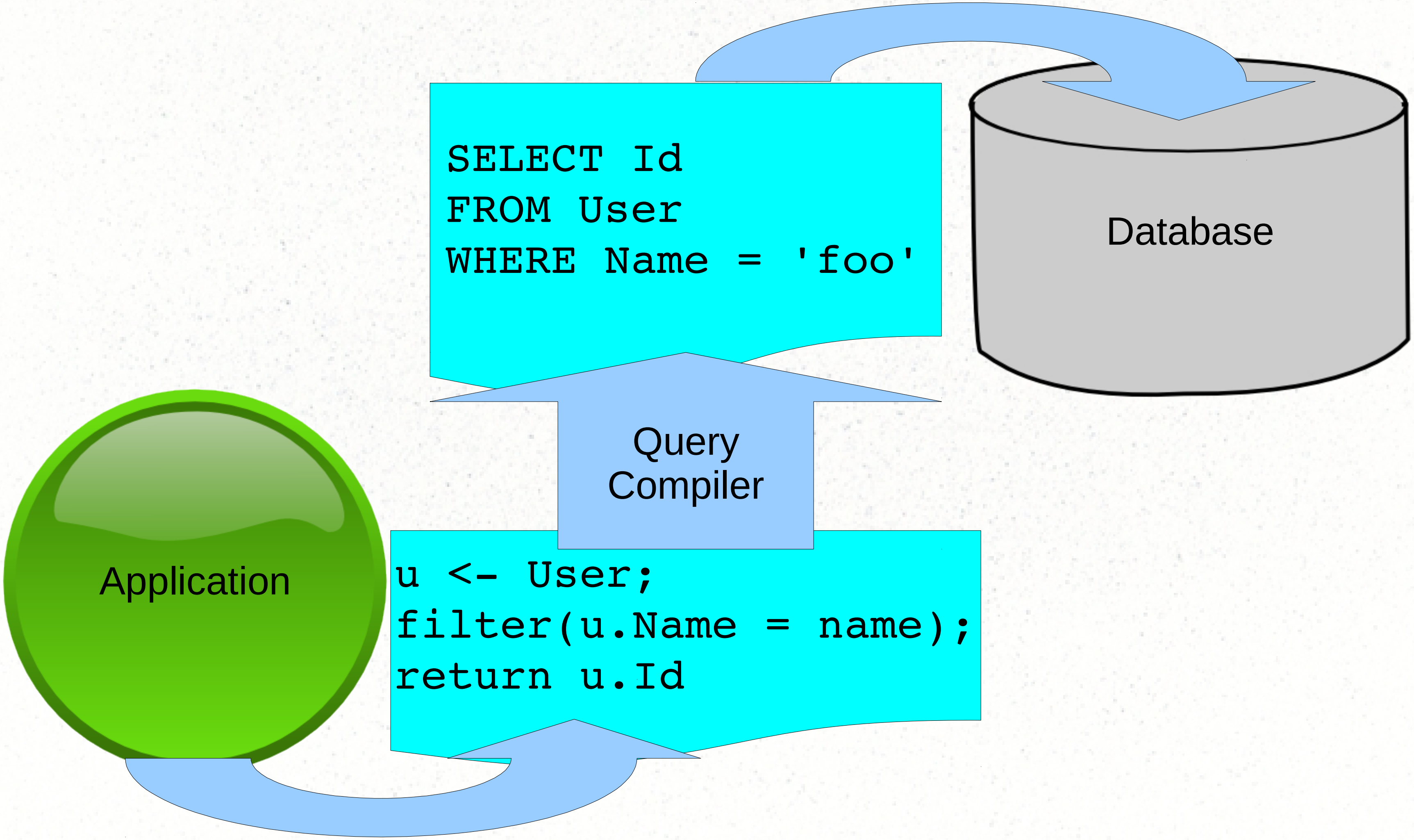
In the Wild...



```
query(“SELECT Id FROM
User WHERE Id = '“ +
escape(name) + “'”)
```

```
SELECT Id
FROM User
WHERE Name = 'foo'
```

