Programmable Semantic Fragments

The Design and Implementation of **typy**

Cyrus Omar
Jonathan Aldrich

Carnegie Mellon University

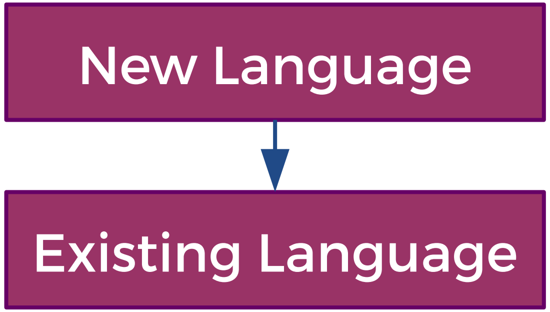
GPCE 2016

New Languages

New Language

Existing Language

New Languages



New Language

Existing Language

→ Safe & Natural Foreign Interface

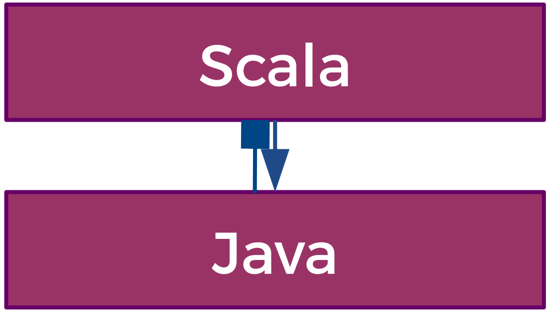
New Languages

Scala

Java

Safe & Natural Foreign Interface

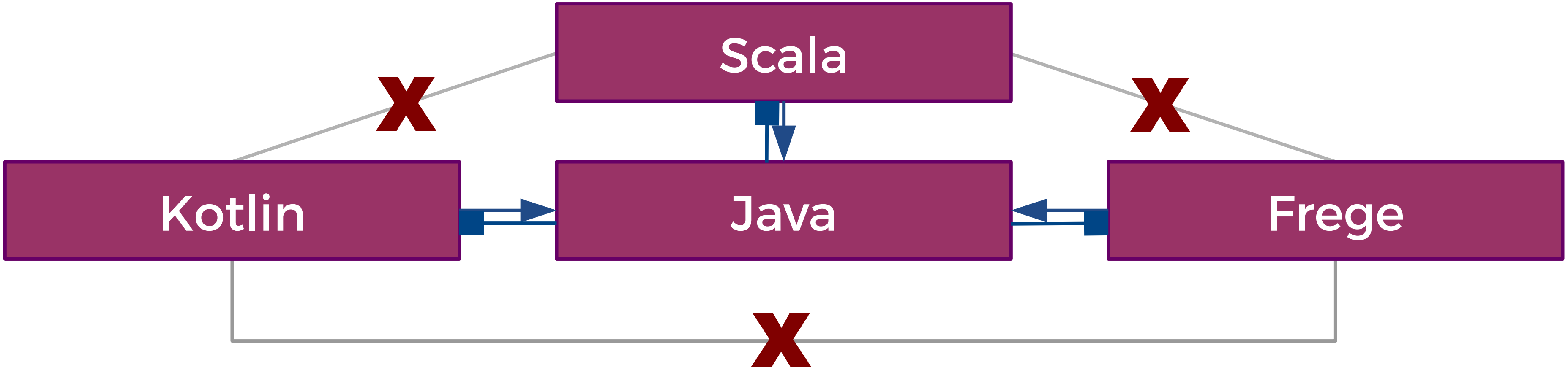
Problem: Backwards Compatibility



→ **Safe & Natural Foreign Interface**

—■ **Unsafe or Unnatural Foreign Interface**

