Introduction to Algebraic Subtyping

1 Overview

这篇文档是我个人对 Algebraic Subtyping [1] 的一些理解与读后感。我会尝试将它写得易于理解。Algebraic Subtyping 是 Stephen Dolan 在 2016 年发表的 thesis，它解决了一个类型推导领域数十年的 open problem，一个：

- 同时有 parametric polymorphism 和 subtyping 的类型系统的
- sound（推导出的类型一定正确）
- complete（对有类型的 term 一定能推导出类型）
- principle（推导出的类型是最 general 的，所有其他可能的类型都是它的子类型）

的类型推导算法。除此之外，我认为 Algebraic Subtyping 中还提出了很多对于类型系统/类型推导的设计非常有意义的思想和方法。

原 thesis 中大量使用了抽象代数，不过在这篇文档中，我会尝试避免对（我认为的）技术细节的深入探讨，对数学工具的使用提供直觉的、“PL”的解释。如果想要更精确的描述，可以去阅读原 thesis。

阅读这篇文档需要对 ML 的类型系统有基本的了解。最好能对 Hindley Milner 的类型推导算法也有一些基本的概念。在这篇文档中会常常将 Algebraic Subtyping 中的概念与 ML 中对应的概念进行对比。在进行上述对比的过程中，我发现了许多我以前所不知道的、ML 中的概念的直觉解释与联系。所以，即使你对 subtyping 不感兴趣，阅读这篇文档也可能会带来一些有趣的启发。

这篇文档中会不可避免地掺杂大量我的个人观点。我会尽量在我的个人观点前面加上“我认为”一类的标记。

这篇文档中许多构造的呈现方式可能与原 thesis 有所出入，内容的顺序也和原 thesis 不完全一致。我采用的是（我个人认为的）用于理解时最为便利的呈现方式和顺序。

在本文档中，我会用 A, B, C, D 来作为类型的 meta variable，用 t, u, s, r 作为表达式的 meta variable。

[1] https://www.cs.tufts.edu/~nr/cs257/archive/stephen-dolan/thesis.pdf

2 Subtyping 与类型推导

类型的最简单的数学 denotation 之一就是集合：我们可以把类型看作集合，把某个类型的表达式看作集合中的元素。在这个视角下，subtyping 关系 $A \preceq B$ 就是类型对应的集合之间的子集关系 $[\![A]\!] \subseteq [\![B]\!]$。子集的性质是：

$$\forall t \in A, t \in B \qquad (A \subseteq B)$$

所以把子集的性质翻译回类型中，就是：

$$\frac{\Gamma \vdash t : A \qquad A \preceq B}{\Gamma \vdash t : B} \ (Sub)$$

subtyping 也有很多具体的应用：

- 通过 $\texttt{int} \preceq \texttt{float}$，我们可以实现 C 中的整数到浮点数的隐式转换（在一个比 C 安全得多的类型系统中）
- 通过诸如 $\{x : \texttt{int}; y : \texttt{int}\} \preceq \{x : \texttt{int}\}$ 这样的、record type 上的 subtyping 关系，我们可以实现 polymorphic record。例如，函数`fun rcd -> rcd.x`可以接受任何含有`x`这个 field 的 record。
- 把上面的 record type 上的 subtyping 关系应用到 class 的 method list 上，或者使用用户声明的继承关系作为 subtyping 关系，就可以用 subtyping 来实现面向对象（事实上，这是 1970s 对 subtyping 的大量研究的主要推动力：给人望激热的 Java 找个理论模型）
- Haskell 式的 typeclass 也可以用 subtyping 实现，把一个 typeclass `C`看作一个抽象的类型，实现了`C`的 instance 的类型看作`C`的 subtype。（不过这一种 subtyping 与上面的几个例子有一个重要的不同：上述例子中利用类型规则 (Sub) 来改变类型是不需要运行时操作的。但在 typeclass 的例子里，(Sub) 是有对应的运行时操作的，得把 instance 的字典塞进值里面。这叫 coercive subtyping，因为 $A \preceq B$ 会对应一个 $A \to B$ 的 coercion function）。

所以，subtyping 看上去很美好：它有清晰的数学含义，也能解决极其大量的现实问题。很遗憾的是，支持 subtyping 的类型系统往往性质奇差，而且 subtyping 与类型系统的其他功能（比如 parametric polymorphism）的互动往往是非常复杂的。类型推导就是这样的一个例子：在一个同时有 parametric polymorphism (a là ML) 和 subtyping 的类型系统中的类型推导，从 1970s 以来都不曾有个能让人满意的成果，直到 algebraic subtyping 的出现。

在介绍 Algebraic Subtyping 本身之前，我将首先列出一些 Subtyping 的类型推导需要支持的功能和解决的问题：

- bounded quantification。考虑函数`fun rcd -> print(rcd.x); rcd`。在一个只有 subtyping 而没有 parametric polymorphism 的类型系统中，它的类型是 $\{x : \texttt{string}\} \to \{x : \texttt{string}\}$。然而，这个类型并不准确：`rcd`中的其他 field 的信息在返回值中丢失了。所以，要准确表达它的类型需要

更高级的类型系统功能。我们可以用一个 type variable α 来指代`rcd`的类型，那么返回值的类型就应该是 α。但同时，α 必须是一个有 field x 的 record type。所以，这个函数的准确类型是：

$$\forall \alpha \preceq \{x : \texttt{string}\}.\alpha \to \alpha$$

这种与 subtyping 结合的 parametric polymorphism 就叫作 bounded quantification。

- equi-recursive type。递归类型一般的语法是 $\mu\alpha.A$，α 在 A 中表示这个类型自身的递归出现，所以有：

  $$\mu\alpha.A = A[\alpha \mapsto \mu\alpha.A]$$

  我们把沿着这个等式从左到右的移动叫作 unfold，从右到左的移动叫 fold。如果 unfold 和 fold 是有表达式中的特殊构造来完成的，这种递归类型就称为 iso-recursive type。如果 unfold 和 fold 可以在任何地方隐式地发生，就称为 equi-recursive type。

  那么，equi-recursive type 和 subtyping 有什么关系呢？在一个支持 subtyping 的类型系统中，有如下一种现象：一个表达式有很多非递归的类型，但最一般（principal）的类型是递归的。下面的例子来自 Algebraic Subtyping 原文：

  ```
  let rec f g = f (g true)
  ```

  在没有 subtyping 的类型系统中，函数`f`不存在非递归的类型。但在有 subtyping 的情况下，由于 bottom type $\bot$ 的存在（它是任何类型的子类型），`f`可以有类型：

  $$\begin{aligned}
  &(\texttt{bool} \to \bot) \to \bot \\
  &(\texttt{bool} \to \texttt{bool} \to \bot) \to \bot \\
  &(\texttt{bool} \to \texttt{bool} \to \texttt{bool} \to \bot) \to \bot \\
  &\cdots
  \end{aligned}$$

  而它的最一般的类型是 $(\mu\alpha.\texttt{bool} \to \alpha) \to \bot$。诚然，函数`f`应当被拒绝。但这一点很难通过类型系统做到。所以如果我们想要 principle 的类型推导，就必须支持 equi-recursive type。

- 任意两个类型 A、B 应该有 least upper bound $A \sqcup B$ 与 greatest lower bound $A \sqcap B$。

  前者出现在诸如`if`的控制流分叉结构中。如果`if`的两个分支分别有类型 A 和 B，由于 subtyping 的存在，A 和 B 不一定相等。这是我们需要找到一个类型 C，使得 $A \preceq C$ 且 $B \preceq C$ （A 和 B 都可以被当作 C 使用），而最好的 C 就是 $A \sqcup B$。$A \sqcup B$ 可以被看作一种 union type。

后者出现在对一个变量的不同使用中。考虑一个函数`fun x -> ...`，在它的函数体中，x 分别被以类型 A、B 使用了。同样的，A 和 B 不一定相等。我们需要一个类型 C，使得 $C \preceq A$ 且 $C \preceq B$ （C 即可以被当作 A 也可以被当作 B 使用），而最好的 C 的选择就是 $A \sqcap B$，也可以叫作 intersection type。