Language-Integrated Query



```
SELECT Id
FROM User
WHERE Name = 'foo'
```

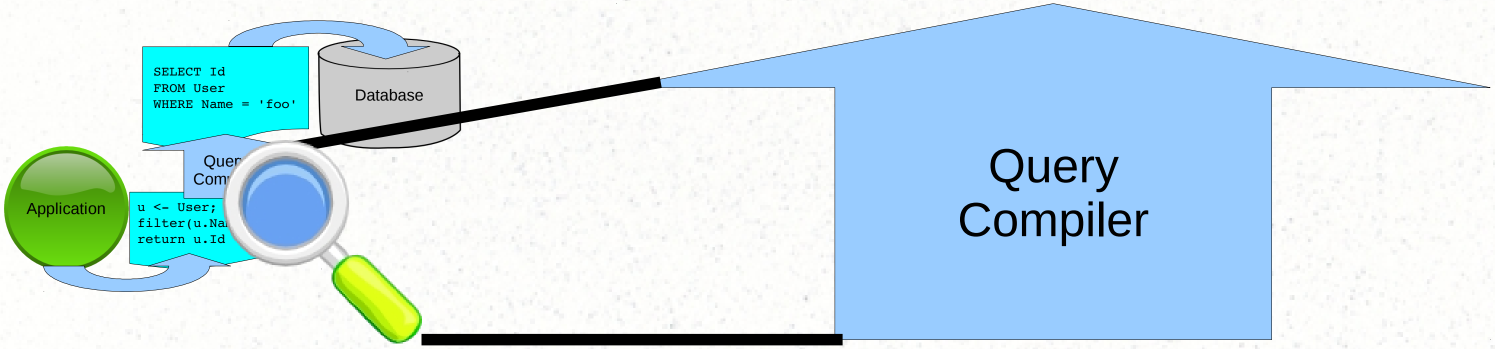
Database

Query
Compiler

Application

```
u <- User;
filter(u.Name = name);
return u.Id
```

Language or Library?



```
SELECT Id
FROM User
WHERE Name = 'foo'
```

```
u <- User;
filter(u.Na
return u.Id
```

