堆逻辑的 FO 完备程序验证

Adithya Murali，威斯康星大学麦迪逊分校，美国HRISHIKESH BALAKRISHNAN， 伊利诺伊大学厄巴纳-香槟分校计算机科学系，美国AARON COUNCILMAN， 伊利诺伊大学厄巴纳-香槟分校计算机科学系，美国P. Madhusudan， 伊利诺伊大学厄巴纳-香槟分校计算机科学系，美国

arXiv:2601.06719v1 [cs.LO] 10 Jan 2026

我们开发了首批两种具有隐式堆片段且支持一阶完备程序验证的堆逻辑。一阶完备性这一概念提供了理论保证：当递归定义被解释为不动点定义（而非最小不动点）时，所有有效定理最终都能由系统证明。我们开发的逻辑包括框架逻辑（框架逻辑）和一种分离逻辑（*SL-FL*），后者采用了受框架逻辑启发的替代语义。我们展示了如何为框架逻辑生成验证条件，并通过量词实例化和 SMT 求解器实现适用于一阶完备推理的方法。我们还展示了如何将 *SL-FL* 转换为框架逻辑，从而实现一阶完备推理。我们实现了可运用该技术的工具，并通过一组用于操作数据结构的基准测试，展示了这些逻辑的表达能力以及该验证技术的有效性。

1 引言

对会以破坏性方式操作堆的程序进行自动化验证，仍然是一个颇具挑战性的开放问题。分离逻辑 [50, 51, 61] 已成为一种流行的规约逻辑，可用于表达堆中结构的性质。尽管分离逻辑广泛应用于交互式定理证明环境，但分离逻辑推理的稳健自动化尚未取得同等程度的成功 [18]。

分离逻辑自动化面临的一个问题是，其中若干方面本质上属于二阶，因此很难使用 SMT 求解器（尤其是针对可判定逻辑的 SMT 求解器）对其进行自动推理。具体而言，分离逻辑中的魔杖(−∗)会对满足某种性质的任意堆进行量化，这本质上是二阶的 [10]。在竞赛（如 SLComp [62]）中评估的分离逻辑推理工具很少支持魔杖运算符，这也印证了对其进行推理的难度。

即使避开魔杖（例如使用不完备的最强后置条件计算，如 [7]），要表达分离合取、合取和析取等性质，也需要对位置集合进行量化；而在证明蕴含或蕴涵时，则会得到同时包含对集合进行存在量化和全称量化的公式。这些公式难以使用一阶求解器进行推理，而且正如本文所示，会导致不完备性。

因此，对于任何表达能力强的堆逻辑，自动程序验证技术的最先进水平始终是不完备的——也就是说，验证条件的有效性不仅不可判定，甚至不是递归可枚举的。（有些堆逻辑是可判定的，但它们通常在表达能力上受到严格限制 [25, 56–58]。）换言之，这些逻辑不存在完备的证明系统，也不可能存在可靠的算法过程，使其对所有有效的验证条件都能停机并判定其有效（即使允许它在无效条件上不终止）。Viper [40] 等系统对魔杖采用了多种启发式方法，并生成包含集合和映射量词以及触发器的验证条件，然后将其发送给可满足性模理论

作者联系信息：AdithyaMurali，adithya5@illinois.edu，威斯康星大学麦迪逊分校，美国威斯康星州麦迪逊；Hrishikesh Balakrishnan，hb19@illinois.edu，伊利诺伊大学厄巴纳-香槟分校计算机科学系，美国伊利诺伊州厄巴纳；Aaron Councilman，aaronjc4@illinois.edu，伊利诺伊大学厄巴纳-香槟分校计算机科学系，美国伊利诺伊州厄巴纳；P. Madhusudan，madhu@illinois.edu，伊利诺伊大学厄巴纳-香槟分校计算机科学系，美国伊利诺伊州厄巴纳。

求解器；对于此类逻辑，该求解器本质上是不完备的 [22]。此外，不完备性在实践中也有所体现（参见近期论文 [18]，该论文利用 Viper 框架研究了分离逻辑中各种算法的不完备性）。

不完备性源于多种原因，本文稍后将从形式上识别其中若干种。首先，内在的二阶量化会导致不完备性，尤其是集合量化。其次，基于堆的有限模型 定义的堆逻辑很快就会变得不完备，即使是一阶逻辑也不例外，这是有限模型理论中的基本不完备定理所致 [37]。第三，堆逻辑通常需要递归定义，且这些定义具有最小不动点语义；即使模型是无限的，这也会导致不完备性，因为它们能够编码整数的加法和乘法，而由哥德尔不完备定理可知，不完备性随之而来 [19]。第四，程序的堆逻辑需要允许背景类型理论的组合，其中包含线性算术等模型，以便定义典型的数据结构（例如二叉搜索树）；带量词理论的组合也会导致不完备性。

这些因素使得表达能力强的堆逻辑不可避免地不完备。因此，针对堆逻辑有效性的实用算法技术无法满足任何关于有效性的理论标准，例如完备性；在基准测试中的经验有效性是评判某项技术成效的唯一方式。最重要的是，我们没有理论基础能够保证：即使给予无限资源，算法技术不会 令人尴尬地无法证明一个简单的验证任务。

一阶完备性的概念 本文提出一个大胆的问题：尽管存在上述挑战，我们能否定义一种完备性概念，以此作为衡量堆逻辑算法技术的标准？这种完备性保证的动机在于，它将为打造比当前最先进水平更可预测的程序验证工具铺平道路。受近期文献中相关成果的启发（这些成果表明，在类似领域中，一阶逻辑完备过程既实用又稳健），我们提出将一阶完备性 作为堆逻辑的一种完备性概念。

堆逻辑不完备性最根本且无可回避的原因，是其中包含递归定义。给定关系（或函数）的递归定义 $f(x) = _{lfp}\ \rho(x,\ f)$，$f$意在解释为满足给定方程的最小不动点。这类定义超出一阶逻辑的表达能力，而且还会导致逻辑如上所述变得不完备。一种简单且在实践中有效的降低这种表达能力的方法，是将递归定义改为不动点 定义 $f(x) = \rho(x,\ f)$，此时我们允许任何满足该方程的谓词/函数 f，而不只是最小不动点。

如果验证条件相对于逻辑中递归定义的这种不动点语义有效，我们就称带有堆逻辑规约的验证条件具有一阶完备 性。之所以称为一阶完备性，是因为这类方程可以用一阶逻辑表示。请注意，如果某个定理相对于满足递归定义的任意不动点都有效，那么该定理相对于最小不动点语义同样有效。

近期文献中的研究结果指出，一些基于展开递归定义、用于程序验证的现有实用、高效且稳健的技术（“展开并匹配”启发式方法、“折叠/展开”操作）实际上是一阶完备的。具体而言，有研究指出，对于带有递归定义的一阶逻辑某些片段，所谓的 自然证明 具有一阶完备性 [38]。另有研究指出，用于 Haskell 程序的 Liquid Haskell [72] 和用于 Scala 程序的 Leon/Stainless [8, 67] 中，基于代数数据类型（ADT）的函数式程序验证技术也具有一阶完备性 [42]。在后一种情形中，为了用一阶逻辑表达性质，标准的代数数据类型全集被其一阶公理化版本取代（该版本允许非标准模型），并且已经证明，实践中使用的技术在这种建模方式下是一阶完备的

在这种建模方式下。请注意，最小不动点语义与不动点语义之间存在差距，而这些框架需要借助归纳引理来弥合这一差距。目前的工具要求用户提供这些引理，但自动合成这些引理仍是一个活跃的研究方向 [34, 41]。以上讨论引出了本文提出的核心问题：

> 是否存在具有递归定义且表达能力强的堆逻辑，并且拥有一阶完备的有效性判定过程？

更准确地说，我们希望有效性判定过程是可靠的——如果它报告某个定理有效，那么该定理在定义的最小不动点语义下确实有效；同时也是一阶完备的——如果某个定理在递归定义的不动点语义下有效，那么该过程保证会终止并证明该定理有效。请注意，如果某个定理在最小不动点语义下有效，但在不动点语义下无效，则不要求工具终止。

我们关注的堆逻辑还应具备以下理想性质：

- 该逻辑应具有隐式堆片段。分离逻辑首创了一项基本设计原则：每个公式 α 都隐式关联一个堆片段，该堆片段精确地捕获与 α相关的位置子集；函数调用契约则使用这个隐式堆片段，指定有权修改堆的位置。我们要求这类隐式堆片段能够让客户端自然地进行框架推理，类似于分离逻辑。
- 该逻辑应支持与背景排序/理论（如线性算术）的标准组合。
- 该逻辑应适用于通过归约到 SMT 求解器进行推理，从而实现可扩展且高效的推理（但请注意，将 SMT 求解器用于不完备逻辑是行不通的）。此外，我们还要求有实验证据表明，尽管采用了一阶逻辑完备过程，推理在实践中仍相当高效。

本文的主要贡献是建立了前两个满足上述要求且具有一阶完备过程的堆逻辑。我们还提供了实验证据，表明我们开发的一阶完备过程能够针对一组标准的数据结构操作程序基准测试进行高效推理，这些程序具有丰富的函数行为规约。第一个堆逻辑是框架逻辑 *[43, 44],* ，第二个则是一种新的分离逻辑，其替代语义受到框架逻辑的启发。

一阶完备框架逻辑 我们的首要研究对象是框架逻辑（FL），这是 Murali 等人近期提出的逻辑。它秉承了分离逻辑的原则，但采用带递归定义的一阶逻辑（*FORD*） [43, 44]的设定。框架逻辑（FL）在形式上十分简洁——它在带递归定义的经典一阶逻辑中加入了一个支撑集运算符Sp；对于任意公式 α，$Sp(\alpha)$的值是堆中一个隐式的位置子集，该子集决定了 α 的真值。支撑集与分离逻辑中的局部堆片段类似。然而，与分离逻辑不同，支撑集具有唯一性，因此无需真正进行集合量化即可对其建模。

框架逻辑不像分离逻辑那样依赖局部化的堆片段语义，而是保留 FORD 在全局堆上的经典语义，同时允许使用支撑算子恢复局部堆片段。框架逻辑可以表达分离合取项——$\alpha * \beta$ 本质上表示为 $\alpha \wedge \beta \wedge Sp(\alpha) \cap Sp(\beta) = \emptyset$。

Murali 等人关于框架逻辑的研究 [43, 44] 指出，带有框架逻辑注解的程序，其最弱前置条件可以用框架逻辑本身表达；而框架逻辑中的验证条件可以转换为 FORD。然而，这并不能实现自动化（尤其是使用 SMT 求解器时），因为带有交替量词的 FORD 复杂度极高。 [43, 44] 中描述的最弱前置条件变换十分复杂，需要多次引入量词，导致最弱前置条件中出现全称量词和存在量词的交替

量化。此外，最弱前置条件还会引入魔杖（“MW”运算符）的一阶形式，其定义错综复杂，难以实现自动化。

本文重新探讨框架逻辑的自动化验证，目标是将验证条件嵌入现有的、可自动化处理的带递归定义的一阶逻辑类别中。自然证明是处理此类逻辑的一种成熟技术——它利用递归定义展开、未解释函数抽象和基于 SMT 的无量词推理，构建可靠的自动推理 [38, 54, 59]。此外，关于其理论基础的研究 [38] 确定了一个特定片段 L_{oneway}，在该片段中，自然证明技术是 *FO*-完备的。

本文的第一个技术贡献是为框架逻辑规约构建一种一阶完备的自动化程序验证范式，在逻辑 L_{oneway}中生成验证条件。我们通过一些新颖的思路克服了若干挑战，包括：(a) 定义一个能力足够强的框架逻辑片段，借助一种新的云算子支持受限形式的量化；(b) 谨慎生成验证条件，使其属于 L_{oneway} 片段，并且仅对前景类型进行存在量化。验证条件通过类似最强后置条件的符号求值生成，而非采用最弱前置条件，从而避免使用类似魔杖的算子。

一种一阶完备的分离逻辑 本文的第二项贡献是一种采用不同于经典语义的替代语义 的分离逻辑，以及一种用于该逻辑的、一阶完备的程序验证过程。新逻辑*SL-FL*是一种具有确定堆片段 的分离逻辑，其灵感来自框架逻辑。据我们所知，这是第一个既足够强大（具有递归定义且有效性问题不可判定），又支持 *FO*-完备程序验证的分离逻辑。

逻辑*SL-FL* 的解释对象是存储和堆片段（s， h），而非传统分离逻辑中的全局堆。不过，该逻辑经过专门设计，使得对于任意公式 α 和任意全局堆 g，至多存在一个堆片段 h 属于 g 且使 α 成立。从这个意义上说，*SL-FL* 具有确定的堆片段。注意，传统分离逻辑不具备这一性质——$\alpha \vee \beta$ 可以在全局堆的多个堆片段中成立，例如一个满足 α 的堆片段和一个满足 β的堆片段；而 *true* 这样的公式则可在多个堆片段上成立。*SL-FL* 的语义使用受框架逻辑启发的辅助支撑算子来定义确定的堆片段。

我们证明，*SL-FL* 公式可以转换为框架逻辑，其中 α 在*SL-FL* 中转换为框架逻辑中的公式 α'。虽然 α' 是在全局堆上解释的，但其支撑集 $Sp(\alpha')$ 恰好是使 α 成立的唯一堆片段。将这一转换与针对框架逻辑规约的程序一阶完备验证相结合，我们便得到针对*SL-FL*的一阶完备程序验证。*SL-FL* 还具有其他若干有助于实现一阶完备性的特性——规约本身是无量词的，但我们允许使用一组递归定义（这需要量化），并且全局堆被解释为可能无限的模型，从而避免了另一个不完备性来源。

评估 本文的第三项贡献是开展了一项评估，旨在回答两个主要研究问题：(a) 我们定义的逻辑是否具有足够的表达能力，能够对标准数据结构操作程序进行完整的函数式规约；(b) 一阶完备程序验证机制在验证此类程序时是否具备实际的效率。我们开发了相关工具：用于验证带有框架逻辑规约的程序的工具 FLV（框架逻辑验证器），以及检查器 SLFLV，该检查器将 *SL-FL* 公式转换为便于针对*SL-FL* 规约进行一阶完备验证的形式。FLV 在带递归定义的一阶逻辑中生成验证条件，特别是在逻辑 L_{oneway}中生成，并利用现有工具进行基于自然证明的一阶完备推理 [38, 41, 54, 59]。我们在一组操作常见数据结构的程序上评估了该工具，

并发现这两种逻辑都具有足够的表达能力，能够为这些方法陈述丰富的函数式规约，并促进高效验证。

总而言之，我们的工作开发了首批具有隐式堆片段并支持 FO-完备有效性判定过程的表达力强大的堆逻辑：一种框架逻辑和一种分离逻辑。我们还提供了相应的程序验证方法，并进行了评估，证明这些逻辑具有强大的表达能力，且自动化效率高。因此，这项工作为堆逻辑确立了新的理论完备性标准。我们希望，未来的堆逻辑若以这一新的 FO 完备性标准为目标，便能推动堆验证变得更加稳健且可预测。

2 堆逻辑不完备性的来源

本节将展示若干结果，揭示堆逻辑不完备性的关键来源。具体而言，我们将说明：(a) 仅考虑有限模型会导致不完备性，因此必须定义包含无限模型在内的模型；(b) 采用最小不动点语义的定义会导致不完备性，这促使我们研究采用任意不动点语义的定义的完备性（称为 FO 完备性）；(c) 即使模型是无限的，且我们考虑不动点定义，经典分离逻辑仍会作出与多个堆片段可满足同一公式有关的设计决策，而这从根本上导致不完备性。

逻辑的完备性与不完备性。 若逻辑 $\mathcal{L}$ 满足 完备，即 $\mathcal{L}$ 中所有有效公式的集合是递归可枚举的（记作 r.e.）。换言之，存在一个可靠的过程，可以证明每个有效公式（但对于无效公式可能不会停机）。反过来，若 $\mathcal{L}$ 满足 不完备，则有效性不是递归可枚举的。请注意，不完备性意味着不可判定性，但反之不成立（例如，纯一阶逻辑不可判定但完备 [19, 70]）。

有限模型 若一个模型的论域大小有限，则称其为有限模型。语义仅在有限模型上定义的逻辑通常是不完备的。这可以由有限模型理论中的一般结果推出 [37]。

定理 2.1（特拉赫滕勃罗特定理）。固定一个一阶签名 Σ，并令 φ 为该签名上的一个公式。判断 φ 在有限模型上是否有效的问题并非递归可枚举。

该证明可通过将图灵机的不停机问题 归约到有限模型上的一阶逻辑有效性来得到。事实上，我们可以用链表编码图灵机的配置序列，并断言它们永远不会编码停机状态。鉴于链表是最基本的数据结构，有限模型并不适合用于设计完备逻辑。分离逻辑通常定义在有限堆上 [61]。我们通过在有限和无限模型上解释逻辑，避开这一不完备性来源。

递归定义 堆逻辑不完备性的第二个来源，源于将堆数据结构和度量建模为采用最小不动点语义的递归定义。即使逻辑同时具有有限模型和无限模型，这一点依然成立；以下结论众所周知：

命题 2.2。 加入最小不动点（*lfp*）递归定义的一阶逻辑（参见第*3.1*节中的定义）是不完备的。

事实上，可以用一个单一的一元lfp 定义来定义“数轴”以及加法和乘法（见 [41]），由此根据算术的哥德尔不完备定理 [17] 得出不完备性。这表明，若要实现完备性，纳入 lfp 定义并不可行。因此，本文考虑一阶完备性，即以任意不动点语义解释定义的完备性概念。

单子二阶量化堆逻辑不完备性的另一个来源，是某些设计选择——无论是隐式还是显式的——允许非嵌套的单子二阶量化（集合量化）。例如，分离逻辑（SL）中的分离合取算子 $*$ 具有如下语义："$s,\ h \models \alpha * \beta$ 如果存在定义域 h 的互不相交子集 $S_1,\ S_2$，使得……"。因此，检查蕴涵 $\alpha * \beta \models \gamma * \delta$ 实质上需要对如下形式的公式进行推理：$(\exists S_1,\ S_2.\text{有效}(S_1,\ S_2) \wedge \Phi) \rightarrow (\exists S_3,\ S_4.\text{有效}(S_3,\ S_4) \wedge \Phi')$。注意，这个公式同时包含对集合的存在量化和全称量化（由于蕴含）。我们提供证据表明，即使是这种非嵌套的二阶量化也会阻碍完备性：

定理 2.3（证明见附录 A）。单子二阶逻辑是不完备的。即使集合量化只有一层（不嵌套），这一结论依然成立。

请注意，在上述结果中，我们考虑了有限模型和无限模型上的有效性。我们不知道上述结果是否已有相关论述，因此在附录 A中给出证明。

在这项工作中，我们试图通过设计公式具有唯一堆片段的逻辑来避免这一不完备性来源。直观地说，当公式具有唯一堆片段时，我们可以用全称量化或存在量化来表述其语义。毕竟，全称量词只能由公式所定义的唯一堆片段满足！因此，我们无需检查 $(\exists S_1,\ S_2.\text{有效}(S_1,\ S_2) \wedge \Phi) \rightarrow (\exists S_3,\ S_4.\text{有效}(S_3,\ S_4) \wedge \Phi')$，而可以检查 $\forall S_1,\ S_2,\ S_3,\ S_4.\ ((\text{有效性}\ (S_1,\ S_2) \wedge \Phi) \rightarrow (\text{有效性}(S_3,\ S_4) \rightarrow \Phi'))$。这个公式只包含对集合的全称量化，因此可以将其消去，并作为一元谓词纳入一阶签名。

我们以 SL 的一个新不完备性结果结束本节。尽管先前工作已指出，魔杖等运算符本质上是二阶的 [10]（这会导致不完备性），但我们证明，即使去除魔杖以及有限模型和最小不动点等其他不完备性来源，SL 仍会因堆片段不唯一而不完备：

定理 2.4（堆片段不唯一使 SL 不完备）。分离逻辑在以下情形下是不完备的：*(a)* 不含魔杖；*(b)* 增添不动点定义；*(c)* 定义在有限和无限堆片段之上。

我们在附录中正式描述扩展到无限堆片段和不动点定义的 SL，以及上述结果的证明：A。我们的证明关键地利用了这样一个事实：SL 公式没有唯一的堆片段。

鉴于这些结果，为了寻找一阶完备堆逻辑，我们设计了能够避开这些陷阱的逻辑。我们设计的逻辑*框架逻辑* 和*SL-FL*定义在任意大小（有限或无限）的模型上，并且其堆片段是确定的；我们证明这两种逻辑都支持一阶完备的程序验证。

3 框架逻辑与程序验证

本节将介绍我们设计一阶完备推理的第一种逻辑：框架逻辑（FL）。Murali 等人在先前工作中提出了 FL [43, 44]，将其作为带递归定义的一阶逻辑（FORD）的扩展，并通过 *支撑集* 算子 Sp 访问公式的堆片段。我们将从技术层面概述 FL 的关键概念，包括其语法和语义，以及霍尔三元组的有效性概念；本工作针对该概念设计了自动化技术。本节介绍的思想并非本文的贡献，而是在先前工作中发展起来的 [38, 43]。

3.1 带递归定义的一阶逻辑

现在我们介绍带递归定义的一阶逻辑（FORD）。FORD 类似于带最小不动点的一阶逻辑 [1, 12, 24, 37, 71], ，不同之处在于，递归定义（具有最小不动点语义）有各自的名称。

形式化地说，我们有一个签名 $\Sigma = (\mathrm{S}、\mathrm{C}、\mathrm{F}、\mathrm{R}、\mathrm{I})$，其中 S 是有限的类型集合，而 C、F 和 R 分别是常量、函数和关系符号的集合。I 是一个与 R 不相交的关系符号集合，其解释通过递归定义给出（而非由模型解释）。符号具有通常的类型，例如，F 中的函数符号具有相应的元数 $n \in \mathbb{N}$，类型为 $\tau_1 \times \tau_2 \ldots \times \tau_n \to \tau$，其中 $\tau_i,\ \tau \in \mathrm{S}$。

我们要求 S 包含一个指定的前景类型Loc。我们使用前景类型来模拟堆位置。其余类型称为背景类型，用于模拟整数、集合等数据值。我们使用 F 中的函数符号来模拟堆位置的指针和数据字段。例如，链表的后继指针可以用符号后继表示：$Loc \to Loc$；同样，位置上存储的键可以用键表示：$Loc \to$ 整数。

