課題研究報告書

K フレームワークの調査研究と見える化

大澤広朗

主指導教員 緒方 和博

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和７年３月

Abstract

This research project introduces the K Framework, a framework that can rigorously defines the syntax and semantics of a programming language using formal methods, and proposes a tool to effectively visualize the formal semantics of a programming language described using the K Framework. The framework is a framework that enables concise and systematic definitions of programming languages based on operational semantics and has the ability to automatically generate tools such as parsers, interpreters, and model checkers. This feature provides great convenience in formalizing existing programming languages and designing new languages, and it is widely used in software development for formal semantics of programming languages such as C, Java, JavaScript, and Ethereum smart contracts. Successful examples such as the formal specification of the EVM language confirm its practicality. However, understanding the formal semantics described in the K Framework requires a precise grasp of a large number of rewriting rules, the complexity of which is a major barrier for learners and practitioners.

Based on operational semantics, the K Framework defines the semantics of a programming language using its syntax, an initial state, and a set of rewriting rules. The syntax is represented by a context-free method using BNF, and the initial state is represented by a tree structure. The state is often the main body of the program or an environment for storing variables. Rewriting rules make it possible for a language designer to explicitly only describe part of an entire state (called a configuration) being changed and omit the other parts being unchanged.

One of the characteristics of the K Framework is the mixture of rewriting rules explicitly described by the designer and implicit rules generated by the framework itself. The latter is, for example, rewriting rules necessary to realize value calls in evaluation strategies, while programming language implicit rules are essential to fully define the behavior of the language, they can also be a source of ambiguity about the designer's intentions. Therefore, when studying the semantics of a language written in the K Framework, it is not easy to identify the rules that the designer considered important and separate them from other less-important information. To mitigate this problem, this study developed a dedicated tool to effectively understand the rewriting rules of the language described in the K Framework. This tool extracts rules explicitly specified by the designer and visualizes before and after states rewritten by the specified rewriting rules.

The proposed tool was applied to various language paradigms, including procedural programing language (IMP), functional programing language (LAMBDA), object-oriented programing language (CLASS), and a concurrent programming language (THREAD), to demonstrate the usefulness of visualization according to their characteristics. CLASS traces the behavior of object creation and method in-

vocation, helping the reader to understand the concepts specific to object-oriented programming. In THREAD, we provide a method to clarify the complex behavior of concurrency by detailing the state transitions associated with concurrent threads creation, synchronization mechanisms, and inter-threads conflicts. These case studies demonstrate the applicability of the proposed tool to various programming language paradigms.

Furthermore, the proposed tool is unique and superior to existing visualization tools (e.g., ShiViz, SMGA). In particular, the ability to selectively visualize rewriting rules that designers consider important is a feature not found in other tools. In addition, by visually showing the order of application of rewriting rules along a time axis, the tool is designed to intuitively understanding the interaction between rules and the priority of application. This approach lowers the hurdle for learning formal semantics and shows its potential for practical as well as academic applications.

The significance of this research is that it provides a new approach to effectively understanding the formal semantics of programming languages utilizing the K Framework. This tool reduces the complexity of the semantics described in the K Framework and increases the practicality of designing and describing the semantics of programming languages using formal methods. Furthermore, due to the inherent flexibility of the K framework, it can be applied to new programming language paradigms and more complex systems in the future and has the potential to further expand the range of applications of formal methods. This research is an important step toward the practical application and dissemination of language design using formal methods and represents a new direction in the field of formal semantics.

Abstract(日本語)

本課題研究では、K Framework という形式手法を用いてプログラミング言語の構文と意味論を厳密に定義することのできるフレームワークを紹介し、K Framework を用いて記述されたプログラミング言語の形式意味論を効果的に視覚化するツールを提案する。K Framework は、操作的意味論に基づいてプログラミング言語の定義を簡潔かつ体系的に記述できるフレームワークであり、構文解析器、インタプリタ、モデル検査器といったツールを自動生成する機能を備えている。この特性により、既存のプログラミング言語の形式化や新しい言語の設計において大きな利便性を提供しており、C、Java、JavaScript といった一般のソフトウェア開発において広くつかれているプログラミング言語の形式意味論や Ethereum のスマートコントラクト言語 EVM の形式的仕様化などの成功例がその実用性を裏付けている。しかし、K Framework で記述された形式意味論を理解するには、多数の書き換え規則を正確に把握しなければならず、その複雑さが学習者や実務者にとって大きな障壁となっている。

K Framework では操作的意味論に基づいて、プログラミング言語の構文、初期状態、書き換え規則の集合を用いてプログラミング言語の意味論を定義する。構文は BNF を用いて文脈自由法、初期状態は木構造で表現される。状態にはプログラム本体や、変数を格納するための環境等が用いられることが多い。書き換え規則は状態を書き換える形で定義される。K Framework の特性として、設計者が明示的に記述した書き換え規則と、フレームワーク自体が生成する暗黙的な規則が混在していることが挙げられる。後者は例えば評価戦略における値呼びを実現するために必要な書き換え規則であり、プログラミング言語の暗黙的な規則は言語の動作を完全に定義するために必要不可欠だが、それらが設計者の意図を曖昧にする要因にもなり得る。このため、K Framework で記述された言語の意味論を学ぶ際に、設計者が重要視した規則を識別し、他の重要度の低い情報から切り分けることは容易ではない。本研究では、この課題を解決するために、K Framework で記述された言語の書き換え規則を効果的に理解するための専用ツールを開発した。このツールは、設計者が明示的に指定した規則を抽出し、指定した書き換え規則によって書き換えられた状態の前後を視覚化するものである。

提案ツールは、手続きプログラミング言語（IMP）、関数プログラミング言語（LAMBDA）、オブジェクト指向プログラミング言語（CLASS）、並行プログラミング言語（THREAD）などの多様な言語パラダイムに適用し、それぞれの特性に応じた視覚化の有用性を示した。IMP では、変数宣言や代入文、制御構造の評価手順を明示的に示し、LAMBDA では実行するコードと環境のペアであるクロージャや再帰関数定義の動作を視覚化することで、関数プログラミング言語の複雑な意味論をわかりやすく提示した。CLASS では、オブジェクト生成やメソッド呼び出しの動作を追跡し、オブジェクト指向プログラミング特有の概念を理解する手助けを行った。THREAD においては、並行スレッドの生成、同期機構、およびスレッド間の競合に関連する状態遷移を詳細に示し、並行処理の複雑な動作を明

確化する手法を提供した。これらのケーススタディにより、提案ツールが様々なプログラミング言語のパラダイムに適用可能なことを示した。

さらに、提案ツールは既存の視覚化ツール（例：ShiViz、SMGA）と比較しても独自性と優位性を有している。特に、設計者が重要と考える書き換え規則を選択的に視覚化する機能は、他のツールでは見られない特徴である。また、書き換え規則の適用順序を時間軸に沿って視覚的に示すことで、規則間の相互作用や適用の優先度を直感的に理解できるよう設計されている。このアプローチにより、形式意味論を学ぶ際のハードルを下げ、学術的な用途に加えて実務的な用途にも適用できる可能性を示した。

本研究の意義は、K Framework を活用したプログラミング言語の形式意味論を効果的に理解するための新たなアプローチを提供する点にある。本ツールは、K Framework で記述された意味論の複雑さを軽減し、形式手法を用いたプログラミング言語の設計や意味論記述の実用性を高める。さらに、K framework 本来の柔軟性により、今後は新しいプログラミング言語パラダイムやより複雑なシステムにも適用可能であり、形式手法の応用範囲をさらに広げる可能性を秘めている。本研究は、形式手法を用いた言語設計の実用化と普及に向けた重要な一歩であり、形式意味論の分野における新たな方向性を示している。

目 次

第1章　はじめに

本章では、本研究の背景、目的、関連研究、および構成について述べる。

1.1　背景

近年、プログラミング言語の設計や意味論の記述において形式手法が注目されている。現在主流となりつつあるクラウドコンピューティングの中核をなしている Amazon Web Services は変更に対する検証の難しさに対して形式手法を用いた。それにより、特定の故障や復旧手順の一連の流れが他の処理と交錯する場合にデータを失う可能性があるバグをモデル検査器が発見した。このバグは非常に微妙なもので、バグを引き起こす最短のエラートレースは 35 個の操作を含んでいた。このバグを含めて複数のバグを発見することが出来たと共に多くの最適化をすることが出来たとの報告 [11] がある。

形式意味論が定義されたプログラミング言語によって記述されたプログラムは、その意味論に従って動作するという仮定のもと、プログラムが望みの性質を有することを検証することができる。これはバグが発生することが許されない、例えば、宇宙船や自動車の制御システムなどの高信頼性を要求されるシステムにおける実用的な検証手法である。

プログラミング言語の構文や意味論を記述できるツールは様々存在 [7] [15] するが、その中でも、K framework は、プログラミング言語の構文と意味論を統一的かつ直感的に記述し、インタプリタやデバッガといったツールを自動生成するための強力なフレームワークとして知られている。

1.2　課題設定

近代のソフトウェア開発では、新規にソフトウェアを開発するよりも、既存のソフトウェアを保守、改良することが多い。形式意味論を持つプログラミング言語を用いたソフトウェア開発に新たに加わる者はその意味論を理解することが必要である。

しかし、K framework は、プログラミング言語の意味論を操作的意味論に基づいて直感的に記述するためのフレームワークであるが、記述されるプログラミング言語の意味論自体が直感的になるとは限らない。K framework で定義されたプ

ログラミング言語の意味論を理解するためには、多くの書き換え規則を理解する必要がある。しかし、書き換え規則は具体的な例を伴って記述されるものではなく、K framework に精通していないものが理解するのは困難である。

また、K framework ではプログラミング言語の意味論を定義する際に、言語設計者が明示的に記述した書き換え規則以外にも多くの書き換え規則が暗黙的に存在する。これらの書き換え規則は、記述されたプログラミング言語の意味論を理解する上で重要ではないが、K framework の出力には多くの書き換え規則が暗黙的、明示的問わず含まれてしまう。そのため、K framework で書かれたプログラミング言語の形式意味論を理解するためには、言語設計者が記述した重要な書き換え規則のみを抽出し、それらを視覚化（見える化）することが有用である。

本研究では、K framework によって記述されたプログラミング言語の意味論のなかでも重要な書き換え規則のみを抽出し、それらを視覚化するツールを提案する。

1.3 関連研究

本節では、システムの視覚化に関する関連研究について述べる。さらに、これらの研究と本研究の位置づけや差異を明確にする。形式手法の視覚化は、システムやプロトコルの特性を理解しやすくするために重要である。

ShiViz [2] は、分散システムによって生成されたログを視覚化するツールである。この文脈におけるログは、イベントのシーケンス、イベントを実行するホスト、およびホストがイベントを実行したタイムスタンプで構成される。ShiViz が生成するグラフは一般的なシーケンス図を生成する。そして特徴的なのは、複数の実行を比較して表示できることである。開発者は２つのアルゴリズムもしくは環境の違いによるシステムの実行の違いを比較することができる。ShiVis には２つの実行の差分を強調表示する実装もある。

著者たちは ShiViz を評価するために以下の 3 つの実験を行った

1. ShiViz を使用して分散システムの実行を学ぶ参加者のグループと、ShiViz なしで学ぶ別のグループを含む 39 人の学部生と大学院生の混合による制御実験
2. ShiViz を使用して実装のデバッグと理解を助けた 70 人の学生による分散システムコースの 2 つの宿題
3. 複雑な分散システムを開発している 2 人のシステム研究者による ShiViz のエンドツーエンドの有用性を評価

評価結果はポジティブであり、ShiViz は学生が分散システムをよりよく理解するのを助けた。また、分散システムの専門エンジニアでさえ、通常であれば発見するのが不可能または非常に時間がかかる微妙なエラーを見つけることができた。ShiViz で用いられている比較による視覚化は、分散システムの実行の違いを理解

するのに役立つことが示された。本研究で提案するツールにおいても、操作的意味論における遷移の前後の状態を比較することで、プログラミング言語の意味論を理解しやすくすることができる。

EVM [10] は、視覚的データ探索においてモデル検査を統合するツールであり、統計モデルの予測分布を観測データと比較することで、データ生成プロセスや仮説を直感的に検証することを目的としている。ユーザーはインタラクティブな操作を通じて、複数のモデルを簡単に作成し、それらを比較しながら洞察を得ることができる。

評価のために、データサイエンティスト 12 名を対象としたユーザースタディが行われた。その結果、EVM はモデル検査を視覚化することで、従来の視覚分析ツールでは得られなかった新しい洞察を提供し、データ分析の信頼性を向上させることが確認された。また、統計モデリングを学ぶ良い機会を提供したと一部のユーザーから評価された。

Java Pathfinder [17] は、プログラミング言語 Java をモデル検査するためのソフトウェアである。Java Pathfinder は初期は Java コードからモデル検査器 Spin [15] の記述言語である Promela への翻訳ソフトウェアであった。しかしカバレッジの問題から、モデル検査が可能な独自の Java 仮想機械の実装へと変更された。Java Pathfinder はその仮想機械上でスケジューリング探索やシンボリック実行により、Java で作成されたプログラムのモデル検査を可能とする。しかし、Java Pathfinder の出力は非常に長く複雑であり、その出力を理解することは困難である。

VA4JVM [1] は、Java Pathfinder の出力を視覚化するツールとして、長大な出力の中から重要な情報を抽出し、ズーム機能やハイライトを通じてユーザーの理解を助けている。

Java Pathfinder と同様に K Framework は記述された意味論のインタプリタの状態を出力することができるが、その出力は非常に長く複雑である。本研究で提案するツールは、インタプリタの出力を視覚化し、更にプログラミング言語設計者が指定した書き換え規則による状態遷移のみを視覚化の対象とすることで観察すべき状態遷移の数を絞り、プログラミング言語の意味論を理解しやすくすることを目指す。

