Notes for Inira's SepLogic

Bowen Zhang

2020/04/05

1 Induction

1.1 approaches to programs verification

1.1.1 程序验证

1. specification用来描述程序要计算什么，而不是如何计算

2. “一个程序是否满足一个specification”是不可判定的，因此验证工具需要能够引导验证的过程

3. 程序验证的传统方法

   - **VGC——Verification Condition Generator**
     将程序进行拆解，对每个语句验证是否满足对应的spec，如果每个语句都可以满足对应的输入和输出，那么说明程序可以满足针对整体的spec。

   - **Interactive Theorem Prover (proof assistant)**
     可以将“程序需要满足的spec”视为一个定理，并采用交互式定理证明器对其证明，但其中有两个难点：
     - 将源语言进行编码，使其成为Programming Language
     - 有关程序的推理，要建立在数学逻辑上

     因此建立一个数学逻辑和PL之间的桥梁，是一个挑战.

1.1.2 交互式证明

旨在构造一个公理化的，对PL有逻辑上的语法和语义定义的，定理证明器。理论上，对于给定的程序，可以证明任意spec是否能满足。而实际

中，由于程序语法的显式定义十分笨拙，因此很难对其进行操作。尽管如此，这种方式还是在底层程序的验证上扮演了十分重要的角色。

人们也可以在定理证明器自带的逻辑系统中，开发编程语言(PL)。而浅度嵌入（**shallow embedding approach**）的理念，就是将**编程语言的程序**与**逻辑的程序**联系起来。这个理念实现起来有三种方式：

- 在逻辑系统中编写程序，将其转换为常规的PL
- 将源代码编写的程序，通过反编译，转换为逻辑程序
- 写一遍传统的PL，并写一遍逻辑程序

然而，所有基于**shallow embedding approach**的实现都面临着一个问题，那就是：解决PL和逻辑上的差异，尤其是对**偏函数和可修改状态**的处理。

定义 1. 偏函数（*partial function*）：*一个偏函数$Pfun[A,B]$是一个一元函数，接收一个类型为A的参数x，返回类型为B的值，但x的选取可以不覆盖A的整个定义域。也就是说，偏函数只处理了定义域中的一个子集。*

该篇论文的方法是对给定的源代码，生成有关代码行为的逻辑命题。换句话说，就是对给定程序所生成的spec，为该spec生成可以使其满足的，一个充分的前提。该前提则是可以被交互证明的。**通过不显式地表达程序语法，可以避免深度嵌入的困难；而通过不依赖逻辑来表示程序，又可以避免浅度嵌入的困难。**

1.1.3 特征公式

Characteristic formulae是一种关于程序行为的描述。它最初并应于与进程验算（一个并行计算的模型）。那是特征公式是采用时态逻辑，根据特定语法来生成描述进程的命题。它的最基本的结果就是：两个进程等价**iff**两个特征公式等价。因此以特征公式来证明两个进程之间的相等或不等关系。

最近Honda等人将特征公式由进程逻辑转向了程序逻辑，以PCF（Programming Language for Computable Function）为研究目标，PCF可以被理解为简化版的ML（如Caml或SML）。Honda等人给出了一个生成给定PCF程序的“total characteristic assertion pair”的算法，这个“完全特征断言对”就是该程序的最弱前置条件，和最强后置条件。注意，PCF程序

并没有spec作为注解，也没有循环不变量，因此最弱前置条件的算法与常规有差异。而有关一个程序的普适spec的概念更为古老，起源于Hoare逻辑的完全性证明。

Honda工作的创新之处就在于，一个PCF程序的普适spec可以不依赖编程语言的语法来表示。Honda等人表明，特征公式可以通过证明“普适的spec 逻辑蕴含目标的spec”，来证明一个给定的程序满足其对应的spec。因此证明过程可以在不需要引用变成语言的语法的情况下进行。但这种idea面临着一个主要的问题，就是“完全特征断言对”的spec语言，是由专门的逻辑系统表示的，其中变量代表了PCF的值（包括了非终止函数），也因此等价被解释成为观测意义上的等价，即对比变量是否相同。由于这种特殊的逻辑系统没法轻易编码，转换为标准逻辑系统。并且实现一个新逻辑系统之上的定理证明器需要投入大量的时间。因此Honda等人的工作只停留在理论层面，并未产生一个有效的程序验证工具。