- subsumption。即使在有完整类型推导的语言里，用户提供的类型标注作为文档和对程序的约束也是很重要的。所以，我们需要判断推导出的类型和某个用户提供的类型是否兼容。当推导出的类型和类型标注都是 pamatetrically polymorphic 的类型（type scheme）的时候，判断它们是否兼容的过程就叫作 subsumption。

最后，我将简单介绍过去的工作是如何为 subtyping 实现类型推导的。通过遍历表达式，我们可以得到一系列形如 $A \preceq B$ 的 constraints，这些 constraints 可以用一些规则来化简，例如：

$$(A \to A') \preceq (B \to B') \Leftrightarrow B \preceq A \wedge A' \preceq B'$$

最终，我们可以把 constraints 化简成关于 type variable 的简单 constraint，我们直接将这些 constraint 记录下来，以 bounded quantification 的形式放到类型里。

那么，这种做法会出现什么问题呢？首先，最终留下的 constraint 往往会非常复杂，导致推导出的类型非常长而复杂、难以阅读和理解。其次，为了实现上面提到的 subsumption 的判断，需要判断两个 constraint list 是否等价/蕴含关系，而这也是一个非常复杂的问题。

3 Algebraic Subtyping 的思想

上面我们简单地回顾了 subtyping 上类型推导这一问题。下面我们正式开始介绍 Algebraic Subtyping 中提出的内容。首先，我将介绍 Algebraic Subtyping 中提出的一些思想与方法。原 thesis 正是通过这些思想和方法来推导出后续的构造的。我认为它们对其他类型系统的设计也十分具有参考价值。

3.1 Subtyping 是类型上的代数结构

上面提到了，subtyping 可以看成集合的子集关系在类型上的反映。如果脱离集合来考虑，那 subtyping 就构成了类型上的一个有序关系。例如，很自然地，subtyping 关系应当是 reflexive 且 transitive 的，那么它就构成了类型上的一个 preorder 关系。如果我们把互为子类型的两个类型看成相等的（$A \preceq B \wedge B \preceq A \to A = B$），那么 subtyping 就变成了一个 partial order。上面提到了，我们希望任意两个类型 A 和 B 之间有 lub $A \sqcup B$ 和 glb $A \sqcap B$。如果 $\sqcup$ 和 $\sqcap$ 是两个 total 的操作，那么此时类型就构成了一个 lattice。

但是，形如 $(A \to B) \sqcap \texttt{bool}$ 的类型的意义是什么呢？一个自然的解法是，我们加入一个 top type $\top$，使得 $\forall A, A \preceq \top$，和一个 bottom type $\bot$，使得

$\forall A, \bot \preceq A$。那么，我们就可以使得 $(A \to B) \sqcap \texttt{bool} = \bot$，$(A \to B) \sqcup \texttt{bool} = \top$。如此一来，类型就构成了一个 bounded lattice。所以，subtyping 对应着类型上的 order theory 结构。这一对应能够帮助我们使用 order theory 的工具与结论来研究 subtyping 关系自身。这就引出了 Algebraic Subtyping 中最重要的一个思想：algebra 先于 syntax。

3.2 Algebra First

上面，我们通过加入 $\top$ 和 $\bot$ 来支持 $\sqcup$ 与 $\sqcap$，使得类型构成了一个 bounded lattice。从 *syntax* 的角度来说，我们往类型的语法中加入了 $\top$ 和 $\bot$ 两个常量，就使得类型有了我们想要的性质。所以，这一构造确实是十分自然的：它是 *syntax* 上最简单的构造。不过注意到，上述构造从 order theory 的角度来看，也是一个把 poset 补全成 lattice 的过程（因为我们想要 $\sqcup$ 和 $\sqcap$）。那么，从 order theory（或者更广义地说，"algebra"）的角度来看，上面的构造又是否自然呢？

注意到，在加入 $\top$ 和 $\bot$ 之前，$\sqcap$ 和 $\sqcup$ 就已经有了部分的定义：

$$(A \to B) \sqcap (A' \to B') = (A \sqcup A') \to (B \sqcap B')$$
$$\texttt{bool} \sqcap \texttt{bool} = \texttt{bool} \sqcup \texttt{bool} = \texttt{bool}$$

类型可以看成诸如 `bool`、$A \to B$ 这样、head constructor 不同的类型的 disjoint union，而每种 constructor 自身已经是一个 lattice 了。所以，从 algebra 的角度来看，直接添加 $\top$ 和 $\bot$ 的构造看起来是这样的：

> 每种 constructor 构成 lattice
>
> $\to$ 忘记它们的 lattice 结构，当成 poset 取 disjoint union
>
> $\to$ 复用它们原来的 lattice 结构，加上 $\top$ 和 $\bot$ 构成 lattice

可以看到，从 algebra 的角度来看，直接添加 $\top$ 和 $\bot$ 是一个非常没有道理的做法。从 *algebra* 的角度来看，最简单的构造应该是某种 lattice 上的 disjoint union。

于是，现在我们有了两种不同的构建类型的方式，它们都能达到目的（使类型变成一个 lattice），它们分别是 *syntax* 上和 *algebra* 上最简单的做法。在这一观察的基础上，Algebraic Subtyping 中最重要的思想就是：

> *我们应当选择 algebra 上简单的构造方式，而非 syntax 上简单的构造方式。*

而这么做的理由也很充分：algebra 上简单的类型系统更容易有好的性质，至于 syntax，我们总能通过一些 heuristic 的设计来让它看起来更简单。

3.3 Type Variables as Skolems

在有 parametric polymorphism 的系统中，我们必须要给带 free type variable 的 open type、给它们中的 type variable 赋予意义。一种自然的解释是，我们把 type variable 看作 meta theory 的 variable，让 type variable 代表某个未知的 ground type（没有 free type variable 的类型）。

然而，还有另外一种不同的解释方法：type variable 也可以被看作 skolem constant，某个只和自身相等、只和自身有关系的类型。这种解释从许多不同的角度都能得到支持：

- 在 ML 中，在进行 subsumption 判断时，由于我们在比较两个 polymorphic 的 type scheme，不能随意 unify 两个 type variable，此时应当把 type variable 当作 skolem constant 处理。
- 通过把 type variable 看作 skolem constant，它们成为了类型的 (algebraic) 构造的一部分，那么对 open type 进行处理就会更加方便。
- 让 type variable 去 quantify over 所有 ground type，相当于做了一种 closed world assumption，即语言中不会有新的类型被加入。但这会使得类型系统的可拓展性变弱，不利于向类型系统加入新的功能。
- 如果我们从 universal algebra 的角度来看类型，把 type variable 看作 generator，那么 skolem constant 的解释下类型才会对应 free object/free algebra。也就是说，skolem constant 的解释是 algebra 上更简单的。这也可以看成是上一点（closed world v.s. open world）的数学表达。

4 类型的构造

在大部分类型系统中，类型本身的语法的构造是一个不需要什么推理与分析的过程，然而，在 Algebraic Subtyping 中，类型本身的构造是一个非常重要的过程。接下来，我将介绍 Algebraic Subtyping 中如何利用上面的思想，完成类型自身的构造。但我不会介绍原 thesis 中 recursive type 的构造。因为它虽然必要，但非常复杂，而且我认为它属于“技术细节”，并不很有意义或参考价值。

4.1 基本类型的构造

在前面我们提到了，通过加入 $\top$ 和 $\bot$ 来将类型补全成 lattice 并不是 algebra 上最简单的方法。所以，Algebraic Subtyping 中使用了 algebra 上的构造方法。首先，让我们回顾我们已经有的原料：

- 我们有诸如 `bool` 这样的 constant type。每个 base type 都有一个 trivial 的 lattice 结构。
- 我们有函数类型 $A \to B$，它具有如下的 lattice 结构：

$$(A \to B) \sqcap (A' \to B') = (A \sqcup A') \to (B \sqcap B')$$
$$(A \to B) \sqcup (A' \to B') = (A \sqcap A') \to (B \sqcup B')$$

- 类似地，record type 也有对应的 lattice 结构。如果我们支持 record polymorphism，那么 $\sqcap$ 会对应 label 集合的并集，$\sqcup$ 会对应 label 集合的交集。