Dang ら [4] は、状態遷移図の設計およびその視覚化ツール「SMGA（State Machine Graphical Animation）」の活用を通じて、形式検証の効率化に寄与する新しいアプローチを提案した。この研究では、Mellor-Crummey と Scott によるメモリ共有型相互排除プロトコル（MCS プロトコル）を例として取り上げ、新しい状態図設計がいかにして視覚的認識を向上させ、状態機械の特性推測を支援するかを詳述している。彼らのアプローチの特徴的な点は、ゲシュタルト原則、「近接性の原則」と「類似性の原則」を活用している点であり、これにより視覚要素の配置や色使いを最適化することで、状態図の直感的な理解が可能となっている。

例えば、

1. key と値を持った連想配列と呼ばれるデータ構造において、値の種類があれ

ば種類ごとに視覚要素をわける、つまり、on という値には明るい色、off という値には暗い色をつかうこと。

2. 連想配列のデータ構造の値に 3 つ以上の種類がある場合、それらの値を視覚要素の位置によって分けること。

といった具体的な視覚化ツールの tips が提案されている。これらの tips は、本研究で提案するツールにも適用可能であり、将来的にゲシュタルト原則を考慮した図式にすることによって、視覚的な理解を向上させることができると考えられる。また、著者らは SMGA を改良した r-SMGA [5] を提案している。r-SMGA は、インタラクティブな視覚表現や、形式仕様言語である Maude との統合が実装されている。結果として、視覚化された状態遷移の観察を容易にし、文脈自由文法によるパターンマッチングが活用できるようになった。これらはユーザーがプロトコルの特徴を推測し、実際に確認することを容易にしている。ケーススタディとして Suzuki-Kasami 分散相互排除プロトコル [16] を用い、分散システムにおける相互排除特性を検証した結果、r-SMGA が非自明な補題を発見し、形式証明を補助する上で有効であることが示された。これにより、r-SMGA は視覚的アプローチを活用することで、推測が困難だった特性の発見を容易にし、形式検証の効率化に寄与している。r-SMGA に対して本研究のツールでは、プログラミング意味論の書き換えが大量に存在するなかで、設計者が記述した重要な書き換え規則のみを抽出し、それらを視覚化するというアプローチをとり視覚化の効果を高めることを目指す。

1.4 本研究の目的

本研究は、上記の研究を基盤として、K framework における書き換え規則の視覚化に特化したツールを提案する。本研究のツールは K Framework を用いて言語設計者が記述した書き換え規則に焦点を当て、それらの適用箇所を直感的に視覚化することで、形式意味論の理解と説明を支援する。

本研究の提案するツールは、K Framework の十分な表現力と合わせて様々なプログラミング言語を対象とすることができる。K Framework のチュートリアル [14] に記載されている単純な手続きプログラミング言語や関数プログラミング言語、オブジェクト指向プログラミング言語や並行プログラミング可能な言語を対象として、ツールの有用性を検証する。論文構成のため、いくつかチュートリアル [14] から編集した例を用いていることに留意されたい。これらの点から、本研究は K framework および形式手法の視覚化における新たな方向性を提供するものである。

1.5 本論文の構成

本論文は以下の構成である。

1. 第 2 章では、K framework の概要について述べる。本章では、K framework がプログラミング言語の構文と意味論を統一的に記述する手法を詳細に説明する。また、形式手法としての特性や、書き換え規則を利用して動作を定義する基本的な仕組みを解説する。

2. 第 3 章では、本研究で提案するツールの設計思想と実装について述べる。このツールは、K framework の出力を視覚的に解析しやすくするためのものであり、ユーザーが K framework の書き換え規則の適用をより直感的に理解できるよう設計されている。

3. 第 4 章では、基礎的な手続きプログラミング言語を対象に、K framework による意味論の定義を具体的に示す。具体例として、典型的な制御フローやデータ操作に焦点を当て、その表現方法やツールの適用可能性を議論する。また、手続きプログラミング言語の意味論を視覚化するためのツールの有用性を示す。

4. 第 5 章では、基礎的な関数プログラミング言語を対象に、意味論の記述と提案ツールの適用例を詳述する。関数の評価戦略や高階関数といった特徴的な概念がどのように K framework で構文と意味論が定義されるかを説明し、それを視覚化するツールの有用性を示す。

5. 第 6 章では、オブジェクト指向プログラミング言語のクラスやメソッドの意味論を K framework を用いて定義する方法を示す。本章では、クラスの継承やオブジェクトの生成といった概念がどのように形式化されるかを具体例を交えて説明し、提案ツールの適用例を示す。

6. 第 7 章では、並行プログラミングのための言語機能に対する提案ツールの適用例を紹介する。競合状態など、並行性に関する特有の課題を解決する方法がどのように形式化されるかを示し、提案ツールの有用性を強調する。

7. 第 8 章では、本研究の成果を総括し、今後の展望について述べる。本章では、提案ツールの現在の限界と期待を議論する。

第2章　K frameworkについて

本課題研究ではK frameworkという形式手法を用いてプログラミング言語の定義を行うことができるフレームワークを紹介する。

K Frameworkの基礎的な概念についてはXiaohong Chen [6] らが、K Frameworkではプログラミング言語の構文と意味論が与えられると、自動的に構文解析器、インタプリタ、モデル検査器などの形式手法解析ツールを生成できることを示している。これにより、プログラミング言語意味論の開発の効率性と一貫性を向上することができる。例えば、プログラミング言語設計者は開発している意味論のインタプリタを設計しなくても、すぐに意味論をテストすることができる。また、与えられた意味論から生成されたモデル検査器を使うことで、意味論を他の言語に翻訳するような作業を行うことなく、一貫性を持ってモデル検査を行うことができる。

K Frameworkを用いて既存のプログラミング言語を形式化する研究は豊富に存在する。たとえば、C言語 [8]、Java [3]、およびJavaScript [12] などが挙げられる。また、Kフレームワークは Ethereumの現在のスマートコントラクト言語であるEVM [9] を形式に仕様化するためにも使用された。実際、EVMを実行可能な意味論として形式化する過程で、その元の英語による仕様書 [18] におけるさまざまな不一致や未定義の挙動が明らかになった。これらはK Frameworkで定義する意味論が十分な表現力を持ち、実際のソフトウェア開発の現場においても有用なツールであることを示している。

また、Grigore Rosu [13] らは、K Frameworkの基本的な概念を説明し、基礎的な手続きプログラミング言語を用いてK Frameworkの構文と意味論を定義する方法を示している。

2.1　K frameworkでのプログラミング言語の定義

この節ではK frameworkでの意味論の定義方法について説明する。また実例として、基礎的な手続きプログラミング言語（IMP）を定義する方法を示す。

2.1.1 K framework の基本的な概念

K framework ではプログラミング言語を構文と書き換え規則によって定義する。構文は、Backus–Naur form(BNF) を用いて文脈自由法によって定義される。書き換え規則は操作的意味論に基づいて定義される。K framework におけるプログラミング言語の意味論は状態の書き換え規則の集合として定義される。

2.1.2 基礎的な手続きプログラミング言語（IMP）の定義

説明のために、K framework で基礎的な手続きプログラミング言語（IMP）を定義する方法を示す。この IMP は K framework のチュートリアル [14] に記載されているものを基にして作成したプログラミング言語の意味論である。

式

```
syntax AExp  ::= Int | Id
               | "-" Int
               | AExp "/" AExp              [left, strict]
               | "(" AExp ")"               [bracket]
               > AExp "+" AExp              [left, strict]
```

K framework では、構文定義は BNF を拡張した構文を用いて記述する。終端記号は引用符"で囲み、非終端記号は大文字で始まる。通常の BNF では見られない > という記号は優先度を定義している。>の左辺の構文要素が右辺の構文要素よりも優先されることを意味する。角括弧 [] は属性を表す。意味的な情報を表すこともあれば、構文解析にのみ影響を与えることもある。left 属性は、構文 `Exp1 * Exp2` が左結合であることを示す。つまり、K framework は、`Exp1 * Exp2 * Exp3` を `(Exp1 * Exp2) * Exp3` と解釈する。bracket 属性は、構文（ `AExp` ）が構文解析において括弧で囲まれていることを示す。例えば、プログラミング言語利用者が最初に加算を適用し、その後に乗算を適用する式を書きたい場合、括弧を用いて明示的に優先順位を指定する必要がある。その際に括弧として使うための記号を指定するための属性である。strict 属性は、引数、この場合は 2 つの Exp が値になるまで評価する評価戦略を用いることを示す。IMP における値については後述する。また、strict 単体では引数の評価の順序を指定することはない。つまり、`Exp1 * Exp2` が `Exp1` と `Exp2` のどちらを先に評価するかは非決定的である。この定義では、IMP の算術式は、自然数、変数、単項のマイナス演算、除算、加算、括弧で括られた算術式からなることを示している。

以下に IMP のブール式の構文を定義する。

```
syntax BExp  ::= Bool
               | AExp "<" AExp              [seqstrict]
               | "!" BExp                    [strict]
               | "(" BExp ")"                [bracket]
               > BExp "&&" BExp              [left, strict(1)]
```

strict(1) は、構文内の非終端記号のうち、一つ目の非終端記号のみ strict 属性を持つことを示す。seqstrict は、引数の評価順序が決定的で左辺から順に値まで評価されることを示す。この定義では、IMP のブール式は、真偽値、算術式の比較、否定、括弧、論理積からなることを示している。

IMP の文の構文

以下に IMP の文の構文を定義する。

```
syntax Block ::= "{" "}"
               | "{" Stmt "}"
syntax Stmt  ::= Block
               | Id "=" AExp ";"             [strict(2)]
               | "if" "(" BExp ")"
                 Block "else" Block          [strict(1)]
               | "while" "(" BExp ")" Block
               > Stmt Stmt                   [left]
```

この定義では、IMP の文は、ブロック、代入、条件分岐、繰り返し、文の連結からなることを示している。

以下に IMP のプログラム全体の構文を定義する。

```
syntax Pgm ::= "int" Ids ";" Stmt
syntax Ids ::= List{Id,","}
```

この定義では、IMP のプログラムは、変数宣言と文からなることを示している。Ids は、変数のリストを表す。コンマ (,) はリストの区切り記号である。

```
syntax KResult ::= Int | Bool
```

KResult は K framework の組み込みの非終端記号で、プログラムの実行結果を表す。KResult はこれ以上簡約できないすべての構文要素を含んでいる集合である。言語設計者は KResult を適切に定義して、計算結果を明示的に指定できる。ここでは IMP の計算結果は整数もしくは真偽値であることを示している。strict 属性は、非終端記号が KResult になるまで評価されることを示す。

configuration

K framework は configuration と呼ばれる状態を設定する。configuration は木構造であり、各ノードはセルと呼ばれる。セルは名前、初期値を持つ。

以下に IMP の初期の configuration を設定する。

```
configuration <T>
                <k> $PGM:Pgm </k>
                <state> .Map </state>
              </T>
```

k は計算を表すセルである。state はプログラム内の変数の名前と値を記録するセルである。この定義では、configuration は、T セルというルートセルを持ち、その下に実行しているプログラムを表す k セルと変数とその値の組を持つ state セルを持つ木構造であることを示している。$PGM は利用者によって作成された

IMP のプログラムが渡されることを示している。.Map は、空の連想配列を表す K framework の組み込みの記号である。つまり、IMP の初期状態は、IMP 利用者が記述したプログラムと空の変数と値の連想配列である。

書き換え規則

K framework は意味論を操作的意味論に基づいて定義する。操作的意味論とは、プログラムの動作を記述し意味を与えるための形式手法である。K framework では、書き換え規則を用いてプログラムの動作を定義する。K framework におけるプログラムの意味論は書き換え規則の集合である。

K における書き換え規則は rule を使用して定義される。rule キーワードで始まり、少なくとも 1 つの書き換え演算子を含む。書き換え演算子は =>という構文で表される。左辺には 0 個以上のパターンがあり、右辺には別のパターンがある。このパターンは configuration 内のセルを含めて、任意の K framework の構文要素を含むことができる。

変数宣言

以下に IMP のプログラム変数宣言の書き換え規則を示す。

```
rule <k> int (X,Xs => Xs);_ </k>
     <state> Rho:Map (.Map => X|->0) </state>
     requires notBool (X in keys(Rho))

rule int .Ids; S => S
```

<k> int (X,Xs => Xs);_ </k>は K の書き換え演算子の特徴的な使用方法である。K の書き換え規則において、=>を演算子として左辺を右辺に置き換える。この場合、IMP のプログラムが int キーワードに続いて変数名のリストがプログラムの先頭に存在する場合にマッチするパターンである。アンダースコア (_) は任意の非終端記号にマッチする。この書き換え規則によって state に変数名とその初期値 0 を記録する。ここで使われているメタ変数 Rho は既存の state セルの内容を表す。既存の state セルに変数 X が含まれていないことを条件としている。2 つ目の rule は宣言されている変数がなくなり、空の変数のリストを表す.Ids が現れた場合、その後に続く文 S が評価されることを示している。

変数への代入

以下に IMP の代入文の書き換え規則を示す。

```
rule <k> X = I:Int; => .K ...</k>
     <state>... X |-> (_ => I) ...</state>
```

この規則は、代入文が評価されると、state セル内の変数 X の値が I になることを示している。代入文は.K に評価される。この.K は、K における特別な終端記号で、評価すべきプログラムがなくなったことを示す。... は、このパターンマッチが K セルの開始部分以外は考慮しないことを示している。つまり、代入文が K セルの先頭にある場合にマッチする。K の書き換え規則は不可分である。つまり、書き換え途中の状態は存在しない。state セル内の両端の... は X |-> _という要素が state 内のどの順番に存在していてもマッチすることを示している。

変数の参照

IMP における変数の参照は以下のように定義される。

```
syntax AExp ::= Id
rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
```

X:Id は、変数 X が Id であることを示している。I は、変数 X が評価された結果である。IMP において Id の集合は算術式に含まれるため、変数内に真偽値が含まれることはない。

算術式の評価

以下に IMP の算術式の評価の書き換え規則を示す。

```
rule I1 / I2 => I1 /Int I2  requires I2 =/=Int 0
rule I1 + I2 => I1 +Int I2
rule - I1 => 0 -Int I1
```

/Int、+Int、-Int は、自然数の除算、加算、単項のマイナス演算を表す K framework の組み込みの記号である。requires は、条件を表す。この場合、除数が 0 でないことを示している。

