Protocols and Associated Types in Swift

Slava Pestov

Apple

2023

What is Swift?

- Safe, expressive, systems language
- Open source: `swift.org`
- Apple, Linux, Windows
- C/Objective-C/C++ interop
- **Separately-compiled generic code**

What is Swift?

- Safe, expressive, systems language
- Open source: `swift.org`
- Apple, Linux, Windows
- C/Objective-C/C++ interop
- **Separately-compiled generic code**
  - Runtime model
  - **Type checking**

What is Swift?

- Safe, expressive, systems language
- Open source: `swift.org`
- Apple, Linux, Windows
- C/Objective-C/C++ interop
- **Separately-compiled generic code**
  - Runtime model
  - **Type checking**
    - Informal examples
    - Formal model
    - **String rewriting**

Functions

```
func identity(x: Int) -> Int {
  return x
}

func identity(x: String) -> String {
  return x
}
```

Generic Functions

```
func identity<T>(x: T) -> T {
  return x
}
```

Value Types

```
struct PairOfInts {
  var first: Int
  var second: Int
}

enum IntOrPairOfInts {
  case first(Int)
  case second(PairOfInts)
}
```

Generic Value Types

```
struct Pair<T> {
  var first: T
  var second: T
}

enum Either<U, V> {
  case first(U)
  case second(V)
}
```

Protocols

```
protocol StreamOfInts {
  mutating func next() -> Int
}
```

Protocols with Associated Types

```
protocol Stream {
  associatedtype Element
  mutating func next() -> Self.Element
}
```

Conformance

```
struct NaturalNumbers: Stream {
  var n = 0

  mutating func next() -> Int {
    n += 1
    return n
  }
}
```

Conformance Requirements

```
<S> ... where S: Stream
```

Conformance Requirements

```
func firstTwo<S>(_ s: inout S) -> Pair<S.Element>
    where S: Stream {
  return Pair(first: s.next(), second: s.next())
}
```

`S.Element`

Same-Type Requirements

```
protocol Equatable {
  static func ==(lhs: Self, rhs: Self)
}

func firstTwoEqual<S1, S2>(_ s1: inout S1,
                           _ s2: inout S2) -> Bool
    where S1: Stream,
          S2: Stream,
          S1.Element: Equatable,
          S1.Element == S2.Element {
  return s1.next() == s2.next()
}
```

`S1.Element == S2.Element`

Generic Signatures

A *generic signature* G is a list of root type parameters and requirements.

Requirements

- Conformance: [`T: P`] for protocol `P`
- Same-type: [`T == U`]

(`T` and `U` are type parameters)

Type parameters

Recursively:

- Root: τ_i for $0 \leq i < n$
- Member: `U.A` for type parameter `U` and associated type name `A`

Associated Requirements

(Warning: not the real `Collection`!)

```
protocol Collection {
  associatedtype Element




  var count: Int { get }
  subscript(index: Int) -> Self.Element

}
```

Associated Requirements

(Warning: not the real `Collection`!)

```
protocol Collection {
  associatedtype Element
  associatedtype Slice




  var count: Int { get }
  subscript(index: Int) -> Self.Element
  subscript(range: Range<Int>) -> Self.Slice
}
```

Associated Requirements

(Warning: not the real `Collection`!)

```
protocol Collection {
  associatedtype Element
  associatedtype Slice
    where Self.Slice: Collection


  var count: Int { get }
  subscript(index: Int) -> Self.Element
  subscript(range: Range<Int>) -> Self.Slice
}
```

Associated Requirements

(Warning: not the real `Collection`!)

```
protocol Collection {
  associatedtype Element
  associatedtype Slice
    where Self.Slice: Collection,
          Self.Element == Self.Slice.Element

  var count: Int { get }
  subscript(index: Int) -> Self.Element
  subscript(range: Range<Int>) -> Self.Slice
}
```

Requirement Signatures

The *requirement signature* of protocol `P` is a list of associated type names and associated requirements.

Associated requirements

- Conformance: $[\texttt{Self.U: Q}]_{\texttt{P}}$ for protocol `Q`
- Same-type: $[\texttt{Self.U == Self.V}]_{\texttt{P}}$

(`Self.U` and `Self.V` are relative type parameters)

Relative type parameters

Recursively:

- Root: `Self`
- Member: `Self.V.A` for relative type parameter `Self.V` and associated type name `A`

Protocol Generic Signature

The generic signature of a protocol `P` is denoted $G_{\texttt{P}}$:

- τ_0
- $[\tau_0\texttt{: P}]$

Relative type parameters of `P` are type parameters of $G_{\texttt{P}}$:

$$\texttt{Self.U} \Leftrightarrow \tau_0\texttt{.U}$$

$G_{\texttt{P}}$ is the "simplest non-trivial" generic signature!

Binary Search Example

```
// Find index of ‘e: E’ within ‘c: C’.
func binarySearch<C, E>(_ c: C, _ e: E) -> Int
    where C: Collection,
          E: Comparable,
          E == C.Element {









}
```

Binary Search Example

Generic signature of `binarySearch()`:

$$G := \tau_0, \tau_1, [\tau_0\texttt{: Collection}], [\tau_1\texttt{: Comparable}], [\tau_1 \texttt{ == } \tau_0\texttt{.Element}]$$

Requirement signature of `Collection`:

- `Element`
- `Slice`
- $[\texttt{Self.Slice: Collection}]_{\texttt{Collection}}$
- $[\texttt{Self.Element == Self.Slice.Element}]_{\texttt{Collection}}$

Binary Search Example

```
// Find index of ‘e: E’ within ‘c: C’.
func binarySearch<C, E>(_ c: C, _ e: E) -> Int
    where C: Collection,
          E: Comparable,
          E == C.Element {
  if c.count == 0 { return 0 }
  let mid = c.count / 2
  if c[mid] == e {
    return mid
  } else if c[mid] < e {
    return binarySearch(c[0 ..< mid], e)
  } else {
    return mid + binarySearch(c[mid ..< c.count], e)
  )
}
```

Binary Search Example

Recursive call:

- `binarySearch(c[0 ..< mid], e)`

Binary Search Example

Recursive call:

- `binarySearch(c[0 ..< mid], e)`
- `c[0 ..< mid]` returns a τ_0`.Slice`

Binary Search Example

Recursive call:

- `binarySearch(c[0 ..< mid], e)`
- `c[0 ..< mid]` returns a τ_0`.Slice`
- Substitution: $\{\tau_0 := \tau_0.\texttt{Slice}, \tau_1 := \tau_1\}$

Binary Search Example

Recursive call:

- `binarySearch(c[0 ..< mid], e)`
- `c[0 ..< mid]` returns a τ_0`.Slice`
- Substitution: $\{\tau_0 := \tau_0.\texttt{Slice}, \tau_1 := \tau_1\}$
- Is the `where` clause satisfied?