换言之，类型作为一个 disjoint union，它的每个组件都已经是一个 lattice 了。那么，algebra 上最简单的方法就是直接取 lattice 上的 disjoint union。在 order theory 中，一个符合要求的构造是 lattice free coproduct。它对 disjoint union 的每个组件内部的 $\sqcup$ 和 $\sqcap$ 不做改动，对于跨组件的 $A \sqcup B (A \sqcap B)$ 的操作，它采用构造 free lattice 的方法：直接加入一个新的元素 $A \sqcup B (A \sqcap B)$，并使它满足必要的运算律。于是，类型的 syntax 就变成了：

$$\texttt{Types} \ni A, B ::= \texttt{bool} \mid A \to B \mid \top \mid \bot \mid A \sqcup B \mid A \sqcap B$$

所以，从 syntax 的角度来看，lattice 的 free coproduct 引入了 union type 和 intersection type。但同一大类的类型（比如两个函数类型）之间的 intersection/union 是可以简化的。所以同一个类型可能会对应多个不同的 syntax。换言之，我们是先定义了类型的 denotation/algebra，再反过来为这个 algebra 设计对应的 syntax。

union type 和 intersection type 的引入能极大地简化类型系统。在传统的 subtyping 系统中，type scheme 里可能含有 subtyping constraint，因为关于 type variable 的 constraint 无法继续被化简。然而，由于我们的类型变成了一个 lattice，众所周知在一个 lattice 中可以通过 $A \sqcap B = A$ 或 $A \sqcup B = B$ 来表达 $A \preceq B$。所以，我们可以将所有 constraint 通过 $\sqcap$ 和 $\sqcup$ 嵌入到类型中，而不再需要在类型中保留 subtyping constraint 了。例如，$A_1 = (\alpha \preceq A \Rightarrow \alpha \to \alpha)$ 可以写作 $A_2 = (\alpha \sqcap A \to \alpha \sqcap A)$。我们可以验证这两个类型是等价的。假设我们要将类型为 B 的参数喂给这个函数：

- 当 $B \not\preceq A$ 时，类型 A_1 将会拒绝这次函数调用。在 A_2 中，我们需要找到一个 α 使得 $B \preceq \alpha \sqcap A$，而这等价于 $B \preceq \alpha \wedge B \preceq A$，由于后者不可能成立，所以我们不可能找到符合要求的 α。
- 当 $B \preceq A$ 时，类型 A_1 给出的返回值类型为 B。在 A_2 中，$B \preceq \alpha \sqcap A$ 的最优解是 $\alpha = B$，由于 $B \preceq A$，所以返回值类型是 $B \sqcap A = B$。

所以，假如我们有 $\alpha \preceq A_i$，那么只需要将 α 替换为 $\bigsqcap A_i$，即可得到一个不包含任何 constraint、但等价的类型。

在原 thesis 中，还额外要求类型构成一个 distributive lattice，也就是说 $A \sqcap (B \sqcup C) = (A \sqcap B) \sqcup (A \sqcap C)$。这一要求从 syntax 上看完全没有意义，但它能让我们复用 distributive lattice 的一些非常优良的性质。在 subsumption 的判断中，可能会出现形如：

$$\bigsqcap_{i=1}^{n} A_i \preceq \bigsqcup_{i=1}^{n} B_i$$

的 constraint。其中每个 A_i（B_i）有不同的 head constructor，也就是说 $\preceq$ 两侧的类型都不能被进一步化简了。在一个一般的 lattice 中，这个 constraint 无法被简单地拆分成更小的 constraint 的组合。但在一个 distributive lattice 的 disjoint union 中，它等价于：

$$\exists k, A_k \preceq B_k$$

因此，如果类型是一个 distributive lattice，我们只需要把每个 head constructor 都尝试一遍，就能解决上面的 constraint。

4.2 Type Variable 的构造

正如前面提到的，Algebraic Subtyping 中认为 type variable 应当被看作 skolem constant，而非 quantify over all ground types 的 meta theoretic variable。所以在类型中引入 type variable 的方法也很简单：从 algebra 的角度，我们把每个 type variable 看作一个和 `bool` 相似的常量类型，并通过 distributive lattice 上的 free coproduct 将它们于其他类型放到一起。从 syntax 的角度，我们把 type variable 当作独立的 syntactic object 加入到类型的语法中（在这一点上两种对 type variable 的解释没有区别，这也能体现出 syntax 表达能力的不足）。

那么，我们该如何解释 substitution 呢？从 algebra 的角度来看。假设 type variable 的集合是 $\mathcal{V}$，类型的集合是 $\mathcal{A}$，substitution ρ 可以看成一个 $\mathcal{V} \to \mathcal{B}$ 的函数。由于 open world assumption，$\mathcal{B}$ 是一个与 $\mathcal{A}$ 可能不同的 domain，而我们希望将 ρ 的作用域从变量拓展到类型，虽然它们都必须是类型的 algebra 的 model。现在，我们想要将 ρ 应用到类型上，得到一个（唯一的）$\hat{\rho} : \mathcal{A} \to \mathcal{B}$。前面提到过，按照 open world assumption 构造出的类型才是一个 free algebra，而 $\hat{\rho}$ 的存在性与唯一性恰恰就是 free algebra 的定义。所以，algebra 的定义可以直接导出 substitution 的优良性质。

那么，从 syntax 上，又该如何理解 open world assumption 的性质呢？事实上，这同样很简单：

- 由于 open world assumption，type variable 可能被替换为当前类型定义中不存在的新类型（$\rho : \mathcal{A} \to \mathcal{B}$）。
- 不过我们可以保证新的类型定义中包含旧的类型定义，而且旧的类型的解释/性质不变（$\mathcal{B}$ 是当前类型 algebra 的一个 model）。
- 我们可以保证将 substitution 应用在类型上时，它依然有我们想要的性质，比如 $\hat{\rho}(A \sqcup B) = \hat{\rho}(A) \sqcup \hat{\rho}(B)$ （$\hat{\rho}$ 存在、唯一且是一个当前类型 algebra 上的 homomorphism）。

4.3 Subsumption

在有了 type variable 之后，我们就能构建出（parametrically）polymorphic 的 type scheme 了。那么，下一步，我们需要研究不同 type scheme 之间的 subtyping 关系：subsumption。直觉上，一个 polymorphic type scheme 能够被 instantiate 成许多不同的类型，而 subsumption 就对应这 type scheme 的 instance 集合之间的子集关系。以 ML 为例，我们可以定义一个 type scheme $\forall \bar{\alpha} A$ 的 instance 集合 $\mathcal{I}(\forall \bar{\alpha}.A)$ 为：

$$\{\rho(A) \mid \mathtt{dom}(\rho) = \bar{\alpha}\} \tag{I1}$$

容易证明，$\mathcal{I}(\forall \bar{\alpha}.A)$ 等价于所有满足下面条件的类型 B 构成的集合：

$$\frac{\Gamma \vdash t : \forall \bar{\alpha}.A}{\Gamma \vdash t : B} \text{ is derivable} \tag{I2}$$

所以，$\mathcal{I}(\_)$ 确实符合我们对“一个 polymorphic type scheme 的所有 instance”的直觉理解。利用 $\mathcal{I}(\_)$，我们可以定义 polymorphic type scheme 上的 subsumption

关系 $\forall\bar{\alpha}.A \preceq^{\forall} \forall\bar{\beta}B$ 为：

$$\mathcal{I}(\forall\bar{\beta}.B) \subseteq \mathcal{I}(\forall\bar{\alpha}.A) \tag{S1}$$

将 $\mathcal{I}(\_)$ 的定义展开，就能得到一个更常见的定义：

$$\forall\bar{B}, \exists\bar{A}, A[\bar{\alpha} \mapsto \bar{A}] = B[\bar{\beta} \mapsto \bar{B}] \tag{S2}$$