此处不提供 FORD 的语法，因为它本质上与图 1 中给出的框架逻辑语法相同，唯一的区别是 FORD 不包含支撑算子 $Sp(\cdot)$。

递归定义 谓词 $I \in \mathrm{I}$ 的递归定义形式为 $I(x) :=^{lfp} \rho(x)$，其中 ρ 是一个无量词公式，且其中提及的 I 中递归定义符号均为正向出现（即处于偶数个否定之下）。这保证了最小不动点始终存在 [69]。我们通过将递归定义函数建模为谓词，对其进行形式化处理；不过，在本文的阐述中，我们将使用带有递归定义的函数符号。我们用 D 表示 I 中符号的定义集合。我们要求 D 对每个 $I \in \mathrm{I}$ 恰好包含一个定义。

语义 我们考虑一阶模型，其中前景类型Loc 未作解释，而背景排序受一阶理论约束。该理论通常是若干单个排序上的理论的组合，使得组合的无量词片段是可判定的 [47]。给定 I 中符号的定义集合 D，FORD 的一个模型由上述类型的一阶模型构成，该模型解释 C、F 和 R 中的符号（遵循各项理论），并且还对 I 中的符号作出解释；这些符号的解释由该一阶模型确定，为定义 D 的最小不动点。公式随后按通常方式求值。

该L^{oneway} 片段 L_{oneway} 是先前工作 [38]中引入的 FORD 语法片段，它仅允许从前景类型到背景排序的“单向”函数。形式上，F 中每个值域类型为前景类型 Loc 且元数为 n 的函数符号，其定义域为 Loc^n。元数为 k 的递归定义符号 I 的类型为 Loc^k。最后，公式只允许对 Loc 进行存在量词量化（用于有效性检查）。L_{oneway} 中的公式有效性检查是一阶完备的，并可使用基于自然证明 [54, 59]的系统化量词实例化过程 [38] 有效地自动化。

3.2 带守卫量化的框架逻辑

框架逻辑（FL） [43, 44] 是一种基于 FORD、带有隐式堆片段的堆逻辑。从语法上看，FL 在 FORD 的基础上扩展了一个支撑集运算符$Sp(\cdot)$，用于访问隐式堆片段。形式上，给定一个公式 φ（项 t），$Sp(\varphi)$ 表示其真值 φ（相应地，其值 t）所依赖的位置（前景元素）子集。随后可使用Sp 运算符表达不相交性并进行框架化推理。例如，公式$List(x) \wedge List(y) \wedge Sp(List(x)) \cap Sp(List(y)) = \emptyset$表示$x$ 和 y 指向不相交的链表。

语法 本文使用图1所示的、带守卫量化的 FL 片段。形式上，我们区分一组可变函数 $\mathcal{F}_m$ 符号中的$\mathcal{F}$，其定义域为Loc，用于对前景类型上的指针和数据字段建模。我们还要求一个背景排序

FL 公式 φ ::= $\bot \mid \top \mid t = t \mid R(t_1, \ldots, t_m) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \exists y.\ y = f(x) \wedge \varphi$ 条件表达式 $(:,):()$，其中 $R \in \mathrm{R} \cup \mathrm{I}$ $y \in \mathrm{Var}_{Loc}$，$f \in \mathrm{F}_m$ 守卫 γ ::= $t = t \mid R(t_1, \ldots, t_m) \mid \gamma \wedge \gamma \mid \gamma \vee \gamma \mid \neg\gamma \mid ite(\gamma : \gamma, \gamma)$，其中 $R \in \mathrm{R}$、t_i、t 不是 Set(Loc) 类型 项 t ::= $c \mid x \mid f(t_1, \ldots, t_m) \mid ite(\gamma : t, t) \mid (\mathrm{Sp}(\varphi)$，若 t 的类型为 Set(Loc)) $\mid (\mathrm{Sp}(t')$，若 t 的类型为 Set(Loc))，其中 c 是常量，x 是变量，而 t_1、…、t_m 属于相应的类型。递归定义 $I(x) := \rho_I(x)$，其中 $I \in \mathrm{I}$、$x \in \mathrm{Loc}^k$ 适用于某个 $k \in \mathbb{N}$，ρ_I 是 FL 公式，其中所有关系符号 $I' \in \mathrm{I}$ 仅以正向形式出现或位于支撑表达式内

图 1. 带守卫量化的框架逻辑。Var_{Loc} 表示前景类型上的变量，$Set(Loc)$表示由前景元素的集合构成的类型。为简化说明，假设项和公式的类型良好。

$Set(Loc)$表示前景位置的集合，并将$Sp(\varphi)$定义为类型$Set(Loc)$的项。我们在语法中使用作用于项和公式的ite （“如果-那么-否则”）表达式，其中$ite(\gamma : \alpha, \beta)$表示“如果 γ 成立，那么 α，否则 β”。请注意，$guard$ γ 不能提及归纳定义的谓词，也不能提及类型为$Set(Loc)$的项，包括支撑表达式。为框架逻辑公式定义良定义语义需要这些技术限制。

本文使用的 FL 片段有两项限制。首先，我们只允许形式为守卫量化 $\exists y : y = f(x).\, \varphi(y)$，其中 y 是前景类型Loc上的变量。该公式的真值与 $\exists y : y = f(x) \wedge \varphi(y)$相同，但其支撑集的定义更为谨慎。下文将对此进行说明。其次，我们将函数符号以及递归定义的谓词/函数的签名限制为“单向”，其含义与上文所述的 FORD 片段 L_{oneway} 中的定义相同。我们将对规约以该片段书写的程序进行推理，归约为判断 L_{oneway}中公式的有效性。

语义 FL 语义扩展了 FORD 的语义。关于 FL 的先前工作 [43]中有详细介绍，读者可参阅。此处我们重点关注 $Sp(\cdot)$ 运算符。 Sp 在图 2中定义为一个递归方程组。给定模型 M，公式 α 在 M 中（项则类似）的支撑集，记作 $[\![Sp(\alpha)]\!]_M$，是图 2中方程组的最小不动点。

支撑集可以理解为这样一组位置（即“堆片段”）：为了计算给定项或公式的值，必须在这些位置上应用（即解引用）可变函数。常量的支撑集为空。项 $f(t)$ 在 M 中对可变函数 $f \in \mathrm{F}_m$ 的应用，其支撑集如预期为 $\{[\![t]\!]_M\}$。应用不可变函数不会对支撑集产生贡献。

$\alpha \wedge \beta$的支撑集直观上是 α 和 β 的支撑集的并集，这在框架逻辑中确实如此。不过请注意，$\alpha \vee \beta$ 的支撑集也是 α 和 β 的支撑集的并集。这是因为框架逻辑为公式定义了唯一的支撑集，无论其真值如何。我们也可以从 $Sp(\neg\alpha)$ 的定义中看到这一点，它等于 $Sp(\alpha)$。这与分离逻辑 [16, 49, 61]，不同，在分离逻辑中，$\alpha \vee \beta$ 的堆片段是任一真值为真的析取项所对应的堆片段（公式可以有多个堆片段）。唯一堆片段是框架逻辑的显著特征，该逻辑为此采取了若干设计决策。关于这些设计决策的影响，请参阅先前工作 [43] 。

表达式 $ite(\gamma : \alpha, \beta)$ 的支撑集始终包含 γ 的支撑集（因为我们需要计算 γ 以确定公式的真假），但随后只会添加根据 γ 评估的分支的支撑集。我们使用 ite 而不是 $\vee$ 来书写具有更细粒度支撑集的表达式。归纳定义的关系项 $Sp(I(x))$ 的支撑集就是函数体的支撑集

$$
\begin{aligned}
[\![Sp(c)]\!]_M = [\![Sp(x)]\!]_M &= \emptyset \text{ for constant } c, \text{ variable } x \\
[\![Sp(\top)]\!]_M = [\![Sp(\bot)]\!]_M &= \emptyset \\
[\![Sp(Sp(t))]\!]_M &= [\![Sp(t)]\!]_M \\
[\![Sp(t_1 = t_2)]\!]_M &= [\![Sp(t_1)]\!]_M \cup [\![Sp(t_2)]\!]_M \\
[\![Sp(\alpha \wedge \beta)]\!]_M &= [\![Sp(\alpha)]\!]_M \cup [\![Sp(\beta)]\!]_M \\
[\![Sp(\alpha \vee \beta)]\!]_M &= [\![Sp(\alpha)]\!]_M \cup [\![Sp(\beta)]\!]_M \\
[\![Sp(Sp(\varphi))]\!]_M &= [\![Sp(\varphi)]\!]_M \\
[\![Sp(\neg\varphi)]\!]_M &= [\![Sp(\varphi)]\!]_M \\
[\![Sp(I(t_1 \ldots, t_n))]\!]_M &= [\![Sp(\rho_I(\overline{x}))]\!]_{M[\overline{x} \leftarrow \overline{u}]} \cup \bigcup_{i=1}^{n} [\![Sp(t_i)]\!]_M \\
&\quad \text{for } I \in \mathcal{I} \text{ with definition} \\
&\quad I(\overline{x}) :=_{lfp} \rho_I(\overline{x}), \text{ and} \\
&\quad \text{where } \overline{u} = ([\![t_1]\!], \ldots, [\![t_n]\!])
\end{aligned}
$$

$$
\begin{aligned}
[\![Sp(R(t_1 \ldots, t_n))]\!]_M &= \bigcup_{i=1}^{n} [\![Sp(t_i)]\!]_M \text{ for } R \in \mathcal{R} \\
[\![Sp(f(t_1 \ldots, t_n))]\!]_M &= \bigcup_{i=1}^{n} \{[\![t_i]\!]_M\} \cup \bigcup_{i=1}^{n} [\![Sp(t_i)]\!]_M \text{ if } f \in \mathcal{F}_m \\
[\![Sp(f(t_1 \ldots, t_n))]\!]_M &= \bigcup_{i=1}^{n} [\![Sp(t_i)]\!]_M \text{ if } f \notin \mathcal{F}_m \\
[\![Sp(ite(\gamma : t_1, t_2))]\!]_M &= [\![Sp(\gamma)]\!]_M \cup \begin{cases} [\![Sp(t_1)]\!]_M \text{ if } M \models \gamma \\ [\![Sp(t_2)]\!]_M \text{ if } M \not\models \gamma \end{cases} \\
[\![Sp(ite(\gamma : \alpha, \beta))]\!]_M &= [\![Sp(\gamma)]\!]_M \cup \begin{cases} [\![Sp(\alpha)]\!]_M \text{ if } M \models \gamma \\ [\![Sp(\beta)]\!]_M \text{ if } M \not\models \gamma \end{cases} \\
[\![Sp(\exists y\colon y = f(x).\ \varphi)]\!]_M &= \{[\![x]\!]_M\} \cup [\![Sp(\varphi)]\!]_{M[y \leftarrow u]} \\
&\quad \text{where } u = [\![f(x)]\!]_M
\end{aligned}
$$

图 2. 支撑算子的语义。$[\![e]\!]_M$ 指模型 M 中表达式 e 的解释。支撑被定义为满足给定方程的最小解释。

$$
\begin{aligned}
P \coloneqq\ & x \coloneqq y \mid x \coloneqq c \mid v \coloneqq be \mid x \coloneqq y.f \mid v \coloneqq y.d \mid y.f \coloneqq x \mid y.f \coloneqq c \mid y.d \coloneqq v \mid y.d \coloneqq be \\
& \mid \text{分配}(x) \mid \text{释放}(x) \mid \bar{q} \coloneqq g(\bar{p}) \mid \text{假设}(\eta) \mid P;\ P \mid \text{如果 } \eta \text{ 那么 } P \text{ 否则 } P \mid \text{返回}
\end{aligned}
$$

图 3. 编程语言的语法。其中，x, y 是类型为 Loc 的位置变量，c 是位置常量，f 是类型为 $Loc \to Loc$ 的指针，d 是类型为 $Loc \to \tau_{\text{bs}}$ 的数据字段，其中 $Loc \to \tau_{\text{bs}}$ 属于某个背景排序 τ_{bs}，be 是背景表达式，v 是背景排序的变量，η 是不包含任何解引用的布尔表达式。最后，g 是类型为 $\tau_1 \times \cdots \times \tau_m \to \tau'_1 \times \cdots \times \tau'_n$ 的函数，其中 m, n 的类型为 $\tau_i, \tau'_i \in \mathsf{S}$。这是一个方法，其函数体本身是一个程序，其变量是输入变量 p_i（类型为 τ_i）和输出变量 q_j（类型为 τ'_j）的超集。

关于 $Sp(\rho(x))$ 的定义。函数体 ρ 当然可以递归地提及 I，而支撑是这些方程的最小不动点。

最后，$\exists y$的支撑：$y = f(x).\ \varphi(y)$ 包含由 x 解释的位置，以及 $\varphi(y)$ 的支撑，其中 y 在给定模型中被解释为与 $f(x)$ 对应的位置。请注意，尽管该公式与 $\exists y.\, y = f(x) \wedge \varphi(y)$ 的求值结果相同，但其支撑仅包含在 y 取与守卫“匹配”的值时 φ 的支撑，即 $f(x)$。

3.3 程序验证

现在，我们将介绍本文为其开发自动化方法的编程语言及其正确性概念。这些概念是第 5 节所述验证条件（VC）生成的背景知识；读者可以放心地先跳过本节，待读完第4节后再回来阅读。

图 3 描述了该语言的语法。它支持常见的命令，包括字段修改、位置分配和释放。该语言还支持函数调用，包括递归调用 [1]。该语言中程序的操作语义是堆程序的常见语义（见附录 B）。程序配置的形式为$(S,\ H,\ A)$其中，S是存储，H是堆，而 A是已分配位置的集合。此外还有一个错误配置 $\bot$。根据操作语义，我们将程序 P上配置 C_1 与 C_2 之间的状态转换记为 $C_1 \xrightarrow{P} C_2$。

[1]为简化技术性阐述，我们没有将*while* 循环纳入形式语法。不过，我们的理论很容易扩展到包含迭代的程序，其规约可将框架逻辑断言作为循环不变式。

霍尔三元组与有效性 我们考虑形如 $\{\alpha\}\ P\{\beta\}$ 的三元组，其中 α 和 β 是上述片段中的 FL 公式， P 是一个程序。我们将 α 和 β 中的自由变量视为常量，并隐式地对它们进行全称量化。接下来定义：

定义 *3.1*（霍尔三元组有效性）。$\{\alpha\}\ P\{\beta\}$ 在有效，当且仅当对于每个配置(S, H, A) 满足$(S, H) \models \alpha$且 $[\![Sp(\alpha)]\!]_{(S,H)} = A$:

(1) (S, H, A) 根据操作语义不会转换到 $\perp$ 上的 P， d

(2) 如果$(S, H, A) \xrightarrow{P} (S', H' A')$，那么$(S', H') \models \beta$且 $[\![Sp(\beta)]\!]_{(S',H')} = A'$

注意，我们要求后状态中的已分配集合等于后置条件的支撑集。

通俗地说，上述定义表示：如果从任意满足前置条件、且已分配集合恰好等于前置条件支撑集的配置开始执行，那么霍尔三元组有效，当且仅当：(a) 程序不会出现错误行为（例如进行不安全解引用）；(b) 如果程序到达最终配置，则后置条件必须成立，且后状态中的已分配集合恰好等于后置条件的支撑集。为确保前者，我们必须在程序的每一步检查被解引用的元素是否包含在已分配集合中。这便引出了以下定义，我们将在第*5*节中为FL进行验证条件生成时使用该定义。

宽松后置条件。 我们引入一种后置条件变体 RP，称为宽松后置条件，表示后状态中的已分配集合对于后置条件而言不必是“紧致的”。我们用 $\{\alpha\}\ P\{RP{:}\ \beta\}$ 表示这类三元组。其正确性的定义与上文 $\{\alpha\}\ P\{\beta\}$ 的定义类似，只是条件 (2) 中我们仅要求 $[\![\text{支撑集}(\beta)]\!]_{(S',H')} \subseteq A'$。类似地，我们也使用这种变体 HP 表示无堆后置条件，其中不检查已分配集合。

4 具有受框架逻辑启发语义的分离逻辑

现在，我们定义一种分离逻辑 *SL-FL*，其替代语义 受框架逻辑语义的启发，并为其给出一个一阶逻辑完备过程。

一个完整示例见附录，其中展示了一个带有 *SL-FL* 规范的程序、该程序到框架逻辑的翻译，以及相应的验证条件生成过程 G。

首要的设计原则是，公式所能成立的堆片段是唯一的。尤其是，这些堆片段的设计受框架逻辑中的支撑算子启发。我们的分离逻辑有几个显著特性：(1) 给定一个全局堆 g 和一个存储 s，任意公式 α 至多能在 g 的一个堆片段中成立，我们将其称为 $Supp(\alpha, s, g)$；(2) 假设堆片段是任意的位置全集（有限或无限），从而避免有限模型带来的不完备性；(3) 我们不允许递归定义以背景类型的值作为参数。递归定义仅在位置类型上定义，不过定义本身可以通过解引用位置上的指针来使用背景类型。这使我们最终能够将公式翻译到具有一阶完备推理的逻辑 L_{oneway} 中。例如，我们不定义一个递归定义 $mem(x, i)$ 来判断由 x 指向的链表中是否包含某个整数，而是在我们的逻辑中定义 $keys(x)$，收集由 x 指向的链表中存储的键，并断言 $i \in keys(x)$。请注意，这类定义翻译成一阶逻辑后，不会涉及对背景类型的量化，这对完备性至关重要。最后，我们不直接在分离逻辑中生成验证条件，而是将分离逻辑注解翻译成框架逻辑，并利用上一节中开发的框架逻辑验证条件生成方法。在分离逻辑中生成精确的验证条件通常需要魔术棒操作，而我们通过使用框架逻辑避免了这些操作。

我们的逻辑功能强大，涵盖了文献中若干已知的精确分离逻辑 [7, 43,51]（精确分离逻辑也被设计为具有唯一堆片段）。具体而言，我们的逻辑

$SL\text{-}FL_b$公式 $\alpha, \beta, \gamma \quad ::= \delta \mid x \mathrel{-\!\!f\!\!\to} y \mid \alpha * \beta \mid \alpha \wedge \beta \mid \alpha \curlywedge \beta \mid$条件表达式$(\gamma, \alpha, \beta) \mid \exists y.(x \mathrel{-\!\!f\!\!\to} y : \alpha)$$SL\text{-}FL_b^{\vee}$
公式 $\alpha, \beta, \gamma \quad ::= \delta \mid x \mathrel{-\!\!f\!\!\to} y \mid \alpha * \beta \mid \alpha \wedge \beta \mid \alpha \curlywedge \beta \mid \alpha \vee \beta \mid$条件表达式$(\gamma, \alpha, \beta) \mid \exists y.(x \mathrel{-\!\!f\!\!\to} y : \alpha)$堆无关 公式 $\delta \quad ::=$ 真 $\mid$假 $\mid x = y \mid x \neq y \mid x =$ 空指针 $\mid x \neq$ 空指针

图 4。B受 FL 启发语义的分离逻辑 $SL\text{-}FL_b$ 和 $SL\text{-}FL_b^{\vee}$；H.I.：“堆无关”

允许对空间公式使用析取，而此前的精确分离逻辑不支持这一点。我们还允许使用递归定义，其求值结果可以是由任意一阶理论（例如算术）约束的背景类型。

尽管我们的替代语义总体上遵循分离逻辑的设计原则，但为了确保唯一堆片段，析取的语义却大不相同。不过，我们的逻辑提供了如果-那么-否则 （条件表达式）构造，其具有更传统的语义，并且可以对分情况进行建模。析取通常用于情况分析（例如，使用析取来表示链表是否为 $x$is nil 或非 nil 的链表定义，本质上就是一种情况划分）。根据我们在评估中为程序添加注释的经验，析取通常可以避免（在我们的基准测试中完全可以避免），并用这种情况分析替代。

我们首先定义一种基础分离逻辑$SL\text{-}FL_b$，其语义差异十分明确；再定义一种加入析取的基础分离逻辑$SL\text{-}FL_b^{\vee}$。随后，我们将其扩展为一种表达能力更强的逻辑（$SL\text{-}FL$），其中包含归纳定义和背景排序。所有这些逻辑都可以转换为框架逻辑（FL），因此，借助我们针对框架逻辑的程序验证方案，可以对它们进行一阶完备程序验证。

4.1 基础逻辑 $SL\text{-}FL_b$ 和$SL\text{-}FL_b^{\vee}$

我们固定一个位置集合 Loc。令 $Loc?$表示 $Loc \cup \{nil\}$，其中nil 是一个特殊符号，并且$nil \notin Loc$。我们固定一个可数变量集Var，并令 x, y, x_1, x_2, x', y' 等取遍Var。这些变量将用于直观地模拟程序变量以及量化变量。我们固定一个有限指针集Ptr，并令 f, f', g等取遍Ptr。

逻辑$SL\text{-}FL_b$ 的语法如图4所示。其中， x, y 取值于Var 和 $f \in Ptr$。该逻辑支持分离合取 $*$、合取 $\wedge$、弱合取 $\curlywedge$、特殊的如果-那么-否则结构ite以及带守卫的存在量化。逻辑$SL\text{-}FL_b^{\vee}$与之相同，但包含析取算子。

语义：A 存储是一个函数 $s : Var \to Loc?$。A 堆片段 h 是函数的集合 $\{h_f \mid f \in Ptr\}$，其中每个函数 h_f: $H \to Loc?$，且 $H \subseteq Loc$ 是所有函数共有的定义域 h_f。注意，我们不假设堆片段是有限的。[2] 注意，所有函数 h_f的定义域相同。我们用 $h(f)$ 表示由 h 定义的函数 h_f，用 $dom(h)$ 表示位置集合 H。此外，对于任意 $S \subseteq dom(h)$，令 $h \restriction S$ 表示一个堆片段，其中 h 中的指针字段的定义域被限制为 S。如果存在某个 $S \subseteq dom(h)$ 使得$h' = h \restriction S$，则堆片段 h' 是 h 的子堆/子堆片段。全局堆是满足 $dom(h) = Loc$的堆片段 h。

基础逻辑的语义定义于图 5 中。公式 α 在一对(s, h)上进行解释，其中 s是一个存储，而 h 是一个堆片段，与传统分离逻辑中的情况相同。然而，语义与支撑算子 $Supp$的定义一起以相互递归的方式定义，其中$Supp$将每个公式映射到位置的一个子集，或映射到表示其未定义的 $\perp$。直观地说，公式的支撑是 h 的一个位置子集，公式在堆片段 h 上的求值取决于该子集；如果 $\perp$ 为 α，则公式无法在该堆片段内求值 h。