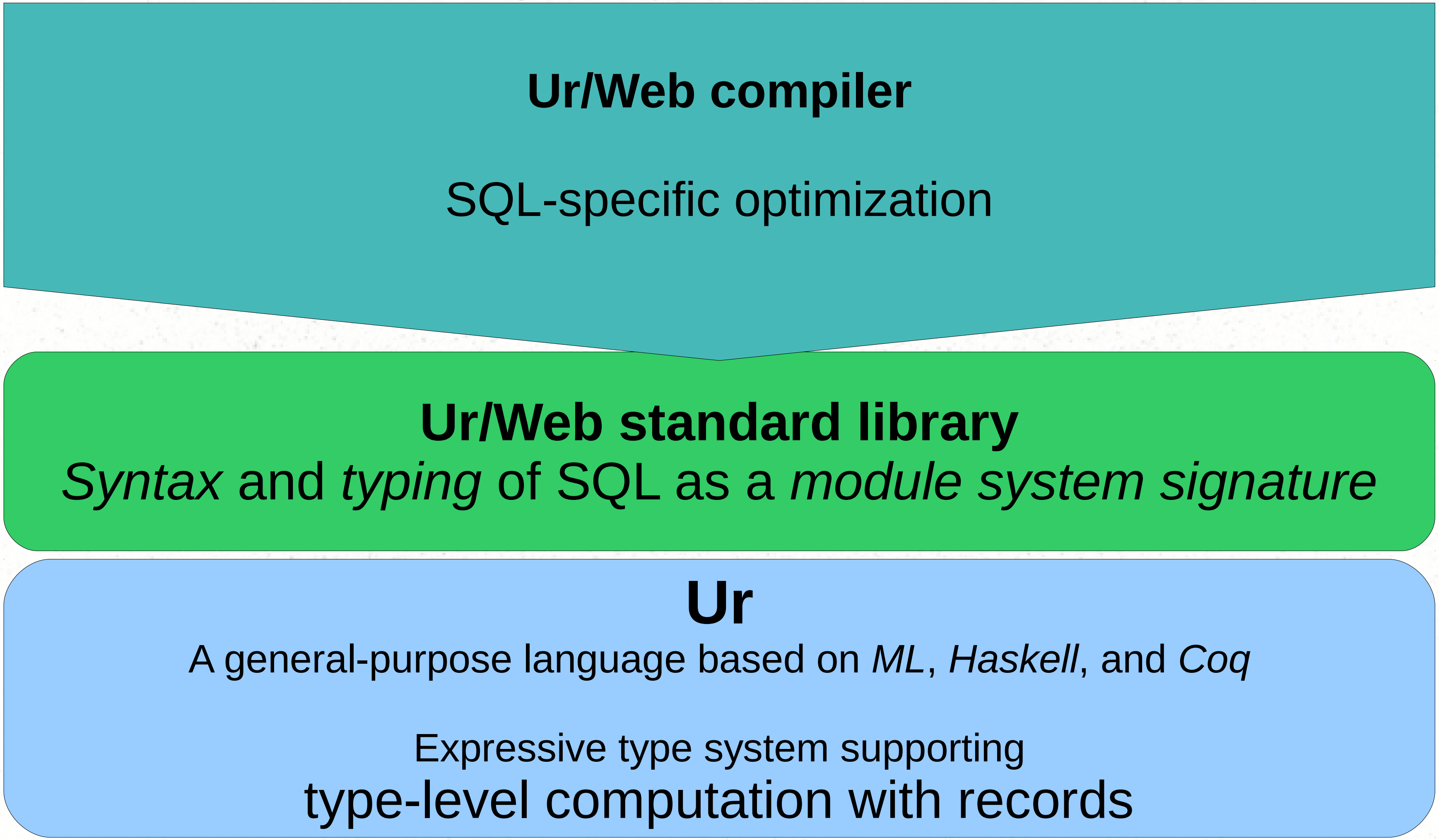
Library? Bridging the gap? *Language?*

How do we know this is done right?

- Programmers can add support for new query languages, without touching the main compiler.
- Can take full advantage of the target language, by building new libraries as needed.

- Static checking of query syntax
- Static guarantee that every query is compiled properly
- Compile-time optimization removes interpretation overhead.