Problem: Lateral Compatibility

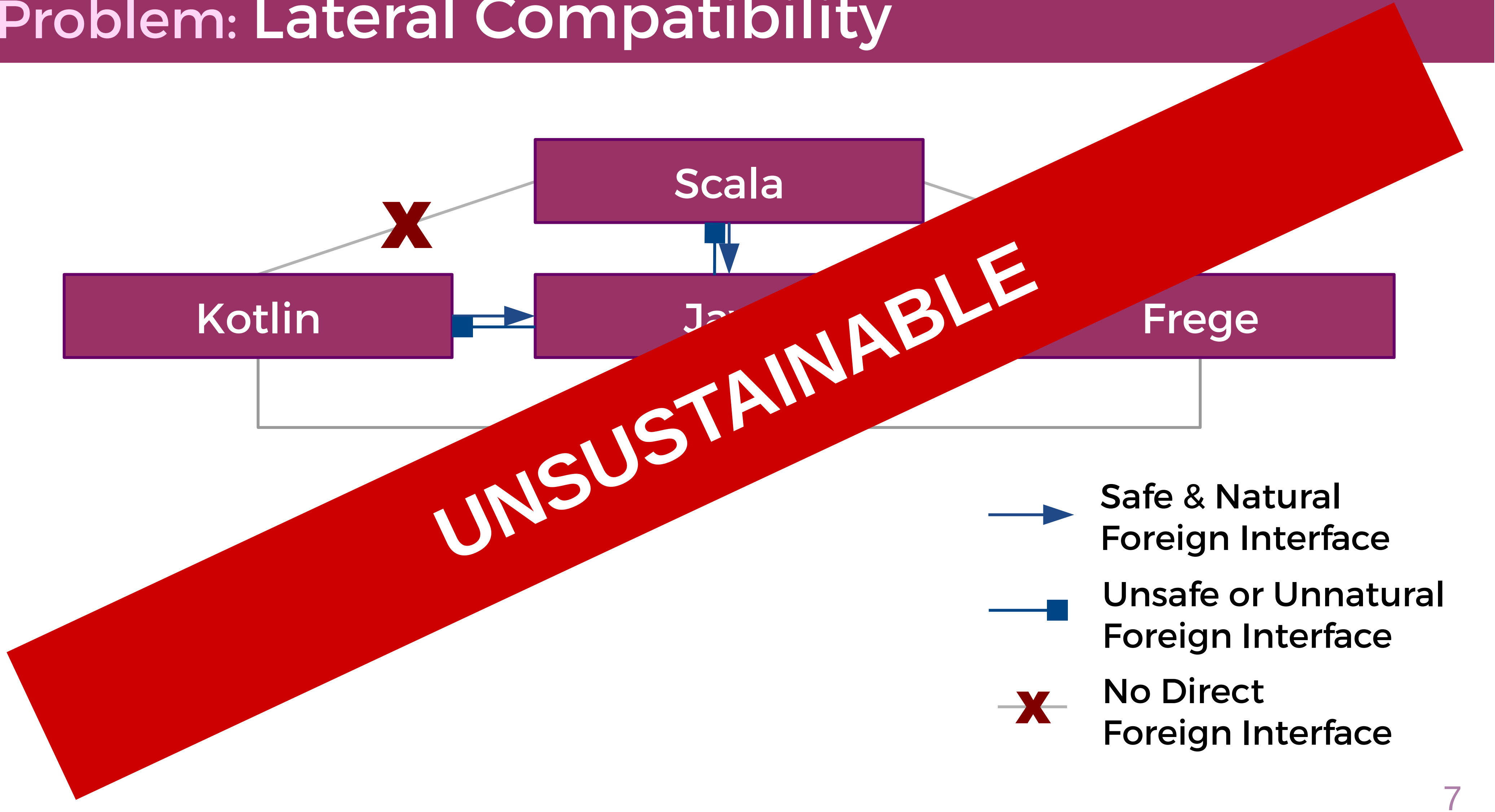



Unsafe or Unnatural Foreign Interface


Safe & Natural Foreign Interface


No Direct Foreign Interface

Problem: Lateral Compatibility



Safe & Natural Foreign Interface

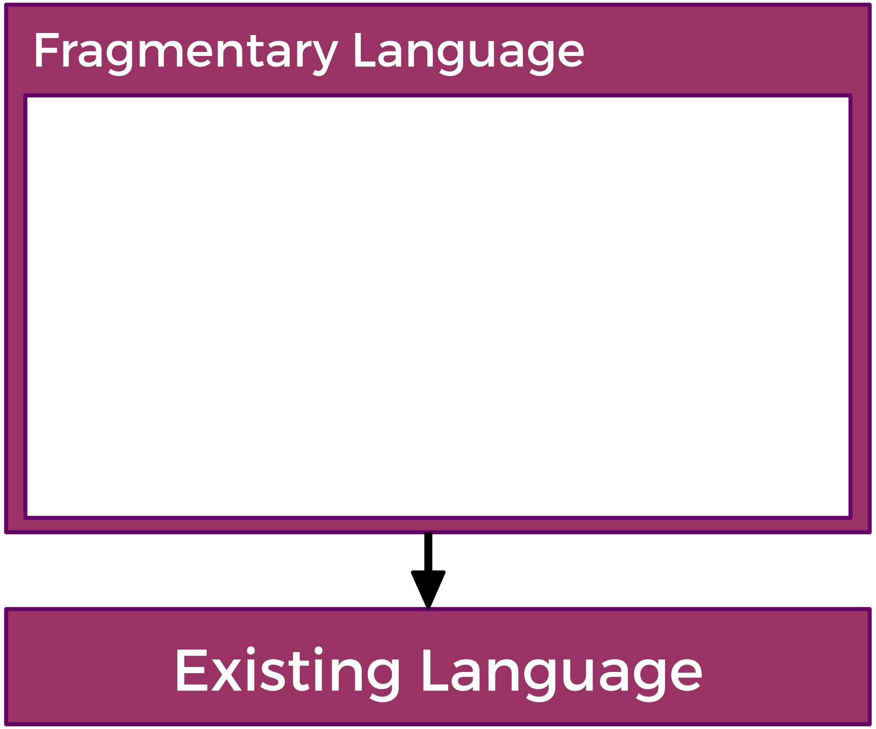
Unsafe or Unnatural Foreign Interface

No Direct Foreign Interface

Our Approach: A Fragmentary Semantics

Existing Language

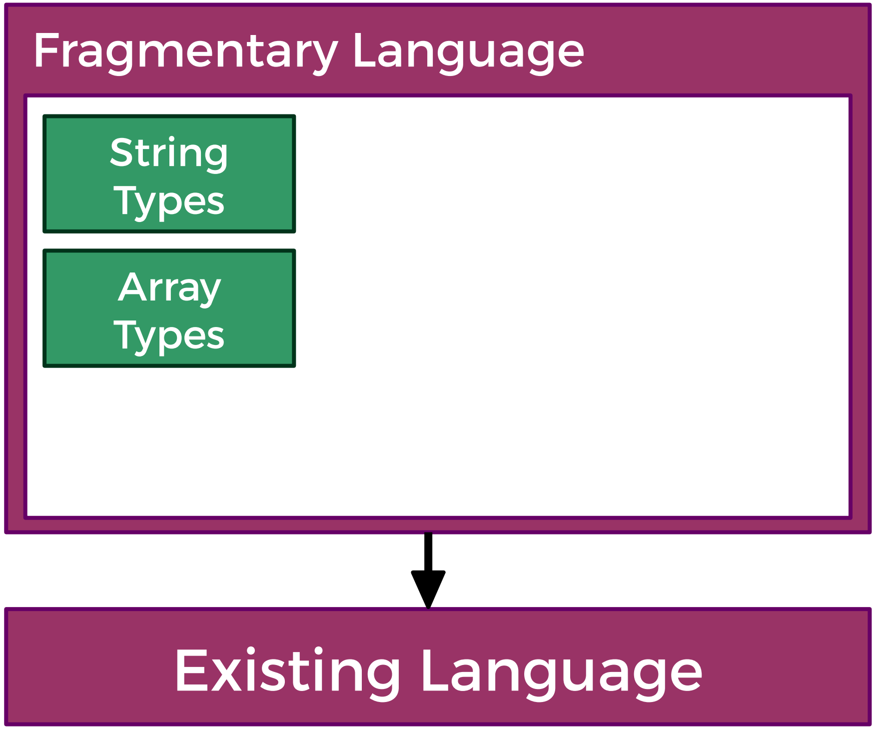
Our Approach: A Fragmentary Semantics



Fragmentary Language

Existing Language

Our Approach: A Fragmentary Semantics



Fragmentary Language