例如，在 ML 中，如果 $A_1 = \forall\alpha.\alpha \to \alpha \to \texttt{bool}$，$A_2 = \forall\alpha\beta.\alpha \to \beta \to \texttt{bool}$，那么 $A_2 \preceq^{\forall} A_1$。Algebraic Subtyping 中，type variable 被看作了类型的 algebra 中实际存在的 skolem constant。所以，我们可以自由地将一个 type variable 替换为另一个 type variable。利用这一点，上述定义可以进一步简化成：

$$\exists\rho, \texttt{dom}(\rho) = \bar{\alpha} \wedge \rho(A) = B \tag{S3}$$

容易证明，上面的所有定义都是等价的，并且都等价于下面的定义：

$$\frac{\Gamma \vdash t : \forall\bar{\alpha}.A}{\Gamma \vdash t : \forall\bar{\beta}.B} \text{ is derivable} \tag{S4}$$

那么，当引入了 subtyping 后，又该如何修改 instance 集合与 subsumption 的定义呢？对于 instance 集合，我们采用上述定义 I2 作为原始定义。通过由于 subtyping 相关的类型规则的引入，定义 I1 会变成：

$$\mathcal{I}(\forall\bar{\alpha}.A) = \{A' \mid \exists\rho, \texttt{dom}(\rho) = \bar{\alpha} \wedge \rho(A) \preceq A'\} \tag{I1'}$$

这一定义在类型推导算法的正确性证明中有着非常重要的作用。对于 subsumption，我们采用上述定义 S4 作为原始定义，加上 subtyping 的类型规则之后，定义 S3 会变成：

$$\exists\rho, \texttt{dom}(\rho) = \bar{\alpha} \wedge \rho(A) \preceq B \tag{S3'}$$

类似地，我们可以构造出对应定义 S2 的定义 S2'，并且证明它们和定义 S1 是等价的。

上面多个不同定义的互相对应能够让我们确信我们得到的 subsumption 关系是正确的。我们定义出的 subsumption 关系是一个 preorder，所以我们可以从它当中得到一个等价关系 $=^{\forall}$：

$$A =^{\forall} B \Leftrightarrow A \preceq^{\forall} B \wedge B \preceq^{\forall} A$$

$=^{\forall}$ 是一个十分强大的关系。它能够帮助我们形式化地表达并证明许多事实。例如，考虑函数 $\lambda bxy.\texttt{if}\ b\ \texttt{then}\ x\ \texttt{else}\ y$。在 ML 中，它的类型是：

$$A_1 = \texttt{bool} \to \alpha \to \alpha \to \alpha$$

这一类型在有 subtyping 的系统中也依然是正确的。但在有 union type 的情况下，上述函数的类型还可以写成：

$$A_2 = \texttt{bool} \to \beta \to \gamma \to \beta \sqcup \gamma$$

那么，这两个类型之间有什么联系呢？$=^{\forall}$ 告诉我们，它们是等价的！由于 $\alpha \sqcup \alpha = \alpha$，$A_2[\beta, \gamma \mapsto \alpha] \preceq A_1$，所以 $A_2 \preceq^{\forall} A_1$。与此同时，由于 $\beta, \gamma \preceq \beta \sqcup \gamma$，$A_1[\alpha \mapsto \beta \sqcup \gamma] \preceq A_2$。所以 $A_1 \preceq^{\forall} A_2$，从而 $A_1 =^{\forall} A_2$！

5 Polar Types 与 Biunification

现在，我们已经完成了 Algebraic Subtyping 中构造类型系统的过程。从现在开始的内容就是关于如何实现类型推导的了。这一节包含类型推导中一些对类型本身的操作。

5.1 Polar Types

我们目前的类型定义已经有许多优秀的性质了。然而，当用于类型推导时，它依然会产生一些难以处理的 constraint。形如 $A \sqcap B \preceq C$ 或 $A \preceq B \sqcup C$ 的 constraint，难以简单地化简为关于 A、B 和 C 的 constraint。如果强行尝试化简，类型推导需要引入 disjunction、尝试不同的可能性，从而变得极其困难或难以接受得低效。

所以，我们希望消除上述难以处理的 constraint。观察到，类型系统中唯一会引入 subtyping constraint 的地方是函数调用：假设 $f : A \to B$，$x : A'$，那么 $f(x)$ 有正确的类型意味着 $A' \preceq A$。其他诸如同一变量的不同使用之间的一致性、控制流分叉的结果的类型的一致性的需求，可以通过 $\sqcap$ 和 $\sqcup$ 完成——在我们 algebra first 的类型设计下，它们永远是 total 的。回到函数调用上来，在上面 constraint $A' \preceq A$ 中，A' 是 x 的类型，它出现在一个 positive 的位置，而 A 是一个函数的参数类型，它出现在一个 negative 的位置。所以，假设 A^- 是处于 negative 位置的类型，A^+ 是处于 positive 位置的类型，那么类型推导时，我们需要处理的全部 constraint 都有 $A^+ \preceq A^-$ 的形式。现在，回到我们最初的目的上来，我们想要消除形如 $A \sqcap B \preceq C$ 或 $A \preceq B \sqcup C$ 的 constraint，那么，我们只需要要求所有 A^+ 都只能包含 union（$\sqcup$），所有 A^- 都只能包含 intersection（$\sqcap$）即可。所以，我们可以将 A^+、A^- 区分开，并将它们的语法分别定义为

$$\begin{aligned} \texttt{PosTypes} \ni \quad A^+, B^+ ::= \quad & \alpha \mid A^- \to B^+ \mid A^+ \sqcup B^+ \\ \texttt{NegTypes} \ni \quad A^-, B^- ::= \quad & \alpha \mid A^+ \to B^- \mid A^- \sqcap B^- \end{aligned}$$

那么，我们就可以保证形如 $A^+ \preceq A^-$ 的 constraint 永远不会引入 disjunction 了。

但是，如何保证 polar types 这一限制是正确的呢？很显然，polar types 中无法表达本来的类型定义中的许多类型，例如 $A \sqcup (B \sqcap C)$。幸运的是，在构建出完整的类型推导算法并证明其正确性后，我们发现：

每个 term 的 principal type 一定是一个 polar type

所以，在类型推导中使用 polar type 并不会使我们失去任何东西。但是，这只是一个幸运的、脆弱的巧合吗？在我看来并不是。我们可以对 positive occurence、negative occurence 以及类型推导作出如下直觉上的解释：

- positive occurence 代表 provider：一段程序*构建*出有 positive type 的值。
- negative occurence 代表 consumer：一段程序*需要*（或者说 abstract over）有 negative type 的值。

- 当我们*构建*一个类型为 $A \to B$ 的函数时，这个函数*需要* A，并*构建* B。所以 $A^- \to B^+ \in \texttt{PosTypes}$。
- 当我们*需要*一个类型为 $A \to B$ 的函数时，我们*构建* A 并提供给它，并*需要*它返回的 B。所以 $A^+ \to B^- \in \texttt{NegTypes}$。
- 类型系统中对不同表达式的类型的唯一限制应当来自于 provider 与 consumer 连接起来时的一致性。所以使用 polar types 的类型推导只会产生形如 $A^+ \preceq A^-$ 的 constraint。
- ML 要求同一个变量的不同使用、`if` 的不同分支的结果的类型相等，*这其实不是必要的，只是为了简化类型的语法*。如果变量被以不一致的类型使用了，那么我们永远不可能找到一个合适的 provider 来与之配对。同理，如果 `if` 的不同分支类型不同，那么，那么任何要使用其参数的 consumer 都无法使用这一结果。也就是说，所有类型错误都可以延后到 provider 和 consumer 配对时再触发。
- 在普通的语言中，provider 唯一的组合方式是 disjunction（例如 `if` 的分支），consumer 唯一的组合方式是 conjunction（例如同一个变量的不同使用）。而 principal type 应当是从表达式的结构中产生的，所以所有 principal type 都符合 polar types 这一形式。