正如我们即将证明的（引理 4.2），这些定义经过精心设计，因此只要 $(s, h) \models \alpha$成立，就必有 $dom(h) =$支撑集(α, s, h)，也就是说，α仅能在支撑映射求值结果为整个堆片段的堆片段上成立。基于这一点，我们来逐一考察其语义。

[2]在我们的逻辑中，堆可以是无限的，堆片段也可以是无限的，例如无限列表的堆片段。

支撑集$(\delta, s, h) = \emptyset$，对于任意堆无关原子公式 δ支撑集$(x \mathrel{-\!\!f\!\!\to} y, s, h) = \{s(x)\}$，如果 $s(x)$ 属于 $dom(h)$，且 $\perp$否则支撑集$(\alpha \oplus \beta, s, h) =$ 支撑集$(\alpha, s, h) \cup$支撑集(β, s, h)，其中 $\oplus \in \{\wedge, *, \between\}$ 支撑集(条件表达式(γ, α, β) $s, h) =$ 支撑集$(\gamma, s, h) \cup$支撑集(α, s, h)，若 $(s, h \downharpoonright$支撑集(γ, s, h) $\models \gamma$且 $(s, h \downharpoonright$支撑集$(\alpha, s, h)) \models \alpha$= 支撑集$(\gamma, s, h) \cup$支撑集$(\beta, s, h)$，否则支撑集$(\exists y.(x \mathrel{-\!\!f\!\!\to} y\colon \alpha)$ $s, h) = \perp$若 $s(x) \notin dom(h) = \{s(x)\} \cup$支撑集$(\alpha, s[y \mapsto h(f)(s(x))], h)$，否则。$(s, h) \models$ δ当且仅当 $dom(h) = \emptyset$且 $s \models \delta$，对于任意堆无关公式 $\delta(s, h) \models$ $x \mathrel{-\!\!f\!\!\to} y$当且仅当 $dom(h) = \{s(x)\}$ 且 $h(f)(s(x)) = y(s, h) \models$ $\alpha * \beta$ 当且仅当存在 h_1, h_2 个 h 的子堆片段，使得 $dom(h_1) =$ 支撑集(α, s, h)，$dom(h_2) =$ 支撑集(β, s, h)，$dom(h_1) \cup dom(h_2) = dom(h)$，$dom(h_1) \cap dom(h_2) = \emptyset$，$(s, h_1) \models \alpha$且 $(s, h_2) \models \beta(s, h) \models$ $\alpha \wedge \beta$当且仅当 $(s, h) \models \alpha$且 $(s, h) \models \beta(s, h) \models$ $\alpha \between \beta$当且仅当存在 h_1, h_2 个 h 的子堆片段，使得 $dom(h_1) \cup dom(h_2) = dom(h)$，$(s, h_1) \models \alpha$且 $(s, h_2) \models \beta(s, h) \models$条件表达式$(\gamma, \alpha, \beta)$ 当且仅当存在 h_1, h_2 个 h 的子堆片段，使得 $dom(h_1) \cup dom(h_2) = dom(h)$，$dom(h_1) =$ 支撑集(γ, s, h_1)，且，$[(s, h_1) \models \gamma$且 $dom(h_2) =$ 支撑集(α, s, h_2) 且 $(s, h_2) \models \alpha)$ $]or[(s, h_1) \not\models \gamma$且 $dom(h_2) =$ 支撑集(β, s, h_2) 且 $(s, h_2) \models \beta)$ $](s, h) \models (\exists y.x \mathrel{-\!\!f\!\!\to} y\colon \alpha)$ 当且仅当 $x \in dom(h)$ 且 $(s[y \mapsto h(f)(x)], h) \models \alpha$

析取规则：

$Supp(\alpha \vee \beta, s, h) = Supp(\alpha, s, h) \cup Supp(\beta, s, h)$ $(s, h) \models$ $\alpha \vee \beta$当且仅当存在 h_1, h_2，它们是 h 的子堆片段，且 $dom(h_1) \cup dom(h_2) = dom(h)$，$dom(h_1) = Supp(\alpha, s, h_1)$，$dom(h_2) = Supp(\beta, s, h_2)$ and，$((s, h_1) \models \alpha$或 $(s, h_2) \models \beta)$

图 5. 基础分离逻辑 SL-FL$_b$ 的支撑集定义与语义，它们是相互定义的。SL-FL$_b^\vee$ 的定义包括析取规则。支撑集上的集合运算若结果为 $\perp$，则其值为 $\perp$。

堆无关公式的语义要求堆片段为空。这使我们能够确保堆片段是确定的；而在传统分离逻辑中，这类公式在任意堆片段上都成立。请注意，我们没有 emp 公式，因为 $true$ 具有相同的语义。 $x \mathrel{-\!\!f\!\!\to} y$、 $\alpha * \beta$ 和 $\alpha \wedge \beta$ 的语义与标准分离逻辑语义相似 [61]。尽管 $*$ 的定义指出，只要能将两个堆片段拆分为不相交部分，并使其满足两个子公式，任何拆分方式都可以，但我们通过归纳可知，实际上只有一种拆分方式。合取和弱合取的语义也类似（弱合取项不要求两个堆片段相同）。

这个条件表达式构造的语义较为复杂。直观来说，要计算条件表达式(γ, α, β)，我们会在它所需的γ堆片段上求值，然后根据 γ是否成立，在它所需的堆片段上计算 α。注意，任意公式 α所需的堆片段都是子堆片段 h'，使得相对于 α计算的支撑集 h'恰好为 $dom(h')$。支撑集的定义经过精心设计，因此这样的子堆片段（如果存在）是唯一的(下文引理4.1中将予以证明)。

公式相对于存储和堆片段的支撑集定义 $Supp(\alpha, s, h)$ 很自然，但请注意，条件表达式的定义引用了堆片段的语义，以便根据 γ是否由某个子堆片段满足来区分情况。

直观而言，s, h捕捉了某个公式是否在一个堆片段上成立，而$Supp(\alpha, s, h)$ 是否成立则捕捉了该堆片段是否足够大，能够对 α 求值；这对于ite 表达式中$else$ 分支的语义至关重要。

$\Pi(\gamma) = \gamma$，对于堆无关原子公式 γ　　$\Pi(x \dashrightarrow y) = f(x) = y$

$\Pi(\alpha * \beta) = \Pi(\alpha) \wedge \Pi(\beta) \wedge$ 支撑集$(\Pi(\alpha)) \cap$支撑集$(\Pi(\beta)) = \emptyset$　　$\Pi(\alpha \vee \beta) = \Pi(\alpha) \vee \Pi(\beta)$

$\Pi(\alpha \wedge \beta) = \Pi(\alpha) \wedge \Pi(\beta) \wedge$支撑集$(\Pi(\alpha)) =$ 支撑集$(\Pi(\beta))$　　$\Pi($条件表达式$(\gamma,\ \alpha,\ \beta)) =$条件表达式$(\gamma,\ \Pi(\alpha)\ \Pi(\beta))$

$\Pi(\alpha \twoheadrightarrow \beta) = \Pi(\alpha) \wedge \Pi(\beta)$　　$\Pi(\exists y.(x \dashrightarrow y :\ \alpha)) = \exists y:\ y = f(x).\Pi(\alpha)$

图 6. 从 $SL\text{-}FL_b^{\vee}$ 到框架逻辑的转换

析取：析取公式 $\alpha \vee \beta$ 的语义与传统语义大不相同。按照传统语义，只要 α 或 β 成立，就会判定该公式成立。但在更大的堆片段 h 中，可能既有使 α 成立的堆片段，也有使 β 成立的堆片段，因此这与我们希望公式拥有唯一堆片段的目标相悖。在我们的语义中，这类 $\alpha \vee \beta$的支撑集被视为 α 与 β支撑集的并集，而 $\alpha \vee \beta$ 在这个合并后的堆片段上成立。

逻辑 $SL\text{-}FL_b^{\vee}$ 包含析取，但正如我们之前提到的，在实验评估中，我们并未发现真正需要析取的情况，因为大多数用法都可以用 条件表达式 表示。

性质：公式的支撑集具有若干值得注意的性质。设 S 是相对于存储和堆片段 h 的公式支撑集，其中 $S \neq \perp$。首先，S 将是 $dom(h)$ 的子集。其次，考虑一个在 S 上与 h 一致的堆片段 h'。那么，相对于 h' 的公式支撑集也将是 S。第三，将公式相对于 h 的支撑集限制在 S 上，结果就是 S 本身。最后，h 中至多存在一个子堆片段 h'，使得 h' 中公式的支撑集本身就是 $dom(h')$。以下引理对此进行了形式化（证明概要见附录 D）。

引理 4.1。设 s为一个存储，α为一个 $SL\text{-}FL_b^{\vee}$ 公式，h为一个堆片段，并且 $S = Supp(\alpha, s, h)$，以及 $S \neq \perp$。

(1) $S \subseteq dom(h)$。*(2)* 设 h'为一个堆片段，使得 $S \subseteq dom(h')$ 且 $h' \restriction S = h \restriction S$。那么 $Supp(\alpha, s, h') = S$。*(3)* $Supp(\alpha, s, h \restriction S) = S$。*(4)* 设 h_1和 h_2为 h的两个子堆片段，并假设 $Supp(\alpha, s, h_1) = dom(h_1)$，且 $Supp(\alpha, s, h_2) = dom(h_2)$。那么 $dom(h_1) = dom(h_2)$。□

上述语义确保了一个关键性质——考虑任意存储 s、堆 h和公式α，那么至多存在一个子堆片段 h'属于 h且满足 α(即 $h \restriction Supp(\alpha, s, h)$)。标准分离逻辑语义不具备这一性质（证明概要见附录D）。

引理 4.2。设 s 是一个存储，h 是一个堆片段，α 是一个 $SL\text{-}FL_b^{\vee}$ 公式。

(1) 如果 $(s,\ h) \models \alpha$，则 $Supp(\alpha,\ s,\ h) =$定义域(h)。

(2) 至多存在一个 h 的子堆片段 h'，使得 $(s,\ h') \models \alpha$。□

向框架逻辑的转换

$SL\text{-}FL_b^{\vee}$ 公式可以直接转换为框架逻辑，其语义使用确定的堆片段/支撑集。转换如图 6 所示，简单而自然。该转换在以下意义下同时保持真值和堆片段语义：

引理 4.3。设 g 为一个全局堆（定义域为 Loc 的堆片段），s 为一个存储。对于框架逻辑公式，我们将该存储解释为变量的一阶逻辑解释。设 α 为一个 $SL\text{-}FL_b^{\vee}$ 公式。则

(1) $g \models_s \Pi(\alpha)$ 当且仅当存在 g 的一个堆片段 h，使得 $(s,\ h) \models \alpha$。*(*

2) 如果 $g \models_s \Pi(\alpha)$，则 $Supp(\alpha,\ s,\ g)$ 等于 $Sp(\Pi(\alpha))$ 在 g 中的值。□

上述内容使我们能够为使用以下公式注解的程序构建自动验证过程：$SL\text{-}FL_b$和$SL\text{-}FL_b^{\vee}$公式。使用分离逻辑公式编写契约的注解可以转换为框架逻辑；其中，分离逻辑公式会隐式定义堆片段，这些堆片段构成相应框架逻辑公式的支撑集（示例见附录G）。

SL-FL 公式 α, α', β ::= 真 | 假 | $x = y$ | $x \neq y$ | $x =$ 空指针 | $x \neq$ 空指针 | $x \xrightarrow{f} y$ | $p(t)$ | $\alpha * \beta$ | $\alpha \wedge \beta$ | $\alpha \twoheadrightarrow \beta$ | $\alpha \vee \beta$ | 条件表达式(α', α, β) | $\exists y.(x \xrightarrow{f} y : \alpha)$ | $R(x)$ 项 $t, t',$ t_i ::= x | $t.f$ | $g(t)$ | 条件表达式(α, t, t') | $F(t)$ 递归定义：$R(x) :=_{lfp} \rho_R(x, R, F)$，对每个 $R \in RF(x) :=_{lfp} \mu_F(x, R, F)$，对每个 $F \in F$

图 7. *SL-FL*：受框架逻辑启发的语义下，完整分离逻辑的语法。*ite* 表达式中的守卫不应提及递归定义的关系 $R \in$ R

4.2 将基础逻辑扩展到背景排序和递归定义

现在，我们将基础逻辑扩展为一种更强大的逻辑，它（a）允许谓词和函数的递归定义，并且（b）纳入背景排序（如算术）、指向此类排序的指针字段，以及求值结果属于背景排序的递归定义函数。

固定背景排序及其相应的函数和关系 G = {g_1, ……, } 和 P = {p_1, ……, }。另外，固定一组函数和关系符号，分别记为 F 和 R，我们将用它们来定义新的递归定义的函数和关系。堆片段扩展后还包括将位置映射到背景排序的指针。

SL-FL 的语法如图 7所示。SL-FL 公式的语法与基础逻辑相似，但允许对背景排序的项上的谓词求值（$p(t)$），并允许对递归定义谓词求值（$R(x)$）。项的语法允许对位置进行解引用（结果可以是位置或背景排序），计算背景排序上的函数（如 + 对整数进行运算）、使用条件构造，或使用返回背景排序值的递归定义函数（例如指向链表的位置的“键集合” x）。

递归定义谓词（R）和函数（F）时，其参数仅定义为位置类型。这是一个至关重要的限制，因为在向框架逻辑以及之后向一阶逻辑转换时，我们会将这些定义视为全称量化方程；而对于使用自然证明实现一阶完备性而言，重要的是这些量化仅针对位置这一前景类型。谓词（R）和函数（F）的定义是相互递归的，并且可以是任意的*SL-FL* 公式 ρ 以及背景排序上的项 μ，其中可以提及 R 和 F。

该逻辑的语义以自然方式扩展了基础逻辑的语义，此处略去形式语义（有关详细信息，请参见附录 E）。我们假设每个背景排序都是一个完备格（对于算术等平坦类型，可以引入一个最底元素和一个最顶元素，以获得完备格）。支撑映射针对递归定义进行了扩展，并将其求值为相应定义的支撑集。对于 *Supp* 和 $\models$，其语义均作为方程组一并处理，并将其语义定义为这些方程的最小不动点。

由此产生的分离逻辑功能强大，能够表达标准数据结构（如链表、树）及其性质（如键、二叉搜索树）等。此外，该逻辑很容易转换为 FL（扩展上述 Π 转换），从而自动验证使用 *SL-FL* 注释的程序（详见附录 E）。

5 使用框架逻辑实现程序验证自动化

本节介绍本文的第二项技术贡献：一种用于生成以 *FL* 注释的程序的验证条件生成技术。为此，我们为 *FL* 扩展了“云”算子，使其能够进行量词消去。随后，我们利用最强后置条件生成验证条件，从而避免了 [43, 44] 中最弱前置条件计算所产生的噩梦般的公式。

5.1 使用云算子简化量化

在这项工作中，我们希望编写无量词规约，在 L_{oneway} 片段中生成无量词验证条件。然而，即使是在框架逻辑中定义像链表这样简单的对象，也需要进行量化 [43, 44]:

$$List(x) :=_{lfp} ite(x = nil, \top, \exists y : y = next(x).\ List(y) \wedge x \notin Sp(List(y)))$$

其中，上述定义使用了存在量词来说明：当 $x \neq$ 为空指针时， y的“后继”位置 x必须指向一个链表，使得 x不属于 链表(y)[3]的支撑集。

很容易想到，只需将 y 替换为 后继(x)，就能消去量词。然而，这样第二个合取项就变成了 $x \notin$ 支撑集(链表 (后继(x)))，但这并不成立，因为提及 后继(x) 的公式的支撑集总是包含 x[4]。

在这项工作中，我们引入一种名为云 算子的全新算子，记作方括号$[\cdot]$。公式（或项） α的云是一个公式 $[\alpha]$，其求值结果与 α相同，但支撑集为空。简单来说， $[\alpha]$ 将 α 视为不含支撑集的表达式。我们为框架逻辑扩展了云算子，从而可以在不使用量化的情况下重写上述定义：

$$链表(x) :=_{lfp} ite(x = nil,\ \top,\ next(x) = next(x) \wedge List([next(x)]) \wedge x \notin Sp(List([next(x)])))$$

其中，所有出现的 y 都替换为 [后继(x)]。我们还添加永真合取项后继 (x) =后继(x)，以确保所得公式的支撑集仍包含 $\{x\}$。形式上，我们将云算子的语义定义如下：

$$Sp([\varphi]) = \emptyset \text{ and } M \models [\alpha] \text{ iff } M \models \alpha \text{ for a formula } \alpha$$
$$Sp([t]) = \emptyset \text{ and } [\![\ [t]\ ]\!]_M = [\![t]\!]_M \text{ for a term } t$$

接下来给出以下引理。固定一个模型 M。

引理 5.1（使用 Cloud 算子消除量化）。 $M \models \exists y.\ y = f(x).\ \varphi$当且仅当 $M \models (f(x) = f(x)) \wedge \varphi([f(x)])$，且 $[\![$支撑集$(\exists y:\ y = f(x).\ \varphi)]\!]_M =$ $[\![$支撑集$((f(x) = f(x)) \wedge \varphi([f(x)]))]\!]_M$ □

由于该引理可直接由定义得出，我们略去其证明。

现在，我们转而讨论如何为带有无量词框架逻辑规约的程序生成验证条件；该逻辑现已扩展云算子。注意，用户仍按照图1所示，使用带守卫量化的框架逻辑编写规约。这些规约会使用引理 5.1中描述的变换自动消除。

我们先约定一些记号。对于给定的程序，假设有一个固定的指针字段和数据字段集合 F_m，并用 f、f'、f_1 等表示这些字段。我们还固定一个有限的递归定义集合 I，符号 I、I'、I_1 等取值于 I。对于每个 $I \in$ I，我们用 定义 表示相应的递归定义，其中 I的定义形式为 $I(\vec{x}) =_{lfp} \rho_I(\vec{x})$。定义可以相互递归。

给定一个三元组 {前置条件} P {后置条件}，我们确定一组由 P调用的方法 G。我们假设每个$g \in G$ 都有自己的契约，其前置条件为 前置条件$_g$，后置条件为 后置条件$_g$。我们的验证是模块化的，因此只需假设被调用方法的三元组有效。

下面给出一个示例。程序 RevPrepend 会将 x 指向的链表反转后，添加到 y 指向的链表前面。前置条件断言这两个链表互不相交。后置条件断言，程序返回一个链表ret，其键集合是程序开始时两个链表的键集合之并。我们用$Old(R(x))$表示通过在方法调用开始时的程序状态上计算 $R(x)$所得到的、属于相应背景排序的元素的引用。具体而言，表达式 $Old(R(x))$的支撑集为空。

---

[3]对于熟悉分离逻辑的读者而言，该公式类似于分离逻辑公式 $\exists y.\ (x \overset{后继}{\mapsto} y) * 链表(y)$ [4]该示例表明，替换在框架逻辑中并不是一种保持语义的操作。实际上，只有当替代表达式与被替换的原子表达式具有相同的真值且具有相同的支撑集时，替换才是有效的。

定义与规约：
链表 (x) := 条件表达式(x = 空指针 $\top$, 链表 (后继 (x)) $\wedge\, x \notin$ 支撑集(链表 ([后继 (x)])) 键集合 (x) := 条件表达式 (x = 空指针 ϕ, 键集合 (后继 (x)) $\cup$ { 键 (x) }) 前置条件: 链表 (x) $\wedge$ 链表 (y) $\wedge$ 支撑集(链表 (x)) $\cap$ 支撑集(链表 (y)) $= \phi$ 后置条件: 链表 (ret) $\wedge$ 键集合(ret) $= Old$ (键集合(x)) $\cup Old$ (键集合(y))

程序： RevPrepend(x, y) 返回(ret): （如果 (x = 空指针), 则 ret := y；否则 tmp := x.next；x.next := y；ret := RevPrepend(tmp, x)；）返回；

5.2 在基本块上生成三元组

对于带有注解的程序，我们首先在基本块上生成若干三元组。这些生成的三元组既描述了沿不同控制流对 P 后置条件的验证，也描述了解引用和释放操作的内存安全等其他“附加条件”。

与程序 P对应的基本块，记为 $BB(P)$，如下所示：

- $BB(c) = \{c\}$ 对于任意原子命令 c
- $BB(\text{如果 } \eta \text{ 那么 } P_1 \text{ 否则 } P_2) = \{assume(\eta);\ P'_1 \mid P'_1 \in BB(P_1)\} \cup \{assume(\neg\eta);\ P'_2 \mid P'_2 \in BB(P_2)\}$
- $BB(P_1; P_2) = \{P'_1; P'_2 \mid P'_1 \in BB(P_1)\ P'_2 \in BB(P_2)\}$

$BB(P)$ 按照常规方式生成基本块，在条件语句处分割路径，并在条件语句中使用假设语句包含条件。假设每个基本块都处于“静态单赋值形式”，即每个变量至多被赋值一次。这并非一种限制，因为我们语言中的每个基本块都可以转换为静态单赋值形式。

Give在三元组 {前置条件} P {后置条件} 中，与基本块 $BB(P)$ 相关联的三元组集合为：

- 对于每个 $s \in BB(P)$, {前置条件} s {后置条件}
- 对于每个 s_1, s_2，其中 s_1； $x := y.f$； s_2 或 s_1； $y.f := x$； s_2 或 s_1；释放(y)； s_2 属于 $BB(P)$，其中 HP 是无堆后置条件 s (参见第 3.3 节)。
- 对于每个 s_1， s_2 使得 s_1； $b := g(\bar{a})$； $s_2 \in BB(P)$，其中 RP 是宽松后置条件 (参见第 3.3 节) $Pre_g[\bar{p} \leftarrow \bar{a}]$}

上面的第一项包含了对程序沿每个基本块的后置条件进行验证。第二组霍尔三元组用于捕获解引用的安全性，并确保已释放位置当前已分配。对于每条解引用位置变量或释放位置的语句，我们为基本块中截至该语句的前缀添加一个霍尔三元组，并使用宽松后置条件检查该变量是否属于当前已分配集合。最后，对于每条调用方法 g的语句，我们引入基本块中截至该调用的前缀，并再次使用宽松后置条件检查被调用方法的前置条件是否成立。上面的宽松后置条件确保作为 g, Pre_g 的前置条件支撑集的堆片段（该支撑集 g 将始终包含在其中）是当前已分配集合的子集。