ブール式の評価

以下に IMP のブール式の評価の書き換え規則を示す。

```
rule I1 < I2 => I1 <Int I2
rule ! T => notBool T
rule true && B => B
rule false && _ => false
```

notBool は、真偽値の否定を表す K framework の組み込みの記号である。構文定義において論理積を strict(1) としていたのは、１つ目の引数さえ評価されれば、論理積の評価が可能であるためである。

条件分岐の評価

以下に IMP の条件分岐の書き換え規則を示す。

```
rule if (true)  S else _ => S
rule if (false) _ else S => S
```

この規則は、条件分岐の評価において、条件が真であれば then 節が評価され、条件が偽であれば else 節が評価されることを示している。条件分岐の構文において strict(1) 属性を設定していたのは、条件式が評価されれば、条件分岐の評価が可能であるためである。

繰り返し

以下に IMP の繰り返しの書き換え規則を示す。

```
rule while (B) S => if (B) {S while (B) S} else {}
```

この規則は、繰り返しの評価において、条件が真であれば本体を評価し、再び繰り返しを評価することを示している。条件が偽であれば、空のブロックを評価し、繰り返しを終了する。

文の逐次合成

以下に IMP の文の逐次合成の書き換え規則を示す。

```
rule S1:Stmt S2:Stmt => S1 ~> S2
```

書き換え規則の左辺には、文の連結がある。この規則は、文の連結が評価されると、文 S1 が評価され、その後に文 S2 が評価されることを示している。~>は、逐次合成を表す K framework の組み込みの記号である。

ブロックの評価

以下に IMP のブロックの書き換え規則を示す。

```
rule {} => .K
rule {S} => S
```

この規則は、ブロックが評価されると、ブロック内の文が評価されることを示している。ブロックが空であれば、.K に評価される。

実行例

ここでは IMP のプログラムを実行する例を示す。以下のプログラムは、IMP を用いて作成された変数 x と y に 1 と 2 を代入し、x に y を加算するプログラムである。

```
int x, y;
x = 1;
y = 2;
x = x + y;
```

このプログラムを K framework で定義した IMP の意味論に従って実行すると、下記のような初期状態となる。

```
<T>
  <k> int x, y; x = 1; y = 2; x = x + y; </k>
  <state>
    .Map
  </state>
</T>
```

計算ステップを一つ進めると、以下のような状態になる。

```
<T>
  <k> int y; x = 1; y = 2; x = x + y; </k>
  <state>
    x |-> 0
  </state>
</T>
```

この初期状態から IMP の書き換え規則に従って当てはまる書き換え規則がなくなるまで実行すると、以下のような状態になる。

```
<T>
  <k> . </k>
  <state>
    x |-> 3
    y |-> 2
  </state>
</T>
```

以下のようなコードでモデル検査を行うことも可能である。

```
claim <k> int $a, $b;
$a = X:Int;
$b = Y:Int;
if ( $a < $b ) {
  $b = $a;
} else {
  $a = $b;
} => .K </k>
<state> STATE => STATE [$a <- X] [$b <- X] </state>
requires notBool(X ==Int Y)
  andBool notBool($a in keys(STATE))
  andBool notBool($b in keys(STATE [ $a <- 0 ]))
```

このコードは X と Y を入力として受け取り、`X < Y` の場合に Y に X を代入し、それ以外の場合に X に Y を代入するプログラムをモデル検査するためのコードである。K 内の書き換え演算子=>の右辺は、k セル内に書き換えられるプログラムがなくなることを意味し、state セル内の=>の右辺は初期の状態を表すメタ変数 STATE に、変数`$a` と`$b` をキーとしてどちらも値が X となることを期待している。条件として、X と Y は常に等しくない入力であること、`$a` と`$b` が STATE 内に存在しないことを示している。これは明らかに `X < Y` が成り立たない場合に失敗する検証である。

SIMP の定義から生成されたモデル検査器を用いて、このモデル検査を実行することができる。モデル検査を行うと、以下のような結果が出力される。

```
#Not ( {
  X
#Equals
  Y
} )
#And
<T>
  <k>
    .K
  </k>
  <state>
    $a |-> Y:Int
    $b |-> Y:Int
    STATE
  </state>
</T>
#And
{
  false
#Equals
  X <Int Y
}
```

モデル検査失敗時の出力は、`claim` 内の書き換え規則どおりに書き換えられていないことを示す部分と、最終状態の configurations と、実行中の分岐に関わる部分の出力によって構成される。この出力の場合、`X < Y` が真ではない場合の分岐において、最終的に state 内の`$a` と`$b` の値がどちらも Y になり、X と Y は異なるので検証に失敗するということがわかる。

このように K framework を用いることで、プログラミング言語設計者は構文と意味論を定義するだけで、プログラムの実行環境やモデル検査器を得ることができる。これは、プログラミング言語設計において、プログラムの意味論を定義する際に非常に有用である。また計算ステップ単位での実行結果を確認することができるため、意味論を理解する際にも有用である。

2.2 定義済みの書き換え規則と課題

意味論の設計者を助けるため、K framework は様々な組み込みの書き換え規則を提供している。これらの書き換え規則は、プログラムの実行やモデル検査を行う際に使用される。この節では、K framework で定義された書き換え規則の一部を紹介する。また、それらの規則と意味論の理解に関する課題についても述べる。

2.2.1 評価戦略

K framework では、評価戦略を指定するための書き換え規則が提供されている。例えば、IMP におけるブール式を値呼び戦略で評価するための書き換え規則は以下のようになる。

```
syntax BExp ::= AExp "<" AExp              [seqstrict]
```

この seqstrict 属性は、左辺の AExp、右辺の AExp が値になるまで評価されたのちに、比較演算子が適用されることを示している。このような評価戦略を指定することで、プログラムの意味論を正確に定義することができる。意味論の設計者が記述するのは一行の構文規則だけだが、K framework はこの定義を元に複数の書き換え規則（今回の場合、RW1、RW2、RW3、RW4 の 4 つの書き換え規則）を自動生成する。

```
syntax KItem ::= freezer_<_1(Exp) | freezer_<_2(Exp)

rule <k> E1:Val < E2:Exp ~> K:K </k>
  => <k> E2 ~> freezer_<_1(E1) ~> K </k> [priority(51)] // RW1
rule <k> E1:Exp < E2:Exp ~> K:K </k>
  => <k> E1 ~> freezer_<_2(E2) ~> K </k> [priority(52)] // RW2

rule <k> E2:Val ~> freezer_<_1(E1) ~> K:K </k>
  => <k> E1 < E2 ~> K </k> // RW3
rule <k> E1:Val ~> freezer_<_2(E2) ~> K:K </k>
  => <k> E1 < E2 ~> K </k> // RW4
```

これは、評価戦略において、左辺と右辺の AExp が値になるまで評価されたのちに、比較演算子が適用されることを示している。RW1 と RW2 は、左辺と右辺の AExp に値でない項がある場合、それらを評価するため、比較演算子の計算を遅延させる書き換え規則である。RW3 と RW4 は、左辺と右辺の AExp が値になった場合、遅延されていた比較演算子に対して、値を適用する書き換え規則である。このような計算を一時的に遅延するために、式や文を一時的に固定するための構文を K framework では慣用的に freezer という形で定義することが多い。KItem という非終端記号を定義し、freezer という形で固定するための構文を定義する。KItem は、K framework の組み込みの非終端記号で、プログラミング言語利用者ではなく、書き換え規則内部でのみ使用する非終端記号である。seqstrict 属性を持つ構文要素に対して、このような書き換え規則が自動的に生成されるため、意味論の記述が簡潔になる。しかし、意味論を実際に簡約していく過程ではこの書き換え規則は意味論設計者が直接記述した書き換え規則と区別されず適用される。
例えば、IMP において `E1 < E2` という比較演算子が評価される際には、以下のように 5 つの書き換え規則が適用される。

- 1: E1 が値でない場合、E1 を値になるまで評価する。
- 2: 値になった E1 を `E1 < E2` に戻す。
- 3: E2 が値でない場合、E2 を値になるまで評価する。
- 4: 値になった E2 を `E1 < E2` に戻す。
- 5: `E1 < E2` を評価する。

これは意味論の実行としては問題ないが、操作的意味論の理解のため、計算ステップを一つづつ追う際には、このような書き換え規則が多くなると、重要な書き換え規則が埋もれてしまう。

2.3 まとめ

この節では、K framework を用いた基礎的な手続きプログラミング言語 IMP の意味論の定義をとおして K framework での意味論の記述について説明した。K framework で意味論を定義するには構文と意味論と初期状態を定義する必要がある。IMP の構文は、変数宣言、代入、条件分岐、繰り返し、文の連結からなる。また、定義した意味論を用いてモデル検査が可能であることも示した。また、K framework の組み込みの書き換え規則である評価戦略のための書き換え規則を紹介し、その書き換え規則によって、意味論を記述する際に簡潔に記述することができることを示した。しかし、この書き換え規則は意味論を理解する際に、重要な書き換え規則が埋もれてしまうことがある。この課題に対して、次章では K framework の書き換え規則の視覚化ツールを提案する。

第3章　K frameworkの書き換え規則の視覚化ツール

この章ではK frameworkの書き換え規則の視覚化ツールを提案する。

3.1　K frameworkの書き換え規則の視覚化ツールの概要

プログラミング言語の意味論を記述するにあたって、書き換え規則を理解することは重要である。しかし、先述したようにK frameworkではプログラミング言語の設計者が記載した書き換え規則意外にも様々な組み込みの書き換え規則が存在する。この書き換え規則はK frameworkで定義された意味論を具体的なプログラムで実行する際に、プログラミング言語設計者が定義した書き換え規則と区別なく適用される。これらの書き換え規則は、プログラムの実行やモデル検査を行う際に必要な規則であるが、意味論を理解するにあたって優先順位が低い情報であり、意味論を理解しようとしている者にとっては重要な書き換え規則が埋もれてしまうことがある。そこで、本研究では、K frameworkの書き換え規則の中でも言語設計者が記載した書き換え規則から、特に重要な書き換え規則を指定し、それのみを視覚化するツールを提案する。このツールは、K frameworkによるプログラミング言語の定義ファイルとそのプログラミング言語で書かれたプログラムを入力として受け取る。その後、言語設計者が指定した書き換え規則を抽出しプログラムを実行する。その実行過程でその書き換え規則がconfigurationに適用されるたびに、その前後のconfigurationを視覚化する。

3.2　K frameworkの書き換え規則の視覚化ツールの実際の動作

視覚化ツールは以下の手順で実行する。

1. プログラミング言語設計者は、K frameworkで定義されたプログラミング言語の定義ファイルを作成する。

2. 定義ファイルから言語設計者が記載した書き換え規則に名前をつけることで指定する。

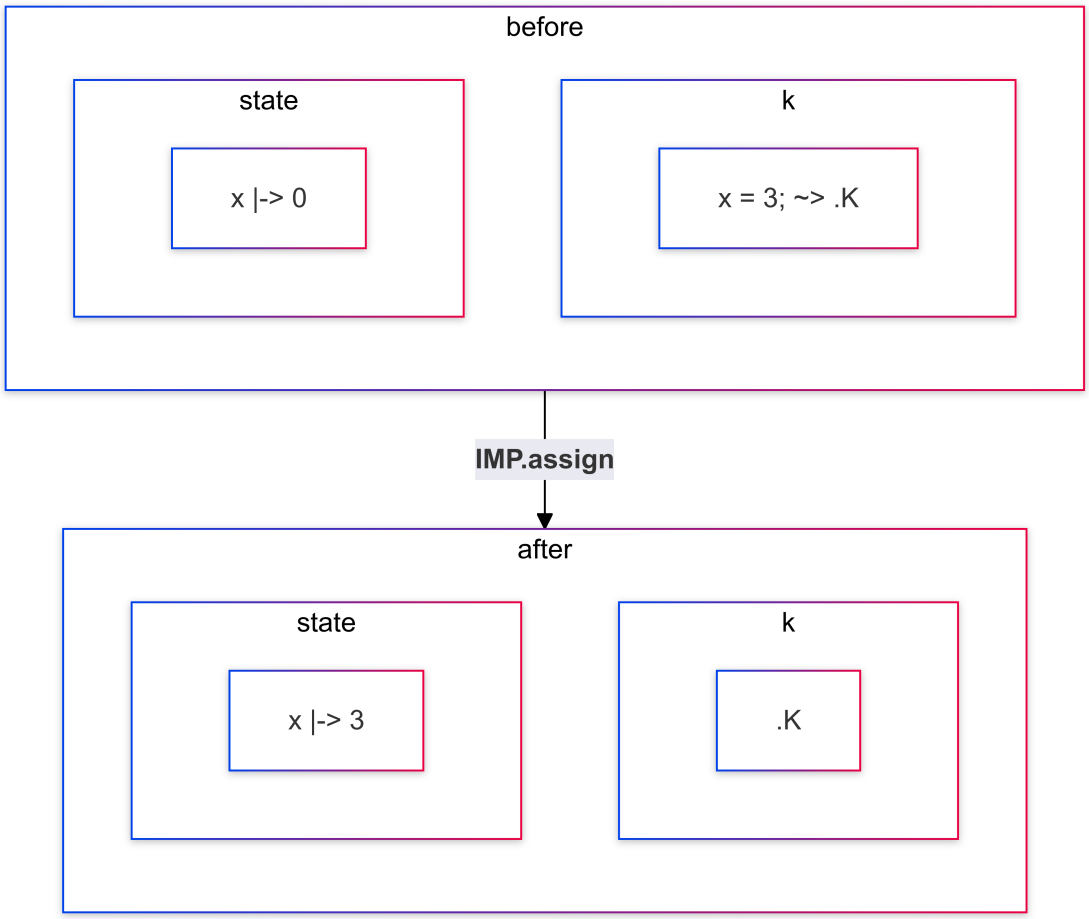
3. 視覚化ツールを用い、設計したプログラミング言語で書かれたプログラムを実行すると、書き換え規則が適用されている箇所を視覚化するファイルが生成される。

プログラミング言語設計者は、書き換え規則に以下のように名前をつける。

```
rule [assign]: <k> X = I:Int; => .K ...</k>
<state>... X |-> (_ => I) ...</state>
```