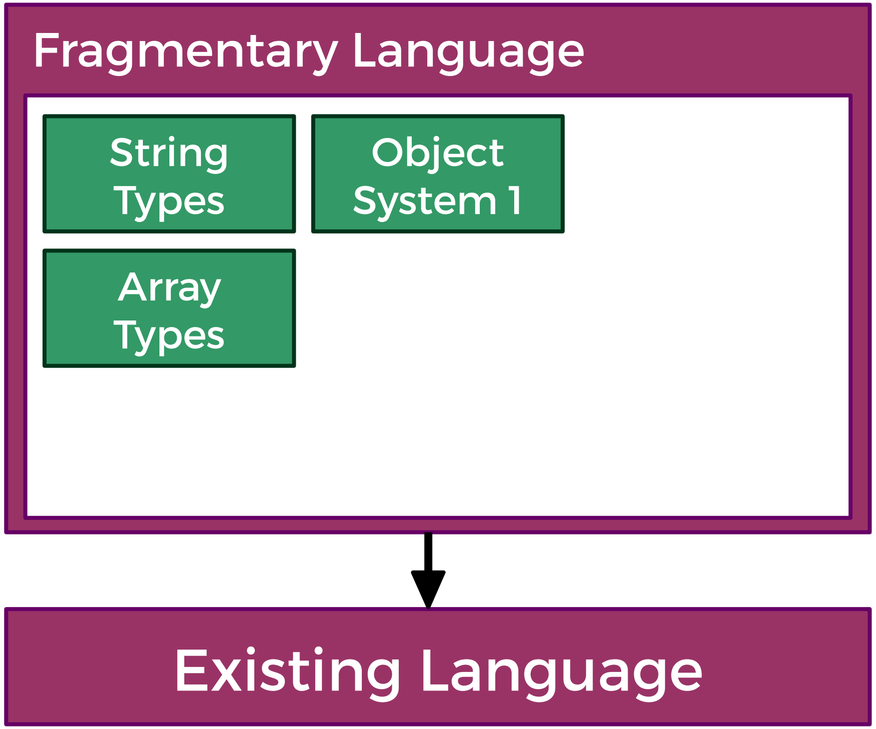
String Types

Array Types

Existing Language

🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics


Fragmentary Language

String Types

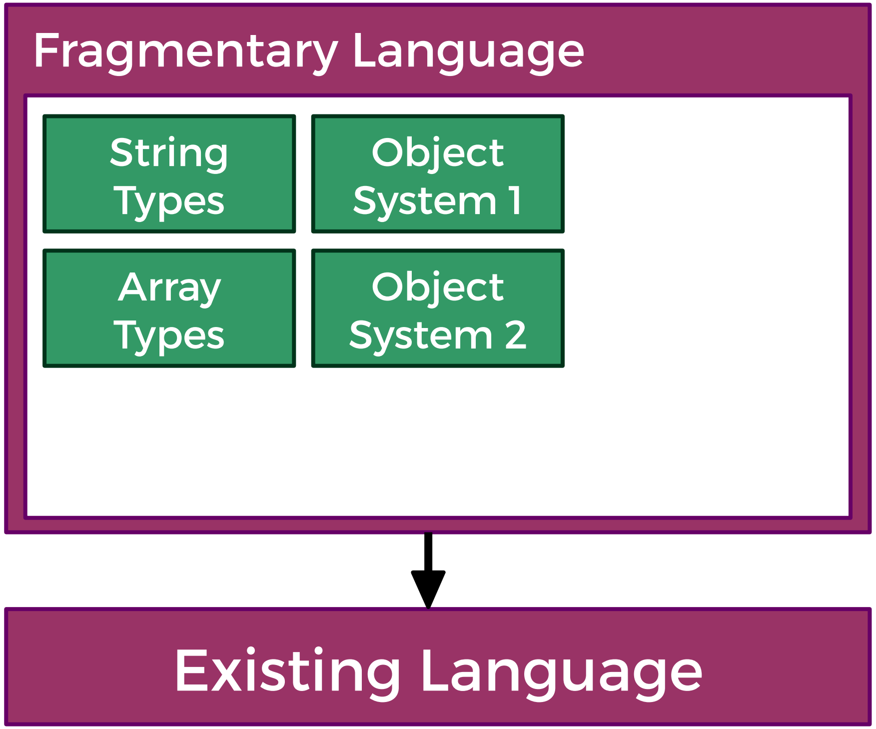
Object System 1

Array Types

Existing Language

🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



Fragmentary Language

String Types

Object System 1

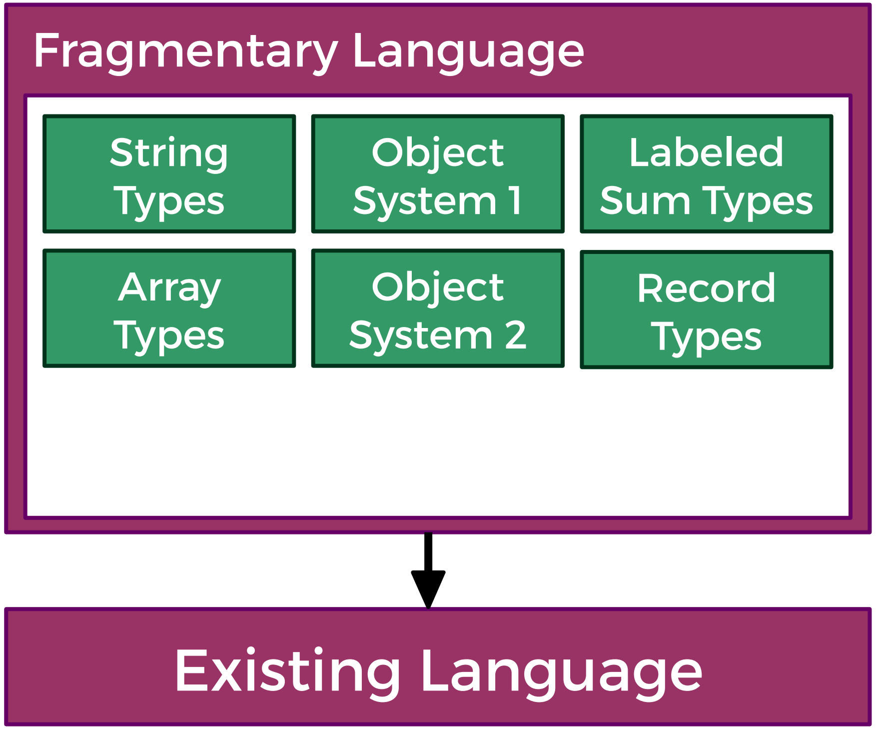
Array Types

Object System 2

Existing Language

🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



Fragmentary Language

String Types

Object System 1