该论文的工作重新发掘了特征公式，同时寻找到一种方法来增强嵌入的深度。在这种方法中，程序语法被显式地表示在定理证明程序中。**该论文通过构建逻辑公式，来捕获能够进行深度嵌入的推理，并且不需要在任何时候以程序语法表示**。从某种意义上，这种方法可以被理解为在深度嵌入之上构建了一个抽象层，隐藏掉技术细节却保留了优点。与Honda等人不同，该论文建立的特征公式是用标准高阶逻辑表示的，也因此能通过特征公式建立一个实用的程序验证工具。

1.1.4 特征公式

一个项t（term t）的特征公式被记为$[\![t]\!]$。特征公式与Hoare三元组之间有很深的联系。Hoare三元组$\{H\}\,t\,\{Q\}$（这里即分离逻辑中的Hoare三元组）断言了：如果一个堆可以满足谓词H时，执行项t终止后返回值v，那么更新后的堆可以使得$(Q\ v)$满足。注意后置条件Q需要输出值和输出堆同时满足。当t的类型为τ时，前置条件H具有类型$Heap \to Prop$，后置条件Q的类型则为$\langle\tau\rangle \to Heap \to Prop$，其中$Heap$是堆的一个类型，而$\langle\tau\rangle$是Coq中的类型，与ML中的$\tau$类型相对应。

特征公式$[\![t]\!]$是使得$[\![t]\!]\ H\ Q$与$\{H\}\,t\,\{Q\}$逻辑等价的谓词，但是特征公式与Hoare三元组具有本质上的区别：三元组$\{H\}\,t\,\{Q\}$是一个三元关系，它的第二个参数需要调用语法来描述项t；相对的，$[\![t]\!]\ H\ Q$是一个逻辑命题，它以标准的高阶逻辑连接词表示，如$\wedge, \exists, \forall$和$\Rightarrow$，并不依赖于项t的语法。而

且Hoare三元组的推导，需要建立在Hoare逻辑特有的推导规则上，然而特征公式只需要运用基本的高阶逻辑就可以进行推理，不需要额外引入推导规则。

本章接下来部分，该论文展示了构造特征公式的核心思想，重点讨论了**let-binding**的处理，包括了函数的应用以及定义应用，并解释了如何处理可以实现局部推理的Frame Rule。

1.1.5 Let-binding

为了评估一个形如"$\textbf{let}\ x = t_1\ \textbf{in}\ t_2$"的项，我们第一步需要分析它的子项$t_1$，接下来带着输出的结果再分析$t_2$。为证明表达式能接受$H$作为前置，$Q$为后置条件。我们需找到$t_1$的一个有效的后置条件$Q'$，该后置条件描述了$t_1$执行后和$t_2$执行前的内存状态，并能接受由$t_1$所生成的结果$x$。因此，可以使用（$Q'\ x$）来表示$t_2$的前置条件。下面是上述内容对应的Hoare规则：

$$\frac{\{H\}\, t_1\, \{Q\} \quad \forall x.\, \{Q'x\}\, t_2\, \{Q\}}{\{H\}\, (\textbf{let}\ t_1\ \textbf{in}\ t_2)\, \{Q\}}$$

而特征公式对let-binding的构建如下：

$$[\![\textbf{let}\ x = t_1\ \textbf{in}\ t_2]\!] \ \equiv\ \lambda H.\lambda Q.\, \exists Q'.\, [\![t_1]\!]\ H\ Q' \ \wedge\ \forall x.\, [\![t_2]\!]\ (Q'\ x)\ Q$$

这个公式与Hoare规则很相似，唯一不同之处在于：在特征公式中，中间的后置条件Q'是明确以存在量词来引入的，而这个量词在Hoare逻辑的规则中是被隐式掉的。之所以能够对未知spec做量化，是建立在高阶逻辑的特性上的，这个特性在该论文的工作中起到了很关键的作用。这种方法与传统的方法形成了鲜明的对比，后者必须在验证源程序时，体现出中间的spec，甚至是循环不变量。

