

# 基于函数式语义的循环和递归程序结构通用证明技术\*

李希萌<sup>1,2</sup>, 王国辉<sup>1,3</sup>, 张倩颖<sup>1,2</sup>, 施智平<sup>1,2</sup>, 关永<sup>1,3</sup>



<sup>1</sup>(首都师范大学 信息工程学院, 北京 100048)

<sup>2</sup>(电子系统可靠性技术北京市重点实验室 (首都师范大学), 北京 100048)

<sup>3</sup>(北京成像理论与技术高精尖创新中心 (首都师范大学), 北京 100048)

通信作者: 施智平, E-mail: [shizp@cnu.edu.cn](mailto:shizp@cnu.edu.cn)

**摘要:** 各类安全攸关系统的可靠运行离不开软件程序的正确执行. 程序的演绎验证技术为程序执行的正确性提供高度保障. 程序语言种类繁多, 且用途覆盖高可靠性场景的新式语言不断涌现, 难以为每种语言设计支撑其程序验证任务的整套逻辑规则, 并证明其相对于形式语义的可靠性和完备性. 语言无关的程序验证技术提供以程序语言的语义为参数的验证过程及其可靠性结果. 对每种程序语言, 提供其形式语义后可直接获得面向该语言的程序验证过程. 提出一种面向大步操作语义的语言无关演绎验证技术, 其核心是对不同语言中循环、递归等可导致无界行为的语法结构进行可靠推理的通用方法. 特别地, 借助大步操作语义的一种函数式形式化提供表达程序中子结构所执行计算的能力, 从而允许借助辅助信息对子结构进行推理. 证明所提出验证技术的可靠性和相对完备性. 通过命令式、函数式语言中的程序验证实例初步评估了该技术的有效性, 并在 Coq 辅助证明工具中形式化了所有理论结果和验证实例, 为基于辅助证明工具实现面向大步语义的语言无关程序验证工具提供了基础.

**关键词:** 程序验证; 大步操作语义; 定理证明; Coq 定理证明器

**中图法分类号:** TP311

中文引用格式: 李希萌, 王国辉, 张倩颖, 施智平, 关永. 基于函数式语义的循环和递归程序结构通用证明技术. 软件学报, 2023, 34(8): 3686–3707. <http://www.jos.org.cn/1000-9825/6616.htm>

英文引用格式: Li XM, Wang GH, Zhang QY, Shi ZP, Guan Y. Unified Proof Technique for Iterative and Recursive Program Structures Based on Functional Semantics. Ruan Jian Xue Bao/Journal of Software, 2023, 34(8): 3686–3707 (in Chinese). <http://www.jos.org.cn/1000-9825/6616.htm>

## Unified Proof Technique for Iterative and Recursive Program Structures Based on Functional Semantics

LI Xi-Meng<sup>1,2</sup>, WANG Guo-Hui<sup>1,3</sup>, ZHANG Qian-Ying<sup>1,2</sup>, SHI Zhi-Ping<sup>1,2</sup>, GUAN Yong<sup>1,3</sup>

<sup>1</sup>(Information Engineering College, Capital Normal University, Beijing 100048, China)

<sup>2</sup>(Beijing Key Laboratory of Electronic System Reliability Technology (Capital Normal University), Beijing 100048, China)

<sup>3</sup>(Beijing Advanced Innovation Center for Imaging Theory and Technology (Capital Normal University), Beijing 100048, China)

**Abstract:** The reliable functioning of safety-critical IT systems depends heavily on the correct execution of programs. Deductive program verification can be performed to guarantee the correct execution to a large extent. There are already a plethora of programming languages in use, and new languages oriented toward high-reliability scenarios are still being invented. As a result, it is difficult to devise a full-fledged logical system for each language to support the verification of programs and prove the soundness and completeness of the logical system with respect to the formal semantics of the language. Language-independent verification techniques offer sound verification

\* 基金项目: 国家重点研发计划 (2019YFB1309900); 国家自然科学基金 (61876111, 61877040, 62002246); 北京市教育委员会科技计划 (KM201910028005, KM202010028010)

本文由“形式化方法与应用”专题特约编辑陈立前副教授、孙猛教授推荐.

收稿时间: 2021-09-04; 修改时间: 2021-10-14, 2022-01-10; 采用时间: 2022-01-27; jos 在线出版时间: 2022-03-24

CNKI 网络首发时间: 2023-02-23

procedures parameterized over the formal semantics of programming languages. The specialization of the verification procedure with the formal semantics of a concrete programming language directly gives rise to a verification procedure for the language. This study proposes a language-independent verification technique based on big-step operational semantics. The technique features a unified procedure for sound reasoning about program structures that potentially cause unbounded behavior, such as iteration and recursion. In particular, the study employs a functional formalization of big-step operational semantics to support the explicit representation of the computation performed by the sub-structures of a program. This representation enables the exploitation of the auxiliary information provided for these sub-structures in the unified reasoning process. In addition, the study has proved the soundness and relative completeness of the proposed technique, evaluated the technique by verification examples in imperative and functional programming languages, and formalized all theoretical results and verification examples in the Coq proof assistant, and thus provides a basis for developing a language-independent program verifier with big-step operational semantics based on a proof assistant.

**Key words:** program verification; big-step operational semantics; theorem proving; Coq proof assistant

软件程序的正确性是影响各类安全攸关系统可靠性的重要因素之一。为提供最高级别的正确性保障, 往往使用形式化方法<sup>[1]</sup>对程序进行验证。在各类程序验证方法中, 基于定理证明思想的演绎验证 (deductive verification) 以程序语言的形式化语义为基础, 对程序的正确性进行证明。当目标程序的正确性依据较为复杂, 依赖于巧妙构思或深入领域知识时, 演绎验证方法可为验证任务的完成提供有力帮助。若基于辅助证明工具 (proof assistant) 实现<sup>[2]</sup>程序的演绎验证, 则能够给出高度可靠的验证结果。

对程序进行演绎验证的基本方法是直接以某种操作语义<sup>[3,4]</sup>的规则作为公理或定理, 构建程序正确性的数学证明。由于循环、函数调用等程序结构可能导致无界的程序行为 (unbounded behavior), 往往须运用数学归纳法完成证明, 其过程较为繁琐。一种简化思路是使用归纳不变式 (inductive invariant)<sup>[5]</sup>, 即构造逻辑表达式, 使其被整个程序的初始条件蕴含, 被程序的所有可能原子执行步骤保持, 且蕴含所需保障的终止条件。此种方法原理简单清晰, 然而构造能够刻画程序所有可达状态的不变式, 则是一项复杂的工作。

一类有效简化程序演绎验证的语义定义方式是公理语义<sup>[3,4]</sup>。它直接提供一系列逻辑规则, 帮助推导关于各类语句执行结果的判断。然而, 公理语义在反映程序语句含义方面的直观性不如操作语义, 其正确性常借助与其他语义的等价性证明加以支撑。基于公理语义思想, 发展出一系列程序逻辑, 提供丰富策略以帮助对指针、函数调用、对象、并发等语言特性进行推理。许多强大的软件验证工具 (如 Frama-C<sup>[6]</sup>、KeY<sup>[7]</sup>、Why3<sup>[8]</sup>、VST<sup>[9]</sup>、Iris<sup>[10]</sup>等) 支持基于程序逻辑对不同语言程序进行演绎验证。与公理语义情形类似, 程序逻辑的正确性往往借助其相对某个基准语义 (常为操作语义) 的可靠性和完备性加以支撑。程序语言数量庞大, 对于每种有验证需求的语言设计程序逻辑并完成可靠性和完备性证明是一项艰巨任务。

语言无关 (language-independent) 的程序验证技术<sup>[11-14]</sup>在一定程度上兼顾直接程序证明方法和程序逻辑的优势。此类技术的核心包括一个以操作语义为参数的验证过程和该过程的可靠性定理。对不同程序语言, 在提供该语言的操作语义后可直接得到一个该语言程序的半自动验证过程。该过程基于用户给出的类似程序逻辑中循环、递归函数等关键结构的辅助信息, 可靠验证程序的正确性。语言无关的程序验证技术解决的核心问题是如何对允许产生任意多执行步骤的程序结构 (如循环、递归函数等), 以一致的形式为其提供辅助信息, 对其进行可靠推理。这是因为图灵完备的程序语言无论具有何种范式和特性, 往往需要此类程序结构以支撑其表达能力。而此类结构的存在导致无法直接借助基于形式语义的符号执行完成程序的证明。

当前支持以辅助证明工具为平台进行语言无关程序验证的技术<sup>[11,12]</sup>均依托于程序的小步操作语义<sup>[15]</sup>。与小步语义相比, 大步语义<sup>[16]</sup>粒度较粗, 较难用来描述并发执行。然而在许多情形下, 大步语义的定义和使用相对简单, 如在大步语义的语义格局中往往无需引入调用栈等控制结构, 在使用大步语义的形式化证明中无需同时考虑推导序列 (derivation sequence) 和推导树 (derivation tree) 等。依托于大步语义的语言无关程序证明技术的研究, 不仅可支撑未定义小步语义的程序语言中的验证任务, 也有助于对一般程序语言, 发挥大步语义本身能够简化某些证明任务的优势。

本文提出一种能够基于大步操作语义进行程序验证的语言无关验证技术。语言无关的程序验证过程并不依赖于程序语言的语法结构, 而是在语义层面利用程序中关键子结构执行相关的辅助信息, 完整证明任务。小步语义描

述程序的单步执行  $c \rightarrow c'$  (其中  $c$  和  $c'$  为语义格局), 在此基础上, 使用单步执行所构成的序列即可表征程序中子结构的执行. 相较之下, 大步语义提供描述程序完整执行的关系  $c \rightarrow t$  (其中  $c$  为初始格局,  $t$  为终止格局), 若要获得其中子结构执行的表示, 需要深入  $c \rightarrow t$  的推导过程进行检视. 为解决这一问题, 本文以特定的函数形式描述程序语言大步语义的语义规则, 使得函数的一次应用对应于某个语义规则的一次使用, 其复合应用则能表征大步语义推导树的一个子树, 从而对程序中子结构的执行进行描述. 此种函数式描述也使本文中理论结果和实例程序的证明往往能借助函数表达式的一系列化简加以实现.

本文所提出技术包含以函数式大步操作语义为参数的验证过程及其可靠性定理. 验证过程将目标语言中程序的验证转化为依托辅助信息的符号执行. 所需辅助信息仅为关于程序中部分关键结构 (如循环、递归函数) 的局部信息, 其作用类似于各种程序逻辑所需要的循环不变式和函数契约. 本文给出该技术的可靠性证明, 即若程序得到验证, 那么程序满足部分正确性性质 (partial correctness)<sup>[17]</sup>. 本文通过命令式语言 While<sup>[3]</sup> 中阶乘计算程序的演绎证明、非确定性卫式命令语言 GC<sub>arr</sub> (guarded command language)<sup>[18]</sup> 中数组极大值查找程序的演绎证明、函数式语言 Fun<sub>lst</sub> 中有序列表合并程序的演绎证明对所提出技术的有效性进行了初步评估. 当辅助信息不足时 (如缺少关键循环的不变式信息), 可能出现形式规约中正确性性质得到满足, 但无法得到证明的情形. 本文证明若表征大步语义的函数为某种意义上的连续函数 (见第 7 节), 则能够向形式规约中添加辅助信息, 以完成正确性证明, 即在一定意义上, 本技术亦具有完备性. 本文所有理论结果和程序验证实例均在 Coq 辅助证明工具<sup>[19]</sup> 中进行了形式化, 除去所使用定理库的代码以及注释和空行, 该形式化在 Coq 中的代码量约 3000 行. 该形式化代码可在地址 <https://gitee.com/lixim/func-verify/tree/master> 处获得.

本文主要贡献可归结为如下 4 个内容.

- 一个以程序语言的函数式大步语义为参数的可靠的程序验证过程.
- 验证过程可靠性和相对完备性的证明.
- 使用验证技术对不同范式程序语言的简单程序实例进行验证的过程和结果.
- 所有理论结果和程序验证实例在 Coq 辅助证明工具中的形式化.

本文第 1 节具体讨论主要相关工作. 第 2 节介绍预备知识和符号约定. 第 3 节给出基于函数式大步操作语义的语言无关验证技术并证明其可靠性. 第 4 节依托简单命令式语言 While, 介绍证明技术的具体使用方法. 第 5、6 节给出含数组的卫式命令语言 GC<sub>arr</sub> 以及含列表的函数式语言 Fun<sub>lst</sub> 中的进一步实例. 第 7 节给出验证技术的相对完备性结果及其证明. 第 8 节讨论本文所提出技术的其他方面, 包括其局限性. 第 9 节总结全文.

## 1 主要相关工作

### 1.1 语言无关程序验证技术

Moore 提出一种对程序进行演绎验证的技术<sup>[11]</sup>, 可根据程序关键点处标注的断言生成归纳不变式 (inductive invariant), 如能验证该归纳不变式被程序的任何原子操作所保持, 则程序得到验证. 实质上该技术将类似于循环不变式的归纳断言 (inductive assertion) 转化为归纳不变式, 从而支持较少辅助信息下的可靠程序验证. 该技术无需对每种程序语言单独证明此种转化的可靠性, 故可视为一种语言无关的演绎验证技术. 该技术针对程序的小步执行模型提出, 建立在表达单步执行的二元关系的基础上. 与之相比较, 本文所提出技术适用于程序的大步操作语义, 在原理上并不基于向归纳不变式的转化.

Moore 等人提出一种在形式规约中辅助信息帮助下通过符号执行对程序进行可靠验证的技术<sup>[12]</sup>. 该技术提供的验证过程以程序语言的小步操作语义为参数, 其可靠性建立在最大不动点和余归纳法 (coinduction) 基础上. 与之相比较, 本文所提出技术适用于程序的大步操作语义, 其可靠性建立在归纳推理基础上, 较为直观. 需要指出, 本文所提出技术在形式规约的方式、相对完备性结果的形式方面从该工作中受到了启发.

先于文献 [12] 的工作, 同一研究组的 Stefanescu 等人提出一种基于重写系统和可达性逻辑 (reachability logic) 的语言无关程序证明技术, 在 K Framework 中进行了实现<sup>[13]</sup>. 由于特定逻辑系统的支撑, 该技术支持多种类型的

程序语义, 包括小步、大步操作语义. 相较之下, 本文工作适用的语义类型较为单一. 但另一方面, 本文所提出技术只需辅助证明工具所提供的逻辑系统 (如高阶逻辑) 作为支撑, 且无需程序语义以重写规则形式进行定义.

由于本文所提出技术基于程序的大步而非小步执行模型, 故较难用来处理程序语言中的并发特性. 而另一方面, 大步执行模型的定义及其在证明中的使用往往相对简单, 如在大步操作语义中, 一般无需通过引入额外程序语句对控制流状态进行记录; 在其使用中, 亦无需同时关注推导树和规约序列. 大步语义同样具有广泛用途.