Labeled Sum Types

Array Types

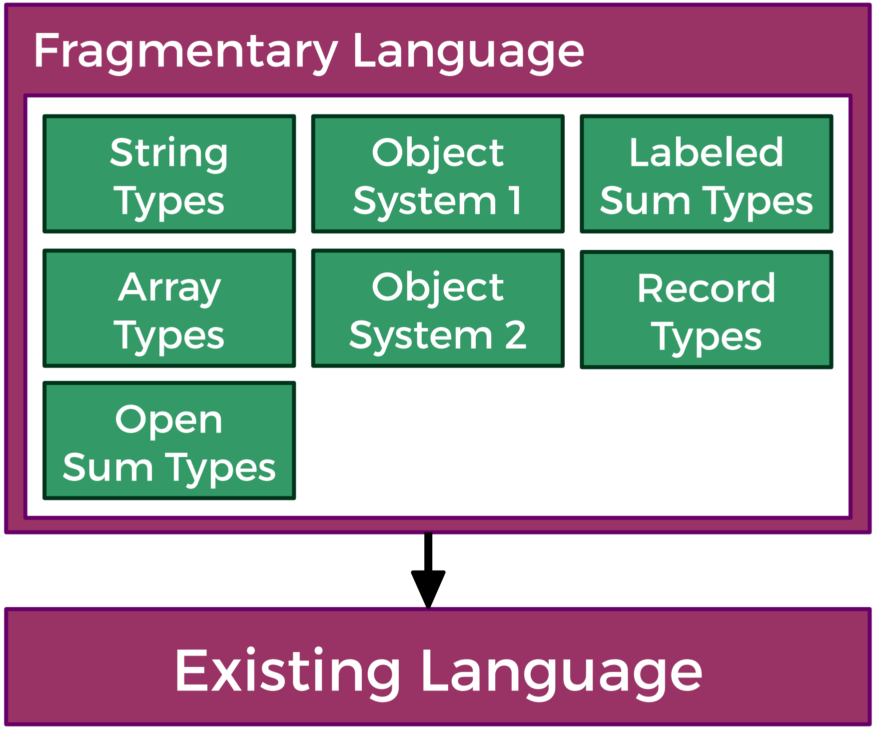
Object System 2

Record Types

Existing Language

🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



Fragmentary Language

String Types

Object System 1

Labeled Sum Types

Array Types

Object System 2

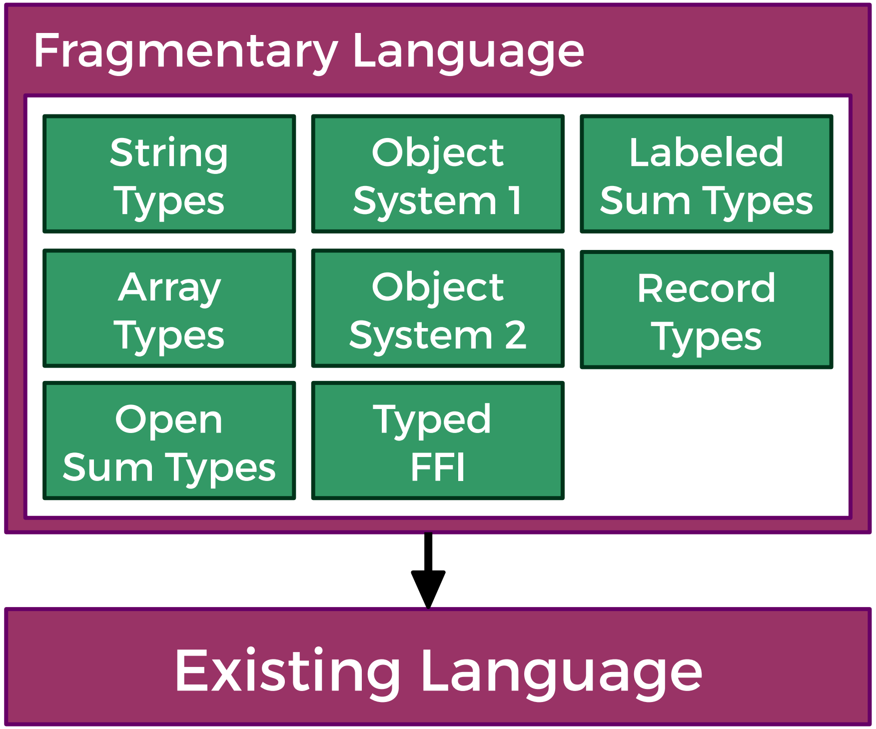
Record Types

Open Sum Types

Existing Language

🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



Fragmentary Language

String Types

Object System 1

Labeled Sum Types

Array Types

Object System 2

Record Types

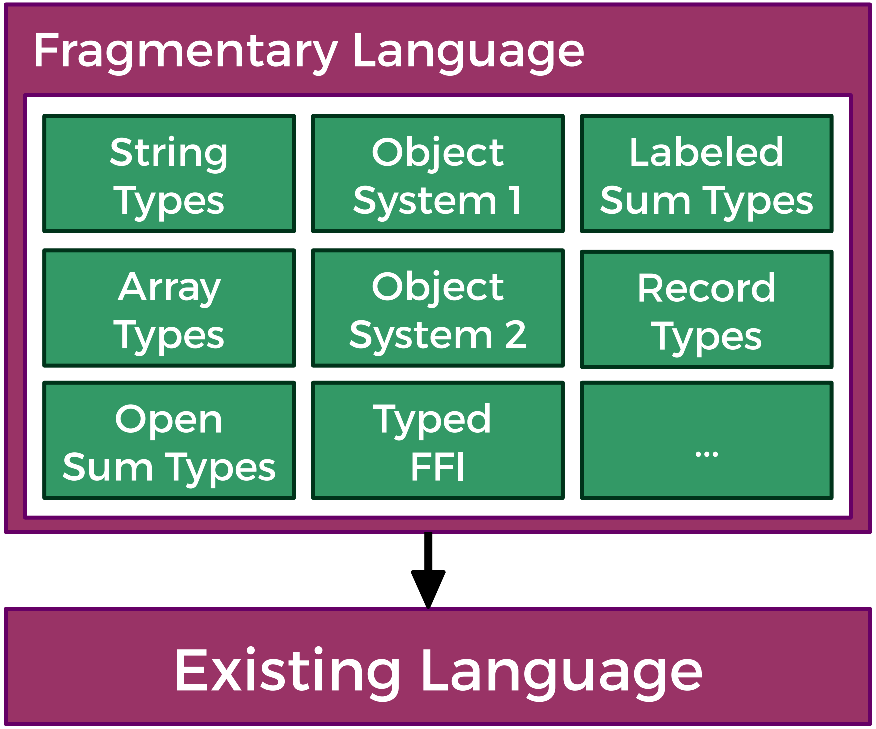
Open Sum Types