Binary Search Example

Recursive call:

- `binarySearch(c[0 ..< mid], e)`
- `c[0 ..< mid]` returns a τ_0`.Slice`
- Substitution: $\{\tau_0 := \tau_0\texttt{.Slice}, \tau_1 := \tau_1\}$
- Is the `where` clause satisfied?
  - $[\tau_1\texttt{: Comparable}]$: trivial
  - $[\tau_0\texttt{: Collection}] \Rightarrow [\tau_0\texttt{.Slice: Collection}]$
  - $[\tau_1\ \texttt{==}\ \tau_0\texttt{.Element}] \Rightarrow [\tau_1\ \texttt{==}\ \tau_0\texttt{.Slice.Element}]$

Are these "consequences" of G?

$$[\tau_0\texttt{.Slice: Collection}]$$

$$[\tau_1 \texttt{ == } \tau_0\texttt{.Slice.Element}]$$

Are these “consequences” of G?

$$[\tau_0\texttt{.Slice: Collection}]$$

$$[\tau_1 \texttt{ == } \tau_0\texttt{.Slice.Element}]$$

What about τ_0`.Slice.Element` itself?

Theory of a Generic Signature

G defines a *theory* of *valid type parameters* and *derived requirements*:

- Elementary derivation steps: $\vdash E_i$
- Inference rules: $D_1, \ldots, D_n \vdash D$

Definition

A *derivation* $G \vDash D$ is a (well-formed) sequence of derivation steps.

Elementary Derivation Steps

- For each root type parameter τ_i of G:

$$\vdash \tau_i \qquad (\textsc{Root})$$

- For each explicit conformance requirement $[\texttt{T: P}]$ of G:

$$\vdash [\texttt{T: P}] \qquad (\textsc{Conf})$$

- For each explicit same-type requirement $[\texttt{T == U}]$ of G:

$$\vdash [\texttt{T == U}] \qquad (\textsc{Same})$$

requirement signature of `P`

$+$

[`T: P`]

$=$

more steps

Inference Rules

For each $G \vDash [\texttt{T: P}]$:

- For each associated type `A` of `P`:

  $[\texttt{T: P}] \vdash \texttt{T.A}$ (Assoc)

- For each $[\texttt{Self.U: Q}]_{\texttt{P}}$:

  $[\texttt{T: P}] \vdash [\texttt{T.U: Q}]$ (AssocConf)

- For each $[\texttt{Self.U == Self.V}]_{\texttt{P}}$ of `P`:

  $[\texttt{T: P}] \vdash [\texttt{T.U == T.V}]$ (AssocSame)

Formal Substitution

Replacement of `Self`:

$$\texttt{T.U} := \texttt{Self.U}/\{\texttt{Self} := \texttt{T}\}$$

Concatenation:

$$\texttt{T.U} := \texttt{T} + \texttt{Self.U}$$

Equivalence relation:

$$G \vDash [\texttt{T == U}]$$

Inference Rules

- For each $G \vDash \texttt{T}$:

$$\texttt{T} \vdash [\texttt{T == T}] \qquad (\textsc{Refl})$$

- For each $G \vDash [\texttt{T == U}]$:

$$[\texttt{T == U}] \vdash [\texttt{U == T}] \qquad (\textsc{Swap})$$

- For each $G \vDash [\texttt{T == U}]$ and $G \vDash [\texttt{U == V}]$:

$$[\texttt{T == U}], [\texttt{U == V}] \vdash [\texttt{T == V}] \qquad (\textsc{Trans})$$

Inference Rules

- For each $G \vDash$ [`U: P`] and $G \vDash$ [`T == U`]:

  [`U: P`], [`T == U`] $\vdash$ [`T: P`] (Equiv)

- For each $G \vDash$ [`T: P`], $G \vDash$ [`T == U`], and each associated type `A` of `P`:

  [`T: P`], [`T == U`] $\vdash$ [`T.A == U.A`] (Member)

Derived Conformance Example

Recall $G \vDash [\tau_0$`.Slice: Collection`$]$ from `binarySearch()`:

Derived Conformance Example

Recall $G \vDash [\tau_0\texttt{.Slice: Collection}]$ from `binarySearch()`:

$\vdash [\tau_0\texttt{: Collection}]$ (Conf)

$[\tau_0\texttt{: Collection}] \vdash [\tau_0\texttt{.Slice: Collection}]$ (AssocConf)

Derived Conformance Example

Recall $G \vDash [\tau_0\texttt{.Slice: Collection}]$ from `binarySearch()`:

$\vdash [\tau_0\texttt{: Collection}]$ (Conf)

$[\tau_0\texttt{: Collection}] \vdash [\tau_0\texttt{.Slice: Collection}]$ (AssocConf)

In fact,

$[\tau_0\texttt{.Slice}^n\texttt{: Collection}] \vdash [\tau_0\texttt{.Slice}^{n+1}\texttt{: Collection}]$

(AssocConf)

Recursive Conformance

$[\texttt{Self.Slice: Collection}]_{\texttt{Collection}}$ generates an infinite family of conformance requirements.

Recursive Conformance

$[\texttt{Self.Slice: Collection}]_{\texttt{Collection}}$ generates an infinite family of conformance requirements.

Definition

$[\texttt{Self.U: Q}]_{\texttt{P}}$ is *recursive* if $G_{\texttt{Q}}$ uses `P`.

Recursive Conformance

$[\texttt{Self.Slice: Collection}]_{\texttt{Collection}}$ generates an infinite family of conformance requirements.

Definition

$[\texttt{Self.U: Q}]_{\texttt{P}}$ is *recursive* if $G_{\texttt{Q}}$ uses `P`.

We just proved:

Theorem

If G uses a protocol with a recursive associated conformance requirement, then G generates an infinite theory.

Derived Same-Type Example

Recall $G \vDash [\tau_1$ `==` τ_0`.Slice.Element`$]$ from `binarySearch()`:

$\vdash [\tau_1$ `==` τ_0`.Element`$]$ (Same)

$[\tau_0$`: Collection`$]$

$\vdash [\tau_0$`.Element == `τ_0`.Slice.Element`$]$ (AssocSame)

$[\tau_1$ `==` τ_0`.Element`$], [\tau_0$`.Element == `τ_0`.Slice.Element`$]$

$\vdash [\tau_1$ `==` τ_0`.Slice.Element`$]$ (Trans)

Derived Same-Type Example

Recall $G \vDash [\tau_1 \texttt{ == } \tau_0\texttt{.Slice.Element}]$ from `binarySearch()`:

$$\vdash [\tau_1 \texttt{ == } \tau_0\texttt{.Element}] \qquad (\textsc{Same})$$

$$[\tau_0\texttt{: Collection}] \vdash [\tau_0\texttt{.Element == }\tau_0\texttt{.Slice.Element}] \qquad (\textsc{AssocSame})$$

$$[\tau_1 \texttt{ == } \tau_0\texttt{.Element}], [\tau_0\texttt{.Element == }\tau_0\texttt{.Slice.Element}] \vdash [\tau_1 \texttt{ == } \tau_0\texttt{.Slice.Element}] \qquad (\textsc{Trans})$$