First-Class Queries in Ur/Web



Ur/Web compiler

SQL-specific optimization

Ur/Web standard library

Syntax and *typing* of SQL as a *module system signature*

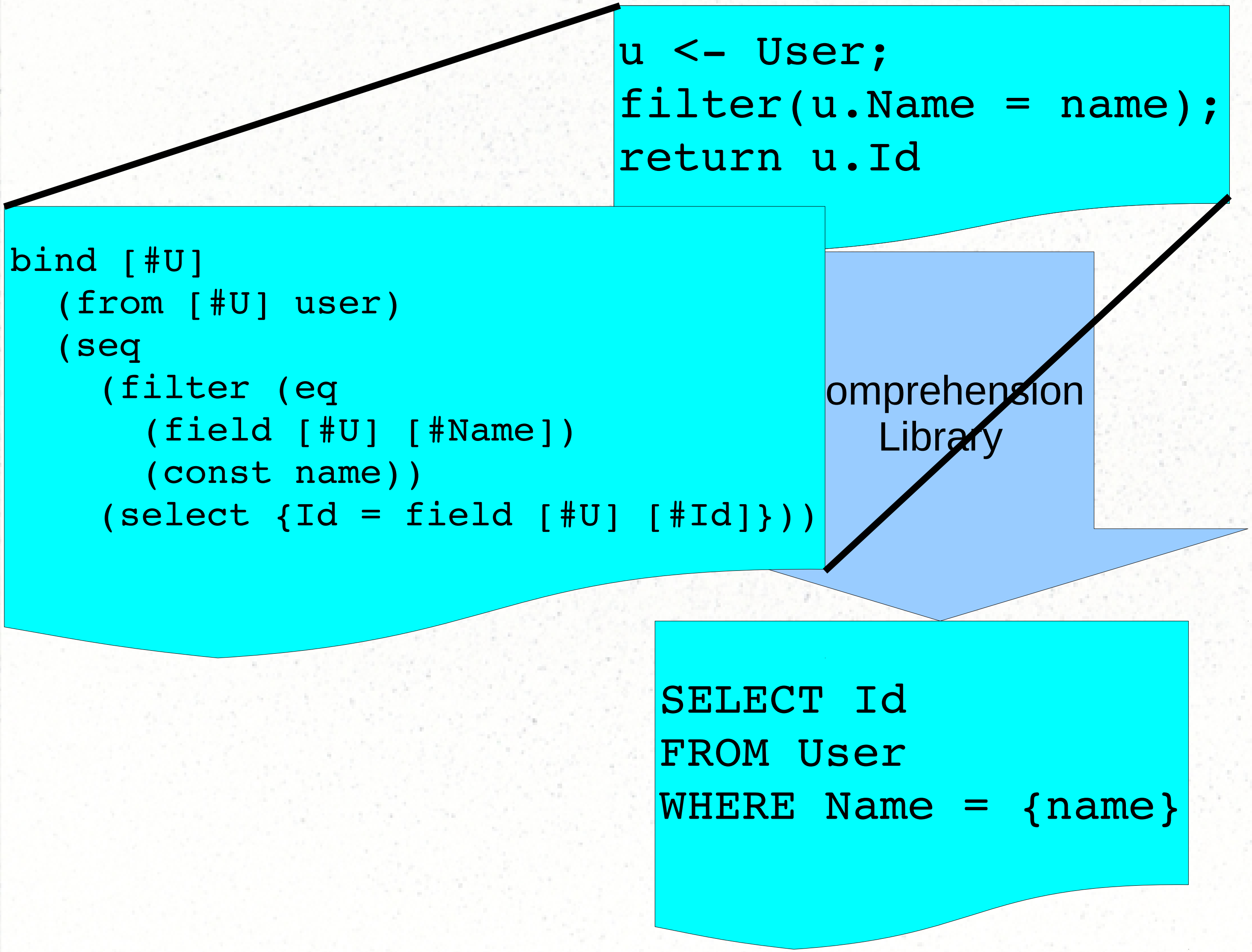
Ur

A general-purpose language based on *ML*, *Haskell*, and *Coq*

Expressive type system supporting

type-level computation with records

Safe Abstractions via Types



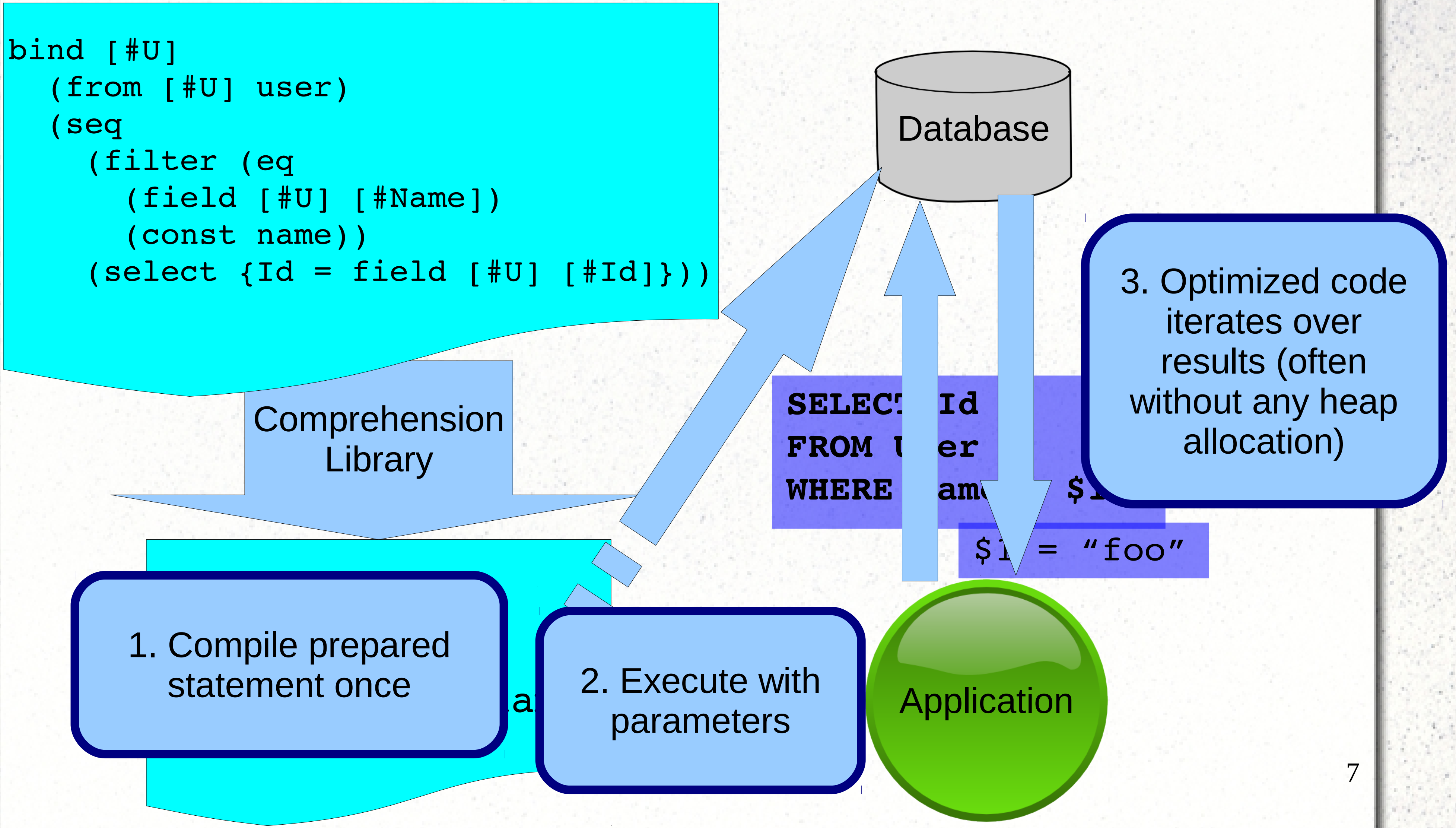
```
u <- User;
filter(u.Name = name);
return u.Id
```

```
bind [#U]
  (from [#U] user)
  (seq
    (filter (eq
      (field [#U] [#Name])
      (const name))
    (select {Id = field [#U] [#Id]}))
```

omprehension
Library

```
SELECT Id
FROM User
WHERE Name = {name}
```

Optimization for Free



```
bind [#U]
  (from [#U] user)
  (seq
    (filter (eq
      (field [#U] [#Name])
      (const name))
    (select {Id = field [#U] [#Id]}))
```