5.2 Bisubstitution

事实上，区分类型的 positive occurence 和 negative occurence 并不是一个 Algebraic Subtyping 首创的做法。仅就我所知的范围内，在 1980s 就有 intersection type 相关的工作中采用了相同的做法。[2] 然而，区分 occurrence 和 parametric polymorphism 混合时，会出现一个矛盾的情况，我认为这也是为什么区分 occurrence 的做法并没有流行起来的重要原因之一。考虑函数：

$$\texttt{id} = \lambda x.x : \forall \alpha.\alpha \to \alpha$$

当我们 instantiate α 时，α 应当被替换为一个 positive type 还是 negative type 呢？显然，无论如何选择，得到的类型都是不符合 polarity 限制的。在上面提到的 intersection type 相关的工作中，作者为了解决这一问题，只允许 type variable 被 instantiate 成同时是 positive type 和 negative type 的类型，例如 `bool`、$\texttt{bool} \to \texttt{bool}$ 等。但在 Algebraic Subtyping 的类型系统中，作者提出了一种更加灵活的办法：将 substitution 拓展为 *bi*substitution。

普通的 substitution 的形式是 $\alpha \mapsto A$，它对每个 type variable 赋予一个类型。而 bisubstitution 则对每个 type variable 赋予两个类型：为所有 positive occurrence 一个，所有 negative occurrence 一个。所以，bisubstitution 的形式是 $\alpha^+ \mapsto A^+, \alpha^- \mapsto A^-$。当应用 bisubstitution 时，$\alpha$ 所有的 positive occurrence 被替换成 A^+，所有 negative occurrence 被替换为 A^-。这样一来，就可以保证一个 polar type 应用 bisubstitution 后依然是一个 polar type。

[2] https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.123.8718&rep=rep1&type=pdf

但是，一个 type variable 如何能被赋予两个类型？我们应该如何解释 bisubstitution？沿用将 occurrence 视作 provider 与 consumer 的解释，我们可以对 bisubstitution 的意义作如下的解读：

- 考虑一个 type scheme θ，它当中包含一个 type variable α。α 可能出现在 negative 的位置，说明 θ 中有对 α 的需求，也可能出现在 positive 的位置，说明 θ 中会构建出 α 的值。
- 当我们使用类型为 θ 的值时，我们需要选定一个类型 A 来 instantiate α。但是对 A 的约束有两个不同的来源：在 θ 需要 α 的地方（negative occurence），我们可能提供了一些类型为 B_i^+ 的值，对于 θ 中构建出来的 α（positive occurence），我们可能将它们当作 C_i^- 来使用了。A 的选择应当同时受到这两种情况的约束，使得：
$$\bigsqcup B_i^+ \preceq A \preceq \bigsqcap C_i^-$$
- 反过来说，由于类型检查的唯一约束来自 provider 与 consumer 间的一致，任何满足上述条件的 A 都可以被用于 instantiate α。
- 对于一个 negative type，我们关心的是*谁是它的 subtype*，也就是说哪些 provider 能被它使用。所以我们可以把 α 的所有 negative occurence 替换成它的上界，$\bigsqcup C_i^-$。
- 对于一个 positive type，我们关心的是*它是谁的 subtype*，也就是说它能被提供给哪些 consumer。所以我们可以把 α 的所有 positive occurrence 替换成它的下界，$\bigsqcup B_i^+$。
- 因此，bisubstitution 背后的直觉来自两个因素。首先，对 polymorphic type scheme 的使用会从 negative、positive 两个方向对 type variable 的 instantiation 带来约束。其次，对于一个 positive/negative type，我们只关心它的下界/上界。

我们也可以直接证明 bisubstitution 的正确性。假定类型系统中有如下两条关于 parametric polymorphism 与 subtyping 的标准规则：

$$\frac{\Gamma \vdash t : \forall \alpha . A}{\Gamma \vdash t : A[\alpha \mapsto A]}\ (inst) \qquad \frac{\Gamma \vdash t : A \qquad A \preceq B}{\Gamma \vdash t : B}\ (sub)$$

给定一个 bisubstitution $\xi = [\alpha^+ \mapsto A^+, \alpha^- \mapsto A^-]$，假设 $A^- \preceq A^+$（在原 thesis 中满足这一条件的 bisubstitution 被称为 stable 的），那么在非 polar 的类型中，我们可以定义一个 substitution $\rho^+ = [\alpha \mapsto A^+]$。通过简单的归纳，容易证明对于任意的 positive type A^+ 和 negative type A^-：

$$\rho^+(A^+) \preceq \xi(A^+) \qquad \xi(A^-) \preceq \rho^+(A^-)$$

所以我们可以导出如下的类型推断：

$$\frac{\rho^+(A^+) \preceq \xi(A^+) \qquad \dfrac{\Gamma \vdash t : \forall \alpha . A^+}{\Gamma \vdash t : \rho^+(A^+)}\ (inst)}{\Gamma \vdash t : \xi(A^+)}\ (sub)$$

所以使用 stable 的 bisubstitution 来 instatiate 一个 polymorphic type scheme 是安全的。此外，假设 Γ^- 是一个只包含 negative type 的 context（由于 negative type 对应 consumer，对应未知量，所以类型推导中所有 context 都应该只包含 negative type），且 $\Gamma^- \vdash t : A^+$，那么通过普通 substitution 的性质，即 $\rho^+(\Gamma^-) \vdash t : \rho^+(A^+)$，以及 (sub)，可以导出 $\xi(\Gamma^-) \vdash t : \xi(A^+)$。换言之，stable 的 bisubstitution 满足 substitution lemma。

那么，stable 这一约束条件、$A^- \preceq A^+$ 又该如何解读呢？考虑一个 polymorphic type scheme θ，它包含 type variable α。由于 α 是被 quantify over 的，θ 应该对任意的 α 表现一致。所以 θ 不能凭空构建出类型 α 的值：它构建出的每个 α^+ 都必定是从别的某处通过某个 α^- 得到的。而在数据从 α^- 流向 α^+ 的过程中只能损失类型信息，而不能凭空产生类型信息，因为 θ 对 α 一无所知。这一要求对应到 bisubstitution 上就是 $\xi(\alpha^-) \preceq \xi(\alpha^+)$。

5.3 Subtyping Constraint 的解

在有了 polar type 和 bisubstitution 这两样工具之后，我们就可以开始着手考虑一系列 polarized 的 constraint $C = A_i^+ \preceq A_i^-$ 了。给定一系列 constraint C，我们希望找到一个最一般的 bisubstitution ξ 作为 C 的解。这一点和 ML 的类型推导中的 unification 非常相似。在原 thesis 中，这一过程被称作 biunification。

在设计 biunification 的算法之前，我们首先需要思考我们希望求得的解满足什么样的性质。在 ML 的 unification 中，一个 substitution ρ 是一系列 equality constraint $A_i \doteq B_i$ 的解意味着 $\rho(A_i) \doteq \rho(B_i)$。然而，对于 bisubstitution，这一定义并不适用。考虑一个最简单的 constraint $\alpha \preceq A$。在前面我们曾经讨论过，union type 和 intersection type 的引入可以消除在类型中的 explicit constraint，类型 $\alpha \preceq A \Rightarrow \alpha \to \alpha$ 会等价于 $\alpha \sqcap A \to \alpha \sqcap A$。后者并不是一个 polarized 的类型，但我们可以把它写成一个等价的（通过 $=^\forall$）、polarized 的形式 $\alpha \sqcap A \to \alpha$。所以，在类型推导中，我们希望有如下过程发生：

- 求解 $\alpha^+ \preceq A^-$ 的结果，会使 $\alpha \to \alpha$ 变成 $\alpha \sqcap A^- \to \alpha$。

显然，这对应着一个 bisubstitution $\xi_0 = [\alpha^- \mapsto \alpha \sqcap A]$。然而，$\xi(\alpha^+ \preceq A) = \alpha^+ \preceq A$，这一 constraint 并不对所有 α 成立！

为了能够准确地定义出“constraint 的解”这一概念，让我们回想我们是如何（informal 地）确认 ξ_0 的正确性的：

- 我们首先考虑了 $C = \alpha^+ \preceq A^-$ 下的某个类型 $B = \alpha \to \alpha$，我们对它的语义有直觉上的理解。
- 我们发现 $\xi(B)$ 与 $C \Rightarrow B$ 的语义是等价的。所以我们认为 ξ 确实是 C 的一个解。