## 1.2 基于翻译或转化的程序验证方法

通过将待验证程序由某种形式语言进行表示, 进而使用该形式语言的验证技术和工具对对象程序进行验证, 亦可得到适用于多种不同语言的验证工具. 此处形式语言可以为某种演算, 如  $\pi$ -演算、 $\lambda$ -演算等, 可以为某种函数式程序语言, 亦可以为某种命令式语言 (如 LLVM IR 等中间语言). 体现此种思想的工作包括但不限于文献 [8, 10, 20–22]. 基于翻译或转化的验证方法具有较强的实用性. 其问题主要在于, 使用形式语言对对象程序进行的表示 (或者源语言向目标形式语言的翻译) 是否恰当反映对象程序的实际语义, 有时无法简单确认. 对于某个语言翻译过程, 若要充分确认其可靠性, 最为稳妥的方法则是基于源语言和目标语言的形式化语义, 证明待验证程序及其翻译结果的等价性. 基于翻译或转化的程序验证技术与本文所提出技术具有不同原理和特点.

## 1.3 程序逻辑

程序逻辑相关研究工作是程序演绎验证方面的重要工作, 既有研究成果众多, 在本文范围内无法进行详细讨论. 基本的霍尔逻辑<sup>[17]</sup>支持顺序程序的正确性证明; 分离逻辑<sup>[23]</sup>提供支持局部推理的语法和规则, 能够简化指针、并发等特性的演绎验证; 假设-担保推理<sup>[24]</sup>适用于并发程序的组合式验证; 动态逻辑<sup>[25]</sup>提供能够区分“存在某个执行”, 或“全部执行”满足目标条件的算子; 关系型程序逻辑 (relational program logics)<sup>[26]</sup>支持对相互关联的多个程序执行的推理等. 许多程序逻辑往往又存在为处理概率系统、实时系统、混成系统、量子程序等所进行的扩展. 与本文所提出技术相比较, 程序逻辑对于某种特定的程序语言提供更加全面的演绎验证支持, 往往提供简化该语言所有特性相关推理的手段和规则. 本文所提出技术和其他语言无关程序验证技术则侧重对循环、递归等可导致无界行为的语言特性推理的简化, 但可同时适用于多种不同语言.

# 2 预备知识

## 2.1 大步操作语义

程序语言的大步操作语义 (或大步语义) 描述各类语法结构如何执行操作, 将初始状态转化为终止状态. 典型的初始状态包括程序的控制结构 (控制状态) 和各个变量的初始值 (变量状态). 控制状态和变量状态的组合通常称为语义格局 (configuration). 表示初始状态的语义格局称为初始格局 (initial configuration). 典型的终止状态给出各个变量的终值. 考虑命名的一致性, 下文将终止状态称为终止格局 (terminal configuration).

大步操作语义可借助一系列语义规则进行纸笔定义. 下面以 While 语言<sup>[3]</sup>为例进行简单介绍.

While 语言是一种高度简化的命令式语言. 其语法结构包括算术表达式  $a$ 、布尔表达式  $b$ 、语句  $S$ . 语句  $S$  可以为执行空操作的语句  $\text{skip}$ 、赋值语句  $x:=a$ 、分支语句  $\text{if } b \text{ then } S_1 \text{ else } S_2$ 、循环语句  $\text{while } b \text{ do } S$ 、顺序组合语句  $S_1; S_2$ . 设所有语句的集合为  $\text{Stmt}$ .

While 语言的大步语义规则如图 1 所示. 其中  $\sigma \in \Sigma$  表示变量状态, 是变量到整数的函数,  $\mathcal{A}[[a]]\sigma$  表示算术表达式  $a$  在变量状态  $\sigma$  下的求值结果,  $\mathcal{B}[[b]]\sigma$  表示布尔表达式  $b$  在变量状态  $\sigma$  下的求值结果,  $\sigma[x \mapsto \mathcal{A}[[a]]\sigma]$  表示将变量  $x$  映射到值  $\mathcal{A}[[a]]\sigma$ , 并将其他变量  $x'$  映射到值  $\sigma x'$  的变量状态. 该语义的初始格局为  $\text{Stmt} \times \Sigma$  中元素, 终止格局为  $\Sigma$  中元素.

使用语义规则推导从某个初始格局  $(S, \sigma)$  出发能够到达的终止格局  $\sigma'$ , 需要构建以  $(S, \sigma) \rightarrow \sigma'$  为根的有限高度的推导树 (derivation tree), 其所有叶子结点对应于无前提条件的规则 (如  $\text{skip}$ 、赋值的规则或 While 循环条件为假的规则). 关于简单顺序语句  $x:=2; x':=10$  执行结果的推导树如下所示:

$$\frac{\overline{(x := 2, \sigma) \rightarrow \sigma[x \mapsto 2]} \quad \overline{(x' := 10, \sigma[x \mapsto 2]) \rightarrow \sigma[x \mapsto 2, x' \mapsto 10]}}{(x := 2; x' := 10, \sigma) \rightarrow \sigma[x \mapsto 2, x' \mapsto 10]}$$

大步操作语义所描述的初始格局与终止格局间的二元关系实为使用语义规则所能推出的最小的初始格局、终止格局对的集合,理论上可通过最小不动点进行刻画.然而进行大步操作语义相关的理论证明或者使用大步操作语义进行程序证明时,往往使用推导树形状上的归纳法<sup>[3]</sup>,而无需考虑不动点概念.

形式语义的定义工作以及基于形式语义的程序证明工作往往较为繁琐,容易出错.辅助证明工具 (proof assistant)<sup>[2]</sup>提供有效的数理逻辑语言和手段,帮助进行精确无歧义的语义定义,并检查程序证明的所有步骤均符合数理逻辑的基本原则.在辅助证明工具中,常以归纳谓词<sup>[27]</sup>、求值函数<sup>[28,29]</sup>等方式对大步操作语义进行形式化,其形式化定义往往无法完全复刻图 1 中的纸笔定义,但反映相同的实质内容.

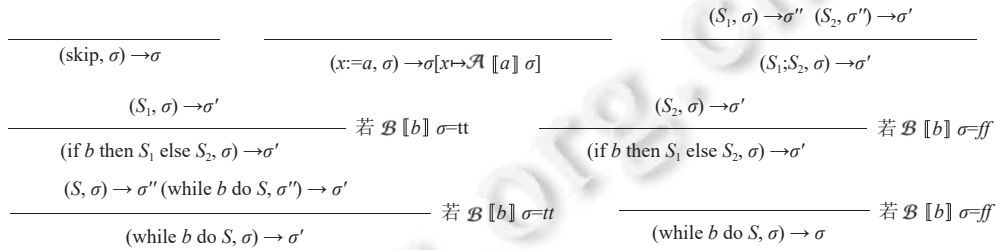


图 1 While 语言大步操作语义的语义规则 (纸笔定义形式)

## 2.2 本文数学记号

本文使用记号  $f a$  表示数学函数  $f$  在参数  $a$  上的应用.有时为显式表示函数  $f$  的自变量  $x$ ,使用记号  $\lambda x.f x$  对函数  $f$  进行表示.函数应用操作具有左结合性,故  $f a b$  表示  $(f a) b$ ,即函数  $f$  应用在参数  $a$  上,结果为另一函数  $g$ ,进而将该函数  $g$  应用在参数  $b$  上.对集合  $A$ ,使用  $\mathcal{P}(A)$  表示  $A$  的幂集,即  $A$  的所有子集构成的集合.本文所讨论的程序语言对整数  $z \in \mathbb{Z}$  和整数列表  $zs \in \mathbb{Z}^*$  进行操作.用  $[\ ]$  表示空列表,用  $z::zs$  表示列表  $zs$  前附加整数  $z$  后所得的列表,用  $|zs|$  表示列表  $zs$  的长度,用  $zs[z_1 \rightarrow z_2]$  表示这样的列表  $zs'$ ,其第  $z_1$  个元素 ( $z_1 \in \{0, \dots, |zs|-1\}$ ) 为  $z_2$ ,而  $zs'$  的长度以及  $zs'$  中其他位置的元素均与列表  $zs$  相同.作为本文所使用元语言 (metalanguage) 的一部分,表达式  $\text{if } b \text{ then } P \text{ else } Q$  在条件  $b$  成立时表示  $P$ ,否则表示  $Q$ .不难根据上下文将其与对象语言中的类似表达式进行区分.

## 3 程序证明技术及其可靠性

### 3.1 程序的形式规约

设  $C$  为所有初始格局的集合,  $T$  为所有终止格局的集合.

设  $D := C \rightarrow \mathcal{P}(T)$ .即  $D$  中函数将每一个初始格局映射为一组可能的终止格局.对  $D$  中元素定义二元关系  $\leq$ ,使得  $d_1 \leq d_2$  当且仅当对任意  $c$ ,有  $d_1 c \subseteq d_2 c$ .用  $\perp$  表示  $D$  中的最小元素.故对任何初始格局  $c$ ,有  $\perp c = \emptyset$ .

$D$  中的函数既可表示程序从具体初始格局所表达状态开始执行的结果,又可表示关于程序的形式规约.

(1) 当  $D$  中函数表示程序从具体初始格局所表达状态开始执行的结果时,该函数将每个初始格局  $c$  映射为由  $c$  开始执行程序所能到达的终止格局集合.故无论对确定性语言还是对非确定性语言,  $D$  中函数均能用来描述该语言程序的执行结果.

(2) 当  $D$  中函数用作程序的形式规约  $\Phi$  时,该规约起到两个作用.首先,  $\Phi$  给出整个程序的正确性需求——程序从每个  $C$  中格局  $c$  开始的实际执行结果应存在于集合  $\Phi c$  中.若  $\Phi c = T$ ,则表示对程序从初始格局  $c$  开始的执行结果不作限制.其次,  $\Phi$  给出关于程序中关键结构 (如循环、递归函数等) 执行结果的辅助信息——对于某个语句和相应的初始格局  $c$ ,  $\Phi c$  为该语句实际终止格局集合的超集.若  $\Phi c = T$ ,则表示对该语句未提供任何辅助信息,因为  $T$  包含形式系统中所有的终止格局.

以下给出用  $D$  中函数表达式程序规约的一个示例. 考虑如下的 While 语言程序  $S_{fac}$ .

$fac := m; \text{ while } 1 < m \text{ do } (m := m - 1; fac := fac * m)$

本程序计算变量  $m$  初始值的阶乘, 计算结果存放在变量  $fac$  中. 以下用  $S_{wh}$  表示  $S_{fac}$  中的 while 循环语句, 用  $S_{seq}$  表示其中的顺序组合语句  $m := m - 1; fac := fac * m$ .

具体化初始格局集合  $C := Stmt \times \Sigma$ 、终止格局集合  $T := \Sigma$ , 其中  $\Sigma := Var \rightarrow \mathbb{Z}$  为变量状态集合. 则本例中  $D = (Stmt \times \Sigma) \rightarrow \mathcal{P}(\Sigma)$ . 对阶乘计算程序  $S_{fac}$  给出如下的形式规约.

$$\begin{aligned} \Phi_{fac}(S_{fac}, \sigma) &:= \{ \sigma' \in \Sigma \mid \sigma'(fac) = (\sigma m)! \} && \text{若 } \sigma m > 0 \\ \Phi_{fac}(S_{wh}, \sigma) &:= \{ \sigma' \in \Sigma \mid \sigma'(fac) = (\sigma fac) * (\sigma m - 1)! \} && \text{若 } \sigma m > 0 \\ \Phi_{fac} c &:= \Sigma && \text{若初始格局 } c \text{ 不满足以上条件} \end{aligned}$$

形式规约  $\Phi_{fac} \in D$  所表达的正确性条件为: 若从状态  $\sigma$  开始执行语句  $S_{fac}$ , 且状态  $\sigma$  下变量  $m$  为正数, 那么执行终止后的状态  $\sigma'$  满足“ $\sigma'(fac) = (\sigma m)!$ ”. 这表达了  $S_{fac}$  终止后, 变量  $fac$  的值应为  $m$  初值的阶乘, 而除去  $fac$ 、 $m$  以外其他变量的值不予关心. 形式规约  $\Phi_{fac}$  关于初始格局  $(S_{wh}, \sigma)$  的部分具有如下含义. 若某次到达循环头部时状态为  $\sigma$ , 其中变量  $m$  为正数, 则整个循环结束后, 变量  $fac$  的值为“ $(\sigma fac) * (\sigma m - 1)!$ ”. 该部分的作用相当于程序逻辑中的循环不变式. 若格局  $c$  不具备  $(S_{fac}, \sigma)$  或  $(S_{wh}, \sigma)$  的形式, 或其状态中变量  $m$  不为正数, 则  $\Phi_{fac} c = \Sigma$ , 表示不关注由此种初始格局  $c$  开始执行的结果, 或不需要提供关于此种初始格局  $c$  的辅助性信息.

上述形式规约中, 在条件  $\sigma m > 0$  下所指定的终止格局集合  $\{ \sigma' \in \Sigma \mid \sigma'(fac) = (\sigma m)! \}$  对应于程序逻辑中的霍尔三元组  $\{ m=n \wedge n>0 \} S_{fac} \{ fac=n! \}$ , 其中  $n$  为逻辑变量 (logical variable), 用于记录程序变量  $m$  的初始值. 形式规约  $\Phi_{fac}$  中允许使用程序状态, 相应省去了该逻辑变量. 对于不同的程序语言, 霍尔三元组使用的断言语言 (assertion language) 及其语义往往需要单独定义, 好处是可以去掉对程序状态的显式引用, 使形式规约更为简洁. 相较之下, 本文中形式规约  $\Phi$  的定义对不同语言具有一致的形式, 同时并未过度牺牲表达方面的简练程度 (见第 5.2、6.2 节实例). 此外, 对于具体的语言或语言特性, 可考虑引入进一步记号以简化形式规约  $\Phi$  的表达.

若使用大步操作语义直接证明规约  $\Phi_{fac}$  得到满足, 即  $fac$  的终值等于  $m$  初值的阶乘, 需要使用基于推导树形状的归纳法 (induction on the shape of derivation trees)<sup>[3]</sup> 证明  $S_{fac}$  中循环结构的不变式. 一般地, 对于每个执行次数没有常数上界的循环, 均需要对其不变式进行归纳证明. 若改为使用霍尔逻辑进行阶乘程序的证明, 则无需显式使用归纳法. 但 While 语言的霍尔逻辑仅适用于 While 语言程序的正确性证明, 对于每种新的程序语言, 则需要为该语言设计专门的程序逻辑, 并证明其关于操作语义的可靠性和完备性, 其证明工作量往往随语言特性繁杂程度的增长而迅速增加. 本节剩余部分所提出技术则允许基于任一程序语言大步操作语义的形式化定义, 直接获得该语言程序的简单验证过程——使用该验证过程无需关于待证程序中循环结构、递归函数的无界行为进行归纳推理.

### 3.2 函数式大步语义形式化和程序正确性条件

某种程序语言大步操作语义的纸笔定义通常由一系列如下形式的语义规则构成.

$$\frac{\dots c_1 \rightarrow t_1 \dots c_n \rightarrow t_n \dots}{c \rightarrow t} \quad \text{若 } \alpha c \quad (1)$$

直观上, 该规则表示在借助逻辑谓词  $\alpha$  表达的条件  $\alpha c$  下, 若从每个初始格局  $c_i$  开始的执行结束时所能到达的终止格局包括  $t_i$ , 且  $c_1 \dots c_n$ 、 $t_1 \dots t_n$  满足一定条件 (未在规则中显示), 那么从初始格局  $c$  开始的执行结束时所能到达的终止格局包括  $t$  ( $t$  与其他参数的联系未在规则中显示).