Comprehension Library

Database

```
SELECT Id
FROM U er
WHERE am   $1
```

$1 = “foo”

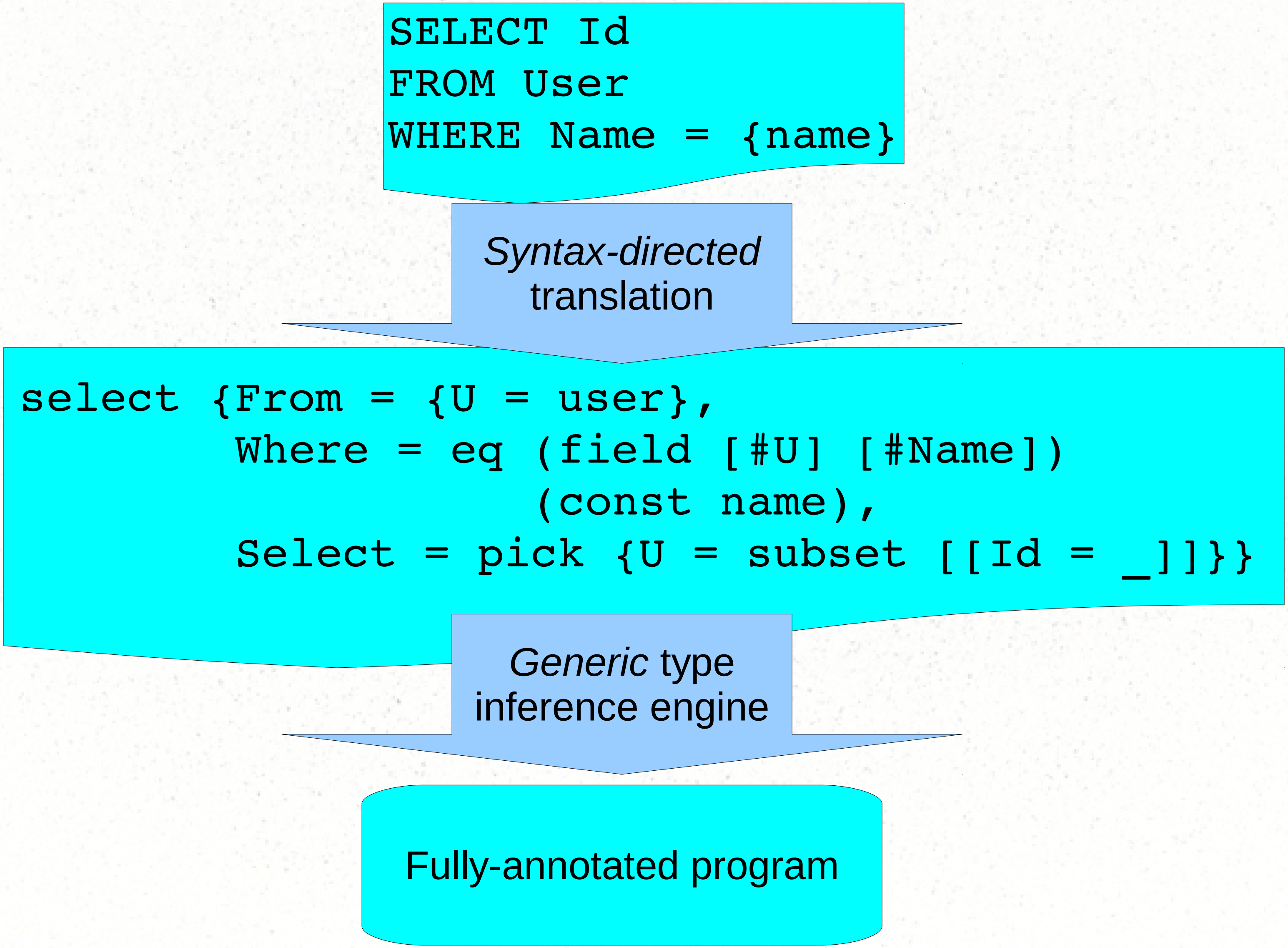
1. Compile prepared statement once

2. Execute with parameters

3. Optimized code iterates over results (often without any heap allocation)

Application

Type Inference for Free



```
SELECT Id
FROM User
WHERE Name = {name}
```

Syntax-directed translation

```
select {From = {U = user},
        Where = eq (field [#U] [#Name])
                   (const name),
        Select = pick {U = subset [[Id = _]]}}
```