In fact,

$$[\tau_0\texttt{.Slice}^n\texttt{: Collection}] \vdash [\tau_0\texttt{.Slice}^n\texttt{.Element == }\tau_0\texttt{.Slice}^{n+1}\texttt{.Element}] \qquad (\textsc{AssocSame})$$

$$[\tau_1 \texttt{ == } \tau_0\texttt{.Element}], [\tau_0\texttt{.Slice}^n\texttt{.Element == }\tau_0\texttt{.Slice}^{n+1}\texttt{.Element}] \vdash [\tau_1 \texttt{ == } \tau_0\texttt{.Slice}^{n+1}\texttt{.Element}] \qquad (\textsc{Trans})$$

Equivalence Classes

Generic signature G of `binarySearch()`:

- $\{\tau_0\}$
- $\{\tau_1, \tau_0\texttt{.Element}, \ldots, \tau_0\texttt{.Slice}^n\texttt{.Element}, \ldots\}$
- $\{\tau_0\texttt{.Slice}\}$
- $\ldots$
- $\{\tau_0\texttt{.Slice}^n\}$
- $\ldots$

Decision Problems

Two fundamental problems:

Problem 1: Conformance

- Instance: Generic signature G, type parameter `T`, protocol `P`.
- Question: Is $G \vDash$ [`T: P`]?

Problem 2: Equivalence

- Instance: Generic signature G, type parameters `T` and `U`.
- Question: Is $G \vDash$ [`T == U`]?

generic signature

$$\Downarrow$$

string rewrite system

The Alphabet

Definition

Finite alphabet A for generic signature G:

- τ_i: for each root type parameter of G.
- `A`: for each unique associated type name `A`.
- [`P`]: for each protocol `P` used by G.

Random example:

```
protocol Chicken { associatedtype Egg }
protocol Egg { associatedtype Egg }
```

$A := \{\tau_0, \tau_1, \tau_2, [\texttt{Chicken}], [\texttt{Egg}], \texttt{Egg}\}$

Free Monoids

Definition

For any set A:

- A^* is the *free monoid* generated by A.
- $t \in A^*$ is a finite string, called a *term*.
- $x \cdot y$ is concatenation of x and y.
- ε is the empty term.

Free Monoids

Definition

For any set A:

- A^* is the *free monoid* generated by A.
- $t \in A^*$ is a finite string, called a *term*.
- $x \cdot y$ is concatenation of x and y.
- ε is the empty term.

Definitions

- $|t| \in \mathbb{N}$ is the *length* of t.
- If $t = x \cdot y \cdot z$, then y is a *subterm* of t.
- If $|y| < |t|$, then y is a *proper* subterm.

Terms

Definition of φ

Type parameter `T` maps to a term $\varphi(\mathtt{T}) \in A^*$:

$$\varphi(\tau_i\,.\,\mathtt{A}_1\,.\,.\,.\,\mathtt{A}_n) := \tau_i \cdot \mathtt{A}_1 \cdots \mathtt{A}_n$$

Terms

Definition of φ

Type parameter `T` maps to a term $\varphi(\mathtt{T}) \in A^*$:

$$\varphi(\tau_i\,.\,\mathtt{A}_1\,.\,.\,.\,\mathtt{A}_n) := \tau_i \cdot \mathtt{A}_1 \cdots \mathtt{A}_n$$

Definition of $\varphi_{\mathtt{P}}$

Relative type parameter `Self.U` maps to a term $\varphi_{\mathtt{P}}(\mathtt{Self.U}) \in A^*$:

$$\varphi_{\mathtt{P}}(\mathtt{Self}\,.\,\mathtt{A}_1\,.\,.\,.\,\mathtt{A}_n) := [\mathtt{P}] \cdot \mathtt{A}_1 \cdots \mathtt{A}_n$$

String Rewrite Systems

$$\langle A;\ R\rangle$$

- $A := \{a_1, \ldots, a_n\}$: finite alphabet of symbols
- $R := \{(u_1 \sim v_1), \ldots, (u_m \sim v_m)\}$: finite set of *rewrite rules*
- $\sim$: the *term equivalence relation* on A^* generated by R.

Term Equivalence Relation

R defines an equivalence relation $\sim$ on A^*:

Term Equivalence Relation

R defines an equivalence relation $\sim$ on A^*:

- If $(u \sim v) \in R$, then $u \sim v$.

Term Equivalence Relation

R defines an equivalence relation $\sim$ on A^*:

- If $(u \sim v) \in R$, then $u \sim v$.
- If $x \sim y$ and $w, z \in A^*$, then $w \cdot x \cdot z \sim w \cdot y \cdot z$.

Term Equivalence Relation

R defines an equivalence relation $\sim$ on A^*:

- If $(u \sim v) \in R$, then $u \sim v$.
- If $x \sim y$ and $w, z \in A^*$, then $w \cdot x \cdot z \sim w \cdot y \cdot z$.
- If $x \in A^*$, then $x \sim x$.
- If $x \sim y$, then $y \sim x$.
- If $x \sim y$ and $y \sim z$, then $x \sim z$.

Term Equivalence Relation

R defines an equivalence relation $\sim$ on A^*:

- If $(u \sim v) \in R$, then $u \sim v$.
- If $x \sim y$ and $w, z \in A^*$, then $w \cdot x \cdot z \sim w \cdot y \cdot z$.
- If $x \in A^*$, then $x \sim x$.
- If $x \sim y$, then $y \sim x$.
- If $x \sim y$ and $y \sim z$, then $x \sim z$.

The derivations of this theory are called *rewrite paths*.

Rewrite Rules

Definition of λ

Generic requirement D maps to a rule $\lambda(D) \in A^* \times A^*$:

$$\lambda(\texttt{[T: P]}) := (\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T}))$$
$$\lambda(\texttt{[T == U]}) := (\varphi(\texttt{T}) \sim \varphi(\texttt{U}))$$

Rewrite Rules

Definition of λ

Generic requirement D maps to a rule $\lambda(D) \in A^* \times A^*$:

$$\lambda([\texttt{T: P}]) := (\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T}))$$
$$\lambda([\texttt{T == U}]) := (\varphi(\texttt{T}) \sim \varphi(\texttt{U}))$$

Definition of $\lambda_{\texttt{P}}$

Associated requirement D in protocol `P` maps to a rule $\lambda_{\texttt{P}}(D) \in A^* \times A^*$:

$$\lambda_{\texttt{P}}([\texttt{Self.U: Q}]_{\texttt{P}}) := (\varphi_{\texttt{P}}(\texttt{Self.U}) \cdot [\texttt{Q}] \sim \varphi_{\texttt{P}}(\texttt{Self.U}))$$
$$\lambda_{\texttt{P}}([\texttt{Self.U == Self.V}]_{\texttt{P}}) := (\varphi_{\texttt{P}}(\texttt{Self.U}) \sim \varphi_{\texttt{P}}(\texttt{Self.V}))$$

Example

Rewrite system for generic signature G of `binarySearch()`:

1. $(\tau_0 \cdot [\texttt{Collection}] \sim \tau_0)$
2. $(\tau_0 \cdot \texttt{Element} \sim \tau_1)$
3. $(\tau_1 \cdot [\texttt{Comparable}] \sim \tau_1)$