现考虑通过函数定义对大步语义进行形式化. 具体引入一个函数  $h \in D \rightarrow D$  (注意  $D = C \rightarrow \mathcal{P}(T)$ ), 由一系列如下形式的定义式给出:

$$hfc := \{ t \mid \dots t_1 \in fc_1 \wedge \dots \wedge t_n \in fc_n \dots \} \quad \text{若 } \alpha c.$$

若函数  $f \in D$  能够对任一初始格局给出通过构造最大高度为  $m$  的推导树 (derivation tree) 所能导出的终止格局集合, 那么  $hfc$  便给出使用语义规则 (1) 和关于  $c_1 \dots c_n$  的子树构造高度为  $m+1$  的推导树所能导出的终止格局集合. 若以此种定义式形式化大步语义的所有语义规则, 则对于这样定义的  $h$ , 其在  $\perp$  上的  $k$  次迭代结果 (亦为  $D$

中的函数) 即能对每个初始格局, 给出通过构造最大高度为  $k+1$  的推导树, 所能导出的相应终止格局集合.

以下给出使用函数  $h$  对 While 语言的大步操作语义进行形式化的示例. 具体化初始格局集合  $C := Stmt \times \Sigma$ 、终止格局集合  $T := \Sigma$ , 其中  $\Sigma := Var \rightarrow \mathbb{Z}$  为变量状态集合. 在此基础上, 使用图 2 中函数  $h$  的定义形式化图 1 中 While 语言的大步操作语义规则. 图 2 中,  $\mathcal{A}$  和  $\mathcal{B}$  分别为算术表达式和布尔表达式的求值函数<sup>[3]</sup>.

$$\begin{aligned}
 hf(skip, \sigma) &:= \{ \sigma \} \\
 hf(x:=a, \sigma) &:= \{ \sigma[x \mapsto \mathcal{A}[a]\sigma] \} \\
 hf(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) &:= f(S_1, \sigma) && (\text{若 } \mathcal{B}[b] \sigma = tt) \\
 hf(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) &:= f(S_2, \sigma) && (\text{若 } \mathcal{B}[b] \sigma = ff) \\
 hf(\text{while } b \text{ do } S, \sigma) &:= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in f(S, \sigma) \wedge \sigma' \in f(\text{while } b \text{ do } S, \sigma'') \} && (\text{若 } \mathcal{B}[b] \sigma = tt) \\
 hf(\text{while } b \text{ do } S, \sigma) &:= \{ \sigma \} && (\text{若 } \mathcal{B}[b] \sigma = ff) \\
 hf(S_1; S_2, \sigma) &:= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in f(S_1, \sigma) \wedge \sigma' \in f(S_2, \sigma'') \}
 \end{aligned}$$

图 2 形式化 While 语言大步语义的函数  $h$

图 2 中定义式  $hf(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma) := f(S_1, \sigma)$  形式化图 1 中 if 语句的第 1 条语义规则, 表示分支条件为真时, 若使用某个高度的操作语义推导树可导出  $(S_1, \sigma)$  对应的终止格局集合  $f(S_1, \sigma)$ , 那么使用该规则构造高度增 1 的语义推导树, 可导出  $(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma)$  对应的终止格局集合  $hf(\text{if } b \text{ then } S_1 \text{ else } S_2, \sigma)$ , 它与  $(S_1, \sigma)$  对应的终止格局集合相同. 其他定义式如何对图 1 中的语义规则进行形式化, 可类似解释.

下面基于用函数  $h$  形式化的大步语义以及形式规约  $\Phi$  给出程序的正确性条件.

**定义 1.** 正确性条件. 若有  $\forall n: h^n \perp \leq \Phi$ , 则称形式规约  $\Phi$  得到满足, 记作  $\models \Phi$ .

本定义中, 实际要求  $\forall n c: h^n \perp c \subseteq \Phi c$ , 即对任意初始格局, 迭代函数  $h$  任意多次可得到的执行结果均在形式规约所规定的范围内. 直观上, 可解释为使用大步语义规则构造任何有限深度的推导树, 所得到的从初始格局开始的执行结果均在形式规约所规定的范围内——亦即程序的任何终止执行, 其结果均在形式规约所规定的范围内. 这是在不假设具体程序语言的情况下, 对部分正确性条件 (partial correctness)<sup>[17]</sup> 的一般性表达. 程序的正确性规约往往只关注执行结果所需满足的部分关键条件, 故  $\Phi c$  中允许包含程序实际执行结果以外的元素. 阶乘计算程序的形式规约即是一个具体例子. 因此, 定义 1 中使用“ $\leq$ ”而非“ $=$ ”来表达程序执行结果和形式规约的关系.

虽然本节仅给出使用函数  $h$  形式化 While 语言大步操作语义的示例, 该方法同样适用于其他确定性、非确定语言. 使用函数  $h$  形式化程序语言大步操作语义的其他示例在本文第 5、6 节中给出.

### 3.3 程序正确性的验证

定义函数  $F \in D \rightarrow D \rightarrow D$ , 使得:

$$\begin{aligned}
 F \Phi f &:= h(f \oplus \Phi) \\
 f \oplus \Phi &:= \lambda c. (\text{if } \Phi c = T \text{ then } f c \text{ else } \Phi c)
 \end{aligned}$$

直观上, 当形式规约  $\Phi$  未能给出关于从初始格局  $c$  开始执行目标程序所得结果的任何信息时 (即  $\Phi c = T$  时),  $F$  通过函数  $f$  计算执行结果的范围, 否则  $F$  通过形式规约  $\Phi$  给出执行结果的可能范围. 直观上, 函数  $F$  借助形式规约  $\Phi$  给出关于程序执行结果的近似范围. 借助简单推导可证, 对将任何初始格局均映射为  $T$  的形式规约  $\Phi_T$ , 有  $F \Phi_T = h$ . 这反映出当形式规约未提供任何有用信息时,  $F$  退化为表达具体执行的函数  $h$ .

以下基于函数  $F \Phi$  给出程序的验证条件, 亦即使用本文所提出技术直接证明的目标.

**定义 2.** 证明目标. 若有  $\forall n: (F \Phi)^n \perp \leq \Phi$ , 则称形式规约  $\Phi$  得到验证, 记作  $\vdash \Phi$ .

在本定义中, 实际要求  $\forall n c: (F \Phi)^n \perp c \subseteq \Phi c$ , 即对任意初始格局, 迭代函数  $(F \Phi)$  给出目标程序产生结果的可能范围均包在形式规约所要求的范围之内. 定义中包含关于迭代次数  $n$  的全称量词, 看似会使证明过程复杂化. 但后文中程序证明实例显示, 在形式规约  $\Phi$  包含循环、递归等程序结构所需关键辅助信息的前提下, 若要完成证明, 只需假设一般  $n$  值, 对  $(F \Phi)^n \perp c$  进行化简 (化简过程中可消去  $n$ ), 并确认最终化简结果在  $\Phi c$  中.

为保证验证的可靠性, 以下证明  $\vdash \Phi$  到  $\models \Phi$  的逻辑蕴含. 该证明无需假设  $h$  定义式的具体形式, 但需要  $h$  单调.

**定义 3.** 单调函数. 若函数  $f \in D \rightarrow D$  满足对任何  $d_1 \leq d_2, f d_1 \leq f d_2$  成立, 则称  $f$  为单调函数.

直观上不难看出, 若函数  $h$  以第 3.2 节中方式对大步语义的规则进行形式化, 则  $h$  为单调函数. 一般地, 可考虑严格刻画  $h$  定义式所具有的形式, 从而证明一类函数  $h$  均为单调函数. 但由于该形式的表达较繁, 本文并不对其进行刻画. 在第 4-6 节中, 具体  $h$  定义的单调性均可通过简单机械的步骤加以证明.

以下证明一关键引理.

**引理 1.** 假设  $h$  为单调函数. 若  $\vdash \Phi$  成立, 则有  $\forall n: h^n \perp \leq (F \Phi)^n \perp$ .

证明: 在  $n$  上使用数学归纳法.

若  $n=0$ , 有  $h^0 \perp = \perp$ , 以及  $(F \Phi)^0 \perp = \perp$ , 故有  $h^0 \perp \leq (F \Phi)^0 \perp$ .

若  $n=k+1$ , 其中  $k \geq 0$ , 则进行如下推导.

$$\begin{aligned}
 (F \Phi)^n \perp &= (F \Phi) ((F \Phi)^k \perp) && \text{(由 } n \text{ 次迭代函数的定义)} \\
 &= F \Phi ((F \Phi)^k \perp) && \text{(由函数应用的左结合性)} \\
 &= h (((F \Phi)^k \perp) \oplus \Phi) && \text{(由 } F \text{ 的定义)} \\
 &= h \lambda c. (\text{if } \Phi c = T \text{ then } ((F \Phi)^k \perp) c \text{ else } \Phi c) && \text{(由 } \oplus \text{ 的定义)} \\
 &\geq h \lambda c. (((F \Phi)^k \perp) c) && \text{(} \vdash \Phi \text{ 说明 } ((F \Phi)^k \perp) c \leq \Phi c, h \text{ 单调)} \\
 &\geq h \lambda c. ((h^k \perp) c) && \text{(由归纳假设、} h \text{ 的单调性)} \\
 &= h^n \perp && \text{证毕.}
 \end{aligned}$$

上述引理指出若形式规约  $\Phi$  能够得到验证, 那么对任何  $n$  值,  $h^n \perp$  不超过  $(F \Phi)^n \perp$ , 直观上表达具体计算结果可由基于  $\Phi$  导出的抽象计算结果进行近似. 基于引理 1 可直接导出下列可靠性定理.

**定理 1.** 验证技术的可靠性. 假设  $h$  为单调函数. 对任一形式规约  $\Phi$ , 若有  $\vdash \Phi$ , 则有  $\models \Phi$ .

基于由单调函数  $h \in D \rightarrow D$  进行形式化的任何程序语言的大步语义, 以上定理说明若能够证明  $(F \Phi)^n \perp \leq \Phi$  对任一自然数  $n$  成立, 则能够保证形式规约  $\Phi$  的正确性. 由于定理 1 的证明不依赖于某个具体的  $h$  定义 (表征某个特定程序语言的语义), 其所支撑的程序证明技术可视为一种语言无关<sup>[11,12]</sup>程序证明技术. 第 4 节中的实例显示, 本定理的使用能够省去基于大步语义的直接正确性证明中为处理每个目标程序中的循环、递归调用所需使用的归纳推理.

## 4 While 语言程序证明示例

本节使用定理 1 证明第 3.1 节中给出的 While 语言阶乘程序的形式规约得到满足, 并讨论本文技术对循环结构的处理方式与霍尔式程序逻辑的差异.

### 4.1 阶乘程序正确性证明

关于第 3.2 节中 While 语言大步语义的形式化, 容易证明下列引理.

**引理 2.** 图 2 中定义的函数  $h$  为单调函数.

本引理的证明转化为对任意满足  $f_1 \leq f_2$  的  $f_1, f_2$ , 和任意  $c$ , 证明  $h f_1 c \leq h f_2 c$ . 对  $c$  中的语句进行分类讨论即可完成证明. 在本引理基础上, 即可使用第 3 节中定理 1 对任何 While 语言程序进行可靠验证.

以下讨论第 3.1 节中形式规约  $\Phi_{fac}$  的证明. 该证明的最终目标是说明  $\Phi_{fac}$  得到满足, 即建立  $\models \Phi_{fac}$  (见定义 1). 使用定理 1, 往证  $\vdash \Phi_{fac}$ . 即证明对每种可能的初始格局  $c$ , 有  $\forall n: (F \Phi_{fac})^n \perp c \leq \Phi_{fac} c$ . 以下对  $c$  进行分类讨论.

(1) 若  $c$  不具有形式  $(S_{fac}, \sigma)$  或  $(S_{wh}, \sigma)$ , 或者根据  $c$  中的状态,  $m$  的值非正, 则  $\Phi_{fac} c = \Sigma$ . 此时显然有  $(F \Phi_{fac})^n \perp c \leq \Phi_{fac} c$ , 因为任何状态的集合均包含在所有状态的全集  $\Sigma$  中.

(2) 若  $c = (S_{fac}, \sigma)$ , 其中  $\sigma m > 0$ , 则对  $(F \Phi_{fac})^n \perp c$  进行计算和化简. 以下过程中, 右侧标有  $(\star)$  的中间结果只在  $n$  足够大, 使  $F \Phi_{fac}$  的迭代能够展开时成立; 否则, 相应的中间结果为  $\emptyset$ , 必定包含于  $\Phi_{fac} c$ .

$$\begin{aligned}
 (F \Phi_{fac})^n \perp (S_{fac}, \sigma) &= (F \Phi_{fac}) ((F \Phi_{fac})^{n-1} \perp) (S_{fac}, \sigma) && (\star) \\
 &= F \Phi_{fac} ((F \Phi_{fac})^{n-1} \perp) (S_{fac}, \sigma)
 \end{aligned}$$



$$\begin{aligned}
&= h(((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{fac}, \sigma) && \text{(由第 3.2 节中 } F \text{ 的定义)} \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(fac := m, \sigma) \wedge && \text{(由 } h \text{ 的定义和 } S_{fac} \text{ 的形式)} \\
&\quad \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in ((F \Phi_{fac})^{n-1} \perp)(fac := m, \sigma) \wedge && \text{(由 } \oplus \text{ 定义)} \\
&\quad \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in (F \Phi_{fac})((F \Phi_{fac})^{n-2} \perp)(fac := m, \sigma) \wedge && (\star) \\
&\quad \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in h(((F \Phi_{fac})^{n-2} \perp) \oplus \Phi_{fac})(fac := m, \sigma) \wedge \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma[fac \mapsto \sigma m]) && \text{(由 } h \text{ 定义可知 } \sigma'' \text{ 唯一值为 } \sigma[fac \mapsto \sigma m]) \\
&= \Phi_{fac}(S_{wh}, \sigma[fac \mapsto \sigma m]) && \text{(由 } \oplus \text{ 定义和 } (\sigma[fac \mapsto \sigma m] m) > 0) \\
&= \{ \sigma' \in \Sigma \mid \sigma'(fac) = (\sigma[fac \mapsto \sigma m] fac) * ((\sigma[fac \mapsto \sigma m] m) - 1)! \} \\
&= \{ \sigma' \in \Sigma \mid \sigma'(fac) = (\sigma m) * (\sigma m - 1)! \} \\
&\subseteq \Phi_{fac}(S_{fac}, \sigma) && \text{(由 } \sigma m > 0 \text{ 可得 } (\sigma m) * (\sigma m - 1)! = (\sigma m)!)
\end{aligned}$$

(3) 若  $c = (S_{wh}, \sigma)$ , 其中  $\sigma m > 0$ , 则对  $(F \Phi_{fac})^n \perp c$  进行计算和化简. 以下过程中, 右侧标有  $(\star)$  的中间结果只在  $n$  足够大, 使  $F \Phi_{fac}$  的迭代能够展开时成立; 否则, 相应的中间结果为  $\emptyset$ , 必定包含于  $\Phi_{fac} c$ .

$$\begin{aligned}
(F \Phi_{fac})^n \perp (S_{wh}, \sigma) &= F \Phi_{fac} ((F \Phi_{fac})^{n-1} \perp) (S_{wh}, \sigma) && (\star) \\
&= h(((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma) && \text{(记为 RHS)}
\end{aligned}$$

(i) 若  $\sigma m = 1$ , 则根据图 2 中  $h$  的定义, RHS 为  $\{ \sigma \}$ . 由于  $\sigma m = 1$ , 且  $0! = 1$ , 可知单元素集  $\{ \sigma \}$  包含于  $\Phi_{fac}(S_{wh}, \sigma) = \{ \sigma' \in \Sigma \mid \sigma'(fac) = (\sigma fac) * (\sigma m - 1)! \}$ .