Generic type inference engine

Fully-annotated program

Typing SELECT

First class polymorphism with higher kinds

Type-level `map` over records

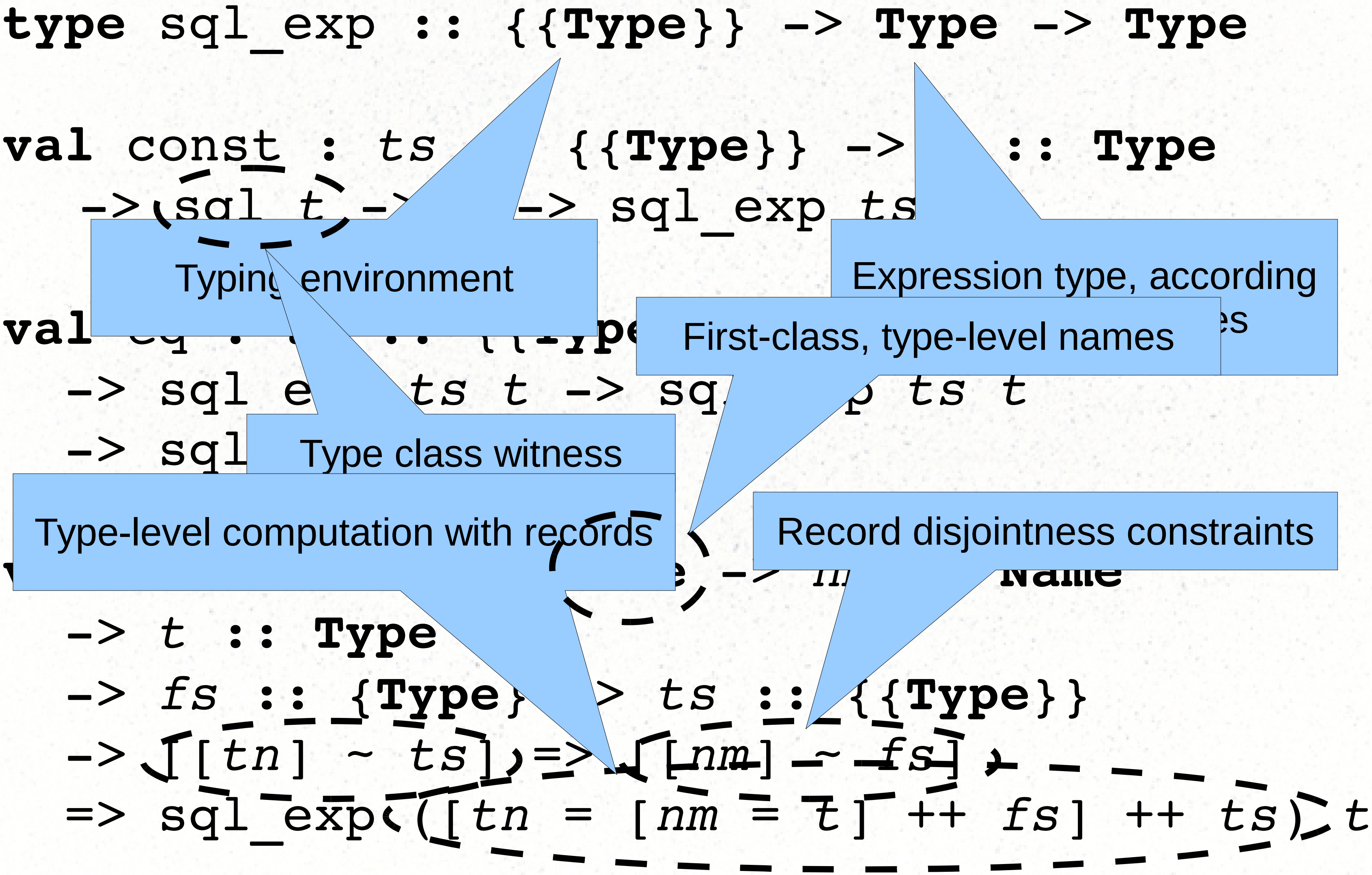
The kind of records of records of types

```
val select : full :: {{Type}}
   -> keep :: {{Type}}
   -> {From : $(map sql_table full),
       Where : sql_exp full bool,
       Select : pick keep full}
   -> sql_query keep
```

Lightweight "proofs"

Richly-typed abstract syntax

SQL Expression Syntax



```
type sql_exp :: {{Type}} -> Type -> Type

val const : ts     {{Type}} ->     :: Type
   -> sql_t ->     -> sql_exp ts

val
  -> sql e     ts t -> sq     p ts t
  -> sql

                        e -> n       Name
  -> t :: Type
  -> fs :: {Type}     > ts :: {{Type}}
  -> [[tn] ~ ts] => [[nm] ~ fs] =>
  => sql_exp ([tn = [nm = t] ++ fs] ++ ts) t
```