この例では、変数の参照の書き換え規則に assign という名前をつけている。これによって、プログラムを実行する際に、この書き換え規則が適用された箇所を視覚化することができる。プログラミング言語設計者は意味論を記述する段階で、視覚化したい書き換え規則を指定できる。

この assign という名前をつけた書き換え規則が適用された箇所を視覚化すると以下のような図が生成される。



この図は、assign という名前をつけた書き換え規則が configration に適用される箇所を視覚化したものである。視覚化の対象は configuration であり図上

部の before と書いてある図は書き換え規則が適用される前の configration を示している。図下部の after と書いてある図は書き換え規則が適用された後の configration を示している。この図では書き換え規則の適用一回のみを図示しているので、before と after という表示になっているが、実際には書き換え規則が適用されるたびに before_1, after_1, before_2, after_2 というように表示される。

configration 前後の間には前述の書き換え規則につけられた名前が表示される。これにより、プログラムの実行過程でどの書き換え規則が適用されたかを視覚化することができる。

3.3 プログラミング言語設計者が指定した規則のみを対象に前後の状態を取得することのメリット

プログラミング言語設計者が指定した規則のみの適用前後の状態を生成させることそのものが K framework の意味論を理解する助けになる。IMP のプログラムの具体的な実行例を用いて説明する。以下のプログラムは 0 から 5 までの整数の和を計算するプログラムである。

```
int n, sum;
n = 5;
sum = 0;
while (!(n < 0)) {
sum = sum + n;
n = n + -1;
}
```

このプログラムを IMP の意味論に従って評価すると、最終的な configuration は以下のようになる。

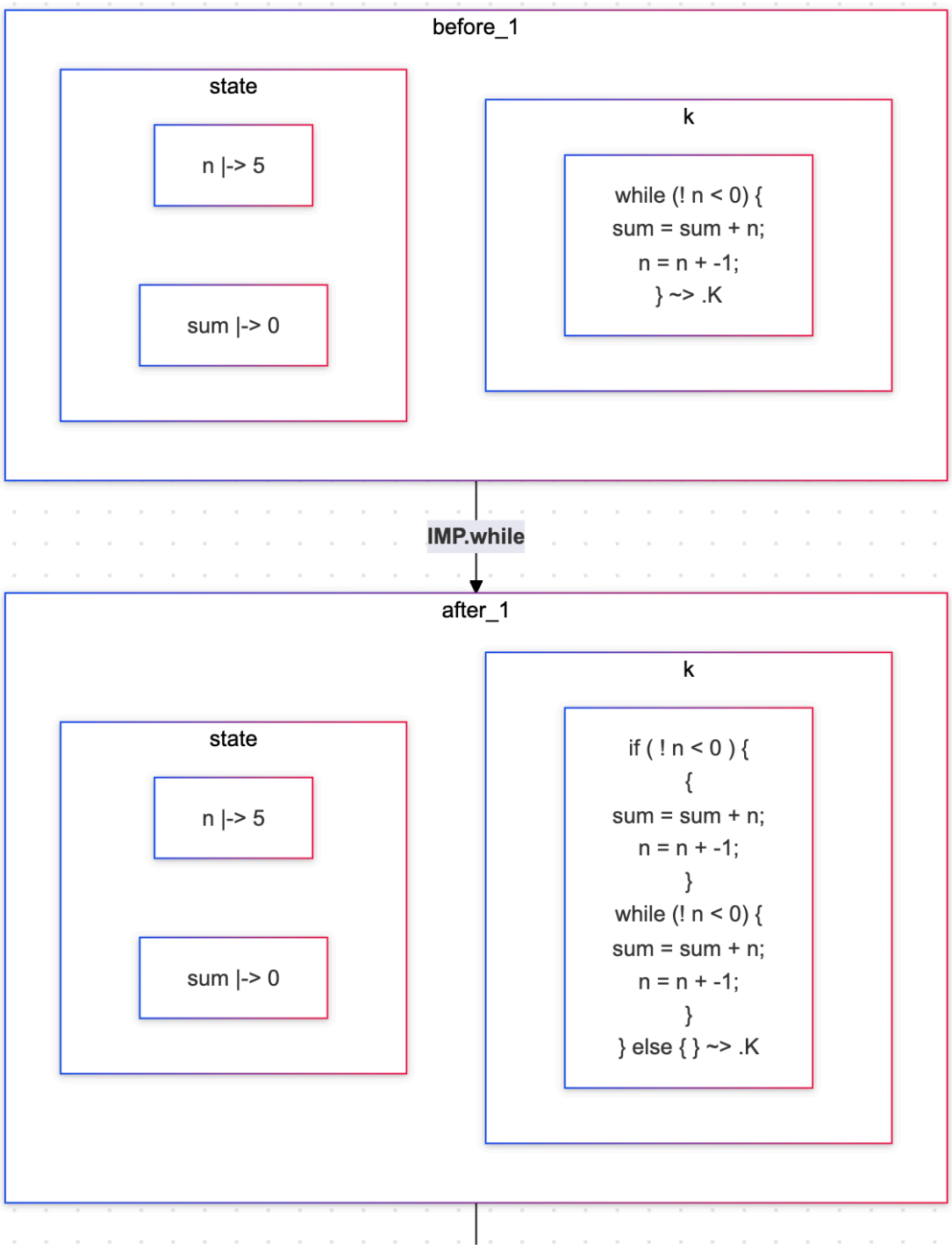
```
<T>
  <k> . </k>
  <state>
    n |-> -1
    sum |-> 15
  </state>
</T>
```

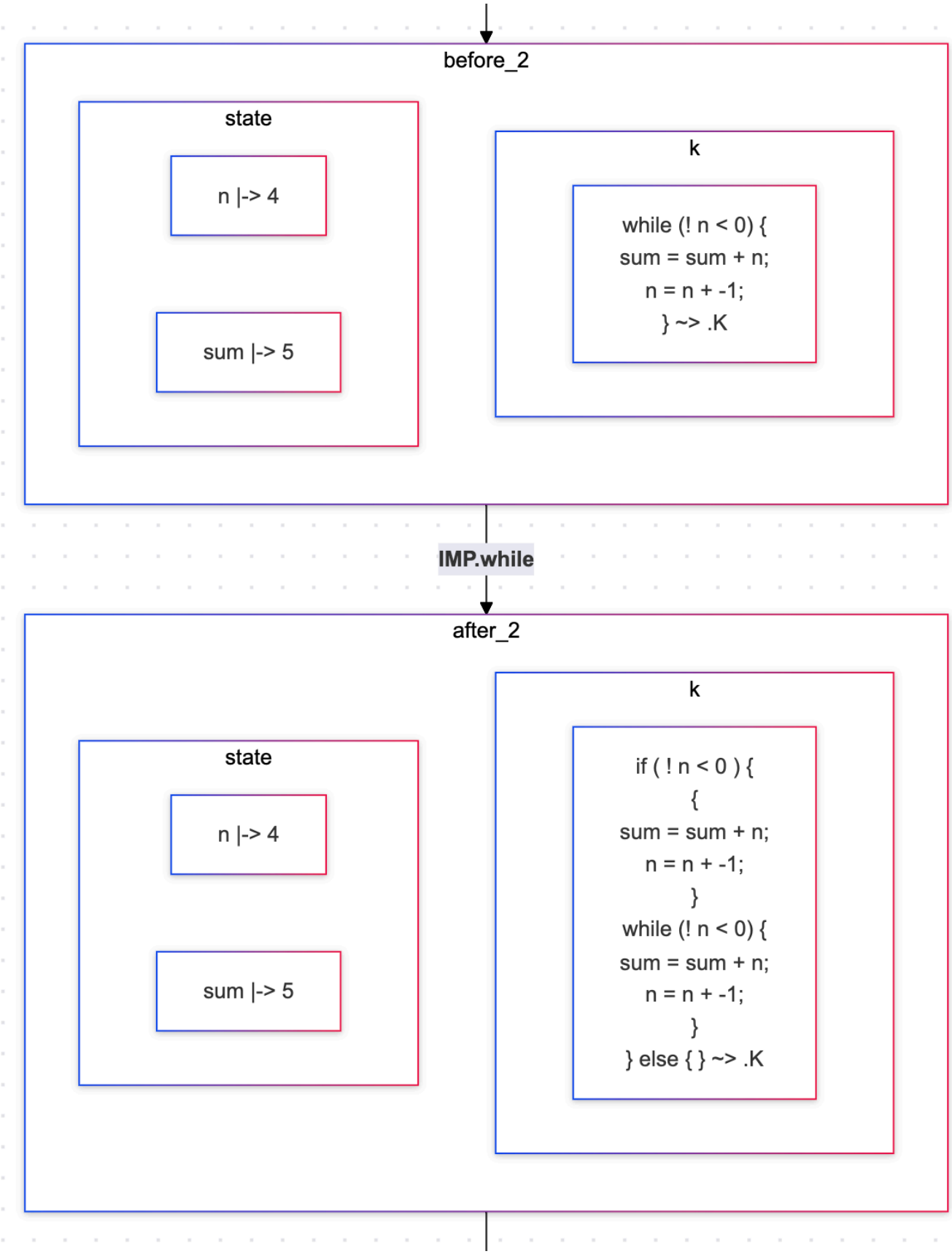
この最終的な configuration に到達するために、K framework では、229 回の書き換え規則が適用される。この 229 回の中には、ユーザーが指定した書き換え規則が含まれているが、先述した値呼びのための書き換え規則や、while 文中に複数回適用される変数の参照や代入の書き換え規則も含まれている。しかし、このプログラムを理解する上で重要なのは、while 文の書き換え規則である。

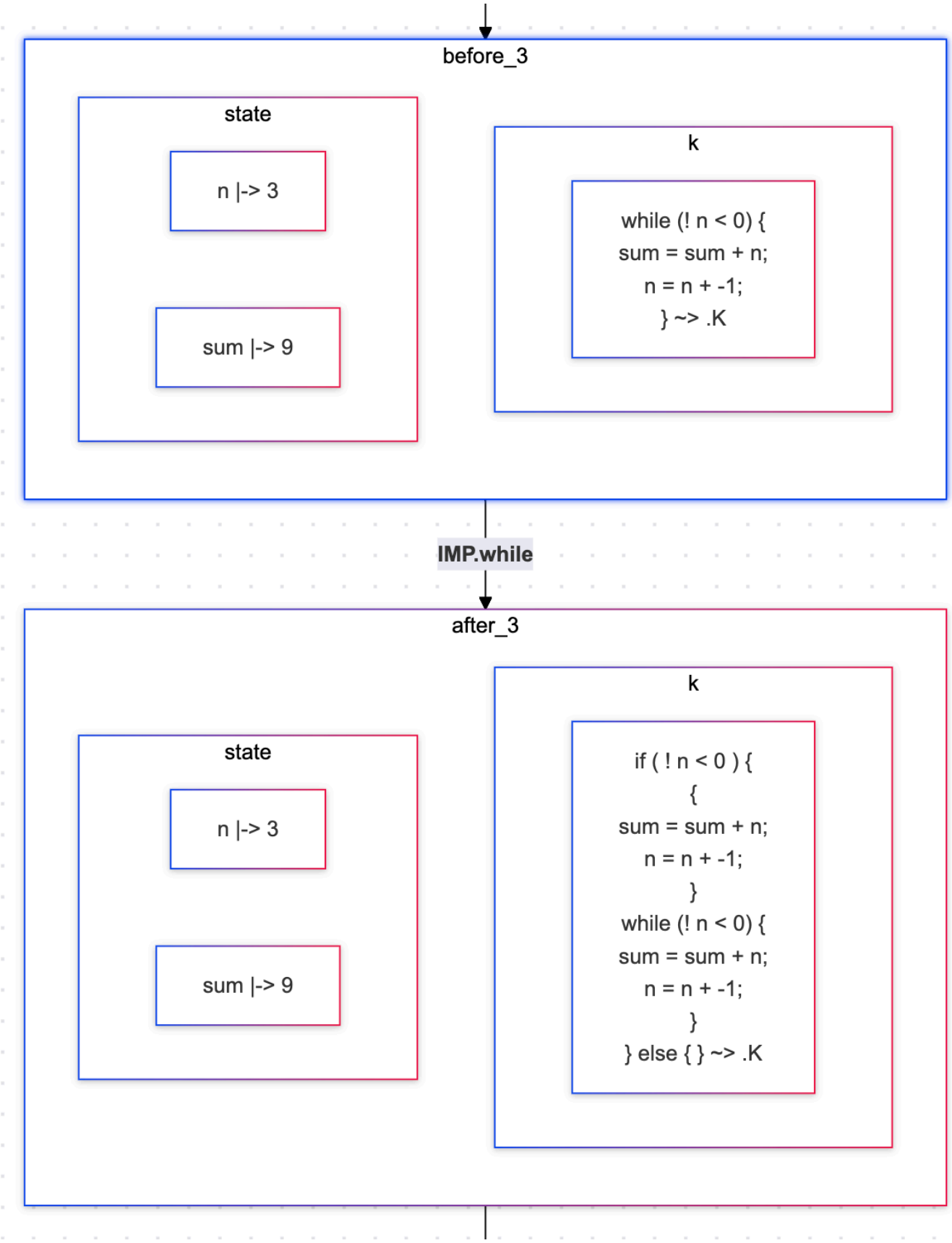
そこで提案する視覚化ツールを用いて while 文の書き換え規則が適用されている部分のみをを視覚化する。IMP の意味論内の while 文の書き換え規則に以下のように名前をつける。