Example

Rewrite system for generic signature G of `binarySearch()`:

1. $(\tau_0 \cdot [\texttt{Collection}] \sim \tau_0)$
2. $(\tau_0 \cdot \texttt{Element} \sim \tau_1)$
3. $(\tau_1 \cdot [\texttt{Comparable}] \sim \tau_1)$
4. $([\texttt{Collection}] \cdot \texttt{Slice} \cdot [\texttt{Collection}] \sim [\texttt{Collection}] \cdot \texttt{Slice})$
5. $([\texttt{Collection}] \cdot \texttt{Slice} \cdot \texttt{Element} \sim [\texttt{Collection}] \cdot \texttt{Element})$

Main Theorem

Theorem

Assume G is valid, $G \vDash \texttt{T}$, $G \vDash \texttt{U}$. Then,

- $G \vDash [\texttt{T: P}]$ **if and only if** $\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T})$.
- $G \vDash [\texttt{T == U}]$ **if and only if** $\varphi(\texttt{T}) \sim \varphi(\texttt{U})$.

Main Theorem

Theorem

Assume G is valid, $G \vDash \texttt{T}$, $G \vDash \texttt{U}$. Then,

- $G \vDash [\texttt{T: P}]$ **if and only if** $\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T})$.
- $G \vDash [\texttt{T == U}]$ **if and only if** $\varphi(\texttt{T}) \sim \varphi(\texttt{U})$.

($\Rightarrow$) Proof by structural induction on derivations:

- Elementary derivation step: property holds by construction.
- For each inference rule, assume property holds for all assumptions, show it holds for consequence.

Derived Conformance Example

To show $\lambda([\tau_0.\texttt{Slice: Collection}])$:

$$\begin{aligned}\tau_0 \cdot \texttt{Slice} \cdot [\texttt{Collection}] &\sim \tau_0 \cdot [\texttt{Collection}] \cdot \texttt{Slice} \cdot [\texttt{Collection}] \\ &\sim \tau_0 \cdot [\texttt{Collection}] \cdot \texttt{Slice} \\ &\sim \tau_0 \cdot \texttt{Slice}\end{aligned}$$

Term Concatenation

$$
\begin{array}{ccccc}
\texttt{T} & + & \texttt{Self.U} & = & \texttt{T.U} \\
\Downarrow & & \Downarrow & & \Downarrow \\
\varphi(\texttt{T}) & \cdot & \varphi_{\texttt{P}}(\texttt{Self.U}) & \neq & \varphi(\texttt{T.U})
\end{array}
$$

Term Concatenation

$$\begin{array}{ccccc} \texttt{T} & + & \texttt{Self.U} & = & \texttt{T.U} \\ \Downarrow & & \Downarrow & & \Downarrow \\ \varphi(\texttt{T}) & \cdot & \varphi_{\texttt{P}}(\texttt{Self.U}) & \neq & \varphi(\texttt{T.U}) \end{array}$$

But if $\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T})$, then $\varphi(\texttt{T}) \cdot \varphi_{\texttt{P}}(\texttt{Self.U}) \sim \varphi(\texttt{T.U})$:

Term Concatenation

$$\begin{array}{ccccc} \texttt{T} & + & \texttt{Self.U} & = & \texttt{T.U} \\ \Downarrow & & \Downarrow & & \Downarrow \\ \varphi(\texttt{T}) & \cdot & \varphi_{\texttt{P}}(\texttt{Self.U}) & \neq & \varphi(\texttt{T.U}) \end{array}$$

But if $\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T})$, then $\varphi(\texttt{T}) \cdot \varphi_{\texttt{P}}(\texttt{Self.U}) \sim \varphi(\texttt{T.U})$:

$$\underbrace{\varphi(\texttt{T})}_{t} \cdot \underbrace{\varphi_{\texttt{P}}(\texttt{Self.U})}_{[\texttt{P}]\cdot u} \sim \underbrace{\varphi(\texttt{T.U})}_{t\cdot u}$$

Because

$$t \cdot [\texttt{P}] \cdot u \sim t \cdot u$$

Sketch of Proof

$$[\texttt{T: P}] \vdash [\texttt{T.U: Q}] \qquad (\textsc{AssocConf})$$

Assume $\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T})$ and show $\varphi(\texttt{T.U}) \cdot [\texttt{Q}] \sim \varphi(\texttt{T.U})$:

$$\begin{aligned}
&\varphi(\texttt{T.U}) \cdot [\texttt{Q}] \\
&\sim \varphi(\texttt{T}) \cdot \varphi_{\texttt{P}}(\texttt{Self.U}) \cdot [\texttt{Q}] \\
&\sim \varphi(\texttt{T}) \cdot \varphi_{\texttt{P}}(\texttt{Self.U}) \\
&\sim \varphi(\texttt{T.U})
\end{aligned}$$

Derived Same-Type Example

To show $\lambda([\tau_1 \texttt{ == } \tau_0\texttt{.Slice.Element}])$:

$$\begin{aligned}\tau_0 \cdot \texttt{Slice} \cdot \texttt{Element} &\sim \tau_0 \cdot [\texttt{Collection}] \cdot \texttt{Slice} \cdot \texttt{Element} \\ &\sim \tau_0 \cdot [\texttt{Collection}] \cdot \texttt{Element} \\ &\sim \tau_0 \cdot \texttt{Element} \\ &\sim \tau_1\end{aligned}$$

Sketch of Proof

$[\texttt{T: P}] \vdash [\texttt{T.U == T.V}]$ (AssocSame)

Assume $\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{T})$ and show $\varphi(\texttt{T.U}) \sim \varphi(\texttt{T.V})$:

$$\begin{aligned}
&\varphi(\texttt{T.U})\\
&\sim \varphi(\texttt{T}) \cdot \varphi_{\texttt{P}}(\texttt{Self.U})\\
&\sim \varphi(\texttt{T}) \cdot \varphi_{\texttt{P}}(\texttt{Self.V})\\
&\sim \varphi(\texttt{T.V})
\end{aligned}$$

The Ultimate Problem

We've reduced everything to...

The word problem

- Instance: String rewrite system $\langle A; R \rangle$, terms x, $y \in A^*$.
- Question: Is $x \sim y$?

Reduction Relation

R defines a *reduction relation* $\rightarrow$ on A^*:

- If $(u \rightarrow v) \in R$, then $u \rightarrow v$.
- If $x \rightarrow y$ and $w, z \in A^*$, then $w \cdot x \cdot z \rightarrow w \cdot y \cdot z$.
- If $x \in A^*$, then $x \rightarrow x$.
- If $x \rightarrow y$ and $y \rightarrow z$, then $x \rightarrow z$.

(Exactly like $\sim$ but not symmetric!)

Normal Form Algorithm

Algorithm

Outputs the *normal form* $\tilde{t}$ of t:

- If $t = xuz$ for some $(u \rightarrow v) \in R$, set t to xvz and continue.
- Otherwise, output t.