Typed FFI

Existing Language

🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



Fragmentary Language

String Types

Object System 1

Labeled Sum Types

Array Types

Object System 2

Record Types

Open Sum Types

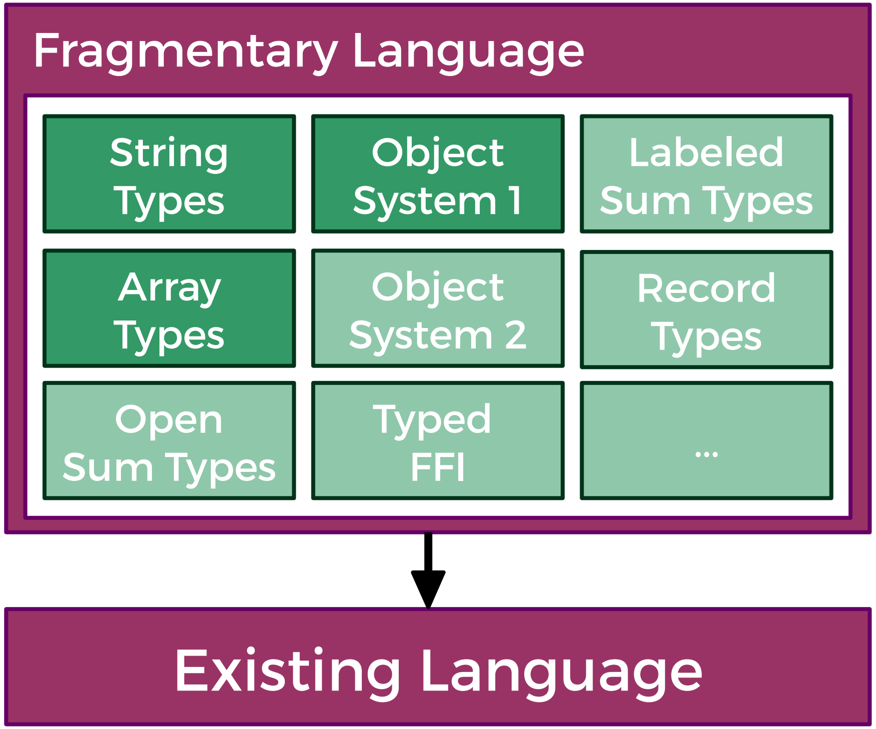
Typed FFI

...

Existing Language

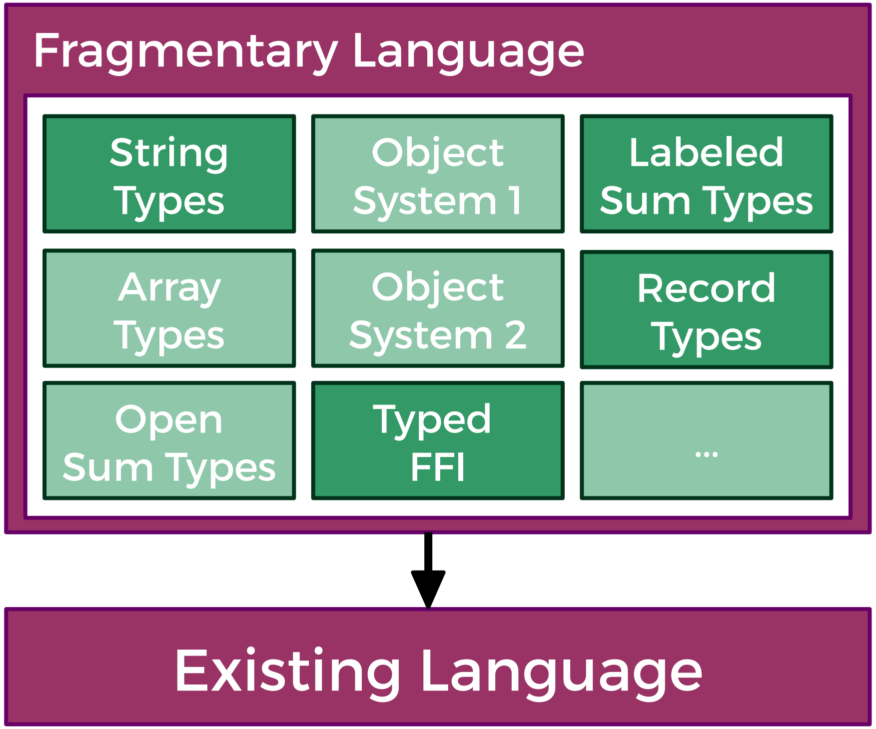
🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



Fragmentary Language

String Types

Object System 1

Labeled Sum Types

Array Types

Object System 2

Record Types

Open Sum Types

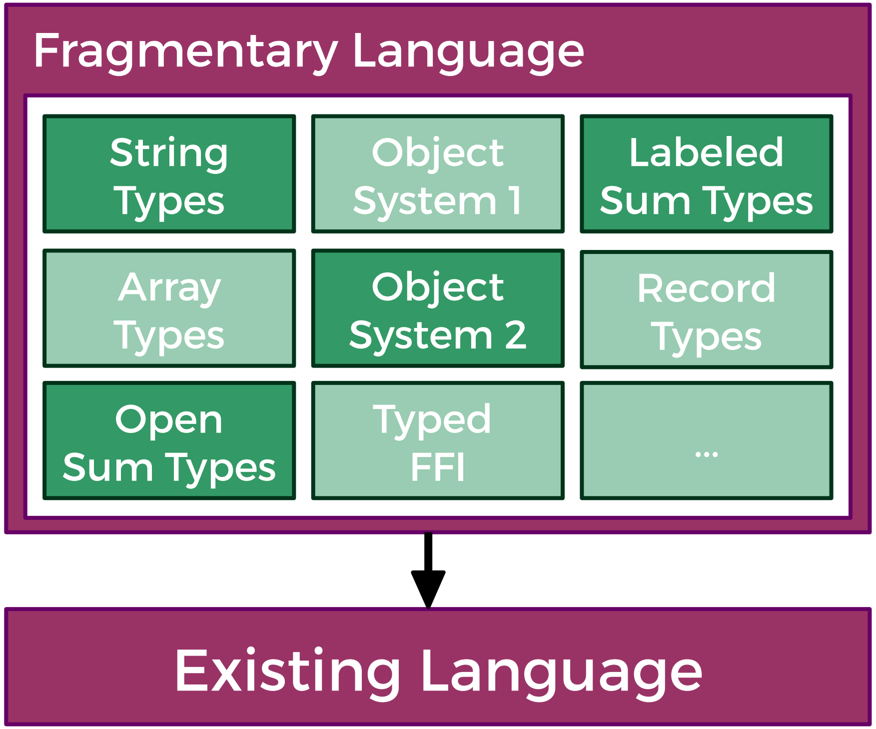
Typed FFI

...