Typing environment

Expression type, according

es

First-class, type-level names

Type class witness

Type-level computation with records

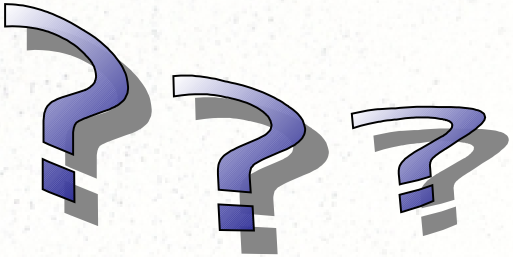
Record disjointness constraints

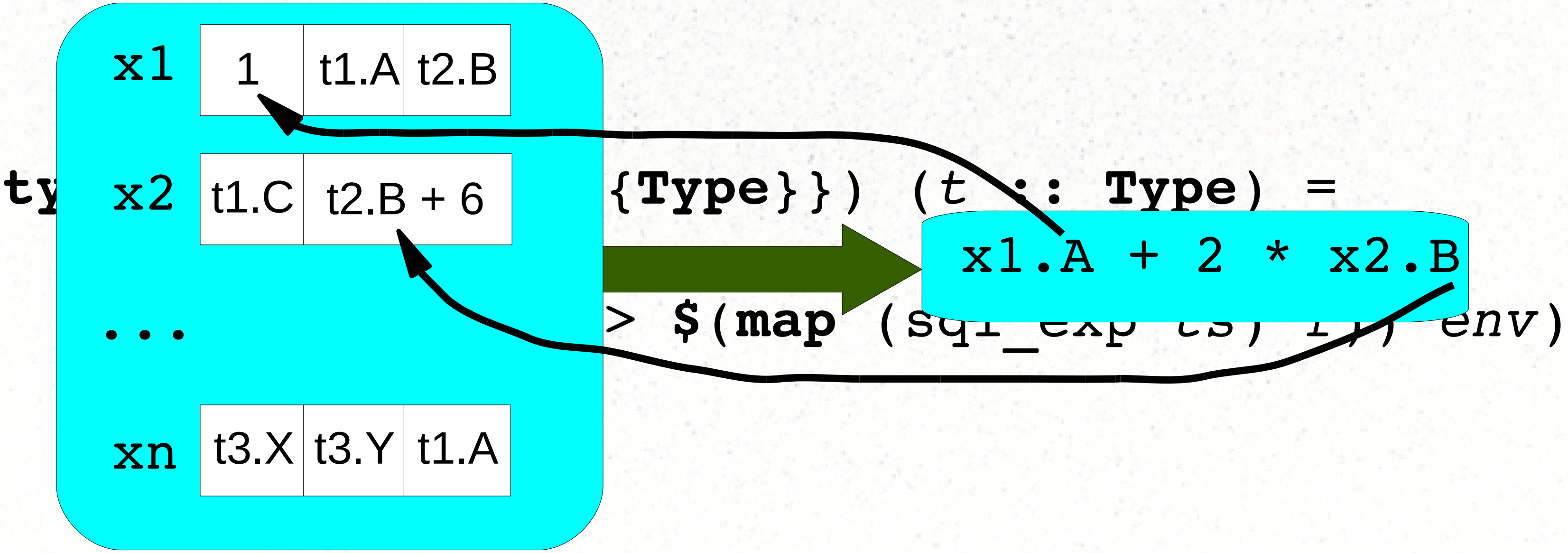
Implementing Comprehensions

```
x1 <- ....;
x2 <- ....;
....
xn <- ....;
return ....
```

`from_table t` → SELECT * FROM ..., t, ...

```
y1 <- ....;
....
ym <- ....;
return ....
```





x1	1	t1.A	t2.B
x2	t1.C	t2.B + 6	
...			
xn	t3.X	t3.Y	t1.A

```
ty        {Type}}) (t :: Type) =
          > $(map (sql_exp ts) f)) env)
```

```
x1.A + 2 * x2.B
```

Supported SQL Features

- Inner and outer joins
- Grouping and aggregation
- Relational operators (union, intersection, ...)
- Subqueries
- Sorting of results (“`ORDER BY`”)
- Table constraints (foreign keys, ...)
- Views
- Insert/update/delete