If $t = \tilde{t}$, then t is *irreducible*.

Normal Form Algorithm

Algorithm

Outputs the *normal form* $\tilde{t}$ of t:

- If $t = xuz$ for some $(u \rightarrow v) \in R$, set t to xvz and continue.
- Otherwise, output t.

If $t = \tilde{t}$, then t is *irreducible*.

Properties

- If $\tilde{t}$ exists, then $t \sim \tilde{t}$.
- If $\tilde{x} = \tilde{y}$, then $x \sim y$.

Problems

Existence

$$\langle a; a \rightarrow aa \rangle$$

Normal form of a does not exist:

$$a \rightarrow aa \rightarrow aaa \rightarrow aaaa \rightarrow \cdots$$

Problems

Existence

$$\langle a; a \rightarrow aa \rangle$$

Normal form of a does not exist:

$$a \rightarrow aa \rightarrow aaa \rightarrow aaaa \rightarrow \cdots$$

Uniqueness

$$\langle a, b, c; ab \rightarrow a, bc \rightarrow b \rangle$$

ac and a are irreducible but also $ac \sim a$:

$$abc \rightarrow ab \rightarrow a$$

$$abc \rightarrow ac$$

Termination

Definition

$\rightarrow$ is *Noetherian* if normal form algorithm terminates.

Termination

Definition

$\to$ is *Noetherian* if normal form algorithm terminates.

Definition

A *reduction order* on A^* satisfies:

- If $u > v$, then $x \cdot u \cdot z > x \cdot v \cdot z$
- No infinite descending chains: $x_1 > x_2 > \cdots > x_n > \cdots$

Theorem

If each $(u \to v) \in R$ is *oriented* so that $u < v$, then $\to$ is Noetherian.

Proof: If $t \to t_1 \to t_2 \to \cdots$, then $t > t_1 > t_2 > \cdots$ (property 1).
Therefore $\tilde{t} = t_n$ for some n (property 2).

Confluence

Definition

$\rightarrow$ is *confluent* whenever $x \rightarrow y$ and $x \rightarrow z$, there exists w such that $y \rightarrow w$ and $z \rightarrow w$.

Example

$$\langle a, b, c; ab \rightarrow a, bc \rightarrow b, ac \rightarrow a \rangle$$

Adding $(ac \rightarrow a)$ does not change $\sim$ but makes $\rightarrow$ confluent. This is called *completion*.

Convergence

Definition

$\langle A;\, R\rangle$ is a *convergent rewrite system* if $\rightarrow$ is Noetherian and confluent.

Church-Rosser Theorem

Assume $\langle A;\, R\rangle$ convergent. Then $x \sim y$ **if and only if** $\tilde{x} = \tilde{y}$.

Completion

Definition

If $t \to t_1$ and $t \to t_2$, (t_1, t_2) is a *critical pair*.

Completion

Definition

If $t \to t_1$ and $t \to t_2$, (t_1, t_2) is a *critical pair*.

Newman's Lemma

Assume $\to$ is Noetherian. Left-hand sides of rewrite rules generate a finite set of critical pairs.

Knuth-Bendix Completion 1/4

Algorithm

Resolving critical pair (t_1, t_2):

- $t_1 \to \tilde{t_1}$, $t_2 \to \tilde{t_2}$.
- If $\tilde{t_1} = \tilde{t_2}$: critical pair is *trivial*.
- If $\tilde{t_1} > \tilde{t_2}$: add $(\tilde{t_1} \to \tilde{t_2})$ to R.
- If $\tilde{t_1} < \tilde{t_2}$: add $(\tilde{t_2} \to \tilde{t_1})$ to R.

Knuth-Bendix Completion 2/4

$$u \cdot v \cdot w$$

$$v$$

Overlap of the first kind

If $(u \cdot v \cdot w \rightarrow x)$, $(v \rightarrow y) \in R$,

- Delete $(u \cdot v \cdot w \rightarrow x)$.
- Resolve $(u \cdot y \cdot w, x)$.

Knuth-Bendix Completion 3/4

$$u \cdot v$$

$$\quad\quad v \cdot w$$

Overlap of the second kind

If $(u \cdot v \rightarrow x)$, $(v \cdot w \rightarrow y) \in R$,

- Resolve $(x \cdot w, u \cdot y)$.

Knuth-Bendix Completion 4/4

Reduction

For all $(u \to v) \in R$ where $v \neq \tilde{v}$,

- Replace $(u \to v)$ with $(u \to \tilde{v})$.

Knuth-Bendix completion might not terminate!

Knuth-Bendix completion might not terminate!

Question

What just defined the *basic lowering*. Does the basic lowering accept *all* generic signatures?

Finite Cross-Section

G has a *finite cross-section* if set of equivalence classes is finite.

$$\text{finite theory} \subsetneq \text{finite cross-section}$$

Finite Cross-Section

G has a *finite cross-section* if set of equivalence classes is finite.

$$\text{finite theory} \subsetneq \text{finite cross-section}$$

```
protocol Z2 {
  associatedtype A where Self.A: Z2, Self.A.A == Self
}
```

$G_{\mathtt{Z2}}$ has an infinite theory: τ_0, τ_0.A, τ_0.A.A, τ_0.A.A.A, ...

Finite Cross-Section

G has a *finite cross-section* if set of equivalence classes is finite.

$$\text{finite theory} \subsetneq \text{finite cross-section}$$

```
protocol Z2 {
  associatedtype A where Self.A: Z2, Self.A.A == Self
}
```

$G_{\texttt{Z2}}$ has an infinite theory: τ_0, $\tau_0\texttt{.A}$, $\tau_0\texttt{.A.A}$, $\tau_0\texttt{.A.A.A}$, ...

$G_{\texttt{Z2}}$ has a finite cross-section:

Representative:	General form:
τ_0	$\tau_0\texttt{.A}^{2n}$
$\tau_0\texttt{.A}$	$\tau_0\texttt{.A}^{2n+1}$

The Basic Lowering

Theorem

$\langle A;\, R\rangle$ is basic lowering of G. Completion terminates **if and only if**:

- G has a finite cross-section.
- For each protocol `P` used by G, $G_{\texttt{P}}$ has a finite cross-section.