Existing Language

🟩 Semantic Fragments

Our Approach: A Fragmentary Semantics



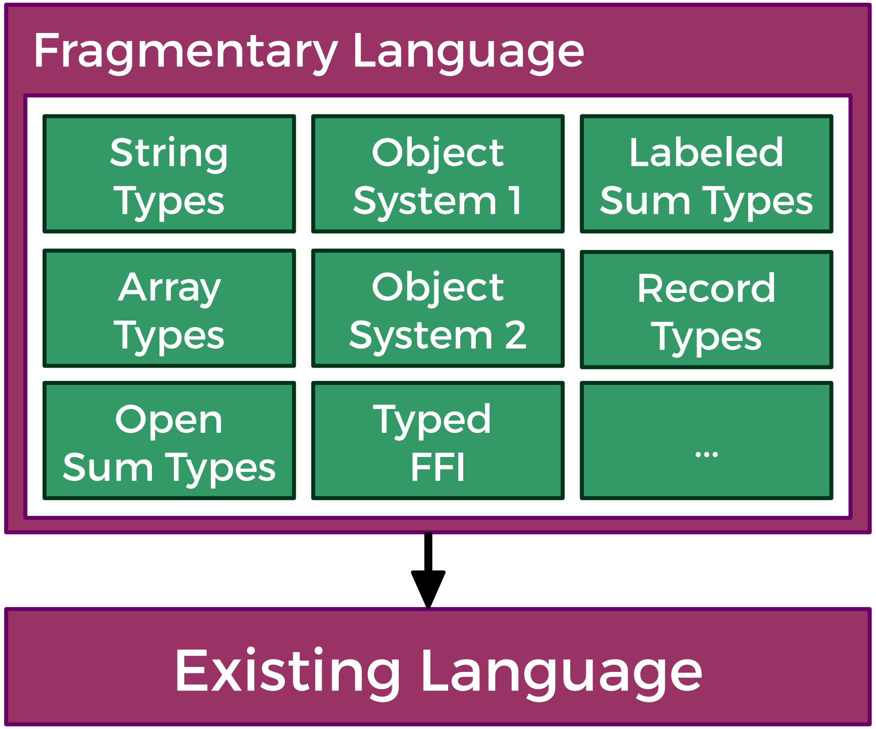
Fragmentary Language

String Types
Object System 1
Labeled Sum Types
Array Types
Object System 2
Record Types
Open Sum Types
Typed FFI
...

Existing Language

🟩 Semantic Fragments

Problem: Composition



🟩 Semantic Fragments

Problem: Composition

```
test_acct = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

Problem: Composition

```
test_acct = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

Is this

- a **record**?
- an **object**?
- a **dynamic dictionary**?
- an **ordered dictionary**?
- a **JSON value**?

Problem: Composition

```
test_acct = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

Is this

- a **record**?
- an **object**?
- a **dynamic dictionary**?
- an **ordered dictionary**?
- a **JSON value**?
- a **set**?

Solution: Type-Directed Disambiguation

```
test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

Is this

- a **record**?
- an **object**?
- a **dynamic dictionary**?
- an **ordered dictionary**?
- a **JSON value**?

Solution: Type-Directed Disambiguation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

Is this

- a **record**?
- an **object**?
- a **dynamic dictionary**?
- an **ordered dictionary**?
- a **JSON value**?

Solution: Type-Directed Disambiguation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

This is a **record**.

Solution: Type-Directed Disambiguation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

This is a **dynamic dictionary**.

Solution: Type-Directed Disambiguation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

This is a **dynamic dictionary**.

(Similar to Omar et al., ECOOP 2014)

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

What is a **record**?

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

Every type is of the form `fragment[idx]`.
(If `[idx]` is omitted, assumed `[()]`)

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

The language applies `fragment.init_idx` to validate the type index.

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

The language applies `fragment.ana_Dict` to perform type analysis of dictionary literal forms.

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

The language then applies `fragment.trans_Dict` to compute the translation of the dictionary literal.

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}

test_acct.name
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx

  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

For targeted forms, the language first recursively synthesizes a type for the target...

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}

test_acct.name
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

For targeted forms, the language first recursively synthesizes a type for the target…
and delegates to the fragment defining that type.

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}

test_acct.name
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …

  def syn_Attribute(ctx, e, idx):
    # … type synthesis …

  def trans_Attribute(ctx, e, idx):
    # … translation …
```

Solution: Fragment Delegation

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}

test_acct.name
```

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx


  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …

  def syn_Attribute(ctx, e, idx):
    # … type synthesis …

  def trans_Attribute(ctx, e, idx):
    # … translation …
```

Summary: Fragment Delegation Protocol

Form	Delegates To
Type forms	Explicitly named fragment
Introductory forms (dicts, sets, lists, tuples, lambdas, ...)	Fragment defining type provided for analysis.
Targeted forms (attributes, subscript, call, guarded, ...)	Fragment defining type that target synthesizes.
Binary operations	Precedent fragment (see paper).
Definition forms	Fragment provided as decorator (see paper).
Patterns	Fragment defining type of scrutinee (see paper).

Key idea: both concrete and abstract syntax are fixed. Each form is assigned static and dynamic meaning by a fragment according to this delegation protocol.

Implementation: typy

```
type Account = record[
  name : string,
  account_num : string,
  memo : dyn
]

test_acct : Account = {
  name: “Harry Q. Bovik”,
  account_num: “00-12345678”,
  memo: { }
}

test_acct.name
```

Implementation: typy

```
from typy import component
from typy.std import (record, string, dyn)

@component
def Listing1():
  type Account = record[
    name : string,
    account_num : string,
    memo : dyn
  ]

  test_acct : Account = {
    name: “Harry Q. Bovik”,
    account_num: “00-12345678”,
    memo: { }
  }

  test_acct.name
```

Implementation: typy

```
from typy import component
from typy.std import (record, string, dyn)

@component
def Listing1():
  Account [type] = record[
    name : string,
    account_num : string,
    memo : dyn
  ]

  test_acct : Account = {
    name: “Harry Q. Bovik”,
    account_num: “00-12345678”,
    memo: { }
  }

  test_acct.name
```

Implementation: typy

```
from typy import component
from typy.std import (record, string, dyn)

@component
def Listing1():
  Account [type] = record[
    name : string,
    account_num : string,
    memo : dyn
  ]

  test_acct [: Account] = {
    name: “Harry Q. Bovik”,
    account_num: “00-12345678”,
    memo: { }
  }

  test_acct.name
```

Implementation: typy

```
from typy import component
from typy.std import (record, string, dyn)

@component
def Listing1():
  Account [type] = record[
    name : string,
    account_num : string,
    memo : dyn
  ]

  test_acct [: Account] = {
    name: “Harry Q. Bovik”,
    account_num: “00-12345678”,
    memo: { }
  }

  test_acct.name
```

Implementation: typy

```
from typy import component
from typy.std import (record, string, dyn)