可以将为基本块生成三元组的过程想象成逐条执行程序中的命令，并逐一履行必要的义务。我们来看看示例中的“else”分支。假设 $x \neq$ 空指针，我们首先使用图 8 中左上角的霍尔三元组，检查将 x 赋给 $x.$后继 到 tmp 时，解引用是否安全[5]。对 $x.$后继 赋给 tmp 的变异，处理方式类似。接下来，我们要求 $RevPrepend$ 的前置条件在方法调用时成立，即输入为 tmp x。此外，我们还要确保其支撑集包含于已分配集合中。我们将调用时前置条件的支撑集视为函数调用修改的位置集合。图 8 中右上角的霍尔三元组描述了这一点。最后，我们通过图8 底部的三元组，检查程序结束时后置条件是否成立。

[5]本节后文将展示，已分配集合 A 可以表示为 FL 公式。因此，诸如 $y \in A$ 这样的检查都可以用 FL 表示。

{List (x) ∧链表 (y) ∧Sp(链表 (x)) ∩Sp(链表 (y)) = ϕ} 假设(x ≠空指针); tmp := x.后继 {List (x) ∧链表 (y) ∧Sp(链表 (x)) ∩Sp(链表 (y)) = ϕ} 假设(x ≠空指针); tmp := x.后继; x.后继 := y{HP: x ∈ A} {RP:链表 (tmp) ∧链表 (x) ∧Sp(链表 (tmp)) ∩Sp(链表 (x)) = ϕ} {List (x) ∧链表 (y) ∧Sp(链表 (x)) ∩Sp(链表 (y)) = ϕ} 假设(x ≠空指针); tmp := x.后继; x.后继 := y; ret := RevPrepend(tmp x) {List (ret) ∧键集合(ret) = Old (键集合(x)) ∪Old (键集合(y)) }

图 8. 示意性示例中为“else”分支生成的霍尔三元组。

5.3 验证条件生成

本节介绍本文的核心技术贡献：一种针对基本块生成验证条件（VC）的机制。我们使用框架逻辑生成验证条件，其中仅对位置进行全称量化（以判断可满足性），并使用从前景类型映射到前景类型或背景类型（整数、位置集合等）的函数。

直观而言，对于每个霍尔三元组 {前置条件} s {后置条件}，我们将构造如下形式的验证条件：((前置条件 $\wedge T$) $\Rightarrow$ 后置条件) $\wedge SC$ 其中 T 刻画基本块 s 对状态和堆所做变换的语义，而 SC 是一个支撑条件，要求 {$Post$} 的支撑集与前置条件 的支撑集相同，但不包括 s 中发生的位置分配和释放（包括对其他函数的调用）。对于带有宽松后置条件的霍尔三元组 {前置条件} s {RP: 后置条件}，验证条件类似，只是支撑条件要求后置条件的支撑集是前置条件的支撑集的子集，但不包括位置的分配和释放。无堆后置条件 {前置条件} s {HP: 后置条件}则没有支撑条件。

考虑一个霍尔三元组 {α} s {β}，其中 s 是一个基本块，α，β 是框架逻辑中的注解。我们使用局部配置 的概念来构造验证条件。我们描述如何在基本块的语句间转换局部配置，每次从左到右处理基本块中的一条语句。局部配置是一个五元组（T，A，H，Fr，RD）。处理完基本块的一个前缀s' 后，s中的 T 对应于一个公式，该公式描述了变换s' 作用于满足前置条件的状态后得到的状态。分量 A 表示该前缀之后已分配位置的集合，H 表示执行该前缀后的堆，而Fr 表示在该前缀对堆进行修改期间收集到的框架规则集合。最后，RD 表示一组新的递归定义，用于表示在执行该前缀 s'期间，我在中间堆上求值的递归定义。

初始时，T 设为Pre，A 设为霍尔三元组的前置条件 α 的支撑集，即$Sp(\alpha)$；H 最初只是表示初始堆的一组未解释函数，而Fr 为空集。集合RD 初始化为 D，即 I 在初始堆上的递归定义。形式化地：

定义 *5.2.* 逻辑配置是一个元组 (T, A, H, Fr, RD) 其中：

- T 是一个无量词逻辑公式，用于描述程序变换。T 是基于程序变量和RD中定义的递归函数表示的。
- A 是一个类型为位置集合的项，表示当前的已分配位置集合。
- H 是一个用于记录当前堆的映射；它将每个指针 $f \in F$ 关联到一个映射$H(f)$：$Loc \to \sigma_\prime$，其中 $\sigma_\prime$ 是位置的前景类型或背景类型。$H(f)$表示为 $\lambda u.t(u)$，其中 t 是关于 u 和程序变量的项。
- Fr 是一个全称量化的一阶公式集合，表示针对堆的每次变异收集的一组框架规则。
- RD是一组函数符号的递归定义，用于刻画中间堆的性质/数据结构。

堆映射 H将每个指针 $f \in F$关联到一个映射 $H(f)$，该映射将位置映射到相应的类型（位置排序或背景排序）。请注意，这些映射因此满足我们所需的一阶完备性单向条件。映射 $H(f)$ 本身通过逻辑中的无量词项来表示，其中涉及其他函数符号，例如刻画初始堆片段的函数。例如， $H(f)$ 可以是公式 $\lambda u.ite(u = z,\ g_1(z)\ g_2(u))$，其中 g_1， g_2是函数符号，z 是程序变量。一般而言，我们会在处理基本块时用新的函数符号扩展签名。

我们从 I 中函数符号的一组递归定义开始。在构造验证条件时，我们将 I 中函数的这些定义调整为引用在新堆上解释的函数定义。

具体而言，对于任意 H和 $I \in \mathrm{I}$，我们引入一个新的函数符号 I_H，并为 I_H添加一个递归定义。该递归定义 $I[H]$是 I的定义，其中递归定义中提及的每个指针和数据字段 $f \in \mathrm{F}_m$ 都替换为 $H(f)$。换言之，对于 I 中每个递归定义函数 $I \in$，其中 I定义为 $I(x) =_{lfp} \rho(x)$，我们给出定义$I[H]$： $I_H(x) =_{lfp} \rho[H(f)/f](x)$，其中 $\rho[H(f)/f]$是 ρ，对于每个 $f \in \mathrm{F}_m$，将 f的每次出现都替换为项 $H(f)$。

例如，考虑上文提到的 H示例，并考虑定义 $lseg(x,\ y) =_{lfp} (x = y) \vee lseg(f(x)\ y)$。那么，$lseg[H]$的定义是 $lseg_H =_{lfp} (x = y) \vee lseg(ite(x = z,\ g_1(z)\ g_2(x))\ y)$。

验证条件的生成如图 9所示。不过，在详细解释规则之前，我们先讨论一下框架规则（形式化描述同样见图9），该规则用于这些规则中。

框架条件： 对于任意递归定义函数符号 $I(\vec{y}) \in$I、两个堆映射 H、 H'以及一个位置集合 X，我们将框架规则定义为

$$fr(I, X, H, H') \ :\ \forall \vec{y} \in Loc, (X \cap Sp(I_H(\vec{y})) = \emptyset) \implies (I_{H'}(\vec{y}) = I_H(\vec{y}))$$

上述公式旨在用于这样的情形：堆 H 被转换为堆 H'，其中修改仅发生在位置 X上。它表示，如果递归定义在 $\vec{y}$上的支撑集（在 H中求值）与 X不相交，那么它在 H'中的值与在 H中的值相同。注意，该公式仅对位置这一前景类型进行全称量化，因此其到 FORD 的翻译将属于 L_{oneway} 片段。

我们用 $fr(X,\ H,\ H')$ 表示对每个 $I \in$I 的 $fr(I,\ X,\ H,\ H')$ 的合取；该合取说明了 I 中每个递归定义函数的框架条件。

验证条件生成器：

定义 *5.3*（基本块的验证条件）。对于基本块的霍尔三元组 $\{\alpha\}\ s\{\beta\}$ 以及具有宽松后置条件的基本块，其对应的验证条件定义如下：

$$VC(\{\alpha\}\ s\ \{\beta\}) = \left(\bigwedge Fr \wedge T\right) \Rightarrow (\beta \wedge Sp(\beta) = A)$$

$$VC(\{\alpha\}\ s\ \{RP : \beta\}) = \left(\bigwedge Fr \wedge T\right) \Rightarrow (\beta \wedge Sp(\beta) \subseteq A)$$

$$VC(\{\alpha\}\ s\ \{HP : \beta\}) = \left(\bigwedge Fr \wedge T\right) \Rightarrow (\beta)$$

其中 $(T, A, H, \mathrm{Fr}, \mathrm{RD})$ =验证条件$((T_0, A_0, H_0, \mathrm{Fr}_0, \mathrm{RD}_0)\ s)$，其中 $T_0 = \alpha, A_0 = \mathrm{Sp}(\alpha)$， $H_0(f)$ $= \lambda u.f(u)$（对每个 $f \in F$），$\mathrm{Fr}_0 = \emptyset$，$\mathrm{RD}_0 = \mathrm{RD}$，而验证条件是图 9 中形式化定义的变换器。

现在我们说明变换器验证条件背后的直观含义。对于赋值 x :=y（**断言**），我们只需将条件 $x = y$ 作为合取项加入变换公式中。请回想，我们假设基本块采用静态单赋值形式，并且这是对变量 x的唯一一次赋值，因此该合取项刻画了一个程序配置，其中 x 与 y 被约束为相等。对背景排序中的变量进行赋值的标量赋值规则（**标量赋值断言**）与此类似。

$$\frac{T' = T \wedge (x = y)}{\text{验证条件}(\sigma, x := y) = (T', A, H, Fr, RD)}\ \textsc{Assn} \qquad \frac{T' = T \wedge (v = \text{是})}{\text{验证条件}(\sigma, v := \text{是}) = (T', A, H, Fr, RD)}\ \text{标量赋值断言}$$

$$\frac{T' = T \wedge (x = H(f)(y))}{\text{验证条件}(\sigma, x := y.f) = (T', A, H, Fr, RD)}\ \text{解引用}$$

$$\frac{\begin{array}{c} H'(f) = \lambda arg.\ ite(arg = y,\ x,\ H(f)(arg)) \text{对每个 } f' \in \mathcal{F}m, f' \neq f \\ Fr' = Fr \cup \{fr(\{y\}\ H, H')\} \qquad RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\} \end{array}}{\text{验证条件}(\sigma, y.f := x) = (T, A, H', Fr', RD')}\ \text{变异}$$

$$\frac{\begin{array}{c} T' = T \wedge (x \notin A) \qquad A' = A \cup \{x\} \\ H'(f) = \lambda \text{参数}.\ \text{条件表达式}(\text{参数} = x, \text{默认值} f,\ H(f)(\text{参数})), \text{对每个 } f \in \mathrm{F}m,\ \text{其定义域为 } A' \\ RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\} \end{array}}{\text{验证条件}(\sigma,\ \text{分配}(x)) = (T, A', H', Fr, RD')}\ \text{分配}$$

$$\frac{A' = A \backslash \{x\} \qquad H'(f) = H(f) \text{ 对每个 } f \in \mathrm{F}m,\ \text{定义域为 } A' \qquad RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\}}{\mathrm{VC}(\sigma, free(x)) = (T, A', H', Fr, RD')}\ \textsc{Free}$$

$$\frac{\text{验证条件}(\sigma,\ P) = \sigma'}{\text{验证条件}(\sigma, P; Q) = \text{验证条件}(\sigma', Q)}\ \text{顺序组合} \qquad \frac{T' = T \wedge \alpha}{\text{验证条件}(\sigma,\ \text{假设}(\alpha)) = (T', A, H, Fr, RD)}\ \text{假设}$$

$$\frac{\begin{array}{c} g \in G \qquad \alpha(\bar{x}) = \text{前置条件}(g) \\ \beta(\bar{x}, \bar{y}) = Post(g)\ H'(f) = \lambda arg.\ \text{条件表达式}(arg \in \text{支撑集}(\alpha_H[\bar{x} \leftarrow \bar{a}])\ f_{fresh}(arg)\ H(f)(arg))\ T' \\ = T \wedge \beta_{H'}[\bar{x} \leftarrow \bar{a}, \bar{y} \leftarrow \bar{b}] \qquad A' = (A \backslash \text{支撑集}(\alpha_H[\bar{x} \leftarrow \bar{a}])) \cup \text{支撑集}(\beta_{H'}[\bar{x} \leftarrow \bar{a}, \bar{y} \leftarrow \bar{b}]) \\ Fr' = Fr \cup \{fr(\text{支撑集}(\alpha_H[\bar{x} \leftarrow \bar{a}])\ H, H')\} \qquad RD' = RD \cup \{I[H'] \mid I \in \mathcal{I}\} \end{array}}{\text{验证条件}(\sigma, \bar{b} := g(\bar{a})) = (T', A', H', Fr', RD')}\ \textsc{Call}$$

图9. 验证条件生成：基本块的谓词转换器。这里，$\sigma = (T, A, H, \mathrm{Fr}, \mathrm{RD})$ 是应用验证条件转换器之前的配置，而 $\mathrm{fr}\,(X, H, H') = \bigwedge_{I \in \mathrm{I}} \mathrm{fr}\,(I, X, H, H')$。我们还假设存在一组（固定的）可调用函数 G。每个函数 $g \in G$都标注有前置条件 $\mathrm{Pre}(g) = \alpha(\bar{x})$ 和后置条件 $\mathrm{Post}(g) = \beta(\bar{x}, \bar{y})$。By α_H，我们指的是 α，其中每个 $f \in \mathrm{F}m$ 都替换为 $H(f)$，每个 $I \in \mathrm{I}$ 都替换为 I_H。By f_{fresh}，我们指的是一个具有 f 签名的新鲜函数符号。

对于解引用 x := y.f（**解引用**），变换会使用函数 $H(f)$ 查找当前指针 f，并添加一个合取项，将 x与 $H(f)(y)$等同。请注意， $H(f)(y)$是公式 $H(f)$，其中将 y 代入其参数，因此结果是一个类型适当的无量词项。**断言** 和**解引用**规则只会改变变换公式，而假设（**假设**）和顺序组合（**顺序组合**）规则则符合预期。

对于**变异** y.f := x，转换组件保持不变，但堆会更新以反映此次变异，其中 $H_{\prime}(f)(y) = x$，而在其他位置上为 $H(f)$。我们还引入了针对当前堆 H' （将其添加到 RD'）调整后的 I 的递归定义。此外，还引入框架规则 $fr(\{y\}\ H, H')$：当 y 不属于这些定义在该元组上的支撑集时，将这些新的递归定义在任意元组上的取值设为旧定义的取值。

对于函数调用（**调用**），我们引入一种优雅的方式来处理堆变换。对于每个指针和数据字段 $f \in \mathrm{F}m$，我们引入一个新鲜的函数符号 f_{fresh}。映射$H_{\prime}(f)$现在是一个 ite（如果-那么-否则）项，其中 $H_{\prime}(f)(u)$ 的求值结果为 $H(f)(u)$，也就是指针 f 的旧值，前提是 u 不属于被调用函数 g 的前置条件的支撑集。否则， $H(f)(u)$ 的求值结果为 $f_{fresh}(u)$，这是一个任意值，因为 f_{fresh} 是

未解释的。转换组件吸收了对 g 的调用所保证的后置条件，其中后置条件中的递归定义函数 R 被修改为函数 $R_{H'}$。递归定义会针对调用后的新堆 H' 进行调整，并添加到 RD' 中。新的已分配集合通过从旧的已分配集合中移除 g 的前置条件支撑集中的位置，并重新加入 g 的后置条件支撑集中的位置而得出。这处理了 g 在执行期间可能分配或释放的位置。最后，我们添加框架条件公式 $fr(Sp(\alpha[\bar{x} \leftarrow \bar{a}])\ M,\ M')$，它表明，对于任何与 g 的前置条件支撑集交集为空的递归定义，在新堆中都保持不变。

分配语句（**分配**）的变换会添加一个假设，指出分配的位置尚未出现在当前已分配集合 A中；随后将其加入新的已分配集合，并更新堆映射，使新位置的指针指向默认值。释放语句（**释放**）的变换只需将位置 x 从已分配集合中移除。尽管释放位置不会改变堆函数 H 的逻辑表达式，但其定义域会改变。因此，每当发生分配或释放时，我们都会通过更新递归定义 分量，在新堆上重新计算递归定义。请注意，我们不会检查被释放的位置是否确实已分配，因为如上一小节所述，这些检查会在其他基本块中进行。

图 8 底部三元组所用的算法如下。黑色文本是待证明的三元组。绿色文本显示了逻辑配置的变更。前两条语句只需将相应的项添加到程序变换中。对于变异 $x.\text{next} := y$，我们更新堆 H，使其反映这一变更。然后，我们在新堆上定义递归函数。最后，我们添加框架规则，将 H 上的递归函数与 H_1 上的递归函数联系起来。函数调用 ret := RevPrepend(tmp x) 使用了类似的思路。首先，我们必须更新堆。我们知道，该调用修改的堆部分是调用前置条件的支撑集，即 $\text{Sp}(\alpha_{H_1}(\text{tmp } x))$。因此，新堆 H_2 仅在 H_1 之外有所不同。此外，调用的后置条件 $\beta_{H_2}(\text{tmp } x, \text{ret})$ 在新堆上成立。新的已分配集合由未被调用修改的部分与该调用后置条件的支撑集取并得到。最后，我们添加一条框架规则，将堆 H_1 和 H_2 联系起来。此基本块生成的验证条件为 $(\text{fr}\ (x, H, H_1) \wedge \text{fr}\ (\text{Sp}(\alpha_{H_1}(\text{tmp } x))\ H_1, H_2) \wedge T_2) \implies \varphi \wedge \text{Sp}(\varphi) = A_2$，其中 $\varphi =$链表$_2$(ret) $\wedge$ (键集合$_2$(ret) =(Old (键集合(x)) $\cup$Old(键集合(y))))。

(链表 (y)) $\wedge$ 假设($x \neq$ 空指针); # $T \leftarrow$ 支撑集(链表$_2$(空指针) *tmp := x.后继; # $T \leftarrow T \wedge$(tmp =后继 (x)) x.后继 := y; # $H_1 = H$[后继 $\leftarrow \lambda$*arg.* 条件表达式(*arg* = x, y, next (arg))]# 链表$_1$ = 链表[H_1], 键集合$_1$ = 键集合[H_1], RD_1 = { 链表, 键集合, 链表$_1$,键集合$_1$} # Fr_1 = {*fr* (x, H, H_1) } ret := RevPrepend(tmp x) # α_{H_1}(tmp x) =(链表$_1$(tmp) $\wedge$ 链表$_1$(x) $\wedge$ 支撑集(链表$_1$(tmp)) $\cap$ 支撑集(链表$_1$(x)) = ϕ) # H_2[f] = λ*arg.*条件表达式(*arg* $\notin$ 支撑集(α_{H_1}(tmp x)) H_1(f) (arg), next$_2$(arg)) for $f \in$ {next, key}. # 链表$_2$ = 链表[H_2], 键集合$_1$ = 键集合[H_2], $RD_2 = RD_1 \cup$ {链表$_2$,键集合$_2$} # β_{H_2}(tmp x, ret) =(链表$_2$(ret) $\wedge$键集合$_2$(ret) =Old (键集合$_1$(tmp)) $\cup$Old (键集合$_1$(x))) # $T_2 = T \wedge \beta_{H_2}$(tmp x, ret) # A_2 =(A|支撑集(α_{H_1}(tmp x))) $\cup$ 支撑集(β_{H_2}(tmp x, ret)) # $Fr_2 = Fr_1 \cup$ {*fr* (支撑集 (α_{H_1}(tmp x)) H_1, H_2) } {*Li*

$st_2(ret) \wedge K\ eys_2(ret) = (Old(K\ eys(x)) \cup Old(K\ eys(y)))\}$

可靠性与完备性。 我们的验证条件生成对于基本块上的霍尔三元组有效性是可靠且完备的。固定一个三元组 $\{\alpha\}\ s\{\beta\}$ 其中 s是一个基本块，并令 验证条件 为定义5.3中的映射。则：

定理 *5.4*（基本块的验证条件生成的可靠性）。如果 *VC({α} s{β})* 是框架逻辑中的有效公式，那么按照定义 *3.1* 的含义，*{α} s{β}* 是有效的。同样，如果 *VC({α} s{RP:β})* 在框架逻辑中有效，那么三元组 *{α} s{RP: β})* 是有效的。

定理 *5.5*（无函数调用基本块的验证条件生成的完备性）。设 s 是一个不包含函数调用的基本块。如果 $\{\alpha\}\ s\{\beta\}$ 有效（定义 *3.1*），那么 $VC(\{\alpha\}\ s\{\beta\})$ 有效。同样，如果 $\{\alpha\}\ s\{RP:\ \beta\}$ 有效，那么 $VC(\{\alpha\}\ s\{RP:\ \beta\})$ 有效。

我们省略这些定理的证明，因为根据验证条件转换器和操作语义的定义，证明相当直接（见附录 B）。完备性证明中的关键论点是，我们通过更新堆映射来精确地建模变异 H。类似地，我们也使用符号化已分配集合 A精确地建模分配和释放；对于不含函数调用的代码块，该集合与操作语义所定义的实际已分配集合相对应。虽然上述定理不包含函数调用，但对于含函数调用的基本块，依据模块化操作语义，我们的验证条件同样是精确的。在该语义中，假设函数调用会清除前置条件的堆片段，并将其替换为满足后置条件的堆片段。

我们的结果本身就很有意义，也展现了我们框架的强大能力。针对分离逻辑等其他逻辑的验证条件生成，当前的研究通常并不完备 [7]。

5.4 验证验证条件

在前文中，我们介绍了如何将带有无量词 FL 公式注释的霍尔三元组的有效性，归约为 FL 公式的有效性。随后，我们通过以下方式在 FL 中处理生成的验证条件：(a) 将其转换为等可满足性 FORD 公式；(b) 使用自然证明——一种在先前工作中提出的一阶完备自动化技术 [38, 54, 59]，用于检查 L_{oneway} 片段中 FORD 公式的有效性。自然证明基于系统化的量词实例化和 SMT 求解。第一阶段中从 FL 到 FORD 的转换采用了与早期 FL 相关工作中提出的方法类似的构造 [43, 44],，但我们还特别处理了云算子，并确保验证条件的转换结果属于L_{oneway}。直观而言，该转换在 FORD 本身中编码了$Sp(\cdot)$ 和 $[\cdot]$ 算子的语义。我们在附录F中详细介绍这两个阶段。