将上述过程稍微泛化，我们可以用如下方式来定义“constraint 的解”：

- 对于任意类型 B，我们需要定义 B 在 C 下的语义。我们可以拓展 instance 集合 $\mathcal{I}(\_)$ 的定义来达成这一目的。

- 如果对于任意的类型 B，$C \Rightarrow B$ 的语义和 $\xi(B)$ 的语义都是相同的，（即 $\mathcal{I}(C \Rightarrow B) = \mathcal{I}(\xi(B))$），那么 ξ 和 $C \Rightarrow \_$ 有着相同的效果，所以我们可以认为 ξ 是 C 的一个解。由于 ξ 和 $C \Rightarrow \_$ 的效果完全相同，ξ 可以看作是 C 的*最优*解。

让我们首先考虑 $C \Rightarrow B$ 的 instance。直觉上，在 instantiate B 中的 type variable 时，我们必须尊重 C 中的 constraint。所以，我们可以如此定义 $C \Rightarrow B$ 的 instance 集合：

$$\mathcal{I}(C \Rightarrow B) = \{B' \mid \exists \rho \vDash C, \rho(B) \preceq B'\}$$

其中 $\rho \vDash C$ 意味着 $\rho(C)$ 中的 constraint 都是无条件成立的（比如 `int` $\preceq$ `float`）。利用这一定义，ξ 是 C 的最优解意味着：

$$\forall B, \mathcal{I}(C \Rightarrow B) = \mathcal{I}(\xi(B))$$

为了进一步验证这一定义，可以证明，当 C 中包含的是 equality constraint，类型系统中没有 subtyping 时，用上述方式定义出来的“最优解”就是普通 unification 中的 most general unifier。

5.4 Biunification

现在，我们终于可以开始定义 biunification 的算法本身了。在进行了诸多铺垫后，biunification 的算法本身十分简单。对于一侧为 type variable 的 atomic constraint，我们直接给出一个 bisubstitution 作为结果：

$$\begin{aligned}
&\texttt{biunify}(\alpha^+ \preceq A^-) = [\alpha^- \mapsto \alpha \sqcap A^-] && \alpha \text{ not free in } A^- \\
&\texttt{biunify}(A^+ \preceq \alpha^-) = [\alpha^+ \mapsto \alpha \sqcup A^+] && \alpha \text{ not free in } A^+
\end{aligned}$$

对于带 intersection 和 union type 的 constraint，由于 polar type 的良好性质，我们可以直接将它们化简成更简单 constraint 的 conjunction：

$$\begin{aligned}
\texttt{biunify}(A^+ \sqcup B^+ \preceq C^-) &= \texttt{biunify}(A^+ \preceq C^-) \circ \texttt{biunify}(B^+ \preceq C^-) \\
\texttt{biunify}(A^+ \preceq B^- \sqcap C^-) &= \texttt{biunify}(A^+ \preceq B^-) \circ \texttt{biunify}(A^+ \preceq C^-)
\end{aligned}$$

其中 $\_ \circ \_$ 是 bisubstitution 的复合。对于两侧 head constructor 相同的 constraint，我们直接按类型的结构化简即可，例如：

$$\begin{aligned}
\texttt{biunify}(A^- \to B^+ \preceq C^+ \to D^-) &= \texttt{biunify}(C^+ \preceq A^-) \circ \texttt{biunify}(B^+ \preceq D^-) \\
\texttt{biunify}(\texttt{bool} \preceq \texttt{bool}) &= \mathbf{1}
\end{aligned}$$

其中 **1** 是单位 substitution。如果 constraint 两侧 head constructor 不同，那么 `biunify` 就会失败，因为待求解的 constraint 不可能达成。具体来说，可以证明 `biunify` 有如下性质：

- $\texttt{biunify}(C)$ 对于任意的 C 都停机。
- 如果 $\texttt{biunify}(C)$ 成功，并返回了一个 bisubstitution ξ，那么 ξ 是 C 的最优解。

虽然 `biunify` 的算法本身非常简单，它的正确性却十分难以证明。不过原 thesis 中用了很大篇幅给出了详细的证明。此外，原 thesis 中的 `biunify` 为了处理 equi-recursive type 加入了一些额外的记忆化处理，一侧为 type variable 的 atomic constraint 的处理也比这里呈现的版本更复杂。

6 类型推导

终于，我们可以开始讨论类型推导的主算法本身了 ..... 真的吗？Algebraic Subtyping 中使用了一个非常规的类型系统，即使你很熟悉 ML 与 Hindley Milner，看到它的类型系统的第一眼恐怕还是会一头雾水。然而，这个奇怪的类型系统并不是为了刁难读者而设计的。（在我看来），它背后是关于 ML 的类型系统的一些不太为人知的、非常重要的理解。

6.1 Typing v.s. Type，Let v.s. Lambda

对 Hindley-Milner 的类型推导算法有基本了解的人都知道，大名鼎鼎的 Algorithm W 能算出一个表达式的 principal type，这反过来说明 ML 的类型系统中，每个 typable 的表达式都有 principal type。那么，principal typing 又是什么呢？在 ML 的世界里，principal typing 和 principal type 这两个词似乎常常被混用。然而，正如上个世纪的一些工作 [3] 所指出的那样，principal typing 和 principal type 是两样不同的东西。ML 有 principal type，但没有 principal typing。

考虑类型系统中的一条 judgement $\Gamma \vdash t : A$，typing 指的就是这整条 judgement，而 type 指的就是类型 A。如果考虑两这携带的数据的话，type 中包含的数据只有类型 A，而 typing 中除了类型 A 还有 context Γ。现在，当我们拿到一个类型推导算法，一个很自然的问题是：

这个类型推导算法推导出的是 typing 还是 type?

换言之，context 是这个类型推导算法的输入还是输入？如果只考察算法的形式本身，那么 ML 的 Algorithm W 中，context 是输入，只有类型是输出。然而，我们关心的并不是算法的形式，而是：

在这个类型推导算法中，哪些信息必须从外界给入，哪些信息可以由表达式重构出来?

在 ML 中，这个问题的答案是：

- `let` 引入的变量的类型是必须从外界给入的。
- λ 引入的变量的类型是可以由表达式自身重构出来的。

考虑对一个 closed term $\lambda x.t$ 的类型推导，在递归地对 t 进行类型推导时，我们并不知道 x 的实际类型。推导 t 的类型时，我们会将 x 的类型设成某个 fresh type variable α，这其中并不包含任何类型信息。最终推导出的 x 来自推导

[3] https://www.macs.hw.ac.uk/~jbw/papers/Wells:The-Essence-of-Principal-Typings:slightly-longer.pdf

t 的类型得到的 substitution。所以，x 的类型对 t 的类型推导来说是完全的输出。

再考虑对一个 closed term $\texttt{let}\ x = t\ \texttt{in}\ u$ 的类型推导。我们首先推导 t 的类型，并将它 generalize 后，作为 x 的类型传给 u。这里，x 的类型信息来自 t，对于 u 来说，它是输入。由于整个 `let` 表达式是一个 closed term，generalize 之后 x 的类型不会包含任何 free type variable，所以对 u 进行类型推导的结果不会对 x 的类型有任何更新。因此，x 的类型对 u 是完全的输入。

由 `let` 和 λ 引入的变量的类型的这一不同绝不是 trivial 的。它有一个非常知名的后果。众所周知，ML 中，只有 `let` 引入的变量的类型可以是 polymorphic 的，事实上，这并不是一个任意的限制，而是如下事实的反映：

ML 无法从表达式自身重构出 context 中 polymorphic 的类型

λ 引入的变量的类型必须从函数体中重构出来，所以 ML 中函数不能有 polymorphic 的参数类型。`let` 引入的变量的类型对于 `let` 的 body 来说是纯粹的输入，类型推导不需要从 body 中重构出 let-bound variable 的类型。所以 let-bound variable 可以有 polymorphic 的类型。

6.2 Another ML in the World

在大多数 ML 类型系统和 Hindley-Milner 类型推导算法的 presentation 中，上述区别并未被显式地体现出来。然而，这只是因为在 ML 中将 let-bound variable 与 λ-bound variable 混为一谈“恰好”不会引起什么麻烦，并且语言的 surface syntax 中没有两种不同的 variable。实际上，假如我们把两种 variable 的区别显式地写到 syntax 中：

$$
\begin{array}{rrll}
\texttt{LetVariables} \ni & \hat{x}, \hat{y}, \hat{z} & & \\
\texttt{LamVariables} \ni & x, y, z & & \\
\texttt{Types} \ni & A, B, C & ::= & \ldots \\
\texttt{TypeSchemes} \ni & \theta & ::= & \ldots \\
\texttt{LetContext} \ni & \Pi & ::= & \cdot \mid \Pi, \hat{x} : \theta \\
\texttt{LamContext} \ni & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