Infinite Example

$G_{\texttt{Collection}}$ does not have a finite cross-section. Let's remove everything but the recursive "slice" type:

```
protocol N {
  associatedtype A: N
}
```

Rewrite rules:

- $([\texttt{N}] \cdot \texttt{A} \cdot [\texttt{N}] \rightarrow [\texttt{N}] \cdot \texttt{A})$
- $(\tau_0 \cdot [\texttt{N}] \rightarrow \tau_0)$

Overlapping Rules

$$\tau_0 \cdot [\mathtt{N}]$$

$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$

Overlapping Rules

$$\tau_0 \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$

$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot \mathtt{A} \cdot [\mathtt{N}]$$
$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow \tau_0 \cdot \mathtt{A}$$

Overlapping Rules

$$\tau_0 \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$

$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot \mathtt{A} \cdot [\mathtt{N}]$$
$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow \tau_0 \cdot \mathtt{A}$$

$$(\tau_0 \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot \mathtt{A})$$

Overlapping Rules

$$\tau_0 \cdot \mathtt{A} \cdot [\mathtt{N}]$$

$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$

Overlapping Rules

$$\tau_0 \cdot \mathtt{A} \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$

$$(\tau_0 \cdot [\mathtt{N}] \rightarrow \tau_0)$$
$$(\tau_0 \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot \mathtt{A})$$
$$(\tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot \mathtt{A} \cdot \mathtt{A})$$
$$(\tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow \tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A})$$
$$\ldots$$

Full Lowering

$$[\mathtt{P}] \cdot \mathtt{A} \rightarrow [\mathtt{P}]\mathtt{A}$$

Associated Type Symbols

Definition

For each associated type `A` of each protocol `[P]`,

- $A := A \cup \{\texttt{[P]A}\}$
- $R := R \cup \{(\texttt{[P]} \cdot \texttt{A} \rightarrow \texttt{[P]A})\}$

Reduction order: $\texttt{[P]} < \texttt{[P]A} < \texttt{A} < \tau_i$

Theorem

Adding a new symbol a and rewrite rule $(t \rightarrow a)$ does not change the term equivalence relation $\sim$.

Infinite Example, Revisited

```
protocol N {
  associatedtype A: N
}
```

Rewrite rules:

- $([\mathtt{N}] \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A})$—**new!**
- $([\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow [\mathtt{N}] \cdot \mathtt{A})$
- $(\tau_0 \cdot [\mathtt{N}] \rightarrow \tau_0)$

New Overlapping Rules

$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$

$$[\mathtt{N}] \cdot \mathtt{A}$$

New Overlapping Rules

$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A}$$

$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow [\mathtt{N}] \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A}$$

New Overlapping Rules

$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A}$$

$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A} \cdot [\mathtt{N}] \rightarrow [\mathtt{N}] \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A}$$

$$([\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}] \rightarrow [\mathtt{N}]\mathtt{A})$$

New Overlapping Rules

$$
\begin{array}{ll}
[\texttt{N}]\texttt{A} \cdot [\texttt{N}] & \\
 & [\texttt{N}] \cdot \texttt{A}
\end{array}
$$

New Overlapping Rules

$$[\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]$$

$$[\mathtt{N}] \cdot \mathtt{A}$$

$$[\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A} \cdot \mathtt{A}$$

$$[\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]\mathtt{A}$$

New Overlapping Rules

$$[\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A}$$

$$[\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A} \cdot \mathtt{A}$$
$$[\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]\mathtt{A}$$

$$([\mathtt{N}]\mathtt{A} \cdot \mathtt{A} \rightarrow [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]\mathtt{A})$$

New Overlapping Rules

$$\tau_0 \cdot [\mathtt{N}]$$

$$[\mathtt{N}] \cdot \mathtt{A}$$

New Overlapping Rules

$$\tau_0 \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A}$$

$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow \tau_0 \cdot \mathtt{A}$$
$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow \tau_0 \cdot [\mathtt{N}]\mathtt{A}$$

New Overlapping Rules

$$\tau_0 \cdot [\mathtt{N}]$$
$$[\mathtt{N}] \cdot \mathtt{A}$$

$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow \tau_0 \cdot \mathtt{A}$$
$$\tau_0 \cdot [\mathtt{N}] \cdot \mathtt{A} \rightarrow \tau_0 \cdot [\mathtt{N}]\mathtt{A}$$

$$(\tau_0 \cdot \mathtt{A} \rightarrow \tau_0 \cdot [\mathtt{N}]\mathtt{A})$$

Success

We have a convergent rewrite system!

Conformance rules:

- $([\texttt{N}]\texttt{A} \cdot [\texttt{N}] \rightarrow [\texttt{N}]\texttt{A})$
- $(\tau_0 \cdot [\texttt{N}] \rightarrow \tau_0)$

Name reduction rules:

- $([\texttt{N}] \cdot \texttt{A} \rightarrow [\texttt{N}]\texttt{A})$
- $([\texttt{N}]\texttt{A} \cdot \texttt{A} \rightarrow [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A})$
- $(\tau_0 \cdot \texttt{A} \rightarrow \tau_0 \cdot [\texttt{N}]\texttt{A})$

Normal Form Example

$G_{\mathtt{N}} \vDash [\tau_0\mathtt{.A.A.A\colon\ N}]$. Therefore, $\tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \sim \tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A}$:

Normal Form Example

$G_{\mathtt{N}} \vDash [\tau_0\mathtt{.A.A.A:\ N}]$. Therefore, $\tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \cdot [\mathtt{N}] \sim \tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A}$:

$$\begin{aligned}\tau_0 \cdot \mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} &\rightarrow \tau_0 \cdot [\mathtt{N}]\mathtt{A} \cdot \mathtt{A} \cdot \mathtt{A} \\ &\rightarrow \tau_0 \cdot [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]\mathtt{A} \cdot \mathtt{A} \\ &\rightarrow \tau_0 \cdot [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]\mathtt{A} \cdot [\mathtt{N}]\mathtt{A}\end{aligned}$$

Normal Form Example

$G_{\texttt{N}} \vDash [\tau_0\texttt{.A.A.A: N}]$. Therefore, $\tau_0 \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A} \cdot [\texttt{N}] \sim \tau_0 \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A}$:

$$\begin{aligned}\tau_0 \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A} &\rightarrow \tau_0 \cdot [\texttt{N}]\texttt{A} \cdot \texttt{A} \cdot \texttt{A}\\ &\rightarrow \tau_0 \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A} \cdot \texttt{A}\\ &\rightarrow \tau_0 \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A}\end{aligned}$$

Other side:

$$\begin{aligned}\tau_0 \cdot \texttt{A} \cdot \texttt{A} \cdot \texttt{A} \cdot [\texttt{N}] &\rightarrow \tau_0 \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\\ &\rightarrow \tau_0 \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A} \cdot [\texttt{N}]\texttt{A}\end{aligned}$$

Full Lowering

Full lowering accepts:

- All generic signatures with finite cross-section.
- Some generic signatures with infinite cross-section: $G_{\texttt{N}}$, $G_{\texttt{Collection}}$, many more.