(ii) 若  $\sigma m > 1$ , 则继续进行如下计算化简 (略去部分计算步骤及原因).

$$\begin{aligned}
\text{RHS} &= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{seq}, \sigma) \wedge \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in F \Phi_{fac} ((F \Phi_{fac})^{n-2} \perp) (S_{seq}, \sigma) \wedge \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} && (\star) \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in h(((F \Phi_{fac})^{n-2} \perp) \oplus \Phi_{fac})(S_{seq}, \sigma) \wedge \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= \{ \sigma' \mid \exists \sigma''' : \sigma''' \in (((F \Phi_{fac})^{n-2} \perp) \oplus \Phi_{fac})(m := m-1, \sigma) \wedge \\
&\quad \exists \sigma'' : \sigma'' \in (((F \Phi_{fac})^{n-2} \perp) \oplus \Phi_{fac})(fac := fac * m, \sigma'') \wedge \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= \{ \sigma' \mid \exists \sigma''' : \sigma''' \in ((F \Phi_{fac})^{n-2} \perp) (m := m-1, \sigma) \wedge \exists \sigma'' : \sigma'' \in ((F \Phi_{fac})^{n-2} \perp) (fac := fac * m, \sigma'') \wedge \\
&\quad \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} && (\star) \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' \in h(\dots) (m := m-1, \sigma) \wedge \exists \sigma'' : \sigma'' \in h(\dots) (fac := fac * m, \sigma'') \wedge \\
&\quad \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= \{ \sigma' \mid \exists \sigma'' : \sigma'' = \sigma[m \mapsto \sigma m - 1, fac \mapsto (\sigma fac) * (\sigma m - 1)] \wedge \\
&\quad \sigma' \in (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma'') \} \\
&= (((F \Phi_{fac})^{n-1} \perp) \oplus \Phi_{fac})(S_{wh}, \sigma[m \mapsto \sigma m - 1, fac \mapsto (\sigma fac) * (\sigma m - 1)]) \\
&= \Phi_{fac}(S_{wh}, \sigma[m \mapsto \sigma m - 1, fac \mapsto (\sigma fac) * (\sigma m - 1)]) \\
&= \{ \sigma' \in \Sigma \mid \sigma'(fac) = ((\sigma fac) * (\sigma m - 1)) * (\sigma m - 1 - 1)! \} \\
&\subseteq \Phi_{fac}(S_{wh}, \sigma)
\end{aligned}$$

以上计算过程中, 倒数第 4 行之前的步骤对循环  $S_{wh}$  进行一轮执行, 得到该轮结束时的格局  $(S_{wh}, \sigma[m \mapsto \sigma m - 1, fac \mapsto (\sigma fac) * (\sigma m - 1)])$ . 由倒数第 4 行起使用关于该循环的辅助信息——证明根据该辅助信息, 从第一轮循环结束时的格局开始继续执行循环所能到达的终止格局均为由原初始格局  $(S_{wh}, \sigma)$  开始执行循环所能到达的终止格局. 由以上基于分类讨论和计算化简的证明过程可知, 下列定理成立.

**定理 2.** 阶乘程序的正确性.  $\models \Phi_{fac}$  成立.

由本定理的证明过程可见, 虽然条件  $\vdash \Phi_{fac}$  含有关于自然数  $n$  的全称量词, 但无需在  $n$  上使用数学归纳法, 亦无需添加诸如  $n$  值应大于某个具体数值的前提条件. 得益于语义定义本身的函数形式, 证明过程的主体是对函数

求值表达式  $(F \Phi_{fac})^n \perp c$  的展开和化简, 这有助于在辅助证明工具中借助计算和改写等证明策略完成形式化证明. 定理 1 有效支撑了在提供类似于霍尔逻辑中循环不变式的前提下对  $\Phi_{fac}$  正确性的可靠证明. 定理 1 无需对 While 语言单独证明, 而是在确认 While 语言语义函数单调性的前提下即可直接使用.

#### 4.2 在循环结构处理方面与霍尔式程序逻辑的差异

以下结合本例和图 3, 对本文中循环的形式规约和程序逻辑中的循环不变式进行比较. 图 3 包括左右两个子图, 分别呈现程序逻辑中循环不变式以及本文中循环规约的原理. 每个子图中一系列圆圈表示每一轮循环开始时的语义格局以及整个循环结束时的语义格局 ( $c_k$ ). 循环不变式设计为在每轮循环开始处成立的条件, 假设其在某轮循环开始处 (如  $c_1$  处) 成立, 需要往证其在下一轮循环开始处 (如  $c_2$  处) 成立. 相较之下, 本文中循环的形式规约对于任何一轮循环开始处的语义格局 (如  $c_1$ ), 给出如果循环能够终止, 那么其终止格局 (如  $c_k$ ) 所满足的条件. 在阶乘程序的例子中,  $c_1$  为满足条件  $\sigma m > 0$  的语义格局 ( $S_{wh}, \sigma$ ), 而完成整个循环时的条件由集合  $\Phi_{fac}(S_{wh}, \sigma)$  表示. 此种循环规约的证明策略是, 首先模拟第一轮循环的执行, 即导出图 3 右侧子图中对  $c_2$  成立的条件, 而后往证: 根据同一循环的规约, 由  $c_2$  开始结束循环时可能到达的语义格局 (即  $\Phi_{c_2}$  中元素) 均在  $\Phi_{c_1}$  中. 若考虑循环的尾递归实现, 则此种处理方式较易理解.

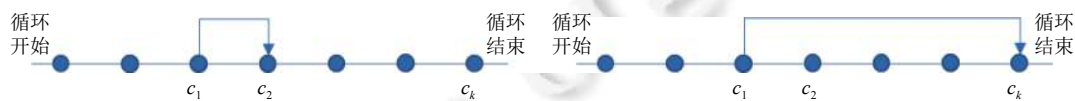


图 3 程序逻辑中循环不变式与本文循环规约的比较

在已有相关工作中, 文献 [11] 对循环的处理类似于程序逻辑中的循环不变式, 而文献 [12,13] 中对循环的处理类似于本文中循环的形式规约.

## 5 卫式命令语言程序证明实例

本节考虑带有声明和一维数组扩展的非确定性卫式命令语言  $GC_{arr}^{[18]}$ , 通过定义函数  $h \in D \rightarrow D$  (其中  $D = C \rightarrow \mathcal{P}(T)$ ) 对  $GC_{arr}$  的大步操作语义进行形式化, 并用定理 1 对一个查找数组极大值的非确定性程序进行证明.

### 5.1 卫式命令语言 $GC_{arr}$ 大步语义的形式化

带有声明、一维数组扩展的卫式命令语言  $GC_{arr}$  语法如下, 其中  $a \in AExp$  为算术表达式,  $b \in BExp$  为布尔表达式,  $G \in GCom$  为卫式命令,  $S \in Stmt$  为语句.

$a ::= nc \mid x \mid X[a] \mid a + a \mid a - a \mid a * a \mid a / a$

$b ::= true \mid false \mid a = a \mid a < a \mid b \ \&\& \ b \mid !b$

$G ::= b \rightarrow S \mid G \square G$

$S ::= var \ x := nc \mid arr \ X := ns \mid skip \mid x := a \mid X[a] := a \mid if \ G \ fi \mid do \ G \ od \mid S; S$

上述语法中  $nc$  为数值常量,  $x$  为整型变量,  $X$  为一维数组变量. 卫式命令  $b \rightarrow S$  仅当条件表达式  $b$  求值为真时执行语句  $S$ . 命令  $G_1 \square G_2$  非确定选择  $G_1$  或  $G_2$  进行执行. 条件分支语句  $if \ G \ fi$  对卫式命令进行执行, 循环语句  $do \ G \ od$  对卫式命令  $G$  进行重复执行. 语句  $var \ x := nc$  声明整型变量  $x$  并用常量  $nc$  对其进行初始化. 语句  $arr \ X := ns$  声明一维数组变量  $X$  并用常数列表  $ns$  对其进行初始化.

程序状态  $\sigma$  具有形式  $(s, \theta)$ , 其中  $s \in Var \rightarrow \mathbb{Z} \cup \{\perp\}$  为变量状态,  $\theta \in Arr \rightarrow \mathbb{Z}^* \cup \{\perp\}$  为数组状态. 变量状态将程序变量映射为该变量的值; 或  $\perp$ , 表示该变量未声明. 数组状态将数组变量映射为由整数列表所表示的数组内容; 或  $\perp$ , 表示该数组未声明. 下文中, 对于程序状态  $\sigma = (s, \theta)$ 、整型变量  $x$ 、数组变量  $X$ 、整数  $z$ 、整数列表  $zs$ , 用  $\sigma x$  表示  $s x$ , 用  $\sigma X$  表示  $\theta X$ , 用  $\sigma[x \mapsto z]$  表示  $s[x \mapsto z]$ , 用  $\sigma[X \mapsto zs]$  表示  $\theta[X \mapsto zs]$ .

具体化初始格局集合  $C := (GCom \cup Stmt) \times \Sigma$ , 终止格局集合  $T := \Sigma$ . 在此基础上, 使用图 4 中定义的函数  $h$  表达  $GC_{arr}$  语言的大步操作语义. 其中,  $\mathcal{A}$  和  $\mathcal{B}$  分别为算术表达式和布尔表达式的求值函数. 这两个函数对表达式的求值结果均可基于子表达式的求值结果直接给出, 其具体定义参见附录.

图中表达式  $hf$  的直观意义与图 2 中类似: 若  $f$  对每个初始格局给出使用限定高度的语义推导树所能导出的终止格局集合, 则  $hf$  对每个初始格局给出使用限高增加 1 的语义推导树所能导出的终止格局集合. 基于图 4 中函数  $h$  的定义, 读者应能很容易地得到  $GC_{arr}$  语言大步操作语义规则的纸笔定义形式.

$$\begin{aligned}
hf(skip, \sigma) &:= \{ \sigma \} \\
hf(\text{var } x := nc, \sigma) &:= \{ \sigma[x \mapsto z] \mid \sigma x = \perp \wedge \mathcal{A}[nc] \sigma = z \wedge z \in \mathbb{Z} \} \\
hf(\text{arr } X := ns, \sigma) &:= \{ \sigma[X \mapsto zs] \mid \sigma X = \perp \wedge \exists nc_1, \dots, nc_k: ns = [nc_1, \dots, nc_k] \wedge \exists z_1, \dots, z_k: zs = [z_1, \dots, z_k] \wedge \\
&\quad \forall j \in \{1, \dots, k\}: \mathcal{A}[nc_j] \sigma = z_j \wedge z_j \in \mathbb{Z} \} \\
hf(x := a, \sigma) &:= \{ \sigma[x \mapsto z] \mid \sigma x \neq \perp \wedge \mathcal{A}[a] \sigma = z \wedge z \in \mathbb{Z} \} \\
hf(X[a_1] := a_2, \sigma) &:= \{ \sigma[X \mapsto zs] \mid \exists z_1, z_2 \in \mathbb{Z}: \exists zs \in \mathbb{Z}^*: \mathcal{A}[a_1] \sigma = z_1 \wedge \mathcal{A}[a_2] \sigma = z_2 \wedge \sigma X = zs \wedge \\
&\quad 0 \leq z_1 < |zs| \wedge zs' = zs[z_1 \mapsto z_2] \} \\
hf(\text{if } G \text{ fi}, \sigma) &:= f(G, \sigma) \\
hf(\text{do } G \text{ od}, \sigma) &:= \{ \sigma' \mid \exists \sigma'': \sigma'' \in f(G, \sigma) \wedge \sigma' \in f(\text{do } G \text{ od}, \sigma'') \} \\
hf(S_1; S_2, \sigma) &:= \{ \sigma' \mid \exists \sigma'': \sigma'' \in f(S_1, \sigma) \wedge \sigma' \in f(S_2, \sigma'') \} \\
hf(b \rightarrow S, \sigma) &:= \{ \sigma' \mid \mathcal{B}[b] \sigma = tt \wedge \sigma' \in f(S, \sigma) \} \\
hf(G_1 \square G_2, \sigma) &:= f(G_1, \sigma) \cup f(G_2, \sigma)
\end{aligned}$$

图 4 形式化卫式命令语言  $GC_{arr}$  大步语义的函数  $h$

图 4 中, 关于整型变量声明  $\text{var } x := nc$  的定义式表示, 所声明变量  $x$  在声明前不存在 ( $\sigma x = \perp$ ), 而声明后  $x$  的值初始化为  $nc$  的整数值. 关于数组变量声明  $\text{arr } X := ns$  的定义式表示, 所声明数组变量  $X$  在声明前不存在 ( $\sigma X = \perp$ ), 而声明后  $X$  的值初始化为数值常量列表  $ns$  所对应的整数列表. 关于整型变量、数组元素赋值的定义式要求被赋值的整型变量或数组变量在赋值前已经存在 (被声明). 关于数组元素赋值  $X[a_1] := a_2$  的定义式使用记号  $zs[z_1 \mapsto z_2]$  表示作为数组变量值的整数列表的更新. 关于条件分支语句  $\text{if } G \text{ fi}$  的定义式表示执行该分支语句的任何结果由其中的卫式命令  $G$  产生. 关于循环语句  $\text{do } G \text{ od}$  的定义式表示, 循环语句的执行结果通过先执行其中的卫式命令  $G$ , 再在所到达状态下继续执行该循环得到. 关于卫式命令  $b \rightarrow S$  的定义式表示, 该卫式命令的执行结果当  $b$  成立时即为  $S$  的执行结果, 当  $b$  不成立时为空集. 关于卫式命令  $G_1 \square G_2$  的定义式表示, 该卫式命令的执行结果包括两个子命令的执行结果. 该定义式导致对某个初始格局  $c, h^k \perp c$  的结果可以包含多个终止格局.

**引理 3.** 图 4 中定义的函数  $h$  为单调函数.

本引理的证明使用与 While 语言中对应引理 (引理 2) 的证明相同的策略, 证明过程简单机械. 在本引理基础上, 即可使用定理 1 对  $GC_{arr}$  语言程序进行可靠证明.