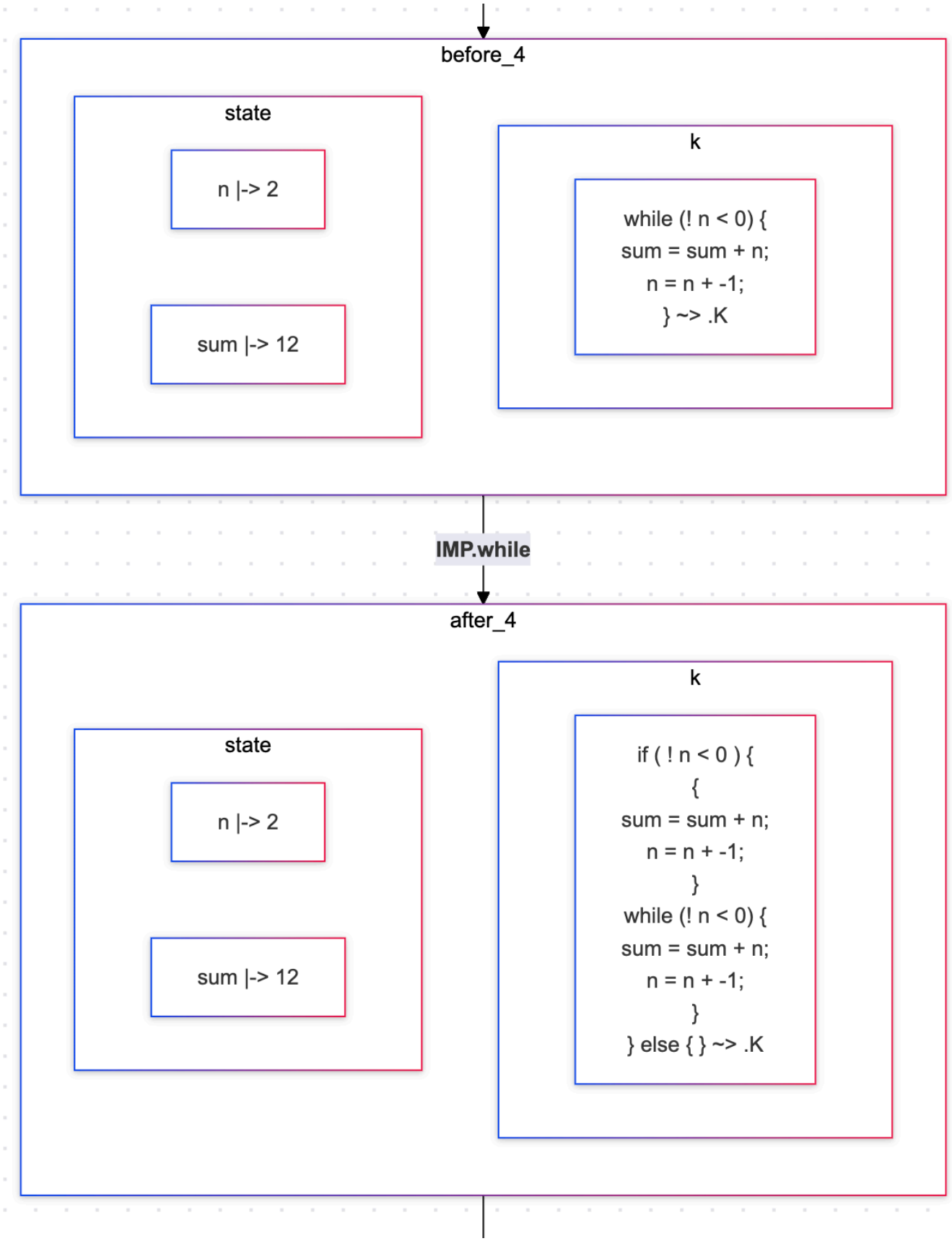
```
rule [while]: while (B) S => if (B) {S while (B) S} else {}
```

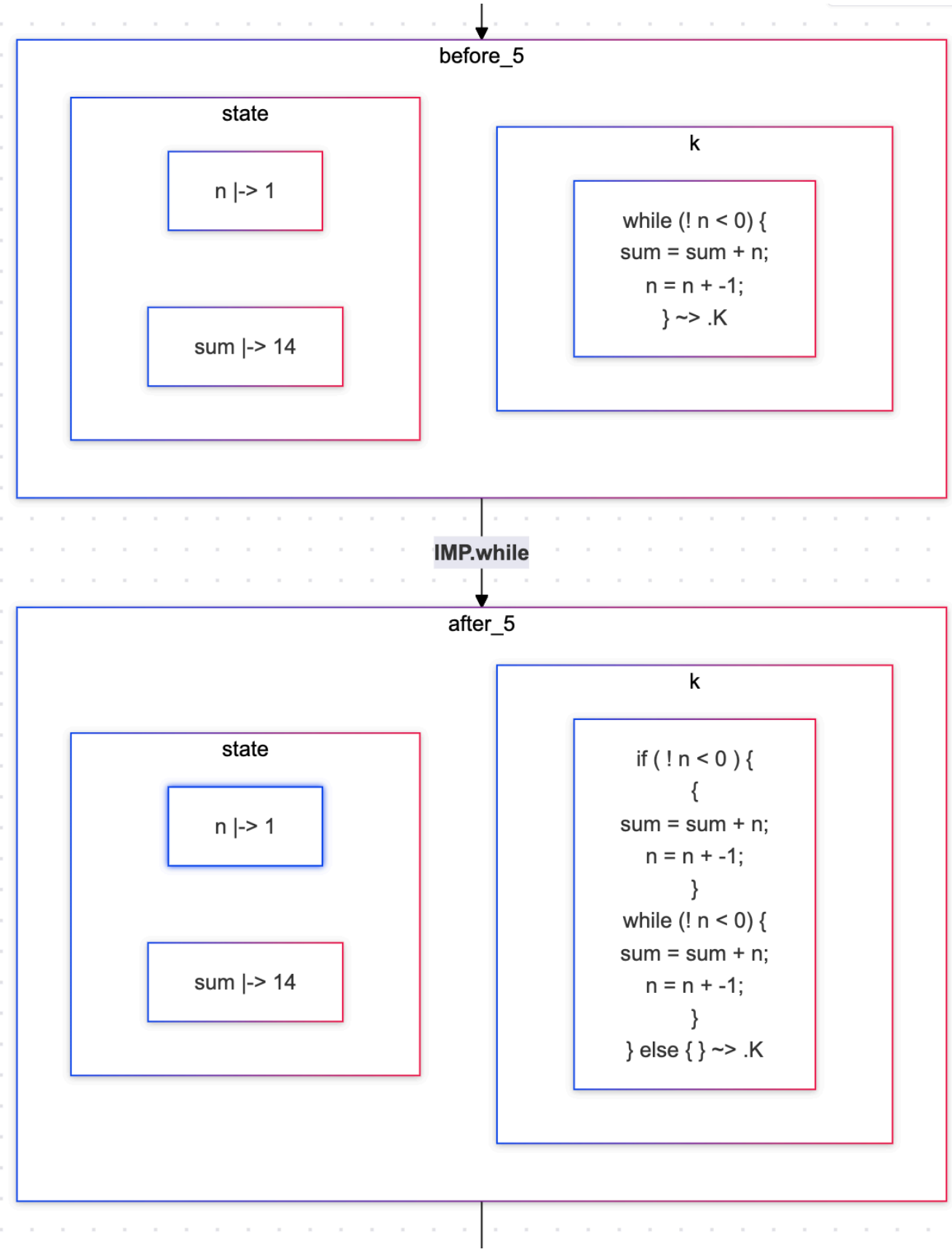
この状態で視覚化ツールを用いて、while 文の書き換え規則が適用されている箇所を視覚化すると以下のような図が生成される。

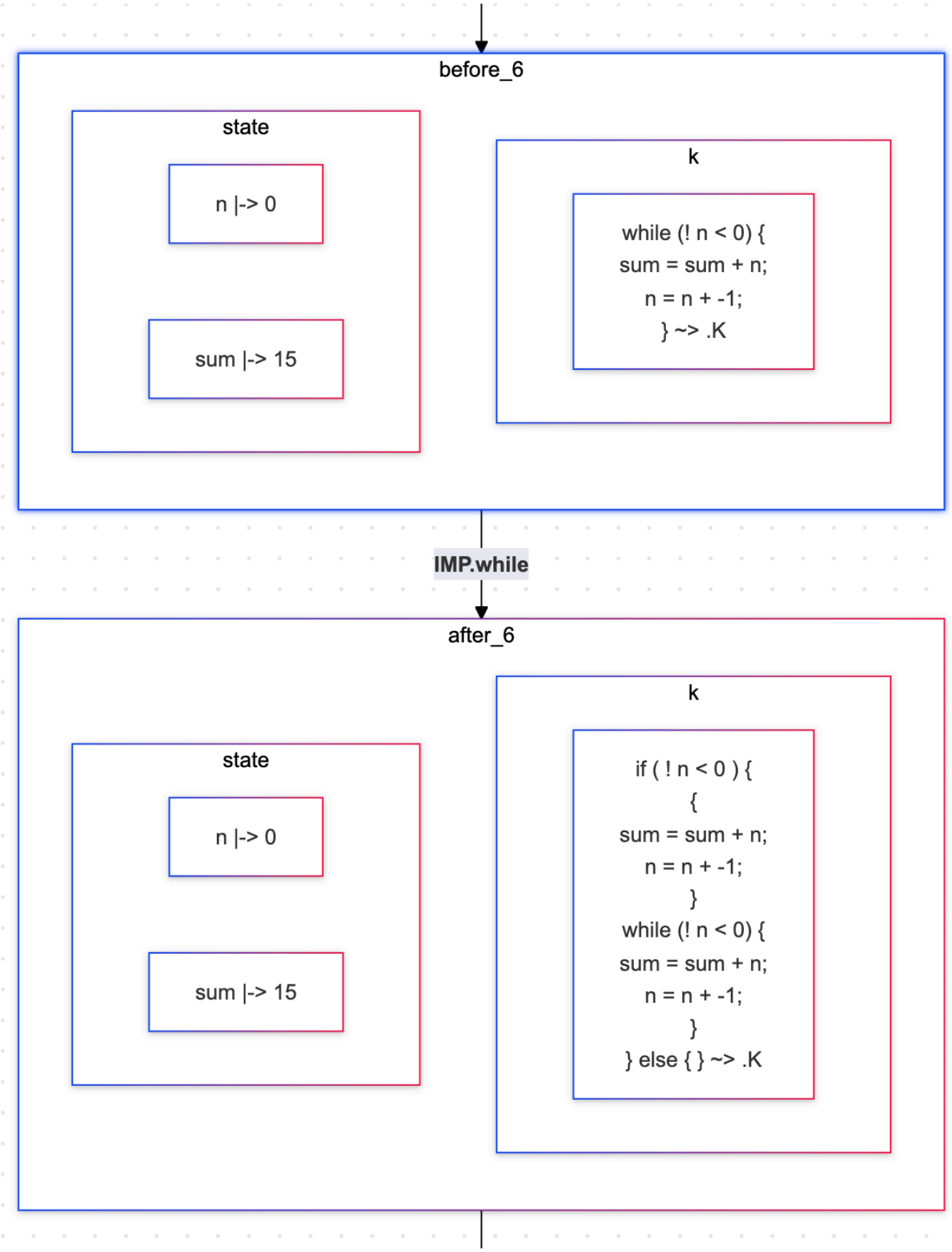


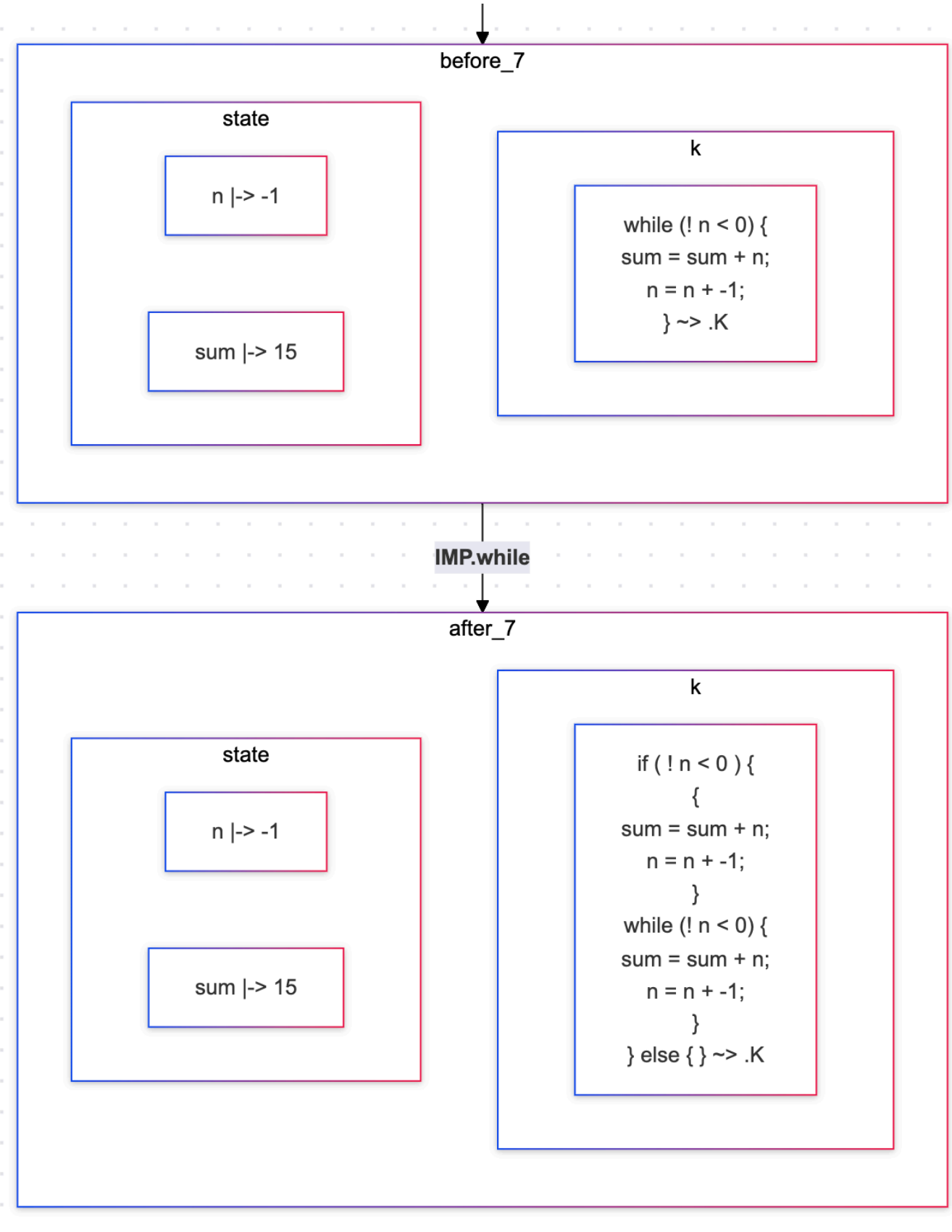

before_2
state
n |-> 4
sum |-> 5
k
while (! n < 0) {
sum = sum + n;
n = n + -1;
} ~> .K
IMP.while
after_2
state
n |-> 4
sum |-> 5
k
if (! n < 0) {
{
sum = sum + n;
n = n + -1;
}
while (! n < 0) {
sum = sum + n;
n = n + -1;
}
} else { } ~> .K