6 实现与评估

6.1 实现

我们基于第5节和第5.4节所述的机制，开发了框架逻辑验证器（*FLV*）工具。该工具允许用户使用支撑集 Supp $[\cdot]$ 运算符，以无量词框架逻辑编写堆操作程序及注解。该工具还允许用户编写归纳引理，并通过基于前不动点的归纳方法自动证明这些引理 [38]。

FLV 工具会生成验证条件，并将其转换为 FORD。这些无量词 FORD 公式随后由一个现有的自然证明求解器进行验证 [41]；该求解器实现了第 F.2 节所述的过程，通过实例化递归定义，并使用 Z3 [15] 对所得的无量词公式进行推理。

实现还通过几种方式进行了优化。首先，我们只在特定关键点引入新的递归定义和框架规则——函数调用之前和之后，以及基本块末尾。请注意，第 5 节所述的表述会在每次变异、分配、释放和函数调用时引入它们（如第 5 节所述）。这些经过优化的框架规则会将单独的修改“批处理”，以涵盖关键点之间程序的足迹。其次，该工具一开始会采用一组精简且常用的实例化，这一做法基于先前关于自然证明的工作（ [54]）：递归定义和引理会针对上一个关键点的相关足迹进行实例化；在函数调用之前和之后，以及基本块末尾，框架规则会针对当前变量及其一层指针解引用进行实例化。最后，我们使用 SMT 求解器简化转换为 FORD 的结果，然后再对其进行验证。

我们基于 FLV 开发了 *SL-FL* 验证器（SLFLV）。该工具接收带有*SL-FL* 规格说明的程序。SLFLV 会将这些程序转换为带有 *FL* 注解的程序（见图 6），然后由 FLV 验证生成的程序。

从 *SL-FL* 到 框架逻辑 的转换会导致公式大小膨胀。我们开发了该工具的优化版本，以控制这种膨胀，尤其是在嵌套合取中。首先，在自底向上将 *SL-FL* 公式转换为框架逻辑公式时，对于每个 *SL-FL* 子公式 α，我们都会计算一个简单的框架逻辑公式 $\alpha^\dagger$，使得 支撑集$(\Pi(\alpha))$ = 支撑集$(\alpha^\dagger)$；每当需要使用 $\Pi(\alpha)$ 的支撑集时，我们改用 $\alpha^\dagger$ 的支撑集。具体来说，对于 $\alpha = \beta \wedge \gamma$，我们令 $\alpha^\dagger$ 简单地等于 $\beta^\dagger$（因为转换过程会独立地约束 支撑集$(\beta^\dagger)$ =支撑集$(\gamma^\dagger)$）。其次，我们通过语法分析跟踪支撑集为空的子公式，并移除用于检查这些支撑集与其他集合的交集是否为空的约束。我们在评估中考察了这些优化的效果。

6.2 基准测试套件

我们在一组数据结构程序上评估了所提方法的表达能力和性能。每个程序都分别标注了 框架逻辑 和 *SL-FL* 规约。我们的套件包含 29 个程序，涉及对多种数据结构的操作，例如单链表和双链表、循环链表、二叉搜索树、红黑树、树堆等。我们从先前工作 [54]中提炼出一组核心基准测试，构成了该套件。我们考察了每种结构的各种关键数据结构操作算法，例如插入、删除、搜索、排序和遍历。基准测试套件的摘要见附录 G，其中还包含规约示例、所用引理以及验证条件生成的说明性示例。

6.2.1 规约。我们为程序标注完整的函数式规约。契约会根据例程对输入数据结构和参数产生的影响，精确规定返回的数据结构及其内容。我们使用递归定义来定义数据结构及其上的各种度量，并用它们编写规约。为了便于编写后置条件，我们引入一个*Old* 运算符。对于递归函数 R，$Old(R(x))$ 等同于在程序开始时假定 $r_x = R(x)$，并将其赋给一个新的变量 r_x，然后用该变量替代$Old(R(x))$。特别是，这意味着$Sp(Old(R(x)))$ 为空，因为它只是变量的一种语法糖。

6.2.2 引理。尽管我们的过程是一阶完备的，即当验证条件相对于定义的不动点解释成立时，它们能保证验证这些验证条件，但这种语义与递归定义 [38]的最小不动点语义之间仍存在差距。用户提供的归纳引理可以弥合这一差距。预期这些归纳引理通过简单归纳来证明，其中引理本身充当归纳假设，并使用不涉及最小不动点解释的一阶逻辑证明（见 [38, 54, 59]）。

引理通常用于关联数据结构的不同性质，并且通常独立于处理这些数据结构的方法（例如，有序列表是一种单链表）。

我们的工具使用归纳和自然证明来证明这些引理，并在证明验证条件时利用已证明的引理。目前，基准测试的引理是用框架逻辑编写的。不过，我们也可以支持使用*SL-FL* 编写引理，在其中我们将蕴涵$\alpha \models \beta$解释为 $\Pi(\alpha) \implies \Pi(\beta) \wedge Sp(\Pi(\alpha)) = Sp(\Pi(\beta))$.

需要注意的是，已有大量正交研究致力于解决自动合成这些引理的问题，无论针对函数式程序还是命令式程序，都取得了相当大的成功（当然，尚未尽善尽美）。参见 [34, 41, 42, 60, 63, 73]。

Data 结构	基准测试	数量 基本块	数量 权限检查	框架逻辑 验证 Time	*SL-FL* 验证时间（未优化）	*SL-FL* 验证时间（已优化）
SLL	追加	2	5	1s	1s	1s
	全部复制	2	6	1s	1s	1s
	删除	3	9	1s	1s	1s
	Find	3	5	1s	1s	1s
	尾部插入	2	7	1s	1s	1s
	头部插入	1	3	1s	1s	1s
	反转	3	7	1s	1s	1s
	插入排序	5	14	3s	4s	3s
	归并排序	10	26	6s	7s	6s
	快速排序	9	25	13s	18s	15s
DLL	尾部插入	2	9	1s	1s	1s
	头部插入	2	8	1s	1s	1s
	删除中间节点	4	11	2s	3s	3s
	中间插入	4	10	2s	3s	3s
已排序 List	删除	3	9	1s	1s	1s
	Find	4	6	1s	1s	1s
	插入	3	10	2s	3s	3s
CL	删除前端	5	12	6s	6s	6s
	Find	6	10	2s	3s	3s
	头部插入	2	6	2s	2s	2s
BST	删除	8	27	8s	11s	10s
	Find	4	8	2s	2s	2s
	插入	4	13	3s	4s	4s
	右旋	1	5	1s	1s	1s
Tree 遍历	二叉搜索树转链表	3	11	15s	22s	19s
	中序遍历	2	8	3s	7s	8s
树堆	删除	9	32	18s	24s	20s
	Find	4	8	3s	3s	3s
RBT	插入	11	83	42s	1 分 6 秒	56s

表 1. 验证工具的性能（SLL = 单向链表、DLL = 双向链表、CL =循环链表、BST =二叉搜索树、RBT = 红黑树）。

6.2.3 具有相同支撑集的定义。用户还可以声明一对递归定义的支撑集相等。例如，对于树，二叉搜索树、键集合、最小值、最大值和高度的支撑集都相同。该工具会证明这一性质（将其视为归纳引理），并减少支撑集递归定义的数量，从而提升性能。

6.3 评估

我们在基准测试套件上评估了工具。实验在一台 MacBook Pro (Mac14,9) 上进行，该设备配备 Apple M2 Pro 处理器、10 个 CPU 核心和 16 GB 内存。

我们在表格中展示评估结果1。表格列出了每个程序中的基本块数量和权限检查数量——检查所有解引用位置是否包含在前置条件的支撑集中或是新分配的位置，并检查对被调用函数授予的权限是否为方法权限的子集，以及返回权限的紧致性（后置条件的支撑集）。表中报告了框架逻辑（FL）的验证时间，以及 SL 基准测试中工具未优化版和优化版所用的时间。该时间包括：(a) 逻辑之间的转换——*SL-FL* 转换为 FL、FL 转换为 FORD——包括优化，以及 (b) 所有验证条件的验证，包括权限检查；(c) 引理；以及 (d) 支撑集相等。

我们的工具能够高效地验证所有基准测试，验证每个基准测试所需的时间都不到一分钟。在这些基准测试中，我们最初确定的实例化已足以证明程序正确，无需进一步实例化。

正如预期，*SL-FL* 基准测试所需的时间比 *FL*更长。我们认为，这既是转换膨胀所致，也与分离逻辑中的典型公式要求执行某些多余检查有关。不过，我们的优化平均将验证时间缩短了 4%，使其更接近 FL 的验证时间。

当证明器失效：错误查找与反例。 我们的验证方法分为两部分：(1) 在 FL（最终也包括 FORD）中生成验证条件；(2) 使用一阶完备有效性技术来证明这些验证条件（见第 5.4 节）。FL/FORD 使用预期的最小不动点语义解释递归定义，但我们采用的验证条件有效性检查技术并非如此——它们仅对某种不动点语义完备。请注意，定义遵循不动点语义的模型集合更大。例如，考虑定义 $\text{链表}(x) :=(x = \text{空指针}) \vee \text{链表}(\text{后继}\ (x))$。在不动点语义下，模型 $e_1 \xrightarrow{\text{后继}} e_2 \xrightarrow{\text{后继}} \text{空指针}$ 中的元素 e_1 当然满足 链表的定义，但它也可能被解释为指向模型 $e_1 \xrightarrow{\text{后继}} e_2 \xrightarrow{\text{后继}} e_1$ 中的一个链表，该链表表示一个二元素循环。后一个模型不满足最小不动点语义下的定义。具体而言，这意味着当我们的 FORD 证明器失效时，它所针对的是这个更大的模型类。实际中，实例化不足也会导致失效。我们的有效性技术基于对定义（“展开”）及其他量化公式（例如框架推理公理）进行系统化实例化。实例化分轮进行，一阶完备性在极限意义下成立：如果验证条件在不动点语义下有效，证明器最终会在某一轮成功。每一轮都会向 SMT 求解器发送一个可判定的无量词查询。

总的来说，证明器的失败有三个来源：(a) 该 VC 在一阶逻辑中有效，但实例化不足；(b) 该 VC 在 FO*lfp* 中有效，但在一阶逻辑中无效；(c) 该 VC 在 FO*lfp* 中无效（即程序/规格存在错误）。SMT 求解器在某一轮返回的模型，只能证明在当前可用的实例化集合下无法证明该 VC，却不能区分上述三种情况。

在构建基准测试套件时，我们通过找出求解器无法证明的基本块，并半自动地追踪证明义务，来识别规格和实现中的错误。例如，我们找出后置条件中无法证明的合取项，或在基本块的某些中间位置添加额外的断言。这帮助我们发现了许多错误来源，包括前置条件不足、实现中遗漏边界情况导致的错误，以及对某些归纳引理的需求。请注意，正如第 6.2.2节所述，归纳引理针对的是第二种失败模式。我们还发现，将后置条件弱化为 Relaxed 或无堆后置条件，特别有助于排查与支撑集及已分配位置集合相关证明中的错误。例如，这使我们能够通过有关支撑集的性质来加强函数调用契约。

有许多方法可以帮助区分真正无效的验证条件与仅非一阶逻辑有效（需要归纳引理）的验证条件。一种可能的方法是让 SMT 求解器寻找一个有界大小（例如包含 5–6 个位置）的真实反例（即具有最小不动点解释的模型，而非非标准模型）。通常这很困难，因为需要表达确定最小不动点所涉及的无界计算；不过，若能为计算长度确立实用的上界，也可能奏效。然而，我们认为，本逻辑的设计使这种方法特别有吸引力，因此将如何制定一种通用技术，以合成一阶完备堆逻辑中无效公式的反例，留待未来工作探讨。

7 相关工作

框架逻辑方面的研究 [43, 44] 是最相关的先前工作，我们已在论文中对此进行了详细讨论。还有许多其他堆逻辑 [3, 9, 23, 31, 54, 61, 64] 在设计选择上各不相同，例如公式中的堆片段是否隐式表示、公式是否具有唯一的堆等。最为流行的是分离逻辑 [16, 49, 50, 61]，其中公式的堆片段是隐式的，但并不唯一。动态帧 [31, 32] 以及区域逻辑等相关方法 [2–4] 允许用户显式指定堆片段，并利用它们推理程序的修改。隐式动态帧方面的研究 [36, 53, 64, 65]结合了分离逻辑中隐式堆片段的思想和动态帧中可显式访问堆片段的思想，由此产生了 Chalice 等实现 [53] 以及 Viper [18, 22, 40]。尤其是，Viper 提供了一个通用框架，可在其中编码多种权限逻辑，包括分离逻辑。然而，它会为 SMT 求解器生成量化验证条件，而这些求解器所使用的逻辑本质上是不完备的。另有研究探讨了将 VeriFast [29] 谓词转换为隐式动态帧 [30]。

关于使用分离逻辑进行推理，已有大量先前工作。第一类先前工作开发了可判定推理的片段 [5–7, 14, 25, 46, 52, 55, 56]。另有工作研究了可用有效命题推理（EPR） [26–28]描述的堆性质的可判定性。第二类工作将分离逻辑转换为一阶逻辑，并使用一阶逻辑（FOL）的推理机制 [13, 38, 39, 54, 56–59, 66]。此外，还有一些技术使用循环证明 [11, 68]进行分离逻辑推理。

[9] 中的工作提出了一种类似于框架逻辑（FL）的逻辑，并使用 Why3 [20, 21]推理验证条件。近期工作 [45] 提出了一种避免不完备性且无需使用归纳引理的技术，但该技术需要预先为一组性质开发一组映射；当性质发生变化时，还必须重新设计这些映射。

我们在工作中采用了自然证明 [38, 39, 54, 59]这一推理技术。自然证明使用的量词实例化技术与文献中的许多其他基于展开的技术 [29, 33, 35, 48, 66]类似。

关于 FL 的研究提出了一个名为精确分离逻辑的分离逻辑片段，该片段具有确定且唯一的堆片段，并提供了将该片段转换为 FL 的方法。其他先前的文献也研究了具有确定堆片段的片段 [51, 54, 59]，因为更容易为这些片段开发自动推理。我们相信，本工作中开发的技术能够为分离逻辑的自动化开辟新途径。

8 结论

我们提出了两种逻辑：框架逻辑和分离逻辑，它们支持一阶完备程序验证；其中后者要求系统最终证明递归定义在不动点语义下的所有有效定理。我们的工作应对了不完备性的各种威胁，精心设计了具有特定语义的逻辑及其推理过程，使其达到一阶完备，并且在实践中也有效。

我们的工作为未来最有前景的方向铺平了道路：构建新的程序逻辑和验证范式，使其达到一阶完备的理论标准。我们的评估表明，通过采用一阶逻辑完备过程，仅对可判定逻辑反复调用 SMT 求解器，就有可能实现高效且一阶完备的过程。因此，借助成熟的验证条件生成管道（例如 Viper/Boogie 管道），为丰富的编程语言实现此类一阶完备推理，是亟待探索的重要方向。

为了对带有递归定义的一阶逻辑进行推理，近期的研究 [34, 41] 提出了归纳引理综合算法。将引理合成纳入我们的验证框架，可以减轻程序员当前编写这些归纳引理的负担（第6节介绍了我们基准测试中的相关内容）。

我们认为，我们为分离逻辑 *SL-FL*提出的替代语义值得进一步研究。特别是，既然我们已经证明可以使用一阶逻辑推理自动化其推理过程，那么在类似 SLComp 的分离逻辑竞赛中设置一个赛道 [62]，以鼓励工具开发领域的竞争，将会很有意义。

数据可用性声明

评审者可以在以下匿名代码库中查找 *SL-FL* 和 *FL* 基准测试的评估结果：https://github.com/verifproj/complete-logics-bench。验证条件生成工具将发布以供制品评估。

参考文献

[1] Alfred V. Aho 和 Jeffrey D. Ullman。1979年。数据检索语言的普适性。载于第6届 ACM SIGACT-SIGPLAN 编程语言原理研讨会论文集（得克萨斯州圣安东尼奥）（POPL '79）。计算机协会，美国纽约州纽约，110 119 10 1145 567752 567763 – 。https://doi.org/ . / 。

[2] Anindya Banerjee 和 David A. Naumann。2013年。全局不变量的局部推理，第二部分：动态边界。《ACM 杂志》60, 3, 19 2013 73 10 1145 2485981 文章（6月），页数。https://doi.org/ . /

[3] Anindya Banerjee、David A. Naumann 和 Stan Rosenberg。2008年。区域逻辑：关于全局不变量的局部推理。载于 ECOOP 2008 – 面向对象编程，Jan Vitek（编）。Springer Berlin Heidelberg，德国柏林，387–411。[4] Anindya Banerjee、David A. Naumann 和 Stan Rosenberg。2013年。全局不变量的局部推理，第一部分：区域逻辑。《ACM 杂志》60, 3, 18 2013 56 10 1145 2485982 文章（6月），页数。http://doi.acm.org/ . / [5] Josh Berdine、Cristiano Calcagno 和 Peter W. O'Hearn。2006年。Smallfoot：基于分离逻辑的模块化自动断言检查。载于第4届国际组件与对象形式化方法会议论文集（荷兰阿姆斯特丹）（FMCO'05）。施普林格出版社，德国柏林，115–137。https://doi.org/10.1007/ 11804192 6_[6] Josh Berdine、Cristiano Calcagno 和 Peter W. O'Hearn。2004年。分离逻辑的一个可判定片段。载于第24届软件技术基础与理论计算机科学国际会议论文集（印度金奈）（FSTTCS' *04* 97 109 10 1007 978 3 540 30538 5_9 ）。施普林格出版社，德国柏林，– 。https://doi.org/ . / [7] - - - - Josh Berdine、Cristiano Calcagno 和 Peter W. O'Hearn。2005年。基于分离逻辑的符号执行。载于第三届亚洲编程语言与系统会议论文集（日本筑波）（APLAS'05）。施普林格出版社，德国柏林，52 68 10 1007 11575467_5 – 。https://doi.org/ . / Régis Blanc、Viktor Kuncak、Etienne Kneuss 和 Philippe Suter。2013年。Leon 验证系统概述：通过转换为递归函数进行验证。载于第4届 Scala 研讨会论文集（法国蒙彼利埃）（SCALA '13）。计算机协会，美国纽约州纽约，文章1，10页。https://doi.org/10.1145/ 2489837.2489838[9] François Bobot 和 Jean-Christophe Filliâtre。2012年。分离谓词：一阶逻辑中的分离逻辑初探。载于形式化方法与软件工程，Toshiaki Aoki 和 Kenji Taguchi（编）。Springer Berlin Heidelberg，德国柏林，167–181。[10] Rémi Brochenin、Stéphane Demri 和 Etienne Lozes。2008年。论全能魔杖。载于计算机科学逻辑，Michael Kaminski 和 Simone Martini（编）。Springer Berlin Heidelberg，德国柏林，323–338。[11] James Brotherston、Dino Distefano 和 Rasmus Lerchedahl Petersen。2011年。分离逻辑中的自动循环蕴涵证明。载于第23届国际自动演绎会议论文集（CADE'11）。施普林格出版社，德国柏林，131 146 =2032266 2032278 – 。

[12] http://dl.acm.org/citation.cfm?id . Ashok K. Chandra 和 David Harel。1980年。关系查询的结构与复杂性。载于第21届计算机科学基础年会论文集（SFCS '80）。IEEE 计算机学会，美国，333–347。https: //doi.org/ 10 1109 1980 41 . /SFCS. . Wei-Ngan Chin、Cristina David、Huu Hai Nguyen 和 Shengchao Qin。

[13] 2007年。形状、大小和包属性的自动验证。载于第12届 IEEE 复杂计算机系统工程国际会议论文集（ICECCS ' *07* 307 320 10 1109 2007 17 ）。IEEE 计算机学会，美

[14] 国，– 。https://doi.org/ . /ICECCS. . Byron Cook、Christoph Haase、Joël Ouaknine、Matthew Parkinson 和 James Worrell。2011年。在分离逻辑一个片段中的易处理推理。载于第22届并发理论国际会议论文集（德国亚琛）（CONCUR'11）。施普林格出版社，德国柏林，235–249。[15] Leonardo De Moura 和 Nikolaj Bjørner。2008年。Z3：一种高效的可满足性模理论求解器。载于第14届系统构造与分析工具和算法国际会议论文集（匈牙利布达佩斯）（TACAS'08）。施普林格出版社，德国柏林，337–340。[16] Stéphane Demri 和 Morgan Deters。2015年。分离逻辑与模态：综述。《非经典逻辑应用杂志》25（2015），50–99。[17] H.D. Ebbinghaus、J. Flum 和 W. Thomas。1996年。数理逻辑。爱思唯尔科学。[18] Marco Eilers、Malte Schwerhoff 和 Peter Müller。2024年。自动分离逻辑验证器的验证算法。载于计算机辅助验证，Arie Gurfinkel 和 Vijay Ganesh（编）。施普林格自然瑞士，Cham，362–386。