@component
def Listing1():
  Account [type] = record[
    name : string,
    account_num : string,
    memo : dyn
  ]

  test_acct [: Account] = {
    name: “Harry Q. Bovik”,
    account_num: “00-12345678”,
    memo: { }
  }

  test_acct.name
```

Implementation: typy

```
fragment record:
  def init_idx(idx_ast):
    # … type formation logic …
    return idx

  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

Implementation: typy

```
class record(typy.Fragment):
  def init_idx(idx_ast):
    # … type formation logic …
    return idx

  def ana_Dict(ctx, e, idx):
    # … type analysis …

  def trans_Dict(ctx, e, idx):
    # … translation …
```

Implementation: typy

```
class record(typy.Fragment):
  @classmethod
  def init_idx(cls, idx_ast):
    # … type formation logic (see paper) …
    return idx

  @classmethod
  def ana_Dict(cls, ctx, e, idx):
    # … type analysis (see paper) …

  @classmethod
  def trans_Dict(cls, ctx, e, idx):
    # … translation (see paper) …
```

Advanced Example: OpenCL FFI

```
add5 = pyopencl.Program(cl_ctx, ‘
  __kernel void add5(__global double* x) {
    size_t gid = get_global_id(0);
    x[gid] = x[gid] + 5;
  }’).build()
```

Advanced Example: OpenCL FFI

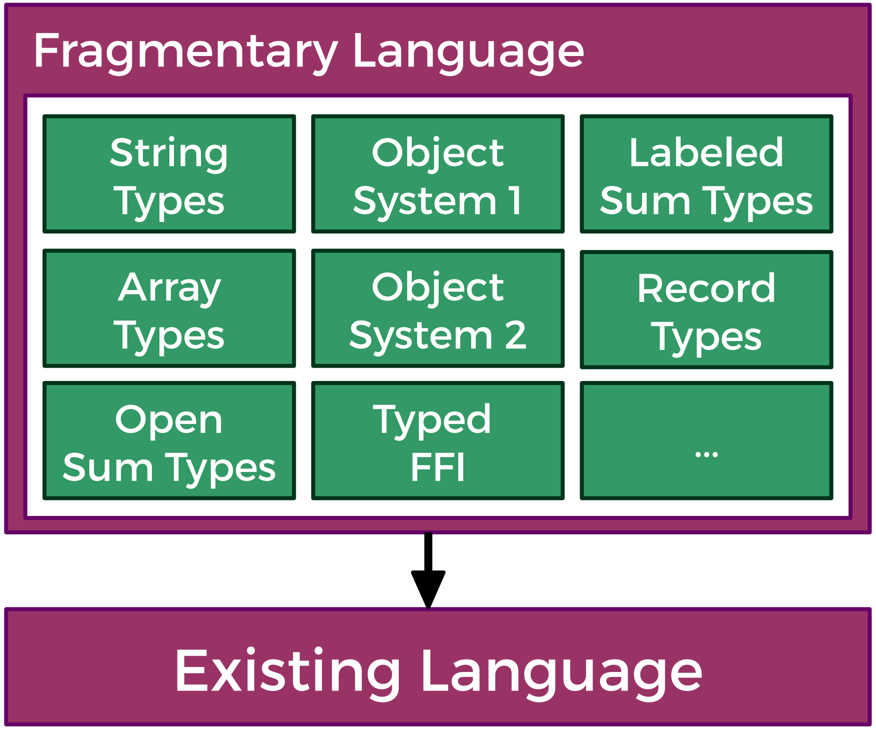
```
from typy import component
from typy.numpy import array, f64
from typy.cl import buffer, to_device, kernel

@component
def Example():
  x [: array[f64]] = [1, 2, 3, 4]
  d_x = to_device(x)

  @kernel
  def add5(x : buffer[f64]):
    gid = get_global_id(0)
    x[gid] = x[gid] + 5

  add5(d_x, global_size=d_x.length)
```

The Idea



Fragmentary Language

String Types

Object System 1

Labeled Sum Types

Array Types

Object System 2

Record Types

Open Sum Types

Typed FFI

...

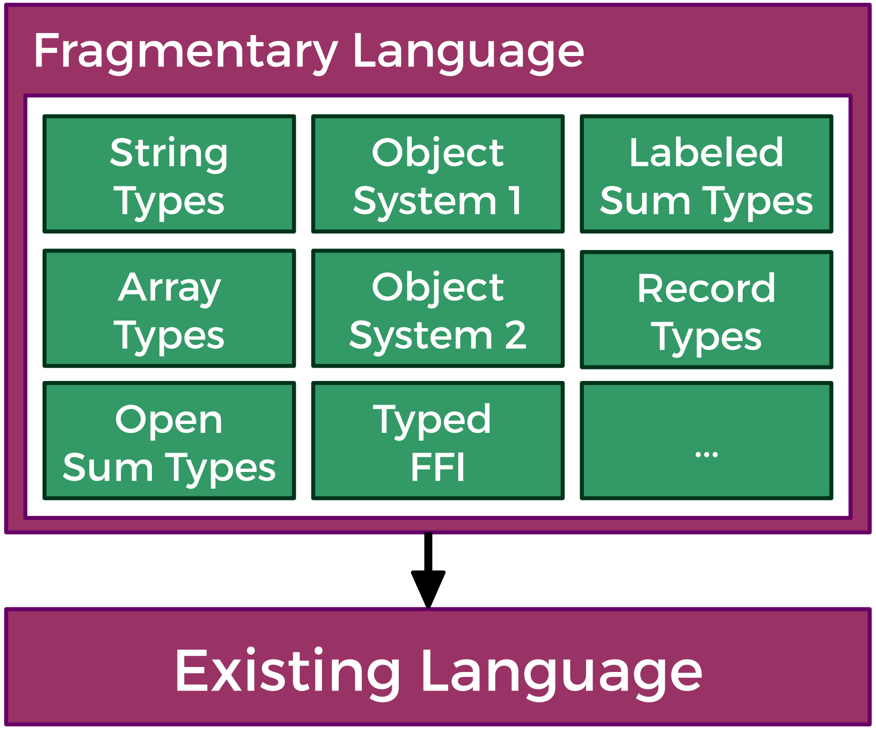
Existing Language

The Idea

Form	Delegates To
Type forms	Explicitly named fragment.
Introductory forms (dicts, sets, lists, tuples, lambdas, ...)	Fragment defining type provided for analysis.
Targeted forms (attributes, subscript, call, guarded, ...)	Fragment defining type that target synthesizes.
Binary operations	Precedent fragment (see paper).
Definition forms	Fragment provided as decorator (see paper).
Patterns	Fragment defining type of scrutinee (see paper).