## 5.2 数组极大元素查找程序的规约和证明

考虑下列卫式命令语言程序  $S_{\max}(ns)$ , 其中  $ns$  是任何整型常量列表. 该程序首先声明整型变量  $k$  和  $j$ 、数组变量  $A$ , 并初始化为 0、1、 $ns$ . 而后在一个循环中使用  $j$  作为下标遍历数组  $A$ , 在循环体中通过卫式命令对  $A[j]$  和  $A[k]$  进行比较, 并相应更新  $j$ 、 $k$  的值. 此处的比较运算符  $\leq$  (小于或等于) 视为由  $GC_{arr}$  语法中其他比较、逻辑运算符所实现的语法糖. 该程序最终完成的功能是找到  $ns$  中某个极大元素, 并将其下标记录在  $k$  中. 由于卫式命令引发的非确定性, 当  $ns$  中含有多个相同元素时, 循环结束后  $k$  的可能取值不唯一.

```

var k := 0; var j := 1; arr A := ns;
do j < |ns| →
  if A[j] ≤ A[k] → j := j+1 □ A[k] ≤ A[j] → k:=j; j:=j+1 fi
od

```

以下通过形式规约描述该程序的功能正确性条件, 并提供关于循环的辅助信息. 其中,  $S_{\text{do}}(ns)$  表示  $S_{\max}(ns)$  中的循环语句.

$$\begin{aligned}
\Phi_{\max}(S_{\max}(ns), \sigma) &:= \{ \sigma' \mid 0 \leq \sigma' k < |ns| \wedge \forall i \in \{0, \dots, |ns|-1\}: \mathcal{A}[A[i]] \sigma' \leq \mathcal{A}[A[k]] \sigma' \} \\
\Phi_{\max}(S_{\text{do}}(ns), \sigma) &:= \{ \sigma' \mid \sigma' j = |ns| \wedge \sigma' k < \sigma' j \wedge \forall i \in \{0, \dots, |ns|-1\}: \mathcal{A}[A[i]] \sigma' \leq \mathcal{A}[A[k]] \sigma' \} \\
&\quad \text{若 } 0 \leq \sigma k < \sigma j \leq |ns| \wedge \forall i \in \{0, \dots, (\sigma j)-1\}: \mathcal{A}[A[i]] \sigma \leq \mathcal{A}[A[k]] \sigma \\
\Phi_{\max} c &:= \Sigma \quad \text{若 } c \text{ 不具有以上形式或不满足相应条件}
\end{aligned}$$

其中,  $\Phi_{\max}(S_{\max}(ns), \sigma)$  的定义式表达当程序  $S_{\max}(ns)$  终止后,  $k$  值在数组  $A$  的下标范围内, 且数组的任何元素均不大于下标为  $k$  的元素. 关于循环  $S_{\text{do}}(ns)$  的辅助信息由  $\Phi_{\max}(S_{\text{do}}(ns), \sigma)$  的定义式给出. 假设某轮循环开始时, 对当时状态  $\sigma$  有  $0 \leq \sigma k < \sigma j \leq |ns| \wedge \forall i \in \{0, \dots, (\sigma j) - 1\}: \mathcal{A}[[A[i]]]\sigma \leq \mathcal{A}[[A[k]]]\sigma$ , 那么若循环能够终止, 则循环终止时  $j$  值为  $ns$  的长度, 即为数组长度,  $k$  值小于  $j$  值, 且数组的任何元素均不大于下标为  $k$  的元素.

以下给出  $\Phi_{\max}$  的正确性结果及其证明思路. 详细证明过程包含在本工作的 Coq 形式化中.

**定理 3.** 数组极大值查找程序的正确性.  $\models \Phi_{\max}$  成立.

本定理的证明中, 使用定理 1 将对  $\models \Phi_{\max}$  的证明任务转化为对  $\vdash \Phi_{\max}$  的证明任务, 而后者通过检查  $(F \Phi_{\max})^n \perp c$  包含于  $\Phi_{\max} c$  加以完成, 无需为  $S_{\max}(ns)$  中存在的递归调用设置归纳证明步骤. 当  $c$  中命令不具有  $S_{\max}(ns)$  或  $S_{\text{do}}(ns)$  形式, 或者  $c$  中命令形如  $S_{\text{do}}(ns)$  但  $c$  中状态  $\sigma$  不满足  $0 \leq \sigma k < \sigma j \leq |ns| \wedge \forall i \in \{0, \dots, \sigma j - 1\}: \mathcal{A}[[A[i]]]\sigma \leq \mathcal{A}[[A[k]]]\sigma$  时, 由于  $\Phi_{\max} c$  为所有可能终止格局的集合  $T = \Sigma$ , 故  $(F \Phi_{\max})^n \perp c$  必定包含于  $\Phi_{\max} c$ . 否则, 若  $c$  中命令为  $S_{\max}(ns)$ , 则对  $(F \Phi_{\max})^n \perp c$  进行展开和化简, 在该过程中使用辅助信息  $\Phi_{\max}(S_{\text{do}}(ns), \sigma)$ , 可在有限步后得到一结果集, 包含于  $\Phi_{\max} c$ . 若  $c$  中命令为  $S_{\text{do}}(ns)$ , 同样对  $(F \Phi_{\max})^n \perp c$  进行展开和化简, 在该过程中使用辅助信息  $\Phi_{\max}(S_{\text{do}}(ns), \sigma)$ , 可在有限步后得到一结果集, 包含于  $\Phi_{\max} c$ . 以上的分类讨论即可完成定理的证明.

定理 1 在本定理的证明中保障了基于化简的证明策略的可靠性. 定理 1 无需对  $\text{GC}_{\text{arr}}$  语言单独证明, 而是在确认  $\text{GC}_{\text{arr}}$  语言语义函数  $h$  单调性的前提下即可直接使用. 因此, 在给出  $\text{GC}_{\text{arr}}$  语言的操作语义后, 无需单独为该语言中循环结构的可靠推理提供相应的基础设施.

## 6 函数式语言程序证明实例

本节考虑一个可对列表进行操作的函数式语言  $\text{Fun}_{\text{lst}}$ , 通过定义函数  $h \in D \rightarrow D$  (其中  $D = C \rightarrow \mathcal{P}(T)$ ) 对该语言的大步操作语义进行形式化, 并使用定理 1 将对两个有序列表合并为单个有序列表的程序进行正确性证明.

### 6.1 函数式语言 $\text{Fun}_{\text{lst}}$ 大步语义的形式化

语言  $\text{Fun}_{\text{lst}}$  为文献 [30] 中所讨论函数式语言的一个子语言, 其程序为一个表达式  $e \in \text{Expr}$ , 抽象语法为:

$$e ::= nc \mid \text{true} \mid \text{false} \mid e + e \mid e - e \mid e * e \mid e / e \mid e = e \mid e < e \mid !e \mid e \&\& e \mid \text{if } e \text{ then } e \text{ else } e \mid \\ \text{nil} \mid e :: e \mid \text{listcase } e \text{ of } (e, e) \mid x \mid e e \mid \text{fn } x: e \mid \text{recf } x = \text{fn } x': e' \text{ in } e$$

这里  $nc$  为数值常量,  $\text{nil}$  为空列表,  $e_1 :: e_2$  为在列表  $e_2$  前添加元素  $e_1$  形成的列表. 表达式  $\text{listcase } e_1 \text{ of } (e_2, e_3)$  判断列表  $e_1$  是否为空: 若为空, 则以表达式  $e_2$  的结果作为最终结果; 否则以  $e_3$  依次应用在  $e_1$  表头和表尾所形成表达式结果作为最终结果. 此外,  $x$  为变量,  $e_1 e_2$  为函数  $e_1$  在  $e_2$  上的应用,  $\text{fn } x: e$  为关于变量  $x$  的函数,  $\text{recf } x = \text{fn } x': e' \text{ in } e$  在表达式  $e$  范围内定义为  $x$  的函数  $\text{fn } x': e'$ , 使得在  $e'$  中能够对该函数进行递归调用.

具体化初始格局集合  $C := \text{Expr}$ 、终止格局集合  $T := \text{Expr}$ . 在此基础上, 使用图 5 中定义的函数  $h$  形式化函数式语言  $\text{Fun}_{\text{lst}}$  的大步操作语义. 其中, 对数值常量  $nc$ , 用  $nc$  表示  $nc$  对应的整数值; 对布尔常量  $bc \in \{\text{true}, \text{false}\}$ , 用  $bc$  表示  $bc$  所对应的逻辑真值; 对二元算术或比较运算符  $op$ , 用  $op$  表示  $op$  对应的数学运算; 对整数  $z$ , 用  $z^\#$  表示  $z$  所对应的数值常量.

图 5 中表达式  $hf$  的直观意义与图 2 中类似: 若  $f$  对每个初始格局给出使用限定高度的语义推导树所能导出的终止格局集合, 则  $hf$  对每个初始格局给出使用限高增加 1 的语义推导树所能导出的终止格局集合. 基于图 5 中函数  $h$  的定义, 读者应能很容易地得到  $\text{Fun}_{\text{lst}}$  语言大步操作语义规则的纸笔定义形式.

对图 5 中较为复杂的定义式解释如下. 关于表达式  $\text{listcase } e \text{ of } (e_1, e_2)$  的定义式指出, 若表达式  $e$  的求值结果为  $\text{nil}$ , 则表达式  $\text{listcase } e \text{ of } (e_1, e_2)$  的求值结果即为表达式  $e_1$  的求值结果; 若表达式  $e$  的求值结果具有形式  $e_h :: e_t$ , 则表达式  $e$  的求值结果与表达式  $(e_2 e_h e_t)$  的求值结果一致, 而后者是  $e_2$  在  $e_h$  和  $e_t$  上的依次应用. 关于表达式  $e_1 e_2$  的定义分句指出, 对该表达式进行求值, 首先将  $e_1$  求值为某个函数  $\text{fn } x: e'_1$ , 将  $e_2$  求值为某个表达式  $e'_2$ , 并将表达式  $e'_1[e'_2/x]$  的求值结果作为原表达式  $e_1 e_2$  的求值结果. 其中, 表达式  $e'_1[e'_2/x]$  是在  $e'_1$  中将自由变量  $x$  替换为  $e'_2$  的结果. 条件  $\text{fv}(e'_2) = \emptyset$  要求  $e'_2$  无自由变量, 从而简化替换的定义 (见附录). 事实上, 由于该函数式语言的程序为无

自由变量的表达式<sup>[30]</sup>,  $e_2'$  无自由变量的条件并不对规则的使用造成实质性限制. 关于表达式  $\text{recf } x_1 = \text{fn } x_2 : e_2 \text{ in } e_1$  的分句指出, 该表达式的求值结果为表达式  $(\text{fn } x_1 : e_1) (\text{fn } x_2 : \text{recf } x_1 = (\text{fn } x_2 : e_2) \text{ in } e_2)$  的求值结果. 后者可视为将函数  $\text{fn } x_1 : e_1$  应用于函数  $\text{fn } x_2 : e_2$ , 只是在  $e_2$  范围内再次引入关于  $x_1$  和函数  $\text{fn } x_2 : e_2$  的绑定, 使名为  $x_1$  的函数  $\text{fn } x_2 : e_2$  可在  $e_2$  中递归调用. 为简单起见,  $\text{Fun}_{\text{lst}}$  未提供同时定义多个相互递归函数的语法.

$$\begin{aligned}
hfnc &:= \{ nc \} \\
hfbc &:= \{ bc \} && (bc \in \{\text{true}, \text{false}\}) \\
hf(e, op e_2) &:= \{ (op(z_1, z_2)) \# \mid nc_1 \in fe_1 \wedge nc_2 \in fe_2 \wedge \underline{nc}_1 \in \mathbb{Z} \wedge \underline{nc}_2 \in \mathbb{Z} \} \quad (op \in \{+, -, *, /, =, <\}) \\
hf(!e) &:= \{ (\neg bc) \# \mid bc \in fe \wedge bc \in \{\text{true}, \text{false}\} \} \\
hf(e_1 \&\& e_2) &:= \{ (b_1 \wedge b_2) \# \mid bc_1 \in fe_1 \wedge bc_2 \in fe_2 \wedge bc_1, bc_2 \in \{\text{true}, \text{false}\} \} \\
hf(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &:= \{ e \mid \text{true} \in fe_1 \wedge e \in fe_2 \vee \text{false} \in fe_1 \wedge e \in fe_3 \} \\
hf(\text{nil}) &:= \{ \text{nil} \} \\
hf(e_1 :: e_2) &:= \{ e' \mid e'_1 \in fe_1 \wedge e'_2 \in fe_2 \} \\
hf(\text{listcase } e \text{ of } (e_1, e_2)) &:= \{ e' \mid \text{nil} \in fe \wedge e' \in fe_1 \vee \exists eh, et : eh : et \in fe \wedge e' \in fe_2 \text{ eh } et \} \\
hfx &:= \emptyset \\
hf(e_1 e_2) &:= \{ e' \mid \exists x, e'_1, e'_2 : (\text{fn } x : e'_1) \in fe_1 \wedge e'_2 \in fe_2 \wedge \text{fn}(e'_2) = \emptyset \wedge e' \in f(e'_1[e'_2/x]) \} \\
hf(\text{fn } x : e) &:= \{ \text{fn } x : e \} \\
hf(\text{recf } x_1 = \text{fn } x_2 : e_2 \text{ in } e_1) &:= \{ e' \mid x_1 \neq x_2 \wedge e' \in f((\text{fn } x_1 : e_1) (\text{fn } x_2 : \text{recf } x_1 = (\text{fn } x_2 : e_2) \text{ in } e_2)) \}
\end{aligned}$$

图 5 形式化函数式语言  $\text{Fun}_{\text{lst}}$  大步语义的函数  $h$

**引理 4.** 图 5 中定义的函数  $h$  为单调函数.

本引理的证明使用与 **While** 语言中对应引理 (引理 2) 的证明相同的策略, 证明过程简单机械. 在本引理基础上, 即可使用定理 1 对  $\text{Fun}_{\text{lst}}$  语言程序进行可靠证明.

## 6.2 有序列表合并程序的规约和证明

考虑图 6 中的  $\text{Fun}_{\text{lst}}$  语言程序, 将该程序记作  $e_{\text{mg}}(l_1, l_2)$ , 其中  $l_1$  和  $l_2$  为表示列表的参量. 对任意两个有序整数列表  $l_1$  和  $l_2$ , 该程序将两个列表合并为单个有序列表. 具体而言, 名为 **merge** 的函数应用时, 两个给定列表绑定到变量  $x$  和  $x'$ , 通过两个 **listcase** 表达式, 若两个列表至少有一个为空表 **nil**, 则未判断为空表的列表作为合并结果返回. 否则, 两个列表的头分别绑定到变量  $i$  和  $i'$ , 两个列表的尾分别绑定到变量  $r$  和  $r'$ . 若  $i$  的绑定值小于等于  $i'$  的绑定值, 则结果列表的头为  $i$ , 尾为递归调用 **merge**  $r x'$  的结果 (其中  $x'$  为第 2 个列表), 否则, 结果列表的头为  $i'$ , 尾为递归调用 **merge**  $x r$  的结果 (其中  $x$  为第 1 个列表).

$$\begin{aligned}
\text{recf merge} &= (\text{fn } x : \text{fn } x' : \\
&\quad \text{listcase } x \text{ of } (x', \\
&\quad \quad \text{fn } i : \text{fn } r : \text{listcase } x' \text{ of } (x, \\
&\quad \quad \quad \text{fn } i' : \text{fn } r' : (\text{if } i \leq i' \text{ then } i :: \text{merge } r x' \text{ else } i' :: \text{merge } x r))) \\
&\text{in merge } l_1 l_2
\end{aligned}$$

图 6 以  $\text{Fun}_{\text{lst}}$  语言编写的有序列表合并程序

对一个  $\text{Fun}_{\text{lst}}$  语言表达式  $e$ , 定义  $(|e|)$  为  $e$  对应的整数列表. 若没有整数列表与  $e$  对应, 则使  $(|e|)$  为未定义值 **undef**. 故  $(|\text{nil}|) = []$ ; 若有整数  $z$  使得  $e_1 = z^\#$ , 且  $(|e_2|) \in \mathbb{Z}^*$ , 则  $(|e_1 :: e_2|) := (z :: (|e_2|))$ , 否则  $(|e_1 :: e_2|) := \text{undef}$ ; 若  $e$  不具有形式 **nil** 或  $e_1 :: e_2$ , 则  $(|e|) := \text{undef}$ . 此外, 令  $\text{occ } zs \ z$  表示整数  $z$  在整数列表  $zs$  中出现的次数, 令  $\text{sorted } zs$  表示整数列表  $zs$  中元素按升序排列.