[19] H.B. Enderton。2001年。逻辑数学导论。爱思唯尔科学。https://books.google.com/books?id=dVncCl EtUkC_[20] Jean-Christophe Filliâtre 和 Andrei Paskevich。2013年。Why3：程序与证明器的交汇之处。载于第22届欧洲编程语言与系统会议论文集（意大利罗马）（ESOP'13）。施普林格出版社，德国柏林，
125 128 10 1007 978 3 642 37036 6_8 – 。https://doi.org/ . / - - - - Jean-Christophe Filliâtre 和 [21] Andrei Paskevich。Why3：程序与证明器的交汇之处。载于第22届欧洲编程语言与系统会议论文集（意大利罗马）（ESOP'13）。施普林格出版社，德国柏林，
2013 3 125 128 10 1007 978 3 642 37036 6_8 – 。https://doi.org/ . / - - - - Stefan Heule、Ioannis T. Kassios、Peter Müller 和 Alexander J. Summers。2013年。利用抽象谓词和抽象函数生成权限逻辑的验证条件。载于 ECOOP 2013 – 面向对象编程，Giuseppe Castagna（编）。Springer Berlin Heidelberg，德国柏林，451–476。[23] Aquinas Hobor 和 Jules Villard。2013年。数据结构中共享的影响。载于第40届 ACM SIGPLAN-SIGACT编程语言原理年会论文集（意大利罗马）（POPL '13）。计算机协会，美国纽约州纽约，
[22] 523 536 10 1145 2429069 2429131 – 。
[24] https://doi.org/ . / 。Neil Immerman。可在多项式时间内计算的关系查询（扩展摘要）。载于第十四届 ACM 计算理论年会论文集（美国加利福尼亚州旧金山）（STOC '82）。计算机协会，美国纽约州纽约，
1982 147 152 10 1145 800070 802187 – 。https://doi.org/ . / 。
[25] Radu Iosif、Adam Rogalewicz 和 Jiri Simacek。带递归定义的分离逻辑的树宽度。载于自动演绎 – CADE-24，Maria Paola Bonacina（编）。Springer Berlin Heidelberg，德国柏林，21–38。[26] Shachar Itzhaky、Anindya Banerjee、Neil Immerman、Ori Lahav、Aleksandar Nanevski 和 Mooly Sagiv。2014年。通过有效命题公式对堆路径进行模块化推理。载于第41届 ACM SIGPLAN-SIGACT编程语言原理研讨会论文集（美国加利福尼亚州圣迭戈）（POPL '14）。ACM，美国纽约州纽约，
2013 385 396 10 1145 2535838 2535854 – 。https://doi.org/ . / 。Shachar Itzhaky、 [27] Anindya Banerjee、Neil Immerman、Aleksandar Nanevski 和 Mooly Sagiv。2013年。关于链式数据结构中可达性的有效命题推理。载于第25届国际计算机辅助验证会议论文集（俄罗斯圣彼得堡）（CAV'13）。施普林格出版社，德国柏林，
756 772 10 1007 978 3 642 39799 8_53 – 。https://doi.org/ . / - - - - Shachar Itzhaky、Nikolaj Bjørner、 [28] Thomas Reps、Mooly Sagiv 和 Aditya Thakur。2014年。性质导向的形状分析。载于第16届国际计算机辅助验证会议论文集（CAV'14）。施普林格出版社，德国柏林，
35 51 10 1007 978 3 319 08867 9_3 – 。
[29] https://doi.org/ . / - - - - Bart Jacobs、Jan Smans、Pieter Philippaerts、Frédéric Vogels、Willem Penninckx 和 Frank Piessens。2011年。VeriFast：一种功能强大、可靠、可预测且快速的 C 和 Java 验证器。载于第三届NASA形式化方法国际会议论文集（帕萨迪纳，加利福尼亚州）（NFM'11）。施普林格出版社，德国柏林，41–55。[30] Daniel Jost 和 Alexander J. Summers。2014年。从 VeriFast 谓词到隐式动态帧的自动编码。载于经过验证的软件：理论、工具与实验，Ernie Cohen 和 Andrey Rybalchenko（编）。Springer Berlin Heidelberg，德国柏林，202–221。[31] Ioannis T. Kassios。2006年。动态帧：在无限制的情况下支持框架化、依赖关系和共享。载于 FM 2006：形式化方法，Jayadev Misra、Tobias Nipkow 和 Emil Sekerinski（编）。施普林格出版社，德国柏林，268–283。[32] I. T. Kassios。2011年。动态帧理论。《形式化方面的计算》23，3（2011年5月），267–288。https://doi.org/10. 1007/s00165-010-0152-5[33] Matt Kaufmann 和 J. S. Moore。1997年。一种基于 Common Lisp 逻辑的工业级定理证明器。《IEEE 软件工程汇刊》23，4（1997年4月），203–213。https://doi.org/10.1109/32.588534[34] Cole Kurashige、Ruyi Ji、Aditya Giridharan、Mark Barbone、Daniel Noor、Shachar Itzhaky、Ranjit Jhala 和 Nadia Polikarpova。2024年。CCLemma：E-图引导的归纳等式证明引理发现。ACM程序设计语言会议录
8, 264 2024 27 10 1145 3674653 ICFP，文章（8月），页数。
[35] https://doi.org/ . / K. Rustan M. Leino。利用 SMT 求解器实现归纳自动化。载于第13届验证、模型检测与抽象解释国际会议论文集（费城，宾夕法尼亚州）（VMCAI'12）。施普林格出版社，德国柏林，
2012 315 331 10 1007 978 3 642 27940 9_21 – 。https://doi.org/ . / - - - - K. Rustan M. Leino 和 [36] Peter Müller。多线程程序验证的基础。载于编程语言与系统，Giuseppe Castagna（编）。Springer Berlin Heidelberg，德国柏林，378–393。https://doi.org/10. 1007/978-3-642-00590-9_27
2009 [37] 2004 10 1007 978 3 662 07003 1 Leonid Libkin。有 [38] 限模型理论要素。施普林格。https://doi.org/ . / - - - - Christof Löding、P. Madhusudan 和 Lucas Peña。自然证明与量词实例化的基础。PACMPL
2018 2, 2018 10 1 10 30 10 1145 3158098 POPL（），：–：。
[39] https://doi.org/ . / P. Madhusudan、Xiaokang Qiu 和 Andrei Ştefănescu。归纳树形数据结构的递归证明。载于第39届 ACM SIGPLAN-SIGACT编程语言原理年会论文集（费城，
2012

PA, USA）（POPL ’ 12 123 136 10 1145 2103656 2103673 ）。ACM，纽约，纽约州，美国，–。https://doi.org/。/。[40] Peter Müller、Malte Schwerhoff 和 Alexander J. Summers。2016。Viper：用于基于权限推理的验证基础设施。载于《第17届国际验证、模型检测与抽象解释会议论文集》第9583卷（美国佛罗里达州圣彼得堡）（VMCAI 2016）。施普林格出版社，柏林，海德堡，41–62。https://doi.org/ 10 1007 978 3 662 49122 5_2 。/ -- -- Adithya Murali、Lucas Peña、Eion Blanchard、Christof Löding 和 P. Madhusudan。2022年。面向带最小不动点一阶逻辑的归纳引理模型引导综合。ACM程序设计语言会议录 6, OOPSLA2，第191篇文章（2022年10月）， 30 10 1145 3563354 页。https://doi.org/。/ [41] Adithya Murali、Lucas Peña、Ranjit Jhala 和 P. Madhusudan。2023年。函数式程序性质的完备一阶推理。ACM程序设计语言会议录 7，OOPSLA2，第259篇文章（2023年10月），30页。https://doi.org/10.1145/ 3622835[43] Adithya Murali、Lucas Peña、Christof Löding 和 P. Madhusudan。2023年。带框架的一阶逻辑。《ACM 程序设计语言与系统汇刊》 45, 2, 7 2023 44 10 1145 3583057 文章（5月），页。https://doi.org/。/ [44] Adithya Murali 、 Lucas Peña 、 Christof Löding 和 P. Madhusudan。2020。带框架的一阶逻辑。载于《编程语言与系统》，Peter Müller（编）。Springer 国际出版，Cham，515–543。[45] Adithya Murali、Cody Rivera 和 P. Madhusudan。2024年。利用内在定义实现可预测验证。ACM程序设计语言会议录 8, 220 2024 26 10 1145 3656450 PLDI，第（6月）篇文章，页。https://doi.org/。/ [46] Juan Antonio Navarro Pérez 和 Andrey Rybalchenko。2011。+分离逻辑超位置演算堆定理证明器。载于《第32届 ACM SIGPLAN 编程语言设计与实现会议论文集》（PLDI ’11）。ACM，纽约，纽约州，美国，556–566。[47] Greg Nelson 和 Derek C. Oppen。1979年。通过协作判定过程进行简化。《ACM 程序设计语言与系统汇刊》 1, 2 1979 245 257 10 1145 357073 357079 （10月），–。https://doi.org/。/。[48] Huu Hai Nguyen 和 Wei-Ngan Chin。2008。通过引理增强程序验证。载于《第20届国际计算机辅助验证会议论文集》（美国新泽西州普林斯顿）（CAV ’08）。施普林格出版社，柏林，海德堡， 355 369 10 1007 978 3 540 70545 1_34 –。https://doi.org/。/ ---- Peter W. O’Hearn。《分离逻辑入门（以及自动程序验证与分析）》。载于《软件安全与保障——分析与验证工具》，Tobias Nipkow、Orna Grumberg 和 Benedikt Hauptmann（编）。北约和平与安全系列——D：信息与通信安全，第33卷。IOS Press， 286 318 10 3233 978 1 61499 028 4 286 –。https://doi.org/。/ ----- Peter W. O’Hearn、John C. Reynolds 和 Hongseok Yang。2001年。对改变数据结构的程序进行局部推理。载于《第15届计算机科学逻辑国际研讨会论文集》（CSL ’01）。施普林格出版社，伦敦，英国，英国， 1 19 =647851 737404 –。http://dl.acm.org/citation.cfm?id。[50] Peter W. O’Hearn、Hongseok Yang 和 John C. Reynolds。2012。分离与信息隐藏。载于《第31届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集》（意大利威尼斯）（POPL ’04）。ACM，纽约，纽约州，美国， 268 280 10 1145 964001 964024 –。https://doi.org/。/。[51] Jens Pagel。2004。分离逻辑判定过程：超越符号堆。博士学位论文。维也纳。[52] Matthew J. Parkinson 和 Alexander J. Summers。2020年。分离逻辑与隐式动态帧之间的关系。载于《编程语言与系统》，Gilles Barthe（编）。Springer Berlin Heidelberg，柏林，海德堡，439–458。[53] Edgar Pek、Xiaokang Qiu 和 P. Madhusudan。2014年。使用分离逻辑在 C 语言中对数据结构操作进行自然证明。载于《第35届 ACM SIGPLAN 编程语言设计与实现会议论文集》（英国爱丁堡）（PLDI ’ 14 440 451 10 1145 2594291 2594325 ）。ACM，纽约，纽约州，美国，–。https://doi.org/。/。[55] Juan Antonio Navarro Pérez 和 Andrey Rybalchenko。2013。模理论分离逻辑。载于《编程语言与系统》（亚洲编程语言与系统研讨会）。Springer 国际出版，Cham，90–106。[56] Ruzica Piskac、Thomas Wies 和 Damien Zufferey。2013年。使用可满足性模理论自动化分离逻辑。载于《第25届国际计算机辅助验证会议论文集》（俄罗斯圣彼得堡）（CAV’13）。施普林格出版社，柏林，海德堡， 773 789 10 1007 978 3 642 39799 8_54 –。https://doi.org/。/ ---- Ruzica Piskac、Thomas Wies 和 Damien Zufferey。2014。使用树与数据自动化分离逻辑。载于《第16届国际计算机辅助验证会议论文集》（CAV’14）。施普林格出版社，柏林，海德堡，711–728。[58] Ruzica Piskac、Thomas Wies 和 Damien Zufferey。2014年。GRASShopper。载于《系统构造与分析工具和算法》，Erika Ábrahám 和 Klaus Havelund（编）。Springer Berlin Heidelberg，柏林，海德堡，124–139。[59] Xiaokang Qiu、Pranav Garg、Andrei Ştefănescu 和 P. Madhusudan。2013年。结构、数据与分离的自然证明。载于《第34届 ACM SIGPLAN 编程语言设计与实现会议论文集》（美国华盛顿州西雅图）（PLDI ’13）。ACM，纽约，纽约州，美国，231–242。https://doi.org/10.1145/2491956.2462169

[60] Andrew Reynolds 和 Viktor Kuncak。2015年。SMT 求解器的归纳。载于《验证、模型检测与抽象解释》，Deepak D'Souza、Akash Lal 和 Kim Guldstrand Larsen（编）。Springer Berlin Heidelberg，柏林，海德堡，80–98。[61] John C. Reynolds。2002年。分离逻辑：一种用于共享可变数据结构的逻辑。载于《第 17 届 IEEE 计算机科学逻辑年会论文集》（LICS '02）。IEEE 计算机学会，美国，55–74。[62] Mihaela Sighireanu。2021年。SL-COMP：分离逻辑求解器竞赛：第三版报告。《国际软件工具与技术转移期刊》23，6（2021年12月），895–903。https://doi.org/10.1007/s10009-021-00628-w [63] Aishwarya Sivaraman、Alex Sanchez-Stern、Bretton Chen、Sorin Lerner 和 Todd Millstein。2022年。交互式证明的数据驱动引理合成。ACM程序设计语言会议录 6，OOPSLA2，第143篇文章（2022年10月），27页。https://doi.org/ 10 1145 3563306 。/ Jan Smans、Bart Jacobs 和 Frank Piessens。2009年。隐式动态帧：结合动态帧与分离逻辑。载于《ECOOP 2009——面向对象编程》，Sophia Drossopoulou（编）。Springer Berlin Heidelberg，柏林，海德堡，148 172 10 1007 978 3 642 03013 0_8 –。https://doi.org/。/ ---- Jan Smans、Bart Jacobs 和 Frank Piessens。隐式动态帧。《ACM 程序设计语言与系统汇刊》34，1，第 2 2012 58 10 1145 2160910 2160911 篇文章（5月），页。https://doi.org/。/。Philippe Suter、Mirco Dotta 和 Viktor Kunćak。具有抽象的代数数据类型判定过程。载于《第 37 届 ACM SIGPLAN-SIGACT 编程语言原理年会论文集》（西班牙马德里）（POPL ' *10* 199 210 10 1145 1706299 1706325 ）。ACM，纽约，纽约州，美国，–。https://doi.org/。/。Philippe Suter、Ali Sinan Köksal 和 Viktor Kuncak。递归程序模可满足性。载于《静态分析》，Eran Yahav（编）。Springer Berlin Heidelberg，柏林，海德堡，298–315。[68] Quang-Trung Ta、Ton Chanh Le、Siau-Cheng Khoo 和 Wei-Ngan Chin。2016年。分离逻辑中的自动化互显式归纳证明。载于《FM 2016：形式化方法》，John Fitzgerald、Constance Heitmeyer、Stefania Gnesi 和 Anna Philippou（编）。Springer 国际出版，Cham，659 676 10 1007 978 3 319 48989 6_40 –。https://doi.org/。/ ---- Alfred Tarski。格论不动点定理及其应用。《太平洋数学杂志》（），–。[70] Alan Turing。1936 年。论可计算数及其在判定问题中的应用。《伦敦数学学会会刊》 42, 1 1936 230 265 10 2307 2268810 （），–。https://doi.org/。/ Moshe Y. Vardi。关系查询语言的复杂性（扩展摘要）。载于《第十四届 ACM 计算理论年会论文集》（美国加利福尼亚州旧金山）（STOC '82）。计算机协会，纽约，纽约州，美国，137 146 10 1145 800070 802186 –。https://doi.org/。/。Niki Vazou、Anish Tondwalkar、Vikraman Choudhury、Ryan G. Scott、Ryan R. Newton、Philip Wadler 和 Ranjit Jhala。2018年。精化反射：使用可满足性模理论实现完备验证。ACM程序设计语言会议录 2，POPL（2018年），53:1–53:31。https://doi.org/ 10 1145 3158141 。/ Weikun Yang、Grigory Fedyukovich 和 Aarti Gupta。2019年。用于自动化代数数据类型归纳的引理合成。载于《约束编程原理与实践》，Thomas Schiex 和 Simon de Givry（编）。Springer 国际出版，Cham，600–617。

A 第2节补充细节

定理 2.3 的证明。 我们将证明一个更强的结果，即该逻辑是高度不可判定的。为此，我们将 MSO 中的有效性归约为图灵机的不停机问题，其中停机的定义是最终状态必须被无限次访问。高度不可判定问题不是递归可枚举的，其补集也不是递归可枚举的，即它们是递归可枚举困难的 ∩ 且余递归可枚举困难。下面介绍证明的关键要素。

首先，我们使用符号 零，后继 和 $<$来定义带比较的自然数。我们按照惯常方式用带量词的一阶公式约束这些符号，例如，要求 $<$ 构成一个全序。不过请注意，模型中可能存在超出与“标准”自然数对应的 ω-前缀的元素。为了筛除 ω-前缀，我们写出以下关于集合变量 S 的带量词公式：

$$\mathit{TrueNat} \equiv \forall S.\big(\text{零} \in S \wedge (\forall x.\, x \in S \Rightarrow \mathit{succ}(x) \in S)) \Rightarrow \forall y.\, y \in S\big)$$

TrueNat 表示每个包含零且对后继运算封闭的集合都是全称量词。这只有在模型中的每个元素都对应一个标准自然数时才成立。

证明的其余部分是标准的。我们使用函数对图灵机进行编码：$\mathit{state}(i)$表示图灵机在时间步 i的状态，$\mathit{head}(i)$表示图灵机在时间步 i的读写头位置，而$\mathit{cell}(i, j)$表示时间步 i时第 j个单元格的内容。我们用一阶逻辑量化的约束来确保这些函数定义正确。将这些约束记为 *TMEncoding*。

现在，我们可以构造表示不停机的公式了。回顾一下，我们对不停机的定义是：图灵机不会无限多次进入停机状态；也就是说，存在某个时刻，此后图灵机不再进入停机状态。不失一般性，设停机状态编码为*zero*。表示不停机的公式如下：

$$\mathit{TrueNat} \Rightarrow \big(\mathit{TMEncoding} \Rightarrow (\exists z \forall t.\, z < t \Rightarrow \mathit{state}(t) \neq \mathit{zero})\big)$$

证明至此完成。请注意，该公式不包含任何关于集合的交替量化。

□

定理证明2.4。 我们首先定义本结果中研究的分离逻辑变体。该逻辑的语法如下：

$$\begin{aligned} \varphi &:= \mathsf{emp} \mid \mathit{true} \mid \mathit{StackFormula} \mid x \xrightarrow{f} y \mid \varphi \wedge \varphi \mid \varphi * \varphi \\ &\quad \mid \mathit{ite}(\mathit{StackFormula}, \varphi, \varphi) \mid \mathit{RecPred}(\bar{t}) \mid \exists y.\, \varphi \\ \mathit{StackFormula} &:= t = t \mid t \neq t \end{aligned}$$

其中，递归定义谓词 表示递归定义谓词， t 表示一个项。项的语法采用通常的定义。

递归谓词 $R(x)$ 使用上述逻辑中的公式 $\rho(x)$ 定义，其中 ρ 可以提及 R （但只能以正面形式提及）。下面描述其语义。我们像分离逻辑一样，在存储 s（从变量到值的映射）和堆 h （从指针符号到函数的映射）上计算公式，但现在允许 h 为无限。此外，我们还要求模型解释递归定义的谓词符号，并将其记为 I。

s, h, I $\models$ 当 h 为空时为 emp$s$, h, I $\models$ 始终为 true$s$, h, I $\models$ 当 StackFormula 根据 ss 计算结果为 true 时为 StackFormula h, $\models x \mathrel{-\!\!f\!\!\rightarrow} y$, I h当 dom($=$) $x\{h\}$ 且 $f(s)(x(=y))\, s\, h$, $\models \alpha \wedge \beta$, I s 当 h, $\models \alpha$, I s且 h, $\models \beta$, I $s\, h$, $\models \alpha * \beta$, I

	如果存在互不相交的堆片段 h_1, h_2 使得 $h_1 \uplus h_2 = h$ and $s, h_1, \mathcal{I} \models \alpha$ and $s, h_2, \mathcal{I} \models \beta$
$s, h, \mathcal{I} \models$ 条件表达式$(cond\ \alpha, \beta)$	如果 $s \models cond$ 且 $s, h,$ I $\models \alpha$ 或者 $s \not\models cond$ 且 $s, h,$ I $\models \beta$
$s, h, \mathcal{I} \models$ 递归定义谓词(t)	如果 I(递归定义谓词)$(s(t))$ $=$ 为真 且 $s, h,$ I $\models \rho(t)$, 其中 ρ 是递归定义谓词的函数体
$s, h,$ 我 $\models \exists y.\varphi$	如果存在某个适当类型的元素 a，使得 $s[y \mapsto a], h, \mathcal{I} \models \varphi$

此外，我们将上述$RecPred$的递归语义视为不动点，而非最小不动点。为说明使用不动点的后果，请考虑以下谓词$List(x)$，它表示（此处为不动点）链表：

$$List(x) \equiv ite(x = nil, \mathrm{emp}, \exists y.\, x \mathrel{-\!\!next\!\!\rightarrow} y * List(y))$$

设 s 为将 x 解释为 0 的存储，并设 h 为定义域是自然数的无限堆片段，其中$next$指针的解释为 $h(next)(i) = i+1$。这表示一个“无限列表”。设 I 是这样一种解释：该无限列表中的每个节点都满足谓词$List$。显然，如果 I 是最小不动点解释，则 $s, h,$ I $\models List(x)$ 不成立（没有节点会满足$List$）。然而，在上述定义的逻辑中，它可以成立，因为我们允许任何对于给定递归定义而言有效的不动点解释。

现在，我们给出在上述逻辑中对不停机进行建模的构造。我们使用带有 后继 指针的链表来表示双计数器机的运行。我们使用整数值指针 状态 c_1 和 c_2，分别表示链表节点所编码配置对应的状态和两个计数器的值，并定义递归谓词 $tcm\_run(x)$，用于检查链表是否编码了该机器的一次有效运行。不失一般性，我们假设初始状态为 0，停机状态为 1。谓词 tcm_run 定义如下：

$$\begin{aligned}
tcm_run(x) := ite\Big(& x = nil, \mathrm{emp}, \\
& \exists y, q.\big((x \mathrel{-\!\!next\!\!\rightarrow} y \wedge x \mathrel{-\!\!state\!\!\rightarrow} q \wedge x \mathrel{-\!\!c_1\!\!\rightarrow} i \wedge x \mathrel{-\!\!c_2\!\!\rightarrow} j) \\
& \qquad * \\
& \qquad ite(y = nil, \mathrm{emp}, \exists q'.((y \mathrel{-\!\!state\!\!\rightarrow} q' \wedge y \mathrel{-\!\!c_1\!\!\rightarrow} i' \wedge y \mathrel{-\!\!c_2\!\!\rightarrow} j') * true) \\
& \qquad\qquad \wedge transition(q, i, j, q', i', j') \\
& \qquad\qquad \wedge tcm_run(y) \\
& \qquad) \\
& \big) \\
\Big)
\end{aligned}$$

其中，状态转换 是一个宏，用于表达双计数器机状态转换的正确性。我们将 $tcm(x)$ 定义为公式 $tcm\_run(x) \wedge (x \rightarrow_{state} 0 * true)$，它表示 x 指向一个链表，该链表包含从初始状态开始的机器有效运行。类似地，我们将 停机(x) 定义为公式 $x \rightarrow_{state} 1$。