为了使证明更加易读，作者介绍了一个关于特征公式的符号系统。比如对于let-binding，他定义了

$$(\textbf{let}\ x = \mathcal{F}_1\ \textbf{in}\ \mathcal{F}_2) \ \equiv\ \lambda H.\lambda Q.\, \exists Q'.\, \mathcal{F}_1\ H\ Q' \ \wedge\ \forall x.\, \mathcal{F}_2\ (Q'\ x)\ Q$$

粗体的关键词对应了逻辑公式中的符号，如（**let** ... **in**...），而非粗体则对应于编程语言语法中的实际构造,如（$x = \mathcal{F}_1$ 或 $\mathcal{F}_2$）。特征公式的生成，就可以归结为对编程语言关键词的重新解释。

这样做的结论就是，特征公式可以像源语言一样，准确且优美地进行表达。用“$[\![t]\!]\ H\ Q$”就可以直接给人们呈现，由源代码t后面接着前/后置

断言的表示方法。注意这种表达方式可以不仅应用于顶层的项t，还可以在证明t正确性的过程中，应用于所有t的子项。

1.1.6 Frame Rule

“局部推导（local reasoning）”指的是能够只验证与内存相关的部分代码的推理方式，它涉及了代码的执行。运用局部推导时，所有没被提及的内存单元都会被默认为保持不变。而局部推导的理念，在分离逻辑中的Frame rule得到了优雅的表示。Frame rule表明了如果程序将一个由谓词H_1描述的完整的堆，转移到了由谓词H_1'描述的堆,那么对于任意的堆谓词H_2，这个程序同样也可以将形如$H_1 * H_2$的堆，转移为$H_1' * H_2$的堆。

比如将内容加一函数**incr**，应用于内存中的地址l。假设$(l \hookrightarrow n)$描述了单堆，那么函数**incr**的应用使一个形如$(l \hookrightarrow n)$的堆转换为$(l \hookrightarrow n+1)$的堆。而通过frame rule，人们可以将函数**incr**的推导，扩大为由一个形如$(l \hookrightarrow n)*(l' \hookrightarrow n')$的堆，转移至$(l \hookrightarrow n+1)*(l' \hookrightarrow n')$的堆，其中分离合取则断定了$l$与$l'$之间互不相等。分离合取的运用，为我们提供了描述除$l$内容加一之外，系统其它属性的可能。

Frame rule以Hoare三元组的形式定义如下：

$$\frac{\{H_1\}\, t\, \{Q_1\}}{\{H_1 * H_2\}\, t\, \{Q_1 \star H_2\}}$$

其中，（$\star$）和（$*$）的区别在于后置条件中，$\{Q_1 \star H_2\}$被用来描述$\lambda x.\,(Q_1\, x) * H_2$,这里$x$描述了输出的值，$(Q_1\, x$描述了输出的堆。

为了将frame rule和特征公式联系在一起，作者引入了一个谓词**frame**（即为课件中的**local**），该谓词的定义为：要证明命题“**frame** $[\![t]\!]\ H\ Q$”，可以通过将H拆解为$H_1 * H_2$,将Q拆解为$Q_1 \star H_2$，进而证明$[\![t]\!]\ H_1\ Q_1$成立，其形式化的定义如下：

$$\textbf{frame}\,\mathcal{F} \;\equiv\; \lambda HQ.\,\exists H_1 H_2 Q_1.\begin{cases} H = H_1 * H_2 \\ \mathcal{F}\ H_1\ Q_1 \\ Q = Q_1 \star H_2 \end{cases}$$

frame rule并不能支持语法推导，也就是说当需要运用该规则时，不能只通过t的形式来猜测。然而作者想通过系统化的方式，直接从源语言得出它的语法表示，进而生成特征公式。但因为推理时在哪插入**frame**谓词是不

固定的，因此作者在每一个特征公式上都加入了**frame**谓词。比如之前定义的let-binding就更新为：

$$(\textbf{let } x = \mathcal{F}_1 \textbf{ in } \mathcal{F}_2) \equiv \textbf{frame} \, (\, \lambda H. \lambda Q. \, \exists Q'. \, \mathcal{F}_1 \, H \, Q' \, \wedge \, \forall x. \, \mathcal{F}_2 \, (Q' \, x) \, Q \,)$$