Question

Does the full lowering accept *all* generic signatures?

string rewrite system

$$\Downarrow$$

generic signature

Encoding a Rewrite System

Example:

$$M := \langle a, b, c; ab \sim a, bc \sim b \rangle$$

```
protocol M {



}
```

Encoding a Rewrite System

Example:

$$M := \langle a, b, c; ab \sim a, bc \sim b \rangle$$

```
protocol M {
  associatedtype A
  associatedtype B
  associatedtype C

}
```

Encoding a Rewrite System

Example:

$$M := \langle a, b, c; ab \sim a, bc \sim b \rangle$$

```
protocol M {
  associatedtype A: M
  associatedtype B: M
  associatedtype C: M

}
```

Encoding a Rewrite System

Example:

$$M := \langle a, b, c; ab \sim a, bc \sim b \rangle$$

```
protocol M {
  associatedtype A: M
  associatedtype B: M
  associatedtype C: M
      where A.B == A, B.C == B
}
```

Solving Word Problems

We have $ac \sim a$ but $ca \not\sim b$:

```
func wordProblems<T: M>(_: T.Type) {
  sameType(T.A.C.self, T.A.self)  // okay
  sameType(T.C.A.self, T.B.self)  // type error
}

func sameType<T>(_: T.Type, _: T.Type) {}
```

Understanding the Encoding

$G_{\mathtt{M}}$ encodes the word problem in $\langle A;\, R\rangle$:

- $t \in A^*$ **if and only if** $G_{\mathtt{M}} \vDash \mathtt{T}$.
- $u \sim v$ **if and only if** $G_{\mathtt{M}} \vDash [\mathtt{U\ ==\ V}]$.

Understanding the Encoding

$G_{\mathtt{M}}$ encodes the word problem in $\langle A;\, R\rangle$:

- $t \in A^*$ **if and only if** $G_{\mathtt{M}} \vDash \mathtt{T}$.
- $u \sim v$ **if and only if** $G_{\mathtt{M}} \vDash [\mathtt{U\ ==\ V}]$.

$$\langle A;\, R\rangle$$

$$\Downarrow$$

$$G_{\mathtt{M}}$$

$$\Downarrow$$

$$\langle A', R'\rangle$$

Undecidability

Historical sketch:

- Thue (1914): word problem
- Gödel (1931): incompleteness
- Turing (1936): effective computability
- Post (1945): word problem is undecidable

Undecidability

Historical sketch:

- Thue (1914): word problem
- Gödel (1931): incompleteness
- Turing (1936): effective computability
- Post (1945): word problem is undecidable

Theorem

(Tseitin, 1956) Undecidable if a given term t is equivalent to aaa:

$$\begin{aligned}\langle a, b, c, d, e; ac \sim ca, ad \sim da, bc \sim cb, bd \sim db, eca \sim ce,\\ cdca \sim cdcae, caaa \sim aaa, daaa \sim aaa\rangle\end{aligned}$$

Undecidable Example

```
// error: cannot build rewrite system for protocol;
// rule length limit exceeded

protocol M {
  associatedtype A: M
  associatedtype B: M
  associatedtype C: M
  associatedtype D: M
  associatedtype E: M
      where A.C == C.A, A.D == D.A,
            B.C == C.B, B.D == D.B,
            E.C.A == C.E, C.D.C.A == C.D.C.A.E,
            C.A.A.A == A.A.A, D.A.A.A == A.A.A
}
```

Undecidable problem

- Instance: Type parameter `T`.
- Question: Is $G_{\mathtt{M}} \vDash [\mathtt{T} \;\mathtt{==}\; \tau_0\mathtt{.A.A.A}]$?

Conclusion

- Formal theory of associated requirements.

Conclusion

- Formal theory of associated requirements.
- Generic signature ⇒ string rewrite system.

Conclusion

- Formal theory of associated requirements.
- Generic signature $\Rightarrow$ string rewrite system.
- Derived requirements $\Rightarrow$ word problem.

Conclusion

- Formal theory of associated requirements.
- Generic signature $\Rightarrow$ string rewrite system.
- Derived requirements $\Rightarrow$ word problem.
- Knuth-Bendix solves word problem if successful.

Conclusion

- Formal theory of associated requirements.
- Generic signature $\Rightarrow$ string rewrite system.
- Derived requirements $\Rightarrow$ word problem.
- Knuth-Bendix solves word problem if successful.
- finite theory $\subsetneq$ finite cross-section $\subsetneq$ all generic signatures

Conclusion

- Formal theory of associated requirements.
- Generic signature $\Rightarrow$ string rewrite system.
- Derived requirements $\Rightarrow$ word problem.
- Knuth-Bendix solves word problem if successful.
- finite theory $\subsetneq$ finite cross-section $\subsetneq$ all generic signatures
- Accept *all* finite cross-section, and *some* infinite cross-section.

Conclusion

- Formal theory of associated requirements.
- Generic signature $\Rightarrow$ string rewrite system.
- Derived requirements $\Rightarrow$ word problem.
- Knuth-Bendix solves word problem if successful.
- finite theory $\subsetneq$ finite cross-section $\subsetneq$ all generic signatures
- Accept *all* finite cross-section, and *some* infinite cross-section.
- No decision procedure can accept *all* generic signatures.

Ten Years of Swift Generics

- Separately-compiled generics (2013)
- Standard library collections (2015)
- Recursive conformances (2017)
- Undecidability (2020)
- Requirement Machine (2022)
- Formal model (2023)

Additional Resources

- Reference guide, *Compiling Swift Generics*:
  `download.swift.org/docs/assets/generics.pdf`
- Recorded talk, *Implementing Swift Generics*:
  `www.youtube.com/watch?v=ctS8FzqcRug`

Thank You!

Backup Slides

Calling Convention Supplement

```
func identity<T>(x: T) -> T {
  return x
}
```

Calling convention of `identity()`:

- Pointer to *type metadata* for `T`
- Pointer to argument of type `T`
- Pointer to return buffer for value of type `T`

Calling Convention Supplement

Type metadata:

- **size**
- **alignment**
- **destroy** value when it leaves scope
- **copy** value if use extends lifetime
- **move** value if this is final use

Implementations:

- Trivial types (`Int`, etc): bitwise copy
- Reference types: copy +1, destroy -1
- Generic structs and enums (`Either<U, V>`, etc):
  - Instantiation function
  - Compute size and alignment from `U` and `V`
  - Generic move, copy, destroy operations

Calling Convention Supplement

```
func firstTwo<S>(_ s: inout S) -> Pair<S.Element>
    where S: Stream
```

Calling convention of `firstTwo()`:

- Pointer to type metadata for `S`
- Pointer to *witness table* for [`S: Stream`]

Witness table layout of [`S: Stream`]:

- Type metadata for `Element`
- Concrete implementation of `next()`

Decision Problems Supplement

Problem 3: Name lookup

- Instance: Generic signature G, type parameter `T`.
- Question: All protocols `P` such that $G \vDash$ [`T: P`]?

Type checking "`foo.bar`" where `foo` is a `T`:

- Some `P` where $G \vDash$ [`T: P`] must declare a member named `bar`.
- Reduces to Problem 1.

Decision Problems Supplement

Problem 4: Type parameter validity

- Instance: Generic signature G, type parameter `T`.
- Question: Is $G \vDash$ `T`?

Reduces to Problem 3:

- τ_i validity: immediate.
- `U.A`: valid iff exists `P` declaring `A` such that $G \vDash$ [`U: P`].

Problem 5: Generic signature validity

- Instance: Generic signature G.
- Question: Is G valid?