最后，我们表达如下蕴涵：

$$x \neq nil \wedge (tcm(x) * true) \wedge (halt(y) * true) \models tcm(x) * halt(y) * true$$

现在我们断言，上述蕴涵有效，当且仅当双计数器机不停机。

机器不停机 $\Rightarrow$ 蕴涵有效。 注意，在任何满足 s, h, I 的 $s, h, \mathrm{I} \models tcm(x)$ 中，从 x 开始且在 后继 指针下封闭的、由链表元素构成的 ω-链，实际上必定编码了该机器从初始配置开始的一次运行。此外， h 必须包含该 ω-链中的所有节点。然而，由于我们考虑的是不动点定义， h 可能包含更多元素。

假设该机器不停机。现在考虑满足蕴涵左侧的 s, h, I。特别地，我们有 $s, h, \mathrm{I} \models tcm(x)$。由于该机器不停机， ω-链中不可能包含处于停机状态的节点。特别地，我们有 $s, h, \mathrm{I} \models halt(y)$，这意味着 y 不可能属于 ω-链。因此，我们可以将 h 拆分为三个互不相交的部分：(a) h_1，包含满足 $tcm(x)$ 的 ω-链；(b) $h_2 = \{y\}$，其中满足 $halt(y)$；以及 (c) 其他任意元素。这表明蕴涵右侧成立，证毕。

机器停机 $\Rightarrow$ 蕴涵无效。 如果机器不停机，那么对于任意满足 s, h, I 且满足$s, h, \mathrm{I} \models tcm(x)$ 的情形，都必定存在一个节点 e，位于从 x 开始的 ω-链上，并且该节点对应停机状态。现在考虑将 y 解释为 e时的蕴涵。显然，蕴涵左侧为真。然而，右侧不可能为真，因为任何满足$s, h, \mathrm{I} \models tcm(x)$ 的 h 都必定包含该 ω-链，尤其必定包含 e。因此，停机(y) 的堆片段无法与$tcm(x)$ 的堆片段分离。

证明至此完成。 □

请注意，在上述证明中，我们关键地用到了这样一个事实：给定一个满足 $s, h, \mathrm{I} \models tcm(x)$ 的情形，如果 h包含的元素多于该 ω-链，那么我们还有 $s, h', \mathrm{I} \models tcm(x)$，其中 h'是 h的任意不动点闭子集，且包含该 ω-链；特别地， h'可以就是该 ω-链本身。该公式可以在多个堆片段上得到满足，而我们在证明的第一个方向中利用了这一点。

B 操作语义

我们考虑形如 (S, H, A) 的配置，其中 S 是存储， H 是堆，而 A 表示已分配位置的集合。形式上， S 是一个部分映射，用于解释常量、变量和不可变函数。 H 是由多个映射组成的元组——每个可变函数 $f \in \mathrm{F}_m$ 对应一个映射——其定义域为位置全集，值域则是相应类型的全集（指针对应位置全集，数据字段对应各种背景排序）。我们将空指针建模为一个特殊位置，从而确保在我们的语义中，有效程序不会解引用空指针。注意， S 和 H 共同定义了一个可用于解释 FL 公式的模型。我们将 SL 公式 φ 在与给定 S, H 对应的模型上的满足关系记为 $S, H \models \varphi$。 A 是位置全集的一个子集，且不包含空指针。我们还引入一个错误配置 $\perp$。

下面通过图10描述操作此类配置的各种命令的操作语义。该语义相当标准，但我们将在此阐明一些关键方面。首先，操作语义会检查解引用是否满足内存安全要求，即查找或

对 $x.f$ 的变异仅在 x 属于已分配集合 A 时发生（特别是，x 不为 空指针）。如果不满足这一条件，则会到达错误状态 $\perp$。我们将 $\perp$ 定义为汇状态，$\perp$ 上的任何命令都会状态转换到其自身 $\perp$。其次，分配会向位置全集以及已分配集合 A 中添加一个新位置（一个不同于当前位置全集中所有位置的新元素）。堆 H 也会扩展，将新位置映射到固定的默认值 默认值 f，对于每个 $f \in \mathrm{F}_m$ 都如此。相应地，释放会从 A 中移除已释放的位置。我们不允许重复释放，否则会到达错误状态 $\perp$。最后，函数采用按值调用语义。

FL 到 FORD 的 C 翻译

FL 到 FORD 的翻译见图11。由于 FL 只是在 FORD 的基础上扩展了支撑集算子和云算子，因此处理好这两种算子后，翻译就很直接了。我们将 φ 到 FORD 的翻译记作 $\nabla(\varphi)$。

回想一下，$M \models [\varphi]$ 当且仅当 $M \models \varphi$。到 FORD 的翻译与此类似：$\nabla([\varphi]) = \varphi$。

直观地说，$\nabla(Sp(\varphi))$ 将是一个集合，其中包含 φ的支撑集中的位置。对于不涉及递归函数的公式，这很简单。例如，$Sp(\text{键}(x) = \text{键}(y))$ 对应的位置集合为 $\{x, y\}$。云算子的交互符合预期：$Sp([\varphi]) = \phi$。

递归函数的支撑集本身将是一个递归定义，定义为该函数支撑集的最小不动点（lfp）。

例如，我们来看下面的列表定义：

$$List(x) := ite(x = nil, \top, (next(x) = next(x)) \wedge List([next(x)]) \wedge x \notin Sp(List([next(x)])))$$

请注意，该定义引用了“列表的支撑”。我们将 $\nabla(Sp(\text{链表}(x)))$ 定义为 $SpList(x)$，其中

$$SpList(x) :=_{lfp} \nabla(Sp(ite(x = nil, \top, (next(x) = next(x)) \wedge List([next(x)]) \wedge x \notin Sp(List([next(x)])))))$$

化简后，

$$SpList(x) :=_{lfp} ite(x = nil, \phi, \{x\} \cup SpList(next(x)))$$

这正是我们凭直觉所预期的。如果 $x =$ 空指针，那么该列表的支撑集为空。否则，它就是列表头部与列表剩余部分的支撑集的并集。

第 D 节详述4：基础逻辑

D.1 引理4.1 的证明

证明。（1）对 α进行结构归纳。大多数规则会组合较小公式的支撑集，因此结果是 $dom(h)$的子集（包括析取规则）。对于 $x \xrightarrow{f} y$ 的规则，当支撑集有定义时，$s(x) \in dom(h)$。

（2）对 α 进行结构归纳。若 $\alpha = \delta$ 是堆无关公式，则 $Supp(\alpha, s, h) = \emptyset = Supp(\alpha, s, h')$。若 $\alpha = x \xrightarrow{f} y$，注意，若 α 关于 h 的支撑集不是 $\perp$，则必为 $S = \{s(x)\}$。于是，由于 h 和 h' 在 $s(x)$ 上一致，$s(x)$ 也属于 $dom(h')$，并且 $Supp(\alpha, s, h') = \{s(x)\} = S$。

若 $\alpha = \exists y.(x \xrightarrow{f} y : \alpha')$，注意到 $S = \{s(x)\} \cup \mathrm{Supp}(\alpha', s', h)$，其中 $s' = s[y \mapsto h(f)(s(x))]$。但由于 $s(x)$ 属于 S，$dom(h')$ 包含它，且在 $s(x)$ 上与指针一致，因此 $h'(f)(s(x)) = h(f)(s(x))$。于是 $\mathrm{Supp}(\exists y.(x \xrightarrow{f} y : \alpha, s, h') = \{s(x)\} \cup \mathrm{Supp}(\alpha', s', h') = \{s(x)\} \cup \mathrm{Supp}(\alpha', s', h)$（由归纳假设）$= \mathrm{Supp}(\alpha, s, h)$。

$$\frac{}{< s, \bot >\Downarrow \bot}\ \text{ERROR} \qquad \frac{< y, \sigma >\Downarrow_e v}{< x := y, \sigma >\Downarrow \sigma(S)[x \leftarrow v]}\ \text{ASSN}$$

$$\frac{S(y) \in A \qquad < f(y), \sigma >\Downarrow_e v}{< x := y.f, \sigma >\Downarrow \sigma(S)[x \leftarrow v]}\ \text{DEREF} \qquad \frac{S(y) \notin A}{< x := y.f, \sigma >\Downarrow \bot}\ \text{BADDEREF}$$

$$\frac{< y, \sigma >\Downarrow_e v \qquad < x, \sigma >\Downarrow_e w}{< y.f := x, \sigma >\Downarrow \sigma(H)[f \leftarrow f[v \leftarrow w]]]}\ \text{MUT} \qquad \frac{\{S(y), S(x)\} \nsubseteq A}{< y.f := x, \sigma >\Downarrow \bot}\ \text{BADMUT}$$

$$\frac{a \notin A \qquad H' = \lambda f, x.ite(x = a, default_f, H(f)(x))}{< alloc(x), \sigma >\Downarrow (S[x \leftarrow a], H', A \cup \{a\})}\ \text{ALLOC}$$

$$\frac{a \in A \qquad < x, \sigma >\Downarrow_e a}{< free(x), \sigma >\Downarrow \sigma[A \leftarrow A \backslash \{a\}]}\ \text{FREE} \qquad \frac{S \models \eta}{< assume(\eta), \sigma >\Downarrow \sigma}\ \text{ASSUME}$$

$$\frac{S \models \eta \qquad < s_1, \sigma >\Downarrow \sigma_1}{< \text{if } \eta \text{ then } s_1 \text{ else } s_2, \sigma >\Downarrow \sigma_1}\ \text{IF-TRUE} \qquad \frac{S \not\models \eta \qquad < s_2, \sigma >\Downarrow \sigma_2}{< \text{if } \eta \text{ then } s_1 \text{ else } s_2, \sigma >\Downarrow \sigma_2}\ \text{IF-FALSE}$$

$$\frac{< s_1, \sigma >\Downarrow \sigma_1 \qquad < s_2, \sigma_1 >\Downarrow \sigma_2}{< s_1; s_2, \sigma >\Downarrow \sigma_2}\ \text{SEQ}$$

$$\frac{\begin{array}{c} < g, \sigma_g >\Downarrow \sigma'_g \qquad \sigma_g \preccurlyeq \sigma \qquad \{\sigma'(S)(y_1), \ldots, \sigma'(S)(y_m)\} \subseteq \sigma'_g(A) \\ (\sigma(A) \backslash \sigma_g(A)) \cap \sigma'_g(A) = \phi \qquad A' = (\sigma(A) \backslash \sigma_g(A)) \cup \sigma'_g(A) \\ S' = \lambda x.ite(S(x) \in \sigma(A) \backslash \sigma_g(A), S(x), \sigma'_g(S)(x)) \text{ with domain } \{x \mid S(x) \in A'\} \\ H' = \lambda f.ite(f \notin dom(\sigma_g(H)), \sigma(H)(f), \lambda x.ite(x \notin \sigma'_g(A), \sigma(H)(f)(x), \sigma'_g(H)(f)(x))) \end{array}}{< y_1, \ldots, y_m := g(x_1, \ldots, x_n), \sigma >\Downarrow (S', H', A')}\ \text{CALL-BY-VALUE}_{n,m}$$

图 10. 编程语言的操作语义。这里， $\sigma = (S, H, A)$ 表示程序的一个非失败状态。错误状态记为 $\bot$。对于每种类型 τ，都固定一个“默认”值。default^{τ} 表示函数陪域所属类型的默认值。 $\sigma(S)[x \leftarrow v]$表示 $(S[x \leftarrow v], H, A)$。$\text{dom}(X)$表示 X 的定义域。$(S_1, H_1, A_1) \preccurlyeq (S_2, H_2, A_2)$表示 $A_1 \subseteq A_2$，并且如果 $S_1(x) \in A_1$，那么 $S_2(x) = S_1(x)$，且对于每个 $f \in \text{dom}(H_1)$， $H_1(f)(x) = H_2(f)(x)$。换言之，(S_1, H_1, A_1) 是 (S_2, H_2, A_2) 的一个限制。这里没有将常量赋值、数据字段解引用等单独列为规则，因为它们与上述规则类似。

现在考虑 α = 条件表达式$(\gamma, \alpha', \beta')$，并假设其在 h, S 中的支撑集不是 $\bot$。那么 $\text{Supp}(\gamma, s, h)$ 不可能是 $\bot$；由于它是 S 的子集，根据归纳假设，$\text{Supp}(\gamma, s, h') = \text{Supp}(\gamma, s, h)$。还要注意， $h \downharpoonright \text{Supp}(\gamma, s, h) = h' \downharpoonright \text{Supp}(\gamma, s, h')$，因此 $(s, h \downharpoonright \text{Supp}(\gamma, s, h)) \models \gamma$ 当且仅当 $(s, h' \downharpoonright \text{Supp}(\gamma, s, h')) \models \gamma$。考虑第一种情况，即当 $(s, h, \text{Supp}(\gamma, s, h)) \models \gamma$ 且 $\text{Supp}(\alpha, s, h)$ 不是 $\bot$ 时。后者是 S 的子集，根据归纳假设，$\text{Supp}(\alpha, s, h') = \text{Supp}(\alpha, s, h)$，这就证明了我们的结论。类似地，在第二种情况下，当

$$
\begin{aligned}
\nabla(\top) &= \top \\
\nabla(\bot) &= \bot \\
\nabla(c) &= c \\
\nabla(x) &= x \\
\nabla(f(t_1,\ldots,t_n)) &= f(\nabla(t_1),\ldots,\nabla(t_n)) \text{ for any function } f \\
\nabla(t_1 = t_2) &= (\nabla(t_1) = \nabla(t_2)) \\
\nabla(R(t_1,\ldots,t_n)) &= R(\nabla(t_1),\ldots,\nabla(t_n)) \text{ for any relation } R \\
\nabla(\alpha \wedge \beta) &= \nabla(\alpha) \wedge \nabla(\beta) \\
\nabla(\alpha \vee \beta) &= \nabla(\alpha) \vee \nabla(\beta) \\
\nabla(\neg\varphi) &= \neg\nabla(\varphi) \\
\nabla(ite(\gamma:\alpha,\beta)) &= ite(\nabla(\gamma):\nabla(\alpha),\nabla(\beta)) \\
\nabla(ite(\gamma:t_1,t_2)]) &= ite(\nabla(\gamma):\nabla(t_1),\nabla(t_2)) \\
\nabla(Sp(\top)) = \nabla(Sp(\bot)) &= \phi \\
\nabla(Sp(c)) &= \nabla(Sp(x)) = \phi \\
\nabla(Sp(f(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n}\{t_i\} \cup \bigcup_{i=1}^{n} \nabla(Sp(t_i)) \text{ if } f \in \mathcal{F}_m \\
\nabla(Sp(f(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n} \nabla(Sp(t_i)) \text{ if } f \notin \mathcal{F}_m \\
\nabla(Sp(Sp(\varphi))) &= \nabla(Sp(\varphi)) \\
\nabla(Sp(Sp(t))) &= \nabla(Sp(t)) \\
\nabla(Sp(t_1 = t_2)) &= \nabla(Sp(t_1)) \cup \nabla(Sp(t_2)) \\
\nabla(Sp(R(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n} \nabla(Sp(t_i)) \text{ for } R \in \mathcal{R} \\
\nabla(Sp(\alpha \wedge \beta)) &= \nabla(Sp(\alpha)) \cup \nabla(Sp(\beta)) \\
\nabla(Sp(\alpha \vee \beta)) &= \nabla(Sp(\alpha)) \cup \nabla(Sp(\beta)) \\
\nabla(Sp(\neg\varphi)) &= \nabla(Sp(\varphi)) \\
\nabla(Sp(ite(\gamma:\alpha,\beta))) &= \ \mathit{if}\ \gamma\ \mathit{then}\ \nabla(Sp(\gamma)) \cup \nabla(Sp(\alpha))\ \mathit{else}\ \nabla(Sp(\gamma)) \cup \nabla(Sp(\beta)) \\
\nabla(Sp(ite(\gamma:t_1,t_2))) &= \ \mathit{if}\ \gamma\ \mathit{then}\ \nabla(Sp(\gamma)) \cup \nabla(Sp(t_1))\ \mathit{else}\ \nabla(Sp(\gamma)) \cup \nabla(Sp(t_2)) \\
\nabla(Sp([\alpha])) &= \phi \\
\nabla(Sp(I(t_1,\ldots,t_n))) &= \textstyle\bigcup_{i=1}^{n} \nabla(Sp(t_i)) \cup SpI(t_1,\ldots,t_n) \text{ for } I \in \mathcal{I}, \\
&\qquad \text{where } SpI(\bar{x}) :=_{lfp} \nabla(Sp(\rho_I(\bar{x}))) \text{ and } I(\bar{x}) :=_{lfp} \rho_I(\bar{x}) \\
\nabla([\alpha]) &= \nabla(\alpha)
\end{aligned}
$$

图 11. 从框架逻辑到 FORD 的翻译。对于 FL 公式 α，翻译得到的 FORD 公式为 $\nabla(\alpha)$，如上所述递归定义。

$(s,\ h\restriction Supp(\gamma,\ s,\ h)) \not\models \gamma$，$Supp(\beta,\ s,\ h)$ 不是 $\bot$ 时，它是 S 的子集，并且根据归纳假设，$Supp(\beta,\ s,\ h_1) = Supp(\beta,\ s,\ h)$，这同样证明了我们的结论。

$\alpha = \alpha' \oplus \beta'$ 的情形是显然的，可直接由归纳假设得出。

(3) 由 (2) 得出，因为 $h\restriction S$ 在 S 上与 h 一致。(4) 堆片段 h 在 $dom(h_1)$ 上与 h_1 一致 $= \mathrm{Supp}(\alpha, s, h_1)$，因此 $\mathrm{Supp}(\alpha, s, h) = dom(h_1)$（由上面的 (2) 得出）。类似地，堆片段 h 在 $dom(h_2)$ 上与 h_2 一致 $= \mathrm{Supp}(\alpha, s, h_2)$，因此 $\mathrm{Supp}(\alpha, s, h) = dom(h_2)$。故 $dom(h_1) = dom(h_2)$。

□

D.2 引理4.2 的证明

(1) 通过对 α进行结构归纳证明。基本情况很简单。

考虑 $\alpha = \exists y.x \dashrightarrow f\rightarrow y$： $\alpha'.$和 $(s,h) \models \alpha$。于是 $s(x) \in dom(h)$，因此 $\mathrm{Supp}(\alpha,s,h) = \{s(x)\} \cup \mathrm{Supp}(\alpha',s',h)$，其中 $s' = s[y \rightarrow h(f)(s(x))]$和 $(s',h) \models \alpha'$。根据归纳假设，$\mathrm{Supp}(\alpha',s',h) = dom(h)$。因此 $\mathrm{Supp}(\alpha,s,h) = dom(h)$。

考虑 $\alpha = \alpha_1 * \alpha_2$，以及 $(s,h) \models \alpha$。那么存在 h 的两个堆片段 h_1, h_2，这些定义域的并集为 h 的定义域，并且 $(s,h_i) \models \alpha_i$, $i = 1,2$。根据归纳假设可得 $\mathrm{Supp}(\alpha_i,s,h_i) = dom(h_i)$。由引理 4.1、(4) 和 (3) 可得，$\mathrm{Supp}(\alpha_i,s,h) = dom(h_i)$。因此，$\mathrm{Supp}(\alpha_1 * \alpha_2,s,h)$ 是 $\mathrm{Supp}(\alpha_1,s,h) \cup \mathrm{Supp}(\alpha_2,s,h)$（根据定义），这等于 $dom(h_1) \cup dom(h_2) = dom(h)$。 $\alpha_1 \oplus \alpha_2$ 的情形，其中$\oplus \in \{\wedge, \curlywedge, \vee\}$，以及 ite 表达式的情形，类似。

（2）假设存在两个满足以下条件的子堆片段 α。根据（1）， α 在这两个子堆片段中的支撑集将分别是它们自身的定义域。但根据引理4.1 （4），这些定义域的堆片段必须相同。因此，这些堆片段是相同的。□

E 第4节详情：扩展逻辑*SL-FL*

该逻辑的语义通过以下方式扩展了基础逻辑上的语义。首先，我们假设每个背景排序都是一个完备格（对于算术等平坦类型，我们可以引入一个底元素和一个顶元素，从而得到一个完备格）。接下来，我们将图*Supp* 5 中定义 的方程视为具有最小不动点语义的递归定义，其中 $\subseteq$上标表示排序关系。

我们为递归定义的谓词和函数添加支撑定义：

$$Supp(R(\overline{x}),s,h) = Supp(\rho_R(\overline{x}),s,h)$$

$$Supp(F(\overline{x}),s,h) = Supp(\mu_F(\overline{x}),s,h)$$

其中， $R^{\frown}(x) = \overline{\rho_R}$ （x）和 F （$x^{\urcorner} = \mu_F$ （x）分别是 R和 F的定义。我们可以证明，上述定义映射*Supp* 的方程的最小不动点同样满足引理4.1 中针对扩展逻辑所描述的性质。

现在很容易定义 *SL-FL* 的语义。我们对图 5中给出的语义作如下修改，以定义递归定义谓词的语义：

$$(s,h) \models R(\overline{x}) \quad iff \quad (s,h) \models \rho(\overline{x}), \ \ where\ R\text{'s definition is } R(\overline{x}) = \rho(\overline{x})$$

我们可以证明与扩展逻辑的引理 4.2 类似的引理。

最后，我们也可以将扩展逻辑转换为框架逻辑，并对图 6 中针对 条件表达式 公式给出的转换作如下修改，同时转换定义：

$$\begin{aligned}
\Pi(ite(\alpha',\alpha,\beta)) &= ite(\Pi(\alpha'),\Pi(\alpha),\Pi(\beta)) \\
\Pi(p(\overline{t})) &= p(\overline{t}) \\
\Pi(R(\vec{x})) &= R(\vec{x}) \\
\Pi(x.f)) &= f(x) \\
\Pi(f(\overline{t})) &= f(\Pi(\overline{t})) \\
\Pi(ite(\alpha,t,t')) &= ite(\alpha,\Pi(t),\Pi(t')) \\
\Pi(F(\overline{t})) &= F(\Pi(\overline{t})) \\
\Pi(\ R(\vec{x}) :=_{lfp} \rho(\vec{x})\ ) &= R(\vec{x}) :=_{lfp} \Pi(\rho(\vec{x})) \\
\Pi(\ F(\vec{x}) :=_{lfp} \mu(\vec{x})\ ) &= R(\vec{x}) :=_{lfp} \Pi(\mu(\vec{x}))
\end{aligned}$$