那么 ML 的 typing judgement 的形式可以写成 $\Pi \vdash t : [\Gamma]A$。在这种新的 judgement 里，t 左侧的就是所有类型推导时必须从外界传入的信息，t 右侧的就是所有可以从 t 中重构出来的信息。而且，可以证明类型推导给出的 $[\Gamma]A$ 这一整体是 principal 的：这是一个比只有 A principal 更强的结果。

在这个新的类型系统中，typing rules 与 ML 中的没有什么区别。下面是和 context 有关的几条：

$$
\frac{\hat{x} : \theta \in \Pi}{\Pi \vdash \hat{x} : [\cdot]\theta} \qquad \frac{(x : A) \in \Gamma}{\Pi \vdash x : [\Gamma]A}
$$

$$
\frac{\Pi \vdash t : [\Gamma, x : A]B}{\Pi \vdash (\lambda x.t) : [\Gamma](A \to B)} \qquad \frac{\Pi \vdash t : [\Gamma]\theta \qquad \Pi, \hat{x} : \theta \vdash u : [\Gamma]\theta'}{\Pi \vdash (\texttt{let}\ \hat{x} = t\ \texttt{in}\ u) : [\Gamma]\theta'}\ (let)
$$

上面的 syntax 中没有给出 polymorphic type scheme θ 的定义，一般来说，θ 的定义与相关的 typing rule 是这样的：

$$\theta ::= \ \forall \bar{\alpha}.A$$

$$\frac{\Gamma \vdash t : A \qquad \bar{\alpha} \notin FreeTypeVars(\Gamma)}{\Gamma \vdash t : \forall \bar{\alpha} A} \qquad\qquad \frac{\Gamma \vdash t : \forall \bar{\alpha}.A}{\Gamma \vdash t : A[\bar{\alpha} \mapsto \bar{B}]}$$

引入 polymorphic type scheme 的规则中有一个 $\bar{\alpha} \notin FreeTypeVars(\Gamma)$ 的限制条件。如果没有这个限制条件，那么 generalization 就可能会错误地“忘记”某个 type variable 在 Γ 和 A 中必须指代同一个类型的事实。然而，将 context 与 type 合在一起变成一个 typing $[\Gamma]A$ 后，我们可以给出一个更优雅的做法。根据 type substitution lemma，如果 $\vdash t : [\Gamma]A$，那么 $\vdash t : [\rho(\Gamma)]\rho(A) = \rho([\Gamma]A)$。也就是说，我们可以随意 instantiate 这个 universally quantified 的 typing。所以整个 typing $[\Gamma]A$ 中的所有 type variable 都可以看成是 universally quantified 的。

基于上面的观察，我们可以进一步简化类型系统，得到 Algebraic Subtyping 最终采用的类型系统。在这个类型系统中，polymorphic type scheme 的定义与相关 typing rule 是：

$$\theta ::= [\Gamma]A \qquad\qquad \Pi ::= \cdot \mid \hat{x} : [\Gamma]A$$

$$\frac{\Pi \vdash t : [\Gamma]A}{\Pi \vdash t : \rho([\Gamma]A)}$$

下面，让我们来考察这个新类型系统中最离奇的部分：let-bound variable 的类型（在 Π 的定义中）中现在包含了一个 context，而且不同 let-bound variable 的 context 不一定相同！想要理解这个定义，就必须将思考方式从“type”转移到“typing”：从 typing 的角度来看，context 是一个 term 携带的信息的一部分。所以每个 term、每个 variable 都可以携带不同的 context，因为那是属于它们各自的信息。在 ML 的类型系统中，几乎所有规则都要求各个子表达式共享相同的 context。在“type”的视点下，context 是整个 typing judgement 共享的，每个子表达式都是在同一个 context 下被考察的。而在“typing”的视点下，子表达式的 context 必须相同是将子表达式组合成更大表达式时，它们之间必须达成的一种共识。就像函数调用 $t\ u$ 中，t 的参数类型和 u 的类型必须相同一样。

于是我们得到了 ML 的类型系统的一个等价的（原 thesis 中有具体的和原版 ML 互相 encode 的方法）形式。而且这个新的类型系统有如下的优点：

- 所有 typing judgement 都有一个统一的形式 $\Pi \vdash t : [\Gamma]A$。t 的右边不再会有 type 和 type scheme 两种情况。
- let-bound variable 的 context Π 中都是形为 $[\Gamma]A$ 的 type scheme，lambda-bound variable 的 context Γ 中都是非 polymorphic 的类型 A。context 中不再会出现 type scheme 和 type 混合的情况。

- 一个 type scheme 中的所有 type variable 都是自动 universally quantified 的（从而 let context Π 中不会有任何 free type variable，所以它的存在没有影响）。不再需要 $\bar{\alpha} \notin FreeTypeVars(\Gamma)$ 的 freshness condition，甚至连 generalization rule 本身都能省去。
- 新的类型系统直接在形式上反映了类型推导的输入输出。通过区分 let-bound context 和 lambda-bound context，在用于更复杂的类型系统的类型推导时（比如 Algebraic Subtyping）更加便利。

6.3 与 subtyping 整合

接下来，我们往上面的“另一个 ML”中添加 subtyping。对于 let context Π，它当中包含的都是没有 free type variable 的 type scheme，所以把不需要做太多处理。由于 lambda context Γ 现在成为了类型信息的一部分，我们需要考虑它与 subtyping 的互动。由于 Γ 就是一串类型，我们可以为它们定义一个 pointwise 的 subtyping 关系：

$$(x_i : A_i) \preceq (x_i : B_i) \Leftrightarrow A_i \preceq B_i (\forall i)$$

接下来我们可以把 subtyping 拓展到完整的 typing $[\Gamma]A$ 上。由于 Γ 中的变量与函数的参数对应，代表 consumer，很自然地，我们可以用与函数参数类似的 subtyping 规则来处理它：

$$[\Gamma]A \preceq [\Gamma']A' \Leftrightarrow \Gamma' \preceq \Gamma \wedge A \preceq A'$$

如果把 Γ 看成逻辑中的假设，那么这条规则可以解读成“假设越弱越好，结论越强越好”。由于 $[\Gamma]A$ 是 polymorphic 的，它包含的 type variable 都是 universally quantified 的，我们也应当在它之上定义 subsumption：

$$[\Gamma]A \preceq^{\forall} [\Gamma']A' \Leftrightarrow \exists \rho, \rho([\Gamma]A) \preceq [\Gamma']A'$$

让 $\rho = \mathbf{1}$，可以看到 $\preceq^{\forall}$ 包含了 $\preceq$。所以，我们可以用一条类型规则：

$$\frac{\Pi \vdash t : [\Gamma]A \qquad [\Gamma]A \preceq^{\forall} [\Gamma']A'}{\Pi \vdash t : [\Gamma']A'} \ (sub)$$

来同时处理 generalization、instantiation 和 subtyping！如果在 $\Gamma \preceq \Gamma'$ 的 subtyping 中假如“变量越少越强”（例如 $\Gamma \preceq \Gamma, x : A$）的规则，那么连（lambda context 的）weakening 规则也可以用 (sub) 来表达！从这个例子中，可以看出我们的新类型系统以及 subtyping 本身的强大与灵活。

接下来，让我们考虑新类型系统与 polarity 的互动。依然将 Γ 中的类型与函数的参数类型作类比，polarized 的 Γ 中的类型应该都是 negative 的。所以，一个 polarized 的 typing 的形式应该是 $[\Gamma^-]A^+$。这里可以发现，let context Π 中的 polymorphic type scheme 是没有 polarity 的。所以如果我们采用原版 ML 作为类型系统的基础，那么 type scheme 与 type 混合的单一 context 与 polarity 的互动会有多么丑陋是很容易想象的。