这个带有侵略性的策略，使得我们在推理的任何时候都可以运用frame rule。而如果不需要frame rule的话，它也可以直接被化简掉。$\mathcal{F} \; H \; Q$永远都是**frame**$\mathcal{F} \; H \; Q$证明时的一个充分的前提。

作者描述的用来处理frame rule的谓词，实际上是广义的。它同时可以在处理复合语句、增强前置条件、弱化后置条件，以及为模拟垃圾回收而允许的内存单元丢弃。

1.1.7 对类型的解释

高阶逻辑能自然地描述基础值（如纯函数化的列表），甚至链表的数据结构在逻辑上的描述和在PL中的能够完美匹配。然而验证和推理程序的函数时，仍需要额外注意。甚至，PL中的函数，并不能直接像逻辑中的函数那样进行直接的表示，究其原因是因为两者的差异：PL中的函数可能出现分歧（字符串类型的函数引入整形参量）或者崩溃（引用null指针），而逻辑中的函数永远都会终止。为了解决这个差异，作者引入了一个新的数据类型**Func**来表示函数。**Func**这个类型在特征公式中被描述为一个抽象的数据类型。在可靠性证明中，一个类型为**Func**的值被解释为源语言中函数对应的语法表示。

另一个是对指针的特殊操作，也就是将Caml中的值对应到Coq中的值。当运用特征公式进行推理时，每个内存空间的类型和内容都通过堆谓词被显式地进行描述。因此就不需要指针携带着所指向内存单元的类型。也因此所有的指针在逻辑中，就都可以通过一个抽象的数据类型来描述，作者将这种类型定义为**Loc**。在可靠性证明中，一个**Loc**类型的值被解释为一个存储地址。

将Caml中的类型形式化转换为Coq中的类型，是通过一个操作符$\langle \cdot \rangle$，它将多有的箭头类型指向**Func**类型，并将所有的引用类型映射到**Loc**类型。一个Caml中类型为τ的值，就可以被表示成Coq中类型为$\langle \tau \rangle$的值，操作符$\langle \cdot \rangle$的定义如下：

$$
\begin{aligned}
\langle int \rangle &\equiv \mathbf{Int} \\
\langle \tau_1 \times \tau_2 \rangle &\equiv \langle \tau_1 \rangle \times \langle \tau_2 \rangle \\
\langle \tau_1 + \tau_2 \rangle &\equiv \langle \tau_1 \rangle + \langle \tau_2 \rangle \\
\langle \tau_1 \to \tau_2 \rangle &\equiv \mathbf{Func} \\
\langle \mathbf{ref}\, \tau \rangle &\equiv \mathbf{Loc}
\end{aligned}
$$

一方面，ML的类型系统对于将Caml的值直接映射为Coq中的值非常有帮助；另一方面，这种类型定义也是带有限制的：有很大数量的程序是正确的，但无法在ML中进行验证。特别是ML的类型系统不支持 空指针调用 和 强制类型转换 的程序。这些问题即使在无视类型安全的情况下，依旧很难处理。然而它们的正确性可以运用程序的正确性验证来证明。因此，作者对Caml引入了空指针和强制类型转换。并运用特征公式验证空指针永远不会被调用，以及从内存中读取的数据永远是预期的类型。

然而，这个方法的正确性并不能被直接且完整地验证。一方面，特征公式是由类型程序生成的；另一方面，空指针和强制类型转换有可能使类型推导出现问题。为了验证特征公式的可靠性，作者引入了一个新的类型系统**weak-ML**。这个类型系统并不具有可靠性，但是它能够携带用来生成特征公式的所有类型和信息和不变量，并证明它们是合理的。

简单来讲，weak-ML对应着一个更为松散的ML版本，即：不追踪指针或函数的类型，不对释放和应用函数的类型施加任何约束。因此，将Caml类型解释为Coq类型实际上需要两步：将Caml类型转译为weak-ML类型，将这个weak-ML类型的值转换为Coq类型。