Key idea: both concrete and abstract syntax are fixed. Each form is assigned static and dynamic meaning by a fragment according to this delegation protocol.

The Idea



Fragmentary Language

String Types

Object System 1

Labeled Sum Types

Array Types

Object System 2

Record Types

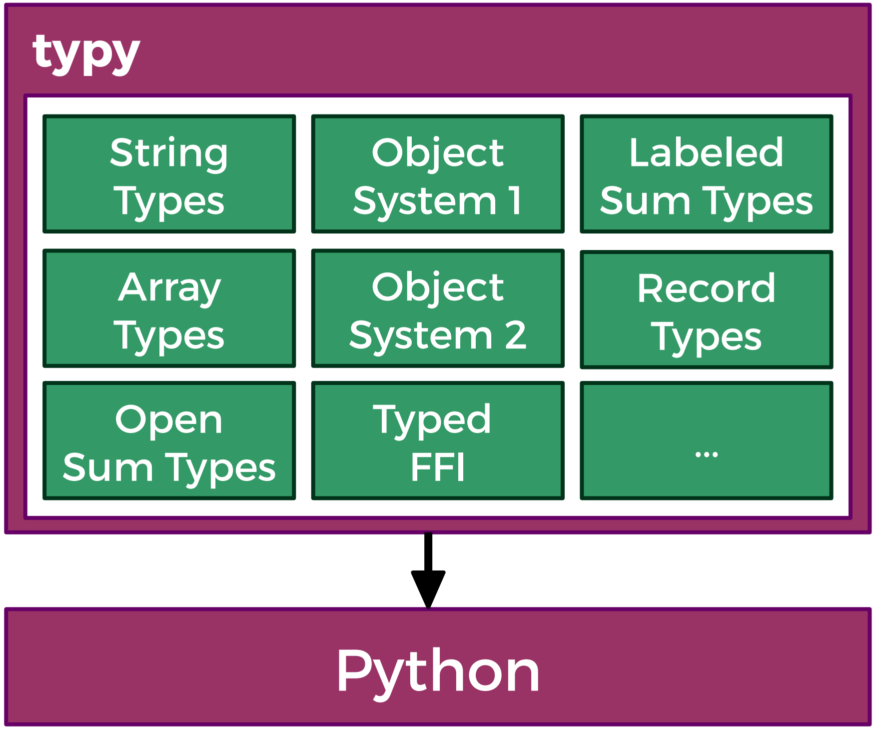
Open Sum Types

Typed FFI

...

Existing Language

The Implementation



typy

String Types

Object System 1

Labeled Sum Types

Array Types

Object System 2

Record Types

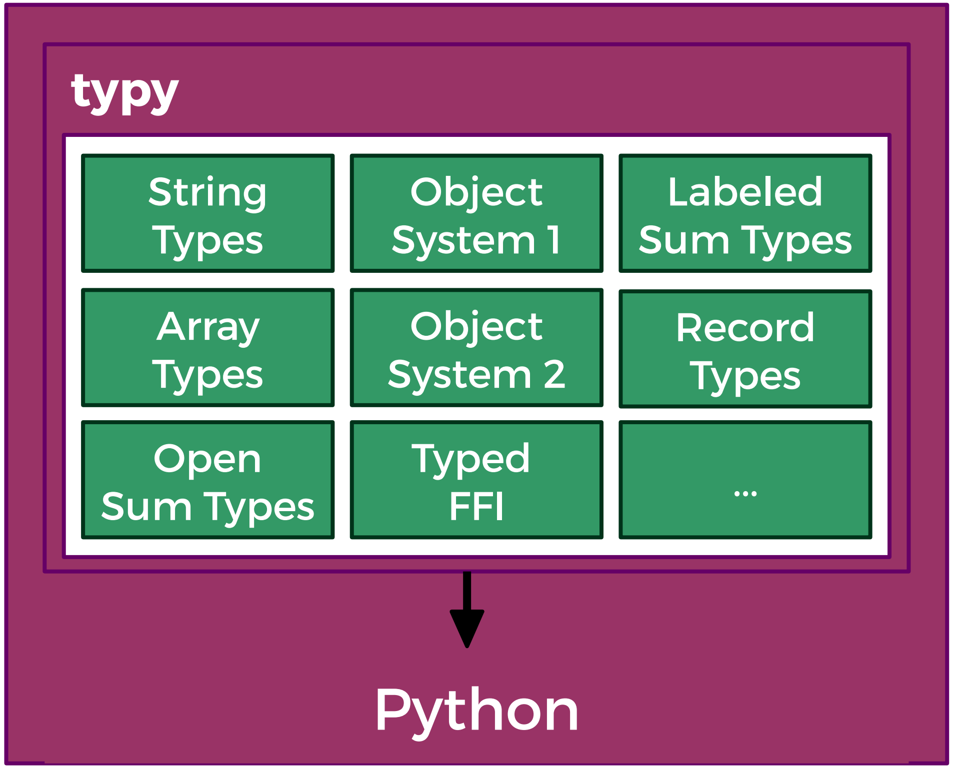
Open Sum Types

Typed FFI

...

Python

The Implementation



Related Work

- **Extensible compilers and language workbenches**
  - No expression problem because syntax is fixed (+ some free tool support)
  - Determinism
  - Stability

- ***Metatheory a la carte*** *[Delaware et al]*
  - **typy** fragments *implement* the statics and dynamics
  - Future: extract typy from a specification in a proof assistant

- **Refinement types / pluggable type systems** (e.g. mypy)
  - These operate over a fixed dynamic semantics
  - **typy** fragments control both the static and dynamic semantics
  - Could layer on a fragmentary refinement system

- **Macro systems** (e.g. Racket, Scala)
  - **typy** fragments manipulate terms but also types
  - Delegation protocol is type-directed, not based on explicit invocation

Limitations & Future Work

- **Type theoretic foundations**
  - Started working on this, more interesting modular reasoning principles…

- **Typed internal language**
  - Use techniques from TIL compiler for ML

- **Hygiene**
  - Need a more disciplined binding structure in IL

- **More implementation work!**

http://github.com/cyrus-/typy

Conclusion

- Semantic fragments support **semantic extensions over a fixed concrete and abstract syntax**.

- Simple!

- You should consider organizing your next statically typed language (or, at least, its compiler) in this way.

- **typy** is a practical implementation of this concept that uses Python:
  - Python's syntax, unchanged
  - Python as the target language
  - Embedded into Python