Checking Security Policies

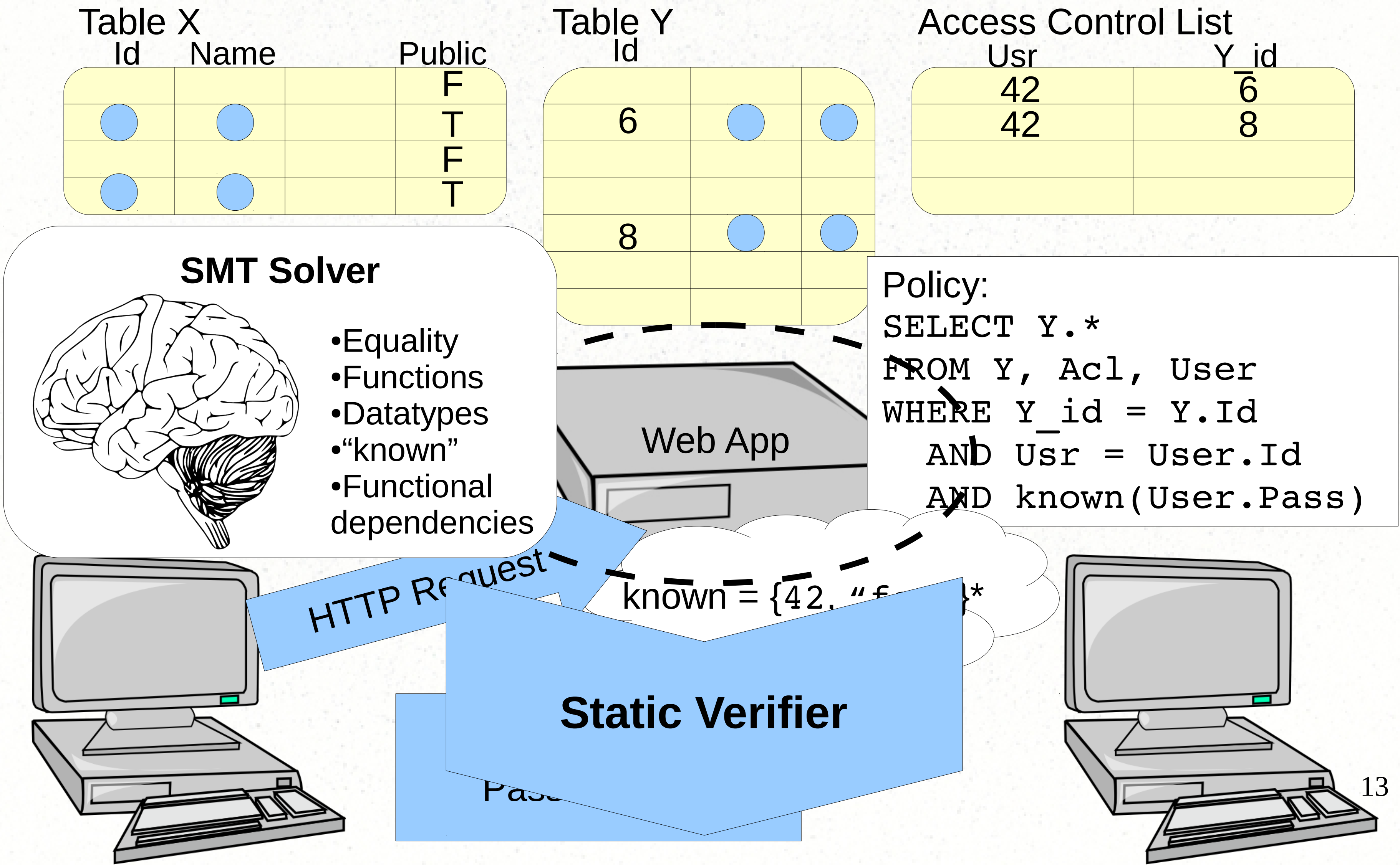


Table X

Id	Name		Public
			F
●	●		T
			F
●	●		T

Table Y

Id		
6	●	●
8	●	●

Access Control List

Usr	Y_id
42	6
42	8

SMT Solver

- •Equality
- •Functions
- •Datatypes
- •“known”
- •Functional dependencies

Policy:

```
SELECT Y.*
FROM Y, Acl, User
WHERE Y_id = Y.Id
  AND Usr = User.Id
  AND known(User.Pass)
```

Ur/Web Available At:

`http://www.impredicative.com/ur/`

Including online demos with syntax-highlighted source code

Smart Type Inference

```
fun foo [ts :: {(Type * Type)}] (fl : folder ts)
    (tabs : $(map (fn (fs, _) => sql_table fs) ts))
    (funcs : $(map (fn (fs, t) => $fs -> t) ts))
    : list $(map (fn (_, t) => t) ts) =
    ....
    select {From = (* build record from tabs *),
            ...}
    ....
```

```
$(map (fn (fs, _) => sql_table fs) ts)
```

The Ur inference engine must apply a **fusion law**!

Example usage:

```
foo {X = t1, Y = t2}
    {X = fn r => r.A, Y = fn r => r.B + r.C}
```

```
val
  _
      ...}
  -> sql_query keep
```