最后，我们可以把 $\sqcup$ 和 $\sqcap$ 拓展到 typing 上：

$$[\Gamma]A \sqcup [\Gamma']A' = [\Gamma \sqcap \Gamma'](A \sqcup A')$$
$$[\Gamma]A \sqcap [\Gamma']A' = [\Gamma \sqcup \Gamma'](A \sqcap A')$$

当 Γ 和 Γ' 中包含的变量不一致时，$\sqcup$ 对应变量集合的并集，$\sqcap$ 对应交集。注意到原版 ML 的 type scheme（$\forall \bar{\alpha}.A$）上很难定义出 $\sqcup$ 和 $\sqcap$，所以这又是一个新类型系统的优势的体现。

6.4 类型推导主算法

终于，经过超长篇幅的铺垫，我们来到了 Algebraic Subtyping 中的类型推导主算法。然而，parametric polymorphism + subtyping 下的 principal typing 这个数十年的 open problem 已经通过 algebra 优先的类型构造、polar type 的限制、bisubstitution 与 biunification 以及新的类型系统呈现方式被一步步解决了。现在，principal type inference，我们的终极目标，已经是一颗低垂的果实了。下面是 type inference 算法的一部分规则，形如 $\Pi \rhd t : [\Gamma^-]A^+$，输入与输出被清晰地放在了 t 的左右两侧。关于 record 和 boolean 的部分被略去了，你可以在原 thesis 中找到完整的算法：

- let-bound variable：
$$\frac{(\hat{x} : [\Gamma^-]A^+) \in \Pi}{\Pi \rhd \hat{x} : [\Gamma^-]A^+}$$
由于 Π 中不包含 free type variable，所以它永远不会被喂给任何 substitution，因此我们不需要像 Algorithm W 中那样把 $[\Gamma^-]A^+$ 中的 type variable 替换成新的 fresh type variable。

- λ-bound variable：
$$\frac{\alpha \text{ fresh}}{\Pi \rhd x : [x : \alpha]\alpha}$$
由于 λ-bound variable 的类型是从它的使用方式中推导出来的，看到这个 variable 自身的时候我们暂时做不了什么。

- λ：
$$\frac{\Pi \rhd t : [\Gamma^-]B^+}{\Pi \rhd \lambda x.t : [\Gamma^- - x](\Gamma^-(x) \to B^+)}$$
如果 Γ^- 中不包含 x，说明 x 在 t 中没有出现，$\Gamma^-(x)$ 直接填个 fresh type variable 或者 top type $\top$ 即可。

- 函数调用：
$$\frac{\Pi \rhd t : [\Gamma_1^-]A_1^+ \qquad \Pi \rhd u : [\Gamma_2^-]A_2^+}{\Pi \rhd t\ u : \xi([\Gamma_1^- \sqcap \Gamma_2^-]\alpha)}$$
其中 $\xi = \texttt{biunify}(A_1^+ \preceq A_2^+ \to \alpha)$，$\alpha$ fresh。

- let：

$$\frac{\Pi \rhd t : [\Gamma_1^-]A^+ \qquad \Pi, \hat{x} : [\Gamma_1^-]A^+ \vdash u : [\Gamma_2^-]B^+}{\Pi \rhd (\texttt{let}\ \hat{x} = t\ \texttt{in}\ u) : [\Gamma_1^- \sqcap \Gamma_2^-]B^+}$$

可以看到，这些规则都是非常直白的：类型系统的结构与 biunification 已经完成了绝大多数的工作。最后，可以证明这个类型推导算法的确是：

- Sound，即如果 $\Pi \rhd t : [\Gamma^-]A^+$，那么 $\Pi \vdash t : [\Gamma^-]A^+$。
- 且 Complete，即如果存在 $[\Gamma_0]A_0$ 使得 $\Pi \vdash t : [\Gamma_0]A_0$，那么 $\Pi \rhd t : [\Gamma^-]A^+$，类型推导会成功。
- 且 Principal 的，即如果 $\Pi \rhd t : [\Gamma^-]A^+$，那么对于任意的 $[\Gamma']A'$，如果 $\Pi \vdash t : [\Gamma']A'$，$[\Gamma^-]A^+ \preceq^\forall [\Gamma']A'$。

7 原 thesis 中的其他内容

原 thesis 中，在证明了类型推导的性质后，探讨了如何高效地实现类型推导中的各种操作，以及如何化简一个 type scheme，得到诸多等价（via $=^\forall$）的 syntax 中最简洁的那一个。原 thesis 中的这一部分内容主要使用了自动机理论。原 thesis 中从每个类型和 type scheme 的 syntax 构建出对应的自动机，并证明了如下的 Representation Theorem：

$A =^\forall B$，*当且仅当* A *和* B *对应的自动机接受同一门 regular language*。

于是，所有自动机理论中用于简化自动机的算法都可以用于简化类型，而且它们的正确性能通过 Representation Theorem 自动得到保障。原 thesis 还通过自动机实现了更高效的 biunification 以及 subsumption 检查。对于将 Algebraic Subtyping 投入实际使用来说，这些内容无疑是非常重要的。但是我个人认为直接在 syntax 上操作的算法以及 heuristic 的化简算法也可以达成相似的目的。所以尽管原 thesis 中自动机的方法有许多优越性，*我认为*除了 Representation Theorem 这一想法本身以外的内容不太有趣。所以这里我并不打算介绍它们。而且我也并不认为我对这部分内容有足够的理解。

8 一些个人评论

下面的内容完全是我个人的观点，与原 thesis 没有任何关系。在我看来，原 thesis 的作者 Stephen Dolan 实在是个天才，他用一种非常 principal 的方式解决了一个多年的 open problem，而他的方法的“principal”性意味着它们有被迁移到其他类型系统的设计中，解决更多问题的巨大潜力。所以，我认为 Algebraic Subtyping 是一篇伟大的工作，值得更多的注意（否则我也不会写这么长的文档来介绍它了）。

那么，Algebraic Subtyping 的出现是否意味着 subtyping 可以完全替代 ML，成为新的 FP 语言的基础类型系统了呢？很遗憾，我并不这么认为。问题并不在 ALgebraic Subtyping，而在于 subtyping 本身。Subtyping 作为“类型上的代数

结构”的解释固然非常优雅，但当把类型和表达式的互动放入考量时，subtyping 的性质依然不能令人满意。我认为，目前的 subtyping 系统接受了*太多的* term。是的，太多、而非太少。由于 subtyping 把类型错误的发生延后了（通过 $\sqcup$、$\sqcap$、$\top$ 和 $\bot$），许多被包裹在了函数内的、不应该被接受的 term 就需要被接受，而这又会反过来使得类型系统的设计必须变得更复杂。对 equi-recursive type 的强需求在我看来就是这样的一个例子。

此外，subtyping 与其他语言功能的兼容性也依然是大问题。从一个比较抽象的角度来看，equality 是一种更普遍、更简单的结构，所以基于 equational reasoning 的类型系统与大部分语言功能都没有兼容性问题。而采用了 subtyping 这种更复杂的类型结构，就意味着所有新的语言功能也必须提供类似的复杂结构。一个典型的例子是可变的 reference cell $\texttt{ref}(A)$，它对 A 是 invariant 的，subtyping 对 A 不适用，因为 reference cell 即可读又可写。Algebraic Subtyping 的框架中就无法支持 invariant 的 type constructor，而本质上这是因为 invariant 的 type constructor 上根本就没有任何 subtyping 需要的 order theoretic 的结构。Algebraic Subtyping 中提出的解决办法（一个更久以前就存在的办法）是把 A 拆分成用于读和用于写的两个类型。这样能解决 variance 的问题、允许类型表达更精细的信息。然而，它也会使类型变得更长、更难理解：传统 subtyping 系统存在的、Algebraic Subtyping 试图解决的问题。

所以，在我看来，subtyping 的时代还未到来，还有很长的路要走，有很多可用性的问题有待实际语言实现的验证。不过，Algebraic Subtyping 填上了缺失已久的、优良 meta theory 性质的空缺。希望这能引发对 subtyping 的一波进一步的可用性提升与研究。