现在我们可以证明与引理 4.3类似的引理：

引理 E.1。 令 g 为一个全局堆（其定义域为 Loc 的堆片段）， s 为一个存储。对于框架逻辑公式，我们将存储解释为变量的一阶逻辑解释。令 α 为一个分离逻辑-框架逻辑公式。那么

- $g \models_s \Pi(\alpha)$ 当且仅当存在 g 的一个堆片段 h，使得$(s, h) \models \alpha$。
- 如果 $g \models_s \Pi(\alpha)$，那么 $Supp(\alpha, s, g)$ 等于 g 中 $Sp(\Pi(\alpha))$ 的值。 □

F 验证验证条件

本节介绍我们用于检查由第5节所述机制生成的无量词框架逻辑验证条件的有效性判定过程。我们的方法分为两个阶段。第一阶段，我们以保持有效性的方式将无量词 FL 公式转换为无量词 FORD 公式。回顾一下，无量词 FL 只是对无量词 FORD 的扩展，加入了支撑集 $Sp(\cdot)$ 和云 $[\cdot]$ 运算符。直观而言，我们的转换在 FORD 本身中编码了这些运算符的语义。第二阶段，我们使用自然证明这一先前工作中提出的机制 [38, 54,59]，对无量词 FORD 公式进行推理。

F.1 阶段 1：将无量词 FL 转换为无量词 FORD

我们的推理机制第一阶段将 FL 公式转换为 FORD 公式。核心思路是在 FORD 本身中编码 Sp 和 $[\cdot]$ 算子的语义。这个过程会引入新的递归定义，对应于现有递归定义的支撑集。

我们将此转换记为 ∇。该函数以包含 Sp 和 $[\cdot]$ 算子的无量词 FL 公式（或项）为输入，并输出 FORD 公式（或项）。我们在附录 C 中介绍形式化转换，并在此给出直观说明。首先，显然，对于任何不涉及支撑集算子或云算子的 FL 公式 α，都有 $\nabla(\alpha) = \alpha$。

接下来，为了编码 Sp 运算符，事实证明，我们基本上可以仿照描述 Sp 语义的方程（图 2）！对于不包含递归定义符号的公式，这一点很容易看出。例如，对于可变函数 $f \in \mathrm{F}_m$，$\nabla(\mathrm{Sp}(f(x) = y)) = \{x\}$，因为$\nabla(f(x) = y) = \nabla(f(x)) \cup \nabla(y) = \{x\} \cup \emptyset$。类似地，$\nabla(\mathrm{Sp}(\alpha \wedge \beta)) = \nabla(\mathrm{Sp}(\alpha)) \cup \nabla(\mathrm{Sp}(\beta))$。这样，只需遵循图 2 中的方程，就能将 Sp 运算符“编译消除”。

然而，这对递归定义符号不起作用。例如，回想一下使用云算子写出的 链表(x) 的定义：

$$List(x) :=_{lfp} ite(x = nil, \top, next(x) = next(x) \wedge List([next(x)]) \wedge x \notin Sp(List([next(x)])))$$

Us按照上述方法，链表(x) 的支撑集给出以下方程：

$$\begin{aligned} Sp(List(x)) &= Sp(ite(x = nil, \top, next(x) = next(x) \wedge List([next(x)]) \wedge x \notin Sp(List([next(x)])))) \\ &= ite(x = nil, \emptyset, \{x\} \cup Sp(List([next(x)]))) \end{aligned}$$

这里，我们使用方程 $Sp([\alpha]) = \emptyset$，展开了云表达式上的 Sp。

在这种情况下，我们无法消去 Sp 算子。不过，回想一下，Sp 算子是满足图 2中方程的最小解释。因此，我们引入一个新的递归定义符号 Sp_{List}，定义如下：

$$Sp_{List}(x) :=_{lfp} ite(x = nil, \emptyset, \{x\} \cup Sp_{List}([next(x)]))$$

然后，我们将 Sp(链表(x)) 翻译为 $Sp_{List}(x)$。后者现在是一个在 FORD 中提及新建递归定义的表达式。一般而言，对于每个对应于其支撑集的 $I \in \mathrm{I}$，我们的翻译都会创建一个新的递归定义符号 Sp_I。

最后，对 $[\cdot]$ 运算符的编码是平凡的。由于支撑表达式已在上一步中消除，$[\alpha]$ 与 α完全相同。因此，我们在翻译过程中直接忽略它。具体而言，上述定义中的递归子表达式 $Sp_{List}([next(x)])$ 在最终翻译为 FORD 时变为 $Sp_{List}(next(x))$。

F.2 St第 2 部分：使用自然证明和 SMT 求解器推理 FORD 公式 rs

上述框架逻辑公式的翻译产生了无量词的 FORD 公式（即所有自由变量都被隐式地作全称量词处理）。此外，它们具有特殊形式：从位置排序映射到位置排序或背景排序的函数只能如此映射，但不存在将背景排序映射到位置排序的函数。因此，它们属于一个具有“单向”定义的特殊片段，即 L*oneway* 片段。

自然证明 [38, 54, 59] 是一种可靠但不完备的技术，用于证明 FORD 公式的有效性。该技术将递归定义视为具有不动点语义（而非最小不动点语义），从而得到一个一阶公式。随后，它在深度为 d（逐渐增大d）的项上实例化递归定义（类似于对这些定义的量词进行实例化），从而得到无量词公式。通过 SMT 求解器验证这些无量词公式。该技术的理论基础 [38]表明，对于一阶公式（其中递归定义具有不动点语义），该技术实际上是一种完备技术。此外，在实践中，自然证明已被证明可用于其他逻辑的堆验证。因此，我们可以使用这一自动化技术验证验证条件。

G 评估详情

G.1 基准测试套件

表 2 总结了我们的基准测试套件。除了代码行数和注解（规约、契约、循环不变式等）外，我们还报告了每个数据结构类所需的引理数量。

数据结构	操作	行数 代码+注解。（框架逻辑）	行数 代码 +注解。（*SL-FL*）	递归定义	唯一项数量 递归定义堆片段	引理
单链表	追加	30	29	2	1	0
	全部复制	32	33	2	1	0
	删除	35	35	2	1	0
	Find	34	35	2	1	0
	尾部插入	32	32	2	1	0
	头部插入	21	21	2	1	0
	反转	33	33	2	1	0
	插入类型	60	62	4	1	1
	归并排序	107	110	4	1	1
	快速排序	145	152	5	1	1
双向链表	尾部插入	38	44	2	1	0
	头部插入	36	41	2	1	
	删除中间节点	50	65	4	2	
	中间插入	56	77	4	2	
有序列表	删除	40	42	4	1	1
	Find	44	47	4	1	
	插入	44	46	3	1	
循环链表	删除前端	61	61	4	2	1
	Find	61	62	4	2	
	头部插入	42	42	4	2	
BST	删除	113	120	4	1	2
	Find	76	76	4	1	
	插入	86	88	4	1	
	右旋	58	59	4	1	
	二叉搜索树转链表	94	90	7	2	
Tree	中序遍历	65	65	3	1	0
树堆	删除	154	158	6	1	4
	Find	95	97			2
RBT	插入	193	218	5	1	2

表2. 链表列表 基准测试、代码行数、递归定义数量以及这些定义的数量
具有不同的 h 堆，以及证明该基准测试所需的引理数量。

表3 列举了示例例程的前置条件和后置条件示例。 框架逻辑 规约使用 支撑集 运算符来表达数据结构的不相交性。我们还注意到，为了进行验证，可能需要加强规约。例如，BST 删除根节点需要在后置条件中加入以下约束 $(Old(\text{最小值}(x)) \leq \text{最小值}(ret)) \wedge (\text{最大值}(ret) \leq Old(\text{最大值}(x)))$ 以完成验证。

基准测试	前置条件	后置条件
有序插入	已排序（x）	已排序(ret) $\wedge$ 键集合(ret) $= Old$ (键集合(x)) $\cup \{k\}$ $\wedge$ 条件表达式(Old (最小值(x)) $< k$, 最小值(ret) $= Old$ (最小值(x)), 最小值(ret) $= k$)
有序连接 (快速排序的一部分)	已排序$(x) \wedge$ 已排序(y) $\wedge$ 最大值(x) $\leq$ 最小值(y) $\wedge (Sp($已排序 $(x))$ $\cap Sp($已排序 $(y))) = \emptyset$	已排序 (ret) $\wedge$ 键集合(ret) $= Old$ (键集合(x)) $\cup Old$ (键集合(y)) $\wedge$ 条件表达式($x =$ 空指针, 最小值(ret) $= Old$ (最小值(y)), 最小值(ret) $= Old$ (最小值(x))) $\wedge$ 条件表达式($y =$ 空指针, 最大值 (ret) $= Old$ (最大值 (x)), 最大值 (ret) $= Old$ (最大值 (y)))

(a) 示例 框架逻辑 前置条件和后置条件

基准测试	前置条件	后置条件
二叉搜索树右旋	($x \neq$空指针) $*$(二叉搜索树 (x) $\wedge$(左子树 (x) $\neq$ 空指针))	二叉搜索树(ret) $\wedge$(键集合(ret) $= Old$ (键集合(x)))
BST 删除根节点 （二叉搜索树删除的一部分）	($x \neq$空指针) $*$ (∃左子树.(x −左子树→左子树 : $(\exists rht.(x$ −右子树→ rht :(($k =$ 键(x)) $*$(二叉搜索树 (左子树) $\wedge$(最大值 (左子树) $< k$)) $*$(二叉搜索树 (rht) $\wedge$($k <$ 最小值(右子树)))))))	二叉搜索树 (ret) $\wedge$(键集合(ret) $= (Old($键集合$(x))) \setminus \{k\})$ $\wedge(Old($最小值$(x))$ $\leq$ 最小值$(ret))$ $\wedge$(最大值 (ret) $\leq Old($最大值$(x)))$

(b)示例 *SL-FL* 前置条件和后置条件表 3。前置条件和后置条件示例。 x 和 y 的类型为Loc， k 的类型为整数。ret 是例程返回的数据结构的指针。

表 4 包含我们基准测试中使用的一些引理示例。

基准测试	引理
插入排序	已排序(x) $\Longrightarrow$链表(x)
二叉搜索树查找	二叉搜索树(x) $\Longrightarrow$最大值(x) $< k$ $\Longrightarrow$ $k \notin$ 键集合(x)
树转列表	二叉搜索树 (x) $\Longrightarrow n \neq$空指针 $\Longrightarrow$ 条件表达式(左子树 (x) $=$ 空指针, 最小值(x) $= key(x)$, 最小值(x) $=$ 最小值(左子树 (x)))
快速排序	$x \neq$空指针 $\Longrightarrow$ 最大值(x) $\leq$ 最小值(y) $\Longrightarrow$ 最小值(x) $\leq$ 最小值(y)

表 4. 示例引理

G.2 数据结构定义

基准测试中使用的 FL 数据结构定义采用显式支撑集和云算子。单链表的定义如下：

$$List(x) := ite(x = nil, \top, List(next(x)) \wedge x \notin Sp(List([next(x)]))$$

双链表的定义类似，但增加了一个约束：如果 后继(x) 不是空指针，则其前驱指针应指回 x，

$$\begin{aligned} Dll(x) := ite(x = nil, \top, ite(next(x) = nil, \top, \\ x = prev(next(x)) \wedge Dll(next(x)) \wedge x \notin Sp(Dll([next(x)])))) \end{aligned}$$

接下来，为了定义有序链表，我们依赖于列表中最小元素的一个辅助定义。然后，（非递减）有序列表就是指首元素键不大于其尾部

最小值的列表，而该列表本身也是有序列表：

$$Min(x) := ite(x = nil, +\infty, ite(key(x) < Min(next(x)), key(x), Min(next(x))))$$
$$Sorted(x) := ite(x = nil, \top, Sorted(next(x)) \wedge x \notin Sp(Sorted([next(x)])) \wedge key(x) \leq Min(next(x)))$$

在我们对该定义的实现中， $+\infty$ 只是一个额外变量，并将其视为具有无限大的值。

对于循环链表，我们使用 $Lseg(\cdot)$ 定义链表段，然后将循环链表定义为：空链表，或沿其后继指针最终指向自身的链表，

$$Lseg(x, y) := ite(x = y, \top, ite(x = nil, \bot, Lseg(next(x), y) \wedge x \notin Sp(Lseg([next(x)], y))))$$
$$Circ(x) := ite(x = nil, \top, Lseg(next(x), x) \wedge x \notin Sp(Lseg([next(x)], x)))$$

与有序列表类似，我们对二叉搜索树的定义依赖于树中最小值和最大值的定义；二叉搜索性质则通过以下要求来指定：左子树中的最大元素小于根节点的键，右子树中的最小元素大于根节点的键。Min 和 Max 的定义与有序列表中的定义类似，因此这里省略；我们指出，当 x 为 nil 时，Max 的默认值为 $-\infty$。于是，二叉搜索树的定义为

$$\begin{aligned} BST(x) := ite(x = nil, \top, & \\ & BST(left(x)) \wedge Max(left(x)) < key(x) \wedge BST(right(x)) \\ & \wedge key(x) < Min(right(x)) \wedge x \notin Sp(BST([left(x)])) \\ & \wedge x \notin Sp(BST([right(x)])) \wedge Sp(BST([left(x)])) \cap Sp(BST([right(x)])) = \emptyset) \end{aligned}$$

除了通常要求 x 不属于任一子树的支撑集之外，我们还在此规定两个子树的支撑集必须互不相交；这些性质共同表明 x 确实是一棵树。

Treap 树和红黑树的定义与二叉搜索树的定义类似，只是增加了相应的约束：Treap 树中优先级需满足最大堆性质，而红黑树需满足红黑性质。

Treap 树中最大堆性质的实现方式与二叉搜索树的二叉搜索性质相同，只不过我们只要求根节点的优先级大于任一子树的最大优先级。

对于红黑树，我们添加一个递归定义，用于确定树的（最大）黑高；若两棵子树的黑高相同，且根节点为黑色或两个子节点均为黑色，则该树为红黑树。

G.3 一个示例

本节将逐步介绍一个程序的验证条件生成，以展示如何将带有 *SL-FL* 注释的程序转换为 FL 验证条件。

下面给出程序 RevPrepend(x, y)，它接受两个链表，并将第一个链表的反转结果前置到第二个链表上。该程序包含两个递归函数，*List* 和 *Keys*，编写于*SL-FL*中。回想第 6 节的内容，*Old*(.) 的支撑集始终为空。

List(x) := 条件表达式(x =空指针, True, Exists y: 后继(x). (后继(x) = 后继(x)) * List(y)) 键集合(x) := 条件表达式(x =空指针, EmptySet整数, SetAdd(键集合(后继(x)), x))

RevPrepend(x, y) returns(ret) @pre: List(x) * List(y) @post: List(ret) and(Keys(ret) = 集合并(Old(Keys(x)), Old(Keys(y)))) (if (x = 空指针) then ret :=y; else tmp :=x.后继; x.后继 := y; ret := RevPrepend(tmp, x);) return;

首先，我们将递归定义和程序注解转换为框架逻辑（见图6 了解转换过程）。例如，链表将变为

$$List(x) := ite(x = nil, \top, \exists y : y = next(x).\ (next(x) = next(x)) \wedge List(y)$$
$$\wedge (Sp(next(x) = next(x)) \cap Sp(List(y))) = \phi)$$

利用引理 5.1，我们消去存在量词，

$$List(x) := ite(x = nil, \top, (next(x) = next(x)) \wedge (next(x) = next(x)) \wedge List([next(x)])$$
$$\wedge (Sp(next(x) = next(x)) \cap Sp(List([next(x)]))) = \phi)$$

注意，这种机械式转换会导致公式规模膨胀。例如，它会检查 后继(x) =后继(x) 与 支撑集(链表([后继 (x)])) 是否不相交。这可以简化为只检查 x 是否属于后者的支撑集。转换引入冗余检查的一个更极端的例子出现在后置条件中，它被转换为

$$List(ret) \wedge (Keys(ret) = Old(Keys(x)) \cup Old(Keys(y))) \wedge$$
$$Sp(List(ret)) = Sp(Keys(ret) = Old(Keys(x)) \cup Old(Keys(y)))$$

可以验证，最后一个合取项始终成立（链表 的支撑集与键集合 的支撑集始终相同），并且整个表达式的支撑集等于 链表(ret) ∧(键集合(ret) =Old(键集合(x)) ∪Old(键集合(y)))。因此，后置条件等价于 链表 (ret) ∧(键集合(ret) = Old(键集合(x)) ∪ Old(键集合(y)))。我们的优化包括通过语法方式执行此类简化，从而减轻 SMT 求解器的负担。下面给出 RevPrepend 的框架逻辑注解版本。

List(x) := 条件表达式(x =空指针, True, (后继(x) = 后继(x)) and List([后继(x)]) and Not(IsMember(x, Sp(List([后继(x)]))))) 键集合(x) := 条件表达式(x =空指针, EmptySetInt, SetAdd(键集合(后继(x)), x))

RevPrepend(x, y) returns(ret) @pre: List(x) and List(y) and IsEmpty(SetIntersect(Sp(List(x)), Sp(List(y)))) @post: List(ret) and (键集合(ret) = 集合并(Old(键集合(x)), Old(键集合(y))))

(如果 (x = 空指针) 则 ret :=y; 否则 tmp :=x.后继;

x.后继 := y; ret := RevPrepend(tmp, x);) 返回;

我们不详细讲解整个验证条件生成过程，只讨论涉及函数调用的几个有趣情况。其他基本块都很直接。例如，“if”情况只是将 T 设为前置条件、 x = 空指针 和 $ret = y$ 的合取项。最后，它会检查后置条件是否成立。由于 $ret = y$，这可直接由前置条件推出。在检查解引用安全性的基本块中，例如检查 tmp:= x.后继， T 将是前置条件，而 $x \neq$ 空指针，验证条件将检查 $x \in A$（其中 A 仅为前置条件的支撑集）。

调用 RevPrepend(tmp, x) 时，我们需要确保内存安全——也就是说，函数调用最初访问的位置包含在已分配集合之内，这也说明了为何需要宽松后置条件。以下基本块展示了这一点。

{List(x) and List(y)andIsEmpty(SetIntersect(Sp(List(x)), Sp(List(y))))} # T = List(x) and List(y)and IsEmpty(SetIntersect(Sp(List(x)), Sp(List(y)))), # A = Sp(T), Fr = {}, RD ={List, Keys}, H = \lambda f. f where f \in {next, key} 假设(x != 空指针); # T <- T and x !=空指针 tmp := x.next; # T <- T and tmp = next(x) x.next :=y; # H1 = H[next <- lambdaarg. ite(arg = x, y, next(arg))]# List1 = List[H1], Keys1 =Keys[H1], RD1 = {List, Keys, List1, Keys1} # Fr1 = {fr({x}, H, H1)} {RP: List1(tmp) and List1(x) and IsEmpty(SetIntersect(Sp(List1(tmp)), Sp(List1(x))))}

为此基本块生成的验证条件是 $(fr(x,\ H,\ H1) \wedge T) \implies (\varphi \wedge$ 支撑集$(\varphi) \subseteq A)$，其中 $\varphi =$(链表1(tmp) $\wedge$链表1(x) $\wedge$ (支撑集(链表1(tmp)) $\cap$ 支撑集(链表1(x)) = ϕ))。

为了确保堆的紧致性，我们必须刻画函数调用之后的堆。根据归纳假设，我们知道函数调用修改的堆部分由 RevPrepend(tmp, x) 的后置条件刻画。这促使我们引入函数调用规则。

{List(x) and List(y)andIsEmpty(SetIntersect(Sp(List(x)), Sp(List(y))))} 假设(x != 空指针) tmp := x.后继 x.后继 :=y ret := RevPrepend(tmp,x) # 令 PreCall = (List1(tmp)and List1(x) and IsEmpty(SetIntersect(Sp(List1(tmp)), Sp(List1(x))))) # H2[后继] = lambda arg.ite(Not(IsMember(arg, Sp(PreCall))), H1(next)(arg), next2(arg)) # H2[键] = lambda arg. ite(Not(IsMember(arg, Sp(PreCall))), H1(key)(arg), key2(arg)) # List2 = 链表1[H2],键集合2 =键集合1[H2]# 令 PostCall =(List2(ret) and (Keys2(ret) = SetUnion(Old(Keys1(tmp)), Old(Keys1(x))))) # T2 = T and PostCall # A2 = (A\Sp(PreCall))U Sp(PostCall) # Fr2 =Fr1 U {fr(Sp(PreCall), H1, H2)} # RD2 = RD1 U {List2, Keys2}

{List2(ret) and (Keys2(ret) =集合并(Old(Keys(x)), Old(Keys(y))))}

为这个基本块生成的验证条件是 $(fr(x, H, H1) \wedge fr(\text{支撑集(PreCall)}\ H1, H2) \wedge T2) \implies \varphi \wedge \text{支撑集}(\varphi) = A2$，其中 $\varphi = \text{链表2}(ret) \wedge (\text{键集合2}(ret) = (Old(\text{键集合}(x)) \cup Old(\text{键集合}(y))))$。

现在我们描述该验证条件有效性的高层证明。回想一下，RevPrepend 的契约告诉我们：如果从两个不相交链表开始执行程序，则 (1) 返回值是一个链表，其键是原来两个链表的键的并集；(2) 后置条件的堆支持集给出了程序的“修改后”堆。

根据前一个基本块的有效性，$RevPrepend(tmp\ x)$ 的前置条件在函数调用时成立。因此，我们可以得出结论，函数调用之后：(1) PostCall $=$ 链表2$(ret) \wedge$(键集合2$(ret) =$ 键集合1$(x) \cup$键集合1(tmp)) 成立；(2) 支撑集 (PostCall) 是此次调用修改后的堆。

现在我们证明 φ。只需证明 键集合1$(x) \cup$键集合1$(tmp) =$ 键集合$(x) \cup$键集合(y)。为此，注意到函数调用前只有 x 被修改。因此，根据框架规则 Fr1，键集合1$(y) =$ 键集合(y)，且 键集合$(tmp) =$ 键集合1(后继(x))。由此即可得出结论。

对$Sp(\varphi) = A2$ 的证明类似。由 Fr1 可得$Sp(\text{链表1}(y)) = Sp(\text{链表}(y))$，以及$Sp(\text{链表1}(tmp)) = Sp(\text{链表}(tmp))$。由此可得$Sp(\text{PreCall}) = A$。因此，$A2 = Sp(\text{PostCall}) = Sp(\varphi)$。