对程序  $e_{\text{mg}}(l_1, l_2)$  给出如下的形式规约.

$$\begin{aligned}
\Phi_{\text{mg}} e_{\text{mg}}(l_1, l_2) &:= \{ l \mid (|l|) \in \mathbb{Z}^* \wedge (\forall z : \text{occ } (|l|) \ z = \text{occ } (|l_1|) \ z + \text{occ } (|l_2|) \ z) \wedge \text{sorted } (|l|) \} \\
&\quad \text{若有 } (|l_1|) \in \mathbb{Z}^* \wedge (|l_2|) \in \mathbb{Z}^* \wedge \text{sorted } (|l_1|) \wedge \text{sorted } (|l_2|) \\
\Phi_{\text{mg}} e_{\text{unfold}}(l_1, l_2) &:= \{ l \mid (|l|) \in \mathbb{Z}^* \wedge (\forall z : \text{occ } (|l|) \ z = \text{occ } (|l_1|) \ z + \text{occ } (|l_2|) \ z) \wedge \text{sorted } (|l|) \} \\
&\quad \text{若有 } (|l_1|) \in \mathbb{Z}^* \wedge (|l_2|) \in \mathbb{Z}^* \wedge \text{sorted } (|l_1|) \wedge \text{sorted } (|l_2|)
\end{aligned}$$

上述形式规约中前两行要求只要  $l_1$  和  $l_2$  均表示升序有序的整数列表, 那么  $e_{\text{mg}}(l_1, l_2)$  的执行结果亦为升序有序的整数列表, 且该结果中每个整数的出现次数等于该整数在  $l_1$  和  $l_2$  中出现次数的和, 这反映出  $\perp$  确实为  $l_1$  和  $l_2$  的有序合并结果. 上述形式规约中的后两行针对表达式  $e_{\text{unfold}}(l_1, l_2) := (\text{fn } x: \text{recf merge} = \text{fn } x': \text{fn } x'': e_{\text{ics}} \text{ in fn } x': e_{\text{ics}})$   $l_1 l_2$ , 其中  $e_{\text{ics}}$  为图 6 中 2-4 行的外层 listcase 表达式. 此两行实为帮助进行程序证明的辅助信息.

直观上, 表达式  $e_{\text{mg}}(l_1, l_2)$  的执行涉及函数 merge 的递归调用, 故似乎应对任何  $l_1, l_2$ , 为表达式 merge  $l_1 l_2$  提供辅助信息, 以帮助验证. 然而, 表达式 merge  $l_1 l_2$  缺乏关于 merge 所指代函数的信息, 无法进行求值. 事实上, 表达式  $e_{\text{mg}}(l_1, l_2)$  的执行会产生形如  $e_{\text{unfold}}(l_a, l_b)$  的展开形式, 而后的执行亦会产生其自身的形式. 表达式  $e_{\text{unfold}}(l_a, l_b)$  实际执行名为 merge 的函数在每次递归调用中的计算. 在建立  $\vdash \Phi_{\text{mg}}$  的证明过程中, 对  $(F \Phi_{\text{mg}})^n \perp$  进行化简, 遇到形如  $e_{\text{unfold}}(l_a, l_b)$  的表达式时,  $\Phi_{\text{mg}}$  定义的后两行允许直接利用  $\Phi_{\text{mg}}$  中关于  $e_{\text{unfold}}(l_a, l_b)$  的信息给出其可能的结果范围, 避免进一步对该表达式进行符号执行 (该执行可能无法终止). 为利用  $\Phi_{\text{mg}}$  中关于  $e_{\text{unfold}}(l_a, l_b)$  的信息而不破坏程序证明的可靠性, 该信息本身亦须得到验证, 体现为在建立  $\vdash \Phi_{\text{mg}}$  的过程中, 对条件  $\forall n: (F \Phi_{\text{mg}})^n \perp e_{\text{unfold}}(l_a, l_b) \subseteq \Phi_{\text{mg}} e_{\text{unfold}}(l_a, l_b)$  的证明.

以下直接给出  $\Phi_{\text{mg}}$  的正确性结果. 详细证明过程包含在本工作的 Coq 形式化中.

**定理 4.** 列表合并程序的正确性.  $\vdash \Phi_{\text{mg}}$  成立.

本定理的证明中, 使用定理 1 将对  $\vdash \Phi_{\text{mg}}$  的证明任务转化为对  $\vdash \Phi_{\text{mg}}$  的证明任务, 而后者通过  $(F \Phi_{\text{mg}})^n \perp c$  的展开和化简, 以及对结果的检查加以完成, 无需关于  $e_{\text{mg}}(l_1, l_2)$  中存在的递归调用设置归纳证明步骤. 定理 1 有效支撑了在提供函数 merge 相关信息的前提下对有序表合并程序的可靠证明. 该定理无需对 Fun<sub>lst</sub> 语言单独证明, 而是在确认 Fun<sub>lst</sub> 语言语义函数  $h$  单调性的前提下即可直接使用. 在一定程度上,  $e_{\text{unfold}}(l_a, l_b)$  的形式规约类似于命令式语言程序验证中所使用的函数契约 (function contracts)——尽管函数契约往往需要同时反映程序执行的结果和副作用 (如内存状态的变化等), 而本例中的形式规约只反映计算结果.

## 7 关于证明技术的完备性

本文第 3 节中, 给出了一种适用于循环和递归程序结构推理的通用证明技术, 并且证明了该技术的可靠性. 此后在第 4-6 节中, 通过程序验证实例初步体现了该技术在处理不同范式程序语言、不同程序结构方面的适用性. 然而, 若要系统性考察证明技术的适用范围, 则须考虑其完备性——是否对任意的形式规约  $\Phi$ , 若该规约得到满足 ( $\models \Phi$ ), 则该规约能够用该技术证明 ( $\vdash \Phi$ ). 不难发现, 这一命题并不成立. 对第 4 节中的阶乘计算程序  $S_{\text{fac}}$ , 若形式规约  $\Phi_{\text{fac}}$  关于  $(S_{\text{wh}}, \sigma)$  的子句改为  $\Phi_{\text{fac}}(S_{\text{wh}}, \sigma) = \Sigma$ , 所得到的形式规约仍然得到满足 (即有  $\models \Phi_{\text{fac}}$ ), 但该形式规约无法使用本文所提出技术进行证明. 一般地, 若要使形式规约可被证明, 其中需要包含必要辅助信息.

仍然考虑  $D := C \rightarrow \mathcal{P}(T)$ . 对两个形式规约  $\Phi_1, \Phi_2 \in D$ , 定义关系  $\leq$ , 使得  $\Phi_1 \leq \Phi_2$  当且仅当  $\Phi_1 c \supseteq \Phi_2 c$ . 直观上,  $\Phi_1 \leq \Phi_2$  表示  $\Phi_2$  关于目标程序执行结果提供相对更精确的信息. 在此基础上可以证明, 若表达程序语言语义的函数  $h$  不仅单调, 而且满足一种连续性性质, 则对任一得到满足的形式规约  $\Phi$ , 存在提供更精确信息的形式规约  $\Phi'$ , 使得  $\Phi'$  能够使用本文所提出技术加以证明. 事实上, 一个可用的  $\Phi'$  即为将每个初始格局映射为所有从该初始格局出发对程序进行具体执行所产生结果集合的形式规约.

**定义 4.** 连续函数. 若函数  $f \in D \rightarrow D$  单调, 且  $f$  对  $D$  中任一无穷递增序列  $\{d_n\}_{n \in \mathbb{N}}$  (即  $d_0 \leq d_1 \leq d_2 \leq \dots$ ) 满足  $(\lambda c. \bigcup_n (d_n c)) = \lambda c. \bigcup_n (f d_n c)$ , 则称  $f$  为  $D$  上的连续函数.

与已有工作 (如文献 [3,30]) 中较为常见的连续性定义相比, 上述定义不要求无穷递增序列具有最小上界.

定义形式规约  $\Phi^\dagger := \lambda c. \bigcup_n (h^n \perp c)$ , 即  $\Phi^\dagger = \lambda c. \{t \mid \exists n: t \in h^n \perp c\}$ . 对任一初始格局  $c$ ,  $\Phi^\dagger c$  为由  $c$  开始进行程序执行, 终止后可得到的所有结果的集合 (对于确定性语言, 该集合为单元集或空集). 若表达程序语义的函数  $h$  为定义 4 意义上的连续函数, 则形式规约  $\Phi^\dagger$  可由本文所提出技术得到证明, 见下列引理.

**引理 5.** 若  $h \in D \rightarrow D$  为连续函数, 则有  $\vdash \Phi^\dagger$ .

证明: 以下证明对任意自然数  $n$ , 有  $(F \Phi^\dagger)^n \perp \leq \Phi^\dagger$ .

若  $n=0$ , 则  $(F \Phi^\dagger)^0 \perp = \perp$ , 显然有  $(F \Phi^\dagger)^0 \perp \leq \Phi^\dagger$ .

以下假设  $n > 0$ . 那么存在  $k \geq 0$ , 使得  $n = k + 1$ . 进行如下推导:

$$\begin{aligned}
(F\Phi^\dagger)^n \perp &= F\Phi^\dagger((F\Phi^\dagger)^k \perp) && \text{(由 } n=k+1\text{)} \\
&= h(((F\Phi^\dagger)^k \perp) \oplus \Phi^\dagger) && \text{(由 } F\text{ 定义)} \\
&= h(\lambda c. \text{ if } \Phi^\dagger c = T \text{ then } ((F\Phi^\dagger)^k \perp) c \text{ else } \Phi^\dagger c) && \text{(由 } \oplus\text{ 定义)} \\
&\leq h(\lambda c. \text{ if } \Phi^\dagger c = T \text{ then } \Phi^\dagger c \text{ else } \Phi^\dagger c) && \text{(若 } \Phi^\dagger c = T, \text{ 显然有 } ((F\Phi^\dagger)^k \perp) c \subseteq \Phi^\dagger c\text{)} \\
&= h\Phi^\dagger \\
&= h\lambda c. \bigcup_n (h^n \perp c) && \text{(由 } \Phi^\dagger\text{ 定义)} \\
&= \lambda c. \bigcup_n (h(h^n \perp) c) && \text{(由 } h\text{ 连续、}\{h^n \perp\}_n\text{ 为 } D\text{ 中无穷递增序列)} \\
&= \lambda c. \bigcup_n ((h^{n+1} \perp) c) \\
&\leq \lambda c. \bigcup_n (h^n \perp c) \\
&= \Phi^\dagger && \text{证毕.}
\end{aligned}$$

**定理 5.** 验证技术的相对完备性. 假设  $h \in D \rightarrow D$  为连续函数. 对任一形式规约  $\Phi$ , 若  $\vdash \Phi$  成立, 则存在规约  $\Phi'$ , 使得  $\Phi \leq \Phi'$ , 且  $\vdash \Phi'$  可被建立.

证明: 以下证明, 可以使用  $\Phi^\dagger$  作为满足定理所要求条件的  $\Phi'$ .

首先证明  $\Phi \leq \Phi^\dagger$ . 由  $\vdash \Phi$  可得,  $\forall n: h^n \perp \leq \Phi$ , 故  $\forall n, c: h^n \perp c \subseteq \Phi c$ . 因此有  $\forall c: \bigcup_n (h^n \perp c) \subseteq \Phi c$ , 亦即  $\forall c: \Phi^\dagger c \subseteq \Phi c$ . 故  $\Phi \leq \Phi^\dagger$  成立. 另在本定理的条件下, 使用引理 4 可建立  $\vdash \Phi^\dagger$ . 证毕.

本定理及其证明并未说明对于一个得到满足的形式规约  $\Phi$ , 若  $\Phi$  无法得到证明, 需对  $\Phi$  所提供信息进行精确化(使之能够得到证明)的最低限度. 直观上, 所提供的辅助信息应能帮助对可能产生无界行为(unbounded behavior)的程序结构进行推理, 如循环结构、递归函数等, 但通常无需提供进一步的信息, 这是因为若不考虑效率问题, 其他程序结构的推理可借助符号执行加以完成.

最后, 不难证明用来形式化 While 语言、GC<sub>arr</sub> 语言、Fun<sub>lst</sub> 语言大步操作语义的函数  $h$  均为连续函数.

**定理 6.** 实例语言函数  $h$  的连续性. 图 2、图 4、图 5 中定义的函数  $h$  均为连续函数.

不难证明, 只要  $h$  为单调函数, 则有  $\lambda c. \bigcup_n (h d_n c) \leq h(\lambda c. \bigcup_n (d_n c))$ . 故对于图 2、图 4、图 5 中的每一个具体语义函数  $h$ , 只需对  $D$  中任一无穷递增序列  $\{d_n\}_{n \in \mathbb{N}}$  (即  $d_0 \leq d_1 \leq d_2 \leq \dots$ ) 证明  $h(\lambda c. \bigcup_n (d_n c)) \leq \lambda c. \bigcup_n (h d_n c)$ , 即可证明  $h$  在定义 4 意义上的连续性. 定理 6 的具体证明过程参见 Coq 中的形式化代码.

定理 6 表明, 基于第 4-6 节中所定义的 While 语言、GC<sub>arr</sub> 语言、Fun<sub>lst</sub> 语言的语义, 获得了定理 5 意义上相对完备的验证过程. 值得注意的是, 定理 1 的成立, 以及第 4-6 节中具体实例验证结果并不依赖于  $h$  的连续性.

## 8 讨论

### 8.1 验证技术与抽象解释的直观联系

本文所提出技术与抽象解释方法<sup>[31]</sup>在直观意义上具有一定联系. 在第 3 节中, 引理 1 体现若形式规约  $\Phi$  通过验证, 则  $(F\Phi)^n \perp$  是  $h^n \perp$  的上近似. 因而  $(F\Phi)^n \perp$  可视为关于  $h^n \perp$  可靠的抽象计算函数. 一般而言, 抽象解释侧重对每类语法结构导出其抽象计算函数, 以支撑自动程序分析. 相较之下, 本文所提出技术对每个具体程序, 给出用户所提供形式规约  $\Phi$  诱导的抽象计算函数, 以支撑较复杂正确性性质的可靠演绎验证. 引理 1 和定理 1 的成立不依赖于  $h$  的具体定义, 这体现出证明技术的语言无关性. 而抽象计算函数  $(F\Phi)^n \perp$  以形式规约  $\Phi$  为参数则体现出验证同一语言的不同程序时可引入不同的关键辅助信息.

抽象解释是极具表达力和灵活性的形式化程序分析方法. 一般而言, 形式化方法不同领域的技术可存在深层次的关联(如文献[32,33]等所给出结果). 但本文并不试图探究语言无关的演绎验证和抽象解释的深层联系.