Generic signature of `binarySearch()`:

$$G := \tau_0, \tau_1, \sout{[\tau_0 \texttt{: Collection}]}, [\tau_1 \texttt{: Comparable}], [\tau_1 \texttt{ == } \tau_0\texttt{.Element}]$$

Now, $G \vDash [\tau_1 \texttt{ == } \tau_0\texttt{.Element}]$ but $G \nvDash \tau_0\texttt{.Element}$!

Valid Generic Signatures Suppment

Valid generic requirements

- $[\mathtt{T:\ P}]$ is valid if $G \vDash \mathtt{T}$.
- $[\mathtt{T\ ==\ U}]$ is valid if $G \vDash \mathtt{T}$ and $G \vDash \mathtt{U}$.

Valid associated requirements

- $[\mathtt{Self.U:\ Q}]_{\mathtt{P}}$ is valid if $G_{\mathtt{P}} \vDash \tau_0\mathtt{.U}$.
- $[\mathtt{Self.U\ ==\ Self.V}]_{\mathtt{P}}$ is valid if $G_{\mathtt{P}} \vDash \tau_0\mathtt{.U}$ and $G_{\mathtt{P}} \vDash \tau_0\mathtt{.V}$.

Theorem

Assume all explicitly-written requirements of G are valid. Then all derived requirements of G are valid.

Validity Proof Supplement

- Base case $\vdash E_i$: E_i is known valid.
- Inductive step $D_1, \ldots, D_n \vdash D$: show D is valid if all D_i are.

Validity Proof Supplement

Interesting cases:

$[\texttt{T: P}] \vdash [\texttt{T.U: Q}]$ (AssocConf)

$[\texttt{T: P}] \vdash [\texttt{T.U == T.V}]$ (AssocSame)

We know $G \vDash \texttt{T}$, must show $G \vDash \texttt{T.U}$ (or $G \vDash \texttt{T.V}$)

Formal substitution on $G_{\texttt{P}} \vDash \tau_0\texttt{.U}$:

- $\vdash \tau_0$ becomes $G \vDash \texttt{T}$
- $\vdash [\tau_0\texttt{: P}]$ becomes $G \vDash [\texttt{T: P}]$
- In all other steps, replace τ_0 with $\texttt{T}$

We get $G \vDash \texttt{T.U}$!

Term Equivalence Proof Supplement

$[\texttt{U: P}], [\texttt{T == U}] \vdash [\texttt{T: P}]$ (Equiv)

Assume $\varphi(\texttt{U}) \cdot [\texttt{P}] \sim \varphi(\texttt{U})$ and $\varphi(\texttt{T}) \sim \varphi(\texttt{U})$. Then,

$$\varphi(\texttt{T}) \cdot [\texttt{P}] \sim \varphi(\texttt{U}) \cdot [\texttt{P}] \sim \varphi(\texttt{U}) \sim \varphi(\texttt{T})$$

$[\texttt{T: P}], [\texttt{T == U}] \vdash [\texttt{T.A == U.A}]$ (Member)

$\varphi(\texttt{T}) \sim \varphi(\texttt{U})$ implies $\varphi(\texttt{T}) \cdot \texttt{A} \sim \varphi(\texttt{U}) \cdot \texttt{A}$.

Finiteness Proof Supplement

Set of irreducible type terms $\mathcal{I}(G) \subset A^*$:

$$\{\text{for all } G \vDash \texttt{T}\colon \tilde{\varphi}(\texttt{T})\}$$
$$\bigcup$$
$$\{\text{for all } G_{\texttt{P}} \vDash \tau_0\texttt{.U}\colon \tilde{\varphi}_{\texttt{P}}(\texttt{Self.U})\}$$

- $\tilde{\varphi}(\texttt{T})$: normal form of $\varphi(\texttt{T})$
- $\tilde{\varphi}_{\texttt{P}}(\texttt{Self.U})$: normal form of $\varphi_{\texttt{P}}(\texttt{Self.U})$

If $\langle A;\, R\rangle$ is convergent (possibly infinite):

$$\langle A;\, R\rangle \text{ finite} \Leftrightarrow \mathcal{I}(G) \text{ finite}$$

Finiteness Proof Supplement

No element of $\mathcal{I}(G)$ is a proper suffix of any other:

$$\tau_i \cdot \mathtt{A}_{i_1} \cdots \mathtt{A}_{i_m} \qquad [\mathtt{Q}] \cdot \mathtt{A}_{j_n} \cdots \mathtt{A}_{j_n}$$

- If $t \in \mathcal{I}(G)$ and $t \cdot [\mathtt{P}] \sim t$, then $(t \cdot [\mathtt{P}] \sim t) \in R$.
- If $(t \cdot [\mathtt{P}] \sim t) \in R$, then $t \in \mathcal{I}(G)$.

Valid Type Parameters Supplement

If $G \vDash \texttt{U.A}$, $G \vDash [\texttt{U: P}]$ for some `P` declaring `A`. Therefore,

$$\begin{aligned}\varphi(\texttt{U.A}) &= \varphi(\texttt{U}) \cdot \texttt{A} \\ &\sim \varphi(\texttt{U}) \cdot [\texttt{P}] \cdot \texttt{A} \\ &\sim \varphi(\texttt{U}) \cdot [\texttt{P}]\texttt{A}\end{aligned}$$

Name symbols reduce to associated type symbols: $[\texttt{P}]\texttt{A} < \texttt{A}$.

Problem 4: Type parameter validity

Assume G is valid. Then $G \vDash \texttt{T}$ **if and only if** $\tilde{\varphi}(\texttt{T})$ does not contain name symbols.

Full Lowering Supplement

Now, $\tilde{\varphi}_{\mathtt{P}}(\mathtt{Self.U})$ might be a *proper suffix* of $\tilde{\varphi}(\mathtt{T})$:

$$\tilde{\varphi}(\mathtt{T}) = \tau_i \cdot [\mathtt{P}_1]\mathtt{A}_1 \cdots [\mathtt{P}_n]\mathtt{A}_n$$

$$\tilde{\varphi}_{\mathtt{P}}(\mathtt{Self.U}) = \begin{cases} [\mathtt{P}] \\ [\mathtt{P}]\mathtt{A}_1 \cdots [\mathtt{P}_n]\mathtt{A}_n \end{cases}$$

Problem 3: Name lookup

If t irreducible, then $t \cdot [\mathtt{P}] \sim t$ **if and only if** $t = u \cdot v$ for some $(v \cdot [\mathtt{P}] \rightarrow v) \in R$, and $u \in A^*$. We build the *property map*:

- Record all $(v \cdot [\mathtt{P}] \rightarrow v)$ in a multi-map with key v.
- Keys stored in suffix trie.
- Given `T`, check every suffix of $\tilde{\varphi}(\mathtt{T})$.

Full Lowering Supplement

Theorem

Given a string rewrite system $\langle A; R \rangle$, the full lowering outputs a convergent rewrite system for $G_{\mathtt{M}}$ **if and only if** $\langle A; R \rangle$ can be extended to a convergent rewrite system over A compatible with shortlex order on A^*.

Completion can fail:

- Bad choice of alphabet
- Reduction order

Some rewrite systems have no convergent presentation over *any* alphabet.