before_3
state
n |-> 3
sum |-> 9
k
while (! n < 0) {
sum = sum + n;
n = n + -1;
} ~> .K
IMP.while
after_3
state
n |-> 3
sum |-> 9
k
if (! n < 0) {
{
sum = sum + n;
n = n + -1;
}
while (! n < 0) {
sum = sum + n;
n = n + -1;
}
} else { } ~> .K


before_4
state
n |-> 2
sum |-> 12
k
while (! n < 0) {
sum = sum + n;
n = n + -1;
} ~> .K
IMP.while
after_4
state
n |-> 2
sum |-> 12
k
if (! n < 0) {
{
sum = sum + n;
n = n + -1;
}
while (! n < 0) {
sum = sum + n;
n = n + -1;
}
} else { } ~> .K


before_5
state
n |-> 1
sum |-> 14
k
while (! n < 0) {
sum = sum + n;
n = n + -1;
} ~> .K
IMP.while
after_5
state
n |-> 1
sum |-> 14
k
if (! n < 0) {
{
sum = sum + n;
n = n + -1;
}
while (! n < 0) {
sum = sum + n;
n = n + -1;
}
} else { } ~> .K


before_6
state
n |-> 0
sum |-> 15
k
while (! n < 0) {
sum = sum + n;
n = n + -1;
} ~> .K
IMP.while
after_6
state
n |-> 0
sum |-> 15
k
if (! n < 0) {
{
sum = sum + n;
n = n + -1;
}
while (! n < 0) {
sum = sum + n;
n = n + -1;
}
} else { } ~> .K


before_7
state
n |-> -1
sum |-> 15
k
while (! n < 0) {
sum = sum + n;
n = n + -1;
} ~> .K
IMP.while
after_7
state
n |-> -1
sum |-> 15
k
if (! n < 0) {
{
sum = sum + n;
n = n + -1;
}
while (! n < 0) {
sum = sum + n;
n = n + -1;
}
} else { } ~> .K

この画像では、while 文の書き換え規則の中でも、sum が初期値の 0、n が初期値の 5 から、sum が 15、n が-1 になるまでの過程が視覚化されている。本来は while 以外にも 229 回の書き換え規則が適用されているが、while 文の書き換え規則のみを視覚化することで、プログラムの意味論を理解する際に重要な書き換え規則が埋もれてしまうことを防ぐことができる。

3.4 視覚化ツールの実装

この節では、K framework の書き換え規則の視覚化ツールの実装の概要を説明する。なお、視覚化ツールプログラムは `https://github.com/QWYNG/KToMD` にて公開している。このツールは、K framework の定義ファイルと実行するプログラムを入力として受け取るコマンドラインツールである。そして定義ファイルから名前が付けられている書き換え規則を抽出する。その後視覚化ツールは K Framework 内のデバッガツール (`https://kframework.org/k-distribution/k-tutorial/1_basic/19_debugging/`) を起動し、全ての書き換えステップに対して、抽出した書き換え規則が適用されるかどうかをデバックして確認する。適用される場合、その前後の configuration の前後の状態を保存する。最後に保存した cofigration の一覧をパースし、Markdown という形式でファイルに出力する。Markdown 形式とは、プレーンテキスト形式で書かれた文書を HTML に変換するための軽量マークアップ言語である。Markdown 形式で出力することで、視覚化ツールで生成した画像をテキストで管理することができ、バージョン管理システムでの差分管理が容易になる。

3.5 まとめ

この章では、K framework の書き換え規則の視覚化ツールについて説明した。このツールは、プログラミング言語設計者が記載した書き換え規則から、特に重要な書き換え規則を指定し、それのみを視覚化するツールである。このツールを用いることで、プログラムの実行過程でどの書き換え規則が適用されたかを視覚化することができる。これにより、プログラムの意味論を理解する際に、重要な書き換え規則が埋もれてしまうことを防ぐことができる。また、実際に while 文の書き換え規則が適用されている箇所を視覚化する例を示し、複数の書き換え規則が適用される際にも、特定の書き換え規則のみを視覚化することができることを示した。

第4章　LAMBDA

この章ではK frameworkを用いてLAMBDAという言語を定義する。LAMBDAは、学習用の関数プログラミング言語である。

4.1　LAMBDAの構文と意味論

LAMBDAの構文と意味論を定義する。なお、K frameworkでは本来、構文定義のモジュールと意味論のモジュールは分けて定義するべきであるが、ここでは説明のために一つのモジュールにまとめて定義する。

4.1.1　LAMBDAの初期状態

LAMBDAにおける初期状態の`configration`は以下のように定義される。

```
configuration <T>
                <k > $PGM:Exp </k>
                <env > .Map </env>
                <store> .Map </store>
              </T>
```

kセルは記述されたプログラムを表すセルである。k セル内の*PGM*はプログラミング言語利用者が作成したプログラムを表す変数である。envセルは変数名とその変数のstore内の位置を対応付けるセルである。storeセルはインデックスと具体的な値の対応を表すセルである。初期状態では、envセル、storeセルは空である。

例えば、変数`X`に値1を束縛する場合には、envセルには`X |-> 0`のように変数名とその変数のstore内の位置が保存され、storeセルには`0 |-> 1`のようにインデックスと具体的な値が保存される。その場合のconfigrationは以下のようになる。

```
configuration <T>
                <k > $PGM:Exp </k>
                <env > X |-> 0 </env>
                <store> 0 |-> 1 </store>
              </T>
```

4.1.2 式と値

この定義では、LAMBDA の式と値が定義されている。LAMBDA において値は整数と真偽値とクロージャである。クロージャについては後の節で説明する。式には、値、変数、関数、関数適用、括弧、負の整数、算術式が含まれる。関数適用は左結合であり、プログラム使用者がむやみに括弧を使わなくてもよいように括弧と優先順位が同じになるよう定義されている。算術式とブール式はどちらも値呼びである。引数の評価順は非決定的である。また LAMBDA において、引数の評価戦略は値呼びである。

```
syntax Val ::= Int | Bool
syntax Exp ::= Val
             | Id
             | "lambda" Id "." Exp
             | Exp Exp               [strict, left]
             | "(" Exp ")"           [bracket]
             > "-" Int
             | Exp "*" Exp           [strict, left]
             | Exp "/" Exp           [strict]
             > Exp "+" Exp           [strict, left]
             > Exp "<=" Exp          [strict]
```

4.1.3 条件分岐

```
syntax Exp ::= "if" Exp "then" Exp "else" Exp  [strict(1)]
rule if true  then E else _ => E
rule if false then _ else E => E
```

この定義では、LAMBDA の条件分岐が定義されている。strict(1) は、条件式のみが値呼びであり、それ以外の部分は名前呼びであることを示している。値呼び、名前呼びとは引数をどのように評価して式に渡すかという評価戦略の名前である。値呼びと名前呼びの違いは、値呼びは引数が評価された後に式が評価されるのに対し、名前呼びは引数が評価される前に式が評価されることである。条件分岐の書き換え規則では、条件が真であれば then 節に進み、偽であれば else 節に進むことを示している。

4.1.4 変数の束縛

```
syntax Exp ::= "let" Id "=" Exp "in" Exp
rule let X = E in E':Exp => (lambda X . E') E
```

let 文は、変数 X に式 E を評価して得られる値を束縛し、式 E' を評価する。let 文は lambda 式への糖衣構文である。例えば、let x = 3 in x + 1 は、(lambda x . x + 1) 3 と等価である。この場合、x には 3 が束縛され、x + 1 が評価される。

4.1.5 クロージャ

LAMBDA ではクロージャを用いて関数を値として扱う。クロージャは、環境、引数、関数本体を持つ。これにより LAMDA 利用者は関数を返す関数や、関数を引数に取る関数を実装することができる。

```
syntax Val ::= closure(Map,Id,Exp)

rule <k> lambda X:Id . E => closure(Rho,X,E) ...</k>
<env> Rho </env>
```

クロージャは lamnda 式を評価すると生成される。クロージャはクロージャが生成された時点での環境 (env セル)、lamda 式の引数名、関数本体を持つ。例えば、`<k> lambda y . x + y </k>, <env> x |-> 3 </env>` は、`closure(x |-> 3, y, x + y)` となる。このクロージャは、x に store 内の 3 をキーとする値が束縛された環境と、引数 y と関数本体 x + y を持つ。

4.1.6 関数適用

関数適用はクロージャに値を適用することで行われる。値にはクロージャも含まれるので、クロージャにクロージャを適用することも可能である。関数適用が行われると、既存の環境を k セルの逐次合成の右辺に退避させておき、クロージャの持つ環境にクロージャの持つ変数名にクロージャを適用する値を対応付けたものを加えた環境をクロージャが持つ関数本体を実行する際に用いる。既存の環境はクロージャ内の関数本体を評価したあとに回復される。

```
rule <k> closure(Rho,X,E) V:Val => E ~> Rho' ...</k>
<env> Rho' => Rho[X <- !N] </env>
<store>... .Map => (!N:Int |-> V) ...</store>
```

例を用いて説明する。以下は x に値 10 が対応している状態で、クロージャ `closure(x |-> 0, y, x + y)` に 4 を適用する例である。

```
<k> closure(x |-> 0, y, x + y) 4 </k>
<env> x |-> 1 </env>
<store>
  0 |-> 3
  1 |-> 10
</store>
```

この例では、まず env セル内の環境 `x |-> 1` が退避され、env セルはクロージャ内の環境 `x |-> 0` に変更される。次に関数適用の右辺である 4 がクロージャ内の変数名 y に対応する値に対応付けられ、env セル内に変数名と store 内の位置、store セル内に store 内の位置と値が対応付けられる。k セルはクロージャの関数本体である x + y を評価するように書き換えられる。

```
<k> x + y ~> x |-> 1</k>
<env>
  x |-> 0
  y |-> 2
</env>
<store>
```

```
  0 |-> 3
  1 |-> 10
  2 |-> 4
</store>
```

次に環境の回復について説明する。環境の回復の書き換え規則は以下のように定義される。

```
rule <k> _:Val ~> (Rho => .K) ...</k> <env> _ => Rho </env>
```

関数適用の評価が終了し値が得られると、環境を回復する。上記の関数適用の例であれば、`x + y`が評価され、`7 ~> x |-> 1`となる。この際に環境の回復の書き換え規則が適用される。環境の回復は、関数適用の際に行われた環境の退避を元に戻す操作である。逐次合成の右辺である `x |-> 1`が空のプログラム本体を表す`.K`に変換され、env セル内の環境が`x |-> 1`に戻る。先述の例以降に環境を回復した場合の`configuration`は以下のようになる。

```
<k> 7 ~> .K </k>
<env> x |-> 1 </env>
<store>
  0 |-> 3
  1 |-> 10
  2 |-> 4
</store>
```

LAMBDA では、関数適用の際に環境を退避し、関数本体を評価した後に環境を回復することで、変数のスコープという概念を実現している。

4.1.7 リスト

LAMBDA ではリストを扱うことができる。リストは値のカンマ区切りである。

```
syntax Vals ::= List{Val,","}
syntax Exp ::= Val
             | "[" Exp "|" Exp "]" [strict]
             | "head" Exp [strict]
             | "tail" Exp [strict]
             | "empty?" Exp [strict]
```

例えば、[1,2,3] は、1,2,3 の値を持つリストである。head [1,2,3] は、1 を返す。tail [1,2,3] は、[2,3] を返す。empty? [1,2,3] は、false を返し、empty? [] は、true を返す。

4.2 視覚化ツールを用いた実行例

以下のコードは、let 式を用いて関数 add3 を定義し、その関数に 4 を適用するプログラムである。

```
let add3 = lambda n . (n + 3) in (add3 4)
```

このプログラムを K framework で定義した LAMBDA の意味論に従って評価すると、最終的な configuration は以下のようになる。

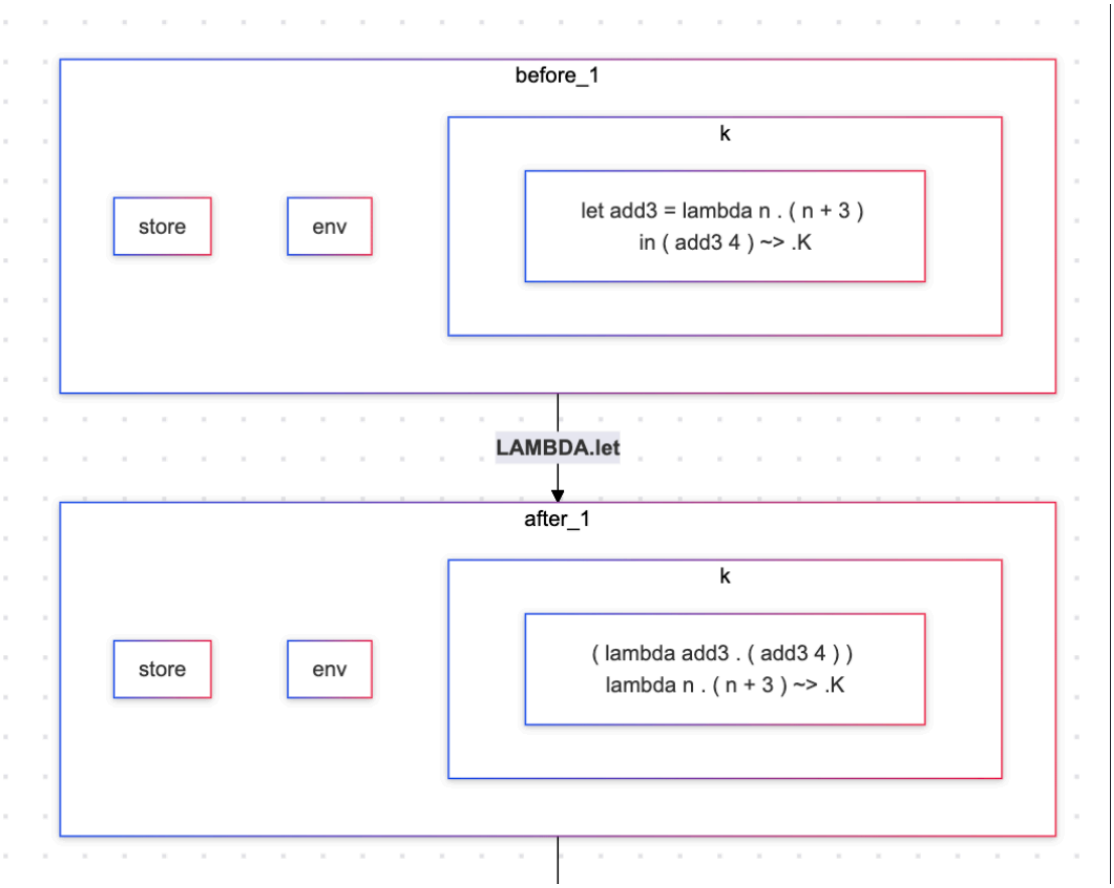
```
<T>
  <k>
    7 ~> .K
  </k>
  <env>
    .Map
  </env>
  <store>
    0 |-> closure ( .Map , n , n + 3 )
    1 |-> 4
  </store>
</T>
```

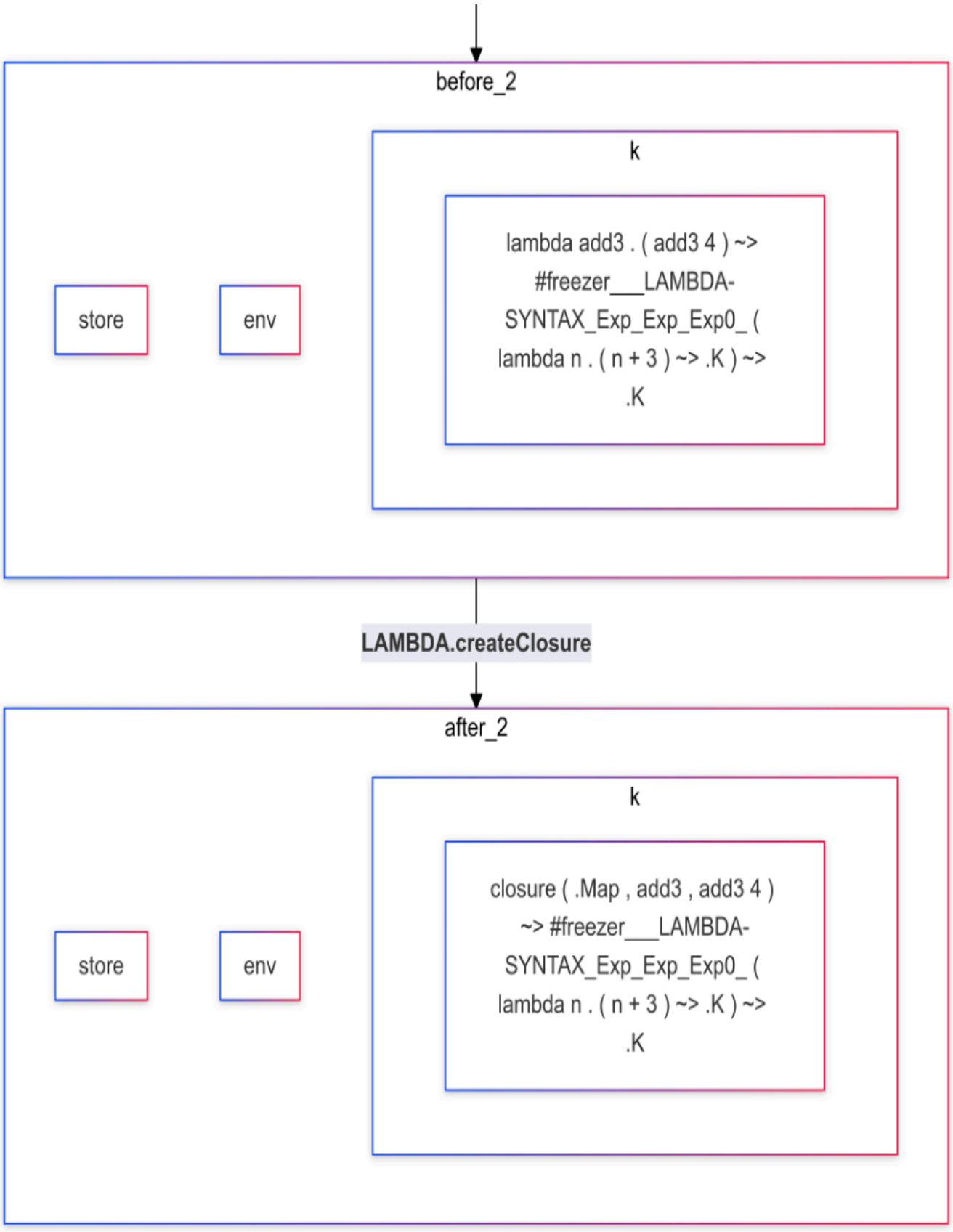
store の 0 番目には let 式で定義した add3 がクロージャとして保存されている。このクロージャは、環境が空であり、引数 n と関数本体 n + 3 を持つ。store の 1 番目には、クロージャの引数 n に束縛された値 4 が保存されている。k セルには add3 に 4 が適用された結果の 7 が残る。環境は最初に let 式が書き換えられたクロージャに add3 の本体のクロージャが適用された際に対比されていた初期状態の空の環境に回復されている。この最終的な configuration に到達するまでには、18 回の書き換え規則が適用される。

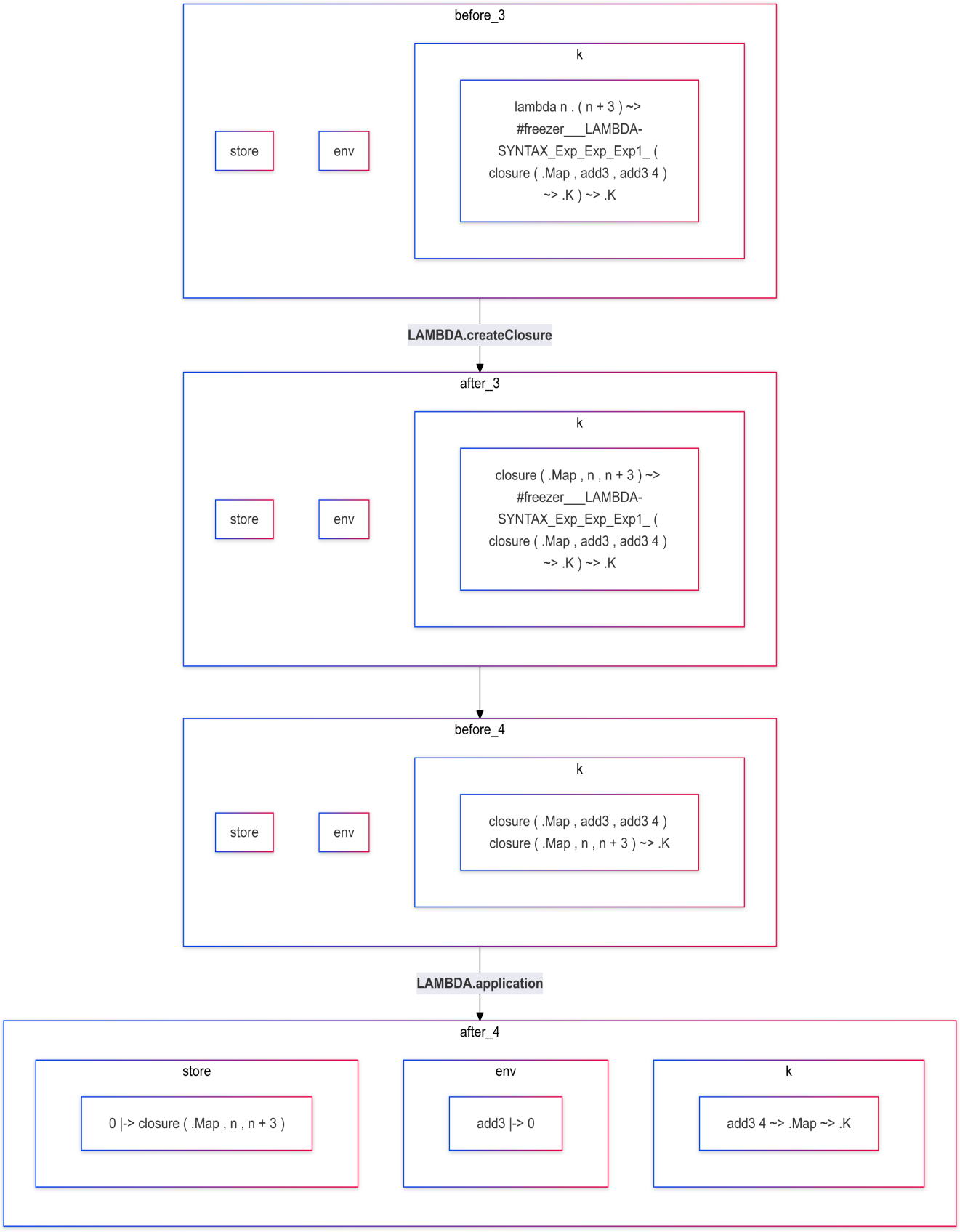
このプログラムにおいて let 式、lambda からクロージャへの変換、関数適用の 3 つの書き換え規則を指定して視覚化を行う。

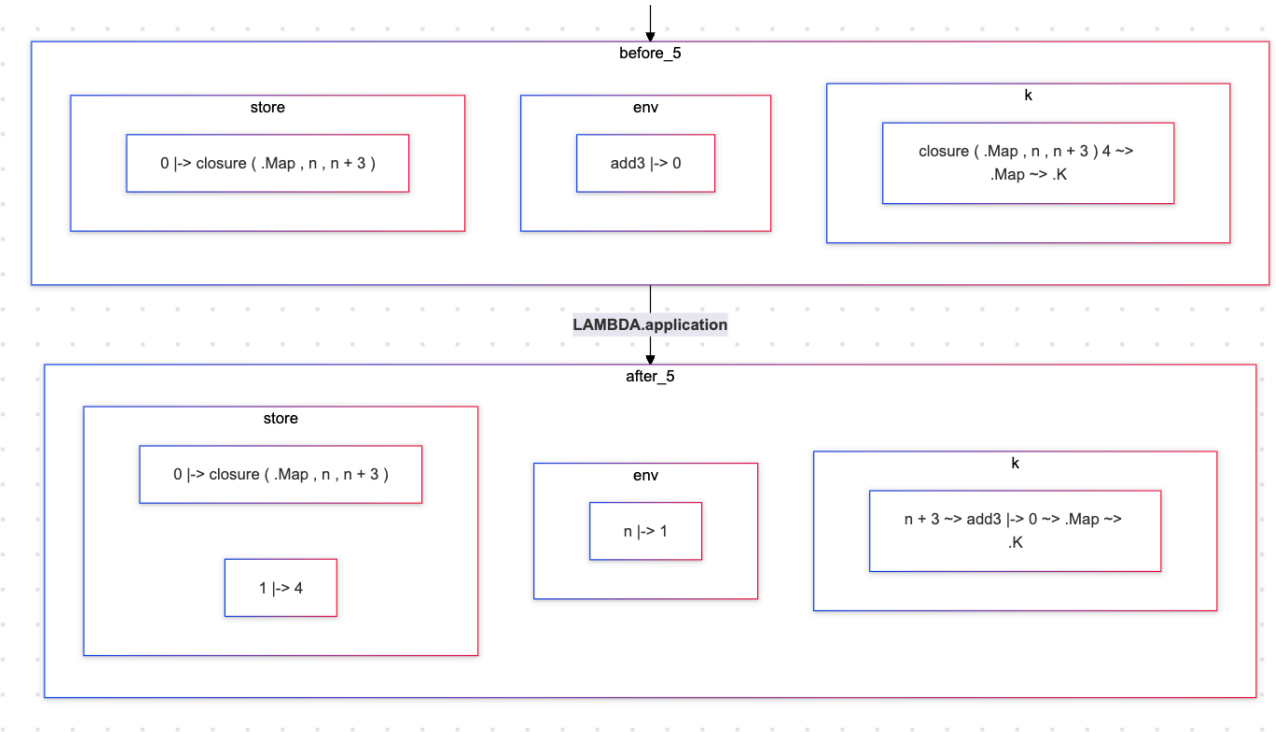
```
rule [let]: let X = E in E':Exp => (lambda X . E') E
rule [createClosure]: <k> lambda X:Id . E => closure(Rho,X,E) ...</k>
                      <env> Rho </env>
rule [application]: <k> closure(Rho,X,E) V:Val => E ~> Rho' ...</k>
     <env> Rho' => Rho[X <- !N] </env>
     <store>... .Map => (!N:Int |-> V) ...</store>
```

視覚化ツールを用いると関数名 add3 に lmanbda 式がクロージャとなって束縛され、そのクロージャに 4 が適用される過程が視覚化される。本来必要な書き換え規則は 18 回であるが、5 回の書き換え規則のみに注目してプログラムの評価過程を観察できる。


before_1
k
store
env
let add3 = lambda n . (n + 3)
in (add3 4) ~> .K
LAMBDA.let
after_1
k
store
env
(lambda add3 . (add3 4))
lambda n . (n + 3) ~> .K


before_2
k
lambda add3 . (add3 4) ~>
#freezer___LAMBDA-
SYNTAX_Exp_Exp_Exp0_ (
lambda n . (n + 3) ~> .K) ~>
.K
store
env
LAMBDA.createClosure
after_2
k
closure (.Map , add3 , add3 4)
~> #freezer___LAMBDA-
SYNTAX_Exp_Exp_Exp0_ (
lambda n . (n + 3) ~> .K) ~>
.K
store
env


before_3
k
lambda n . (n + 3) ~>
#freezer___LAMBDA-
SYNTAX_Exp_Exp_Exp1_ (
closure (.Map , add3 , add3 4)
~> .K) ~> .K
store
env
LAMBDA.createClosure
after_3
k
closure (.Map , n , n + 3) ~>
#freezer___LAMBDA-
SYNTAX_Exp_Exp_Exp1_ (
closure (.Map , add3 , add3 4)
~> .K) ~> .K
store
env
before_4
k
closure (.Map , add3 , add3 4)
closure (.Map , n , n + 3) ~> .K
store
env
LAMBDA.application
after_4
store
0 |-> closure (.Map , n , n + 3)
env
add3 |-> 0
k
add3 4 ~> .Map ~> .K



視覚化を行うことで、let 文自体が env や store に要素を挿入することなく、lambda 式に変換されることがセルの画像の大きさから視覚的に理解可能である。また、lambda 式がクロージャに変換され、引数 add3 をもつクロージャにクロージャを適用することで、add3 が env と store に追加されることがわかる。環境の回復について、まず最初の関数適用である let 式が書き換えられた lambda 式に add3 本体が適用される際に退避されていた空の環境が最終的に回復されることがわかる。

4.3 再帰定義

LAMBDA では再帰も可能である。以下は再帰定義のための構文定義と書き換え規則である。

```
syntax Exp ::= "letrec" Id Id "=" Exp "in" Exp
             | | "mu" Id "." Exp
syntax Val ::= recClosure(Map, Exp)

rule letrec F:Id X = E in E' => let F = mu F . lambda X . E in E'

rule <k> mu F . E => recClosure(Rho[F <- !N], E) ...</k>
      <env> Rho </env>
      <store>... .Map => (!N:Int |-> recClosure(Rho[F <- !N], E)) ...</store>

rule <k> recClosure(Rho, E) => E ~> Rho' ...</k>
      <env> Rho' => Rho </env>
```

letrec 文は再帰関数定義を行う。mu 式は再帰関数を表す。mu 式は recClosure に変換される。recClosure は環境とパラメータを含めた関数本体を持つ。recClosure の

環境には recClosure 自身への参照が含まれる。これにより再帰関数定義が可能となる。

4.3.1 再帰関数の実行の視覚化

再帰的な定義について説明する。map 関数は、与えられた関数 f をリストの各要素に適用し、その結果から新しいリストを構築する高階関数である。この操作は以下の手続きによって実現される：

1. リストが空である場合、結果として空のリストを返す。
2. リストが空でない場合、リストの先頭要素に関数 f を適用し、その結果をリストの先頭要素とする。次にリストの残りの部分に対して再帰的に map を適用し、それをリストの残りの部分とする。

具体的なプログラムは以下のようになる。

```
letrec map f = lambda list . if (empty? list)
                               then []
                               else [f(head list) | map f (tail list)]
  in let add3 = lambda n . (n + 3)
    in (map add3 [1, 2, 3])
```

このプログラムは、map 関数を再帰関数定義を用いて定義し、map 関数に add3 関数と [1,2,3] を適用するプログラムである。add3 は引数に 3 を加える関数である。

このプログラムを LAMBDA の意味論に従って評価すると最終的な configuration は以下のようになる。

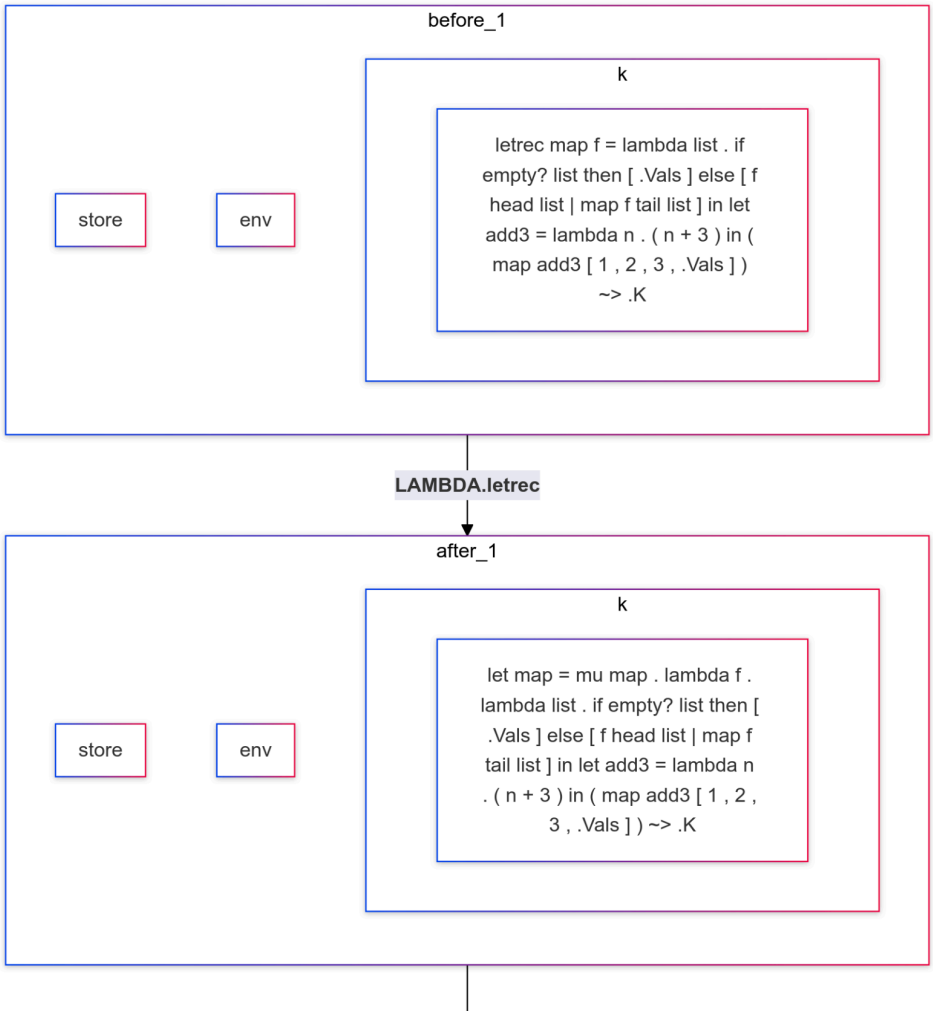
```
<T>
  <k>
    [ 4 , 5 , 6 , .Vals ] ~> .K
  </k>
  <env>
    .Map
  </env>
  <store>
    0 |-> recClosure ( map |-> 0 , lambda f . lambda list . if empty? list then [
         .Vals ] else [ f head list | map f tail list ] )
    1 |-> closure ( map |-> 0 , f , lambda list . if empty? list then [ .Vals ]
        else [ f head list | map f tail list ] )
    2 |-> closure ( map |-> 1 , n , n + 3 )
    3 |-> closure ( map |-> 1 , n , n + 3 )
    4 |-> [ 1 , 2 , 3 , .Vals ]
    5 |-> 1
    6 |-> closure ( map |-> 1 , n , n + 3 )
    7 |-> [ 2 , 3 , .Vals ]
    8 |-> 2
    9 |-> closure ( map |-> 1 , n , n + 3 )
    10 |-> [ 3 , .Vals ]
    11 |-> 3
    12 |-> closure ( map |-> 1 , n , n + 3 )
    13 |-> [ .Vals ]
  </store>
</T>
```

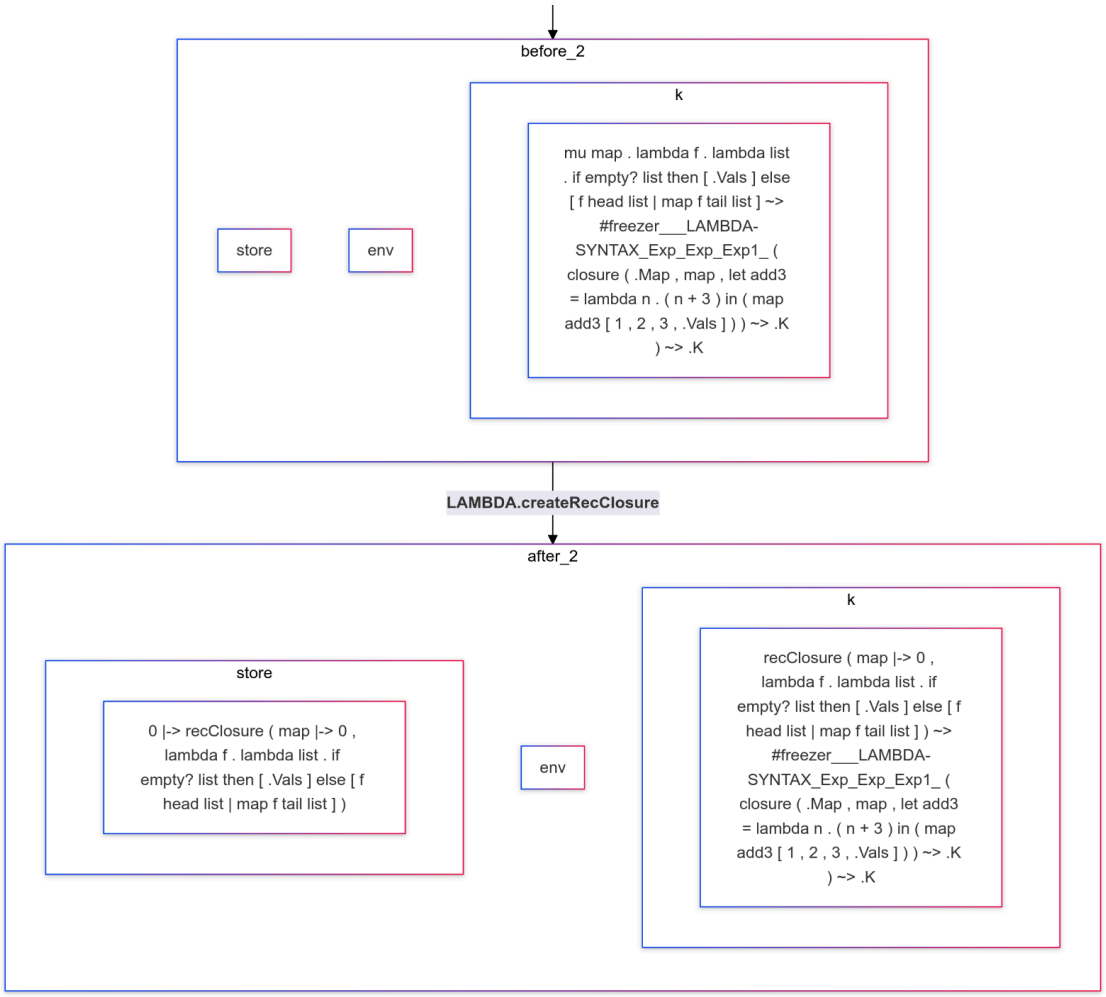
再帰関数の適用を行っているので、store 内には add3 が複数回 map の引数 f に束縛されている。また、add3 の引数 n にリストの要素が順番に束縛されている。この最終的な configuration に到達するまでには、189 回の書き換え規則が適用される。

視覚化ツールを用いて再帰的な関数定義が適用される箇所を視覚化する。視覚化する書き換え規則は先述した letrec 式と mu 式の 2 つを指定する。

```
rule [letrec]: letrec F:Id X = E in E' => let F = mu F . lambda X . E in E'
rule [createRecClosure]: <k> mu F . E => recClosure(Rho[F <- !N], E) ...</k>
     <env> Rho </env>
     <store>... .Map => (!N:Int |-> recClosure(Rho[F <- !N], E)) ...</store>
```

視覚化ツールを用いて、map 関数の再帰関数定義が適用されている箇所を視覚化すると以下のような図が生成される。





この画像では、`letrec` が `mu` に変換され、`mu` が `recClosure` に変換される過程が視覚化されている。store に何も影響を及ぼさない lambda 式からクロージャへの変換とは異なり、再帰関数定義の mu 式は store 内に recClosure を生成する。store 内に生成された recClosure は、k セル内の recClosure に値が適用されることで、recClosure がもつ環境 `map |-> 0` によって利用されることになる。このように、再帰関数定義の適用過程を視覚化することで、通常の let 式の動作と letrec 式の動作を比較し違いを確認することができる。

4.4 まとめ

この章では、K framework を用いて LAMBDA という言語を定義した。LAMBDA は、学習用の関数プログラミング言語であり、関数適用、条件分岐、変数の束縛、クロージャ、リスト、再帰定義を持つ。
また、視覚化ツールを用いて LAMBDA の let 式、letrec 式の書き換え規則に注目し、その書き換え規則のみを視覚化することで、大量の書き換え規則の中から観

察したい書き換え規則のみを観察することができることを示した。これは、意味論を理解する必要のある者が、大量の書き換え規則の中から特定の書き換え規則を見つける際に有用である。また、書き換え規則適用の前後の configuration を示すことで書き換え規則によって、どのように状態が変更されるのかを比較して確認することができることを示した。configurationno 各セルの内部のアイテムの追加や削除や更新についても視覚化されるため、書き換え規則の適用によってどのような変化が生じるのかを視覚的に理解することができる。

第5章　CLASS

この章ではK frameworkを用いてCLASSという言語を定義する。

5.1　CLASSの構文と意味論

CLASSはIMPのスーパーセットであり、IMPの機能に加えてクラスとオブジェクト指向の機能を持つ。オブジェクト指向のプログラミングでは、オブジェクトが持つデータとそのデータを操作する関数を一つのまとまりとして扱う。これにより、プログラムの再利用性や保守性が向上する。オブジェクトが持つ関数をメソッド、データをメンバ変数と呼ぶ。クラスとは、オブジェクトの設計図である。オブジェクトはクラスのインスタンスであり、クラスの設計図に基づいて作られる。オブジェクトは生成されると、クラスの設計図に基づいてデータと関数を持つ。

はじめにCLASSのconfigrationの初期状態を定義する。

```
configuration <T>
              <k> $PGM:Stmt ~> execute </k>
              <control>
                <fstack> .List </fstack>
                <crntObj>
                    <crntClass> Object </crntClass>
                    <envStack> .List </envStack>
                    <location