### 8.2 与基于小步语义的同类验证技术的比较

相较于小步语义, 大步语义常可省去语义格局中的复杂控制信息, 如函数调用栈等. 语言无关的程序证明技术建立在语义格局这一抽象概念上, 无法将程序规约信息标注在具体程序的语法结构中. 故若语义格局中存在调用

栈等控制结构, 则使用语言无关的程序证明技术往往需要在形式规约中引入这些控制结构. 因此, 基于大步语义的语言无关程序验证技术有利于简化在程序证明过程中关于语义格局所需考虑的信息.

对某种程序语言, 一个经验性结论是若该语言具有大步语义, 则亦应能定义其小步语义, 并证明其与大步语义等价. 故若某种程序语言已有大步语义, 则可尝试定义其小步语义, 完成等价性证明, 并使用基于小步语义的语言无关程序证明方法验证该语言程序. 然而, 对于实用程序语言, 小步语义的定义和形式化 (如文献 [29,34]), 以及语义等价性证明 (如文献 [27]) 均是相当繁琐的工作. 而本工作使得直接基于大步语义进行可靠程序证明成为可能.

### 8.3 与基于指称语义的程序证明的异同

在指称语义中, 显式刻画单调函数迭代的不动点, 本文所提出的验证技术亦涉及单调函数的迭代, 但该技术与基于指称语义的程序验证技术亦有如下重要区别.

本文中迭代的函数  $h$  虽具有函数形式, 但反映的是大步操作语义的语义规则, 其迭代结果直接反映大步语义中使用高度在某个上界之内的推导树能够导出的执行结果. 和使用大步操作语义进行程序证明类似, 使用本技术时在语义定义和程序证明中无需考虑不动点概念. 相较于基于大步操作语义的直接证明方式, 本技术提供的额外价值是将不同程序语言所需要的归纳证明结构简化为无需归纳的通用验证过程, 并给出统一的可靠性证明 (定理 1). 使用函数对大步操作语义的规则进行刻画, 一个优势在于能够借助函数的化简和符号求值完成验证技术自身性质的证明以及各种程序语言中目标程序的验证.

在指称语义中, 需要显式刻画其最小不动点的函数不同于本文中的函数  $h$  (参见本文中  $h$  定义的实例), 无论是定义某个程序语言的指称语义还是使用其指称语义进行程序证明均涉及对最小不动点及其性质的显式运用. 与基于操作语义的技术相比, 其概念复杂度相对较高.

### 8.4 本文技术的局限性

本文所提出技术适用于许多主流辅助证明工具 (proof assistant), 如 Coq、Isabelle/HOL、HOL4 等. 然而其程序规约和验证条件中所涉及的无穷集合、逻辑量词等特性使其难以直接借助一般编程语言实现为自动化程序验证工具. 文献 [11,12] 所提出技术与本文技术在这一方面具有类似特点.

本文所提出技术适用于部分正确性条件<sup>[17]</sup>的证明, 在此点上类似于文献 [11,12]. 考虑部分正确性条件意味着对于无法终止的程序, 形式规约中所描述的后条件不必得到满足. 尽管如此, 由于程序的终止性或执行时间上界等性质往往可单独证明等原因, 程序演绎验证方面许多工作仍将部分正确性作为证明目标. 语言无关的完全正确性 (total correctness) 证明需要进一步工作加以支撑, 这是本工作的一个可能的扩展方向. 一种可能的思路是探究如何基于形式规约  $\Phi$  的一般结构给出程序的终止性证明技术, 将其与本文的部分正确性证明技术结合, 从而得到一种语言无关的完全正确性证明技术.

由于本文的验证技术基于程序的大步语义, 该技术较难用于并发程序验证, 除非在语义设计中将终止格局设置为程序所有可能执行路径前缀的集合, 借以反映程序执行的中间状态.

## 9 总 结

程序的演绎验证技术能够可靠验证程序的复杂性质, 然而该技术的使用具有较大难度. 难点之一在于, 对不同程序语言, 其演绎验证系统的可靠性需要单独进行证明. 语言无关的程序验证技术<sup>[11-13]</sup>提供以程序语言的形式语义为参数的通用验证过程, 其可靠性 (soundness) 证明亦可在不同程序语言间复用. 当前支持辅助证明工具的语言无关验证技术<sup>[11,12]</sup>均依托程序语言的小步操作语义进行设计. 若能设计基于大步操作语义 (即以大步语义为参数) 的验证过程, 则可扩展此类验证技术的适用范围, 并利用大步语义在语义格局、语义推导方面的结构特点简化部分程序的证明. 然而, 基于大步语义的通用验证过程的设计, 需要解决程序中循环语句、递归函数等子结构执行的表征问题, 使得关于子结构执行的辅助信息能够加以利用.

注意到任何程序语言的大步操作语义均可形式化为一个函数, 刻画其语义推导树或其子树的顶层构造. 以此作为切入点, 本文解决了依托大步语义为程序子结构的执行提供辅助信息的问题, 提出了基于大步语义的语言无关



程序验证技术. 该技术的核心是一个以程序语言的函数式大步操作语义为参数的验证过程以及该验证过程的可靠性定理. 对于某种具体的程序语言, 以函数形式提供其大步操作语义后, 即可直接得到对该语言具体化的程序验证过程及其可靠性定理. 将程序的演绎验证任务转化为在形式规约中辅助信息帮助下对程序的符号执行和对结果的检查. 形式规约中所需辅助信息仅为循环、递归函数等的不变式和契约——类似于使用语言特定的程序逻辑进行验证所需的辅助信息. 不同范式下确定性、非确定性程序语言中的程序验证实例初步体现了本文所提出技术的有效性. 在 Coq 辅助证明工具中, 所有理论结果和程序验证实例均得到了机械化验证, 而本文所提出技术亦可在 Isabelle/HOL、HOL4 等基于高阶逻辑的辅助证明工具中进行形式化.

本工作为基于辅助证明工具的语言无关程序验证工具的实现提供基础. 若要完整实现此类工具, 使之能够有效处理针对实用程序语言的验证任务, 仍需在核心定理的基础上集成额外组件, 以简化关于内存布局、对象等常见语言特性的推理, 并支持依托领域知识(如数学理论)的推理. 对此, 程序逻辑以及辅助证明工具定理库方面的研究成果将具有宝贵借鉴价值. 未来工作的一个重要方向即是探究如何集成对非跨语言共享特性的推理能力和基于领域知识的推理能力, 并通过进一步的实例程序验证来检验其推理能力的增强.

## References:

- [1] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. *Ruan Jian Xue Bao/Journal of Software*, 2019, 30(1): 33–61 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/5652.htm> [doi: 10.13328/j.cnki.jos.005652]
- [2] Harrison J, Urban J, Wiedijk F. History of interactive theorem proving. *Handbook of the History of Logic*, 2014, 9: 135–214. [doi: 10.1016/B978-0-444-51624-4.50004-6]
- [3] Nielson HR, Nielson F. *Semantics with Applications: An Appetizer*. London: Springer, 2007. [doi: 10.1007/978-1-84628-692-6]
- [4] Zhou CC, Zhan NJ. *Formal Semantics*. 2nd ed., Beijing: Science Press, 2017 (in Chinese).
- [5] McCarthy J. Towards a mathematical science of computation. In: *Proc. of the 1962 Information Processing Congress*. 1962. 21–28.
- [6] Kirchner F, Kosmatov N, Prevosto V, Signoles J, Yakobowski B. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 2015, 27(3): 573–609. [doi: 10.1007/s00165-014-0326-7]
- [7] Ahrendt W, Beckert B, Bubel R, Hähle R, Schmitt PH, Ulbrich M. *Deductive Software Verification—The KeY Book: From Theory to Practice*. Cham: Springer, 2016. [doi: 10.1007/978-3-319-49812-6]
- [8] Filliâtre JC, Paskevich A. Why3—Where programs meet provers. In: *Proc. of the 22nd European Symp. on Programming*. Rome: Springer, 2013. 125–128. [doi: 10.1007/978-3-642-37036-6\_8]
- [9] Appel AW. Verified software toolchain. In: *Proc. of the 20th European Symp. on Programming*. Saarbrücken: Springer, 2011. 1–17. [doi: 10.1007/978-3-642-19718-5\_1]
- [10] Jung R, Krebbers R, Jourdan JH, Bizjak A, Birkedal L, Dreyer D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 2018, 28: e20. [doi: 10.1017/S0956796818000151]
- [11] Moore JS. Inductive assertions and operational semantics. In: *Proc. of the 12th IFIP WG 10.5 Advanced Research Working Conf. on Correct Hardware Design and Verification Methods*. L’Aquila: Springer, 2003. 289–303. [doi: 10.1007/978-3-540-39724-3\_27]
- [12] Moore B, Peña L, Rosu G. Program verification by coinduction. In: *Proc. of the 27th European Symp. on Programming*. Thessaloniki: Springer, 2018. 589–618. [doi: 10.1007/978-3-319-89884-1\_21]
- [13] Stăfănescu A, Park D, Yuwen SJ, Li YL, Roşu G. Semantics-based program verifiers for all languages. In: *Proc. of the 2016 ACM SIGPLAN Int’l Conf. on Object-oriented Programming, Systems, Languages, and Applications*. Amsterdam: ACM, 2016. 74–91. [doi: 10.1145/2983990.2984027]
- [14] Rosu G. K: A semantic framework for programming languages and formal analysis tools. In: *Proc. of the 2017 Dependable Software Systems Engineering*. IOS Press, 2017. 186–206.
- [15] Plotkin GD. A structural approach to operational semantics. Technical Report, Aarhus: Aarhus University, 1981.
- [16] Clément D, Despeyroux J, Despeyroux T, Kahn G. A simple applicative language: Mini-ML. In: *Proc. of the 1986 ACM Conf. on LISP and Functional Programming*. Cambridge: ACM, 1986. 13–27. [doi: 10.1145/319838.319847]
- [17] Hoare CAR. An axiomatic basis for computer programming. *Communications of the ACM*, 1969, 12(10): 576–580. [doi: 10.1145/363235.363259]
- [18] Dijkstra EW. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 1975, 18(8): 453–457. [doi: 10.1145/360933.360975]

- [19] Bertot Y, Castéran P. Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Berlin: Springer, 2004. [doi: 10.1007/978-3-662-07964-5]
- [20] Greenaway D. Automated proof-producing abstraction of C code [Ph.D. Thesis]. Sydney: University of New South Wales, 2015.
- [21] Bulwahn L, Krauss A, Haftmann F, Erkök L, Matthews J. Imperative functional programming with Isabelle/HOL. In: Proc. of the 21st Int'l Conf. on Theorem Proving in Higher Order Logics. Montreal: Springer, 2008. 134–149. [doi: 10.1007/978-3-540-71067-7\_14]
- [22] Zhang XL, Zhu YF, Gu CX, Chen X. C2P: Formal abstraction method and tool for C protocol code based on Pi calculus. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1581–1596 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/6238.htm> [doi: 10.13328/j.cnki.jos.006238]
- [23] Reynolds JC. Separation logic: A logic for shared mutable data structures. In: Proc. of the 17th IEEE Symp. on Logic in Computer Science. Copenhagen: IEEE, 2002. 55–74. [doi: 10.1109/LICS.2002.1029817]
- [24] Jones CB. Tentative steps toward a development method for interfering programs. ACM Trans. on Programming Languages and Systems, 1983, 5(4): 596–619. [doi: 10.1145/69575.69577]
- [25] Harel D, Kozen D, Tiuryn J. Dynamic Logic. Cambridge: MIT Press, 2000.
- [26] Benton N. Simple relational correctness proofs for static analyses and program transformations. In: Proc. of the 31st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Venice: ACM, 2004. 14–25. [doi: 10.1145/964001.964003]
- [27] Blazy S, Leroy X. Mechanized semantics for the Clight subset of the C language. Journal of Automated Reasoning, 2009, 43(3): 263–288. [doi: 10.1007/s10817-009-9148-3]
- [28] Owens S, Myreen MO, Kumar R, Tan YK. Functional big-step semantics. In: Proc. of the 25th European Symp. on Programming. Eindhoven: Springer, 2016. 589–615. [doi: 10.1007/978-3-662-49498-1\_23]
- [29] Han N, Li XM, Zhang QY, Wang GH, Shi ZP, Guan Y. Executable semantics of ethereum intermediate language. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1717–1732 (in Chinese with English abstract). <http://www.jos.org.cn/1000-9825/6246.htm> [doi: 10.13328/j.cnki.jos.006246]
- [30] Reynolds JC. Theories of Programming Languages. Cambridge: Cambridge University Press, 1998.
- [31] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. Los Angeles: ACM, 1977. 238–252. [doi: 10.1145/512950.512973]
- [32] Zhang FY, Nielson F, Nielson HR. Model checking as static analysis: Revisited. In: Proc. of the 9th Int'l Conf. on Integrated Formal Methods. Pisa: Springer, 2012. 99–112. [doi: 10.1007/978-3-642-30729-4\_8]
- [33] Schmidt D, Steffen B. Program analysis as model checking of abstract interpretations. In: Proc. of 5th Int'l Symp. on Static Analysis. Pisa: Springer, 1998. 351–380. [doi: 10.1007/3-540-49727-7\_22]
- [34] Ellison C, Rosu G. An executable formal semantics of C with applications. In: Proc. of the 39th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. Philadelphia: ACM, 2012. 533–544. [doi: 10.1145/2103656.2103719]

#### 附中文参考文献:

- [1] 王戟, 詹乃军, 冯新宇, 刘志明. 形式化方法概貌. 软件学报, 2019, 30(1): 33–61. <http://www.jos.org.cn/1000-9825/5652.htm> [doi: 10.13328/j.cnki.jos.005652]
- [4] 周巢尘, 詹乃军. 形式语义学引论. 第2版, 北京: 科学出版社, 2017.
- [22] 张协力, 祝跃飞, 顾纯祥, 陈熹. C2P: 基于Pi演算的协议C代码形式化抽象方法和工具. 软件学报, 2021, 32(6): 1581–1596. <http://www.jos.org.cn/1000-9825/6238.htm> [doi: 10.13328/j.cnki.jos.006238]
- [29] 韩宁, 李希萌, 张倩颖, 王国辉, 施智平, 关永. 以太坊中间语言的可执行语义. 软件学报, 2021, 32(6): 1717–1732. <http://www.jos.org.cn/1000-9825/6246.htm> [doi: 10.13328/j.cnki.jos.006246]

#### 附录 A. GC<sub>arr</sub> 语言表达式求值函数的定义

以下给出第 5 节语义形式化中辅助函数  $\mathcal{A}$  和  $\mathcal{B}$  的定义. 其中, 沿用第 5 节中的约定: 对状态  $\sigma=(s, \theta)$ , 整型变量  $x$ , 数组变量  $X$ , 用  $\sigma x$  表示  $s x$ , 用  $\sigma X$  表示  $\theta X$ .

函数  $\mathcal{A}$  属于函数空间  $AExp \rightarrow State \rightarrow \mathbb{Z} \cup \{\perp\}$ , 将给定算术表达式在给定状态下求值. 若得一整数, 表示求值结果; 若得到  $\perp$ , 则表示该算术表达式在该状态下无意义. 函数  $\mathcal{A}$  的定义如下:

$$\begin{aligned}
\mathcal{A} \llbracket n \rrbracket \sigma &:= n \\
\mathcal{A} \llbracket x \rrbracket \sigma &:= \sigma x \\
\mathcal{A} \llbracket X[a] \rrbracket \sigma &:= z_i && \text{若 } \exists n: \sigma X = [z_0, \dots, z_{n-1}] \wedge i \in \{0, \dots, n-1\} \\
\mathcal{A} \llbracket X[a] \rrbracket \sigma &:= \perp && \text{若 } \sigma X = \perp \vee \exists n: \sigma X = [z_0, \dots, z_{n-1}] \wedge \mathcal{A} \llbracket a \rrbracket \sigma \notin \{0, \dots, n-1\} \\
\mathcal{A} \llbracket a_1 \text{ aop } a_2 \rrbracket \sigma &:= z_1 \text{ aop } z_2 && \text{若 } \mathcal{A} \llbracket a_1 \rrbracket \sigma = z_1 \wedge \mathcal{A} \llbracket a_2 \rrbracket \sigma = z_2 \wedge z_1, z_2 \in \mathbb{Z} \\
\mathcal{A} \llbracket a_1 \text{ aop } a_2 \rrbracket \sigma &:= \perp && \text{若 } \mathcal{A} \llbracket a_1 \rrbracket \sigma = \perp \vee \mathcal{A} \llbracket a_2 \rrbracket \sigma = \perp
\end{aligned}$$

本定义中, 对数值常量  $n$ , 其所表示的整数记作  $n$ , 对算术运算符  $\text{aop} \in \{+, -, *, /\}$ , 其所表示的整数运算记作  $\text{aop}$ .

本文第 5 节中函数  $\mathcal{B}$  属于函数空间  $BExp \rightarrow State \rightarrow \{tt, ff\} \cup \{\perp\}$ , 将给定布尔表达式在给定状态下求值. 若得一布尔值, 则表示求值结果; 若得到  $\perp$ , 则表示该布尔表达式在该状态下无意义. 函数  $\mathcal{B}$  的定义如下:

$$\begin{aligned}
\mathcal{B} \llbracket \text{true} \rrbracket \sigma &:= tt \\
\mathcal{B} \llbracket \text{false} \rrbracket \sigma &:= ff \\
\mathcal{B} \llbracket a_1 \text{ cop } a_2 \rrbracket \sigma &:= z_1 \text{ cop } z_2 && \text{若 } \mathcal{A} \llbracket a_1 \rrbracket \sigma = z_1 \wedge \mathcal{A} \llbracket a_2 \rrbracket \sigma = z_2 \wedge z_1, z_2 \in \mathbb{Z} \\
\mathcal{B} \llbracket a_1 \text{ cop } a_2 \rrbracket \sigma &:= \perp && \text{若 } \mathcal{A} \llbracket a_1 \rrbracket \sigma = \perp \vee \mathcal{A} \llbracket a_2 \rrbracket \sigma = \perp \\
\mathcal{B} \llbracket b_1 \ \&\& \ b_2 \rrbracket \sigma &:= tt && \text{若 } \mathcal{B} \llbracket b_1 \rrbracket \sigma = tt \wedge \mathcal{B} \llbracket b_2 \rrbracket \sigma = tt \\
\mathcal{B} \llbracket b_1 \ \&\& \ b_2 \rrbracket \sigma &:= ff && \text{若 } \mathcal{B} \llbracket b_1 \rrbracket \sigma = ff \vee \mathcal{B} \llbracket b_2 \rrbracket \sigma = ff \\
\mathcal{B} \llbracket b_1 \ \&\& \ b_2 \rrbracket \sigma &:= \perp && \text{若 } \mathcal{B} \llbracket b_1 \rrbracket \sigma = \perp \vee \mathcal{B} \llbracket b_2 \rrbracket \sigma = \perp \\
\mathcal{B} \llbracket !b \rrbracket \sigma &:= \neg t && \text{若 } \mathcal{B} \llbracket b \rrbracket \sigma = t \\
\mathcal{B} \llbracket !b \rrbracket \sigma &:= \perp && \text{若 } \mathcal{B} \llbracket b \rrbracket \sigma = \perp
\end{aligned}$$

本定义中, 对比较运算符  $\text{cop} \in \{=, <\}$ , 将其所表示的整数上的相应比较运算记作  $\text{cop}$ .

## 附录 B. 关于 $\text{Fun}_{\text{lst}}$ 语言的辅助定义

第 6 节中,  $f_v(e)$  给出表达式  $e$  中自由变量的集合,  $f_v(e)$  递归定义如下.

$$\begin{aligned}
f_v(n) &:= \emptyset \\
f_v(bc) &:= \emptyset && \text{其中, } bc \in \{\text{true}, \text{false}\} \\
f_v(e_1 \text{ op } e_2) &:= f_v(e_1) \cup f_v(e_2) && \text{其中, } \text{op} \in \{+, -, *, /, =, <, \&\&\} \\
f_v(!e) &:= f_v(e) \\
f_v(\text{if } e_1 \text{ then } e_2 \text{ else } e_3) &:= f_v(e_1) \cup f_v(e_2) \cup f_v(e_3) \\
f_v(\text{nil}) &:= \emptyset \\
f_v(e_1 :: e_2) &:= f_v(e_1) \cup f_v(e_2) \\
f_v(\text{listcase } e_1 \text{ of } (e_2, e_3)) &:= f_v(e_1) \cup f_v(e_2) \cup f_v(e_3) \\
f_v(x) &:= \{x\} \\
f_v(e_1 \ e_2) &:= f_v(e_1) \cup f_v(e_2) \\
f_v(\text{fn } x:e) &:= f_v(e) \setminus \{x\} \\
f_v(\text{recf } x=fn \ x:e' \text{ in } e) &:= (f_v(e) \cup f_v(e') \setminus \{x\}) \setminus \{x\}
\end{aligned}$$

第 6 节中,  $e[e'/x]$  为将表达式  $e$  中自由变量  $x$  替换为闭表达式  $e'$  ( $f_v(e') = \emptyset$ ) 的结果, 其定义如下.

$$\begin{aligned}
n[e'/x] &:= n \\
bc[e'/x] &:= bc && \text{其中, } bc \in \{\text{true}, \text{false}\} \\
(e_1 \text{ op } e_2)[e'/x] &:= e_1[e'/x] \text{ op } e_2[e'/x] \\
(!e)[e'/x] &:= !(e[e'/x]) \\
(\text{if } e_1 \text{ then } e_2 \text{ else } e_3)[e'/x] &:= \text{if } e_1[e'/x] \text{ then } e_2[e'/x] \text{ else } e_3[e'/x] \\
\text{nil}[e'/x] &:= \text{nil} \\
(e_1 :: e_2)[e'/x] &:= (e_1[e'/x]) :: (e_2[e'/x])
\end{aligned}$$

$$\begin{aligned}
(\text{listcase } e_1 \text{ of } (e_2, e_3))[e'/x] &:= \text{listcase } e_1[e'/x] \text{ of } (e_2[e'/x], e_3[e'/x]) \\
x[e'/x] &:= e' \\
x[e'/x'] &:= x \quad \text{若 } x \neq x' \\
(e_1 e_2)[e'/x] &:= e_1[e'/x] e_2[e'/x] \\
(\text{fn } x:e[e'/x] &:= \text{fn } x:e \\
(\text{fn } x:e)[e'/x'] &:= \text{fn } x:(e[e'/x']) \quad \text{若 } x \neq x' \\
(\text{recf } x=\text{fn } x':e' \text{ in } e)[e''/x] &:= \text{recf } x=\text{fn } x':e' \text{ in } e \\
(\text{recf } x=\text{fn } x':i \text{ in } e)[e''/x'] &:= \text{recf } x=(\text{fn } x':e')[e''/x'] \text{ in } e[e''/x'] \quad \text{若 } x \neq x''
\end{aligned}$$

### 附录 C.3 种示例程序语言大步操作语义的函数式形式化

在 Coq 辅助证明工具中, 使用函数  $eval$  对第 4–6 节中的 3 种示例程序语言的大步操作语义进行形式化 (机械化). 函数  $eval$  接受类型为  $config \rightarrow rconfig \rightarrow Prop$  的参数  $f$ , 返回同样类型的结果. 该函数对应于文中的函数  $h \in (C \rightarrow \mathcal{P}(T)) \rightarrow (C \rightarrow \mathcal{P}(T))$ . 其中, 初始格局 (即  $C$  中元素) 表示为具有 Coq 类型  $config$  的项, 终止格局集合 (即  $\mathcal{P}(T)$  中元素) 表示为具有类型  $rconfig \rightarrow Prop$  的项. 这里  $Prop$  为 Coq 中表示命题的类型,  $rconfig \rightarrow Prop$  可视为关于终止格局的断言, 对用于推理而非计算的终止格局集合进行形式化表示.

#### (1) While 语言大步操作语义的函数式形式化

Definition  $eval (f: config \rightarrow rconfig \rightarrow Prop) : (config \rightarrow rconfig \rightarrow Prop) :=$

```

(fun (cfg: config) (r: rconfig) =>
  (let (c, st) := cfg in (
    match c with
    | CmSkp => r = st
    | CmAssg x a => r = (st_upd st x (aval a st))
    | CmSeq c1 c2 => (exists st'', f(c1, st) st'' ^ f(c2, st'') r)
    | CmIf b c1 c2 => (if (bval b st) then f(c1, st) r else f(c2, st) r)
    | CmWh b c1 => (
      if (bval b st) then (exists st'', f(c1, st) st'' ^ f((CmWh b c1), st'') r) else r = st)
    end))).

```

#### (2) GC<sub>arr</sub> 语言大步操作语义的函数式形式化

下列函数  $eval$  的定义是第 5 节中函数  $h$  的定义在 Coq 中的表达. 直接使用 Coq 中的整数  $z$  表示数值常量  $nc$ , 使用 Coq 中的整数列表  $zs$  表示数值常量列表  $ns$ . 相应地, 将常量声明  $\text{var } x:=nc$  表示为  $\text{CmVDecl } x z$ , 数组声明  $\text{arr } X:=ns$  表示为  $\text{CmADecl } X zs$ . 这样省去了第 5 节函数  $h$  定义中对数值常量、数值常量列表的求值操作.

Definition  $eval (f: config \rightarrow rconfig \rightarrow Prop) : (config \rightarrow rconfig \rightarrow Prop) :=$

```

(fun (cfg: config) (r: rconfig) =>
  match cfg with
  | Com_Cfg cm st => (
    match cm with
    | CmSkp => r = st
    | CmVDecl x z => varst_of st x = None ^ r = st_upd_var st x z
    | CmADecl X zs => arrst_of st X = None ^ r = st_upd_arr st X zs
    | CmAssg x a => (varst_of st x <> None ^ exists z, aval a st = Some z ^ r = st_upd_var st x z)
    | CMAAssg X a1 a2 => (
      arrst_of st X <> None ^

```

```

exists z1 z2 z s z s',
  aval a1 st = Some z1 ∧ aval a2 st = Some z2 ∧
  arrst_of st X = Some z s ∧ (0 ≤ z1 ∧ z1 < Z.of_nat (length z s))%Z ∧
  z s' = (take (Z.to_nat z1) z s) ++ [:: z2] ++ (drop (Z.to_nat z1 + 1) z s) ∧
  r = st_upd_arr st X z s'
| CmIf gc => f(GCom_Cfg gc st) r
| CmDo gc => (exists st'', f(GCom_Cfg gc st) st'' ∧ f(Com_Cfg (CmDo gc) st'') r)
| CmSeq cm1 cm2 => (exists st'', f(Com_Cfg cm1 st) st'' ∧ f(Com_Cfg cm2 st'') r)
end)
GCom_Cfg gcm st => (
  match gcm with
  | GCom1 b cm => bval b st = Some true ∧ f(Com_Cfg cm st) r
  | GCom2 gc1 gc2 => f(GCom_Cfg gc1 st) r ∨ f(GCom_Cfg gc2 st) r
  end)
end).

```

### (3) Fun<sub>ist</sub> 语言大步操作语义的函数式形式化

Definition *eval* (*f*: *config* → *rconfig* → *Prop*) : (*config* → *rconfig* → *Prop*) :=

```

(fun (cfg: config) (r: rconfig) =>
  match cfg with
  | NE z => r = NE z
  | BE b => r = BE b
  | AddE e1 e2 => exists z1 z2, f e1 (NE z1) ∧ f e2 (NE z2) ∧ r = NE (z1+z2)%Z
  | SubE e1 e2 => exists z1 z2, f e1 (NE z1) ∧ f e2 (NE z2) ∧ r = NE (z1-z2)%Z
  | MulE e1 e2 => exists z1 z2, f e1 (NE z1) ∧ f e2 (NE z2) ∧ r = NE (z1*z2)%Z
  | DivE e1 e2 => exists z1 z2, f e1 (NE z1) ∧ f e2 (NE z2) ∧ r = NE (z1/z2)%Z
  | EqE e1 e2 => exists z1 z2, f e1 (NE z1) ∧ f e2 (NE z2) ∧ r = BE (Z.eqb z1 z2)
  | LtE e1 e2 => exists z1 z2, f e1 (NE z1) ∧ f e2 (NE z2) ∧ r = BE (Z.ltb z1 z2)
  | LeE e1 e2 => exists z1 z2, f e1 (NE z1) ∧ f e2 (NE z2) ∧ r = BE (Z.leb z1 z2)
  | NegE e1 => exists b1, f e1 (BE b1) ∧ r = BE (~~b1)
  | AndE e1 e2 => exists b1 b2, f e1 (BE b1) ∧ f e2 (BE b2) ∧ r = BE (b1 && b2)
  | IfE e1 e2 e3 => (f e1 (BE true) ∧ f e2 r) ∨ (f e1 (BE false) ∧ f e3 r)
  | NilE => r = NilE
  | PPendE e1 e2 => exists lcf1 lcf2, f e1 lcf1 ∧ f e2 lcf2 ∧ r = PPendE lcf1 lcf2
  | LCaseE e e' e'' =>
    (f e NilE ∧ f e' r) ∨ (exists cfcf', f e (PPendE cfcf') ∧ f (AppE (AppE e'' cfcf') cfcf') r)
  | VarE x => False
  | AppE e e' => exists x e'' cfcf', f e (AbsE x e'') ∧ f e' cfcf' ∧ f (e_subst e'' x cfcf') r
  | AbsE x e => r = AbsE x e
  | LetRecE x e'' e' =>
    (exists x' e', e'' = AbsE x' e' ∧ x < x' ∧
      f (AppE (AbsE x e) (AbsE x' (LetRecE x (AbsE x' e') e'))) r)
  end).

```



李希萌(1987-), 男, 博士, CCF 专业会员, 主要研究领域为形式化方法, 软件验证.



施智平(1974-), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为形式化方法, 人工智能.



王国辉(1984-), 男, 博士, 高级实验师, CCF 专业会员, 主要研究领域为形式化验证, 高可靠嵌入式系统.



关永(1966-), 男, 博士, 教授, 博士生导师, CCF 专业会员, 主要研究领域为形式化验证, 高可靠嵌入式系统, 机器人.



张倩颖(1986-), 女, 博士, 副教授, CCF 专业会员, 主要研究领域为形式化验证, 系统安全, 操作系统.

www.jos.org.cn

www.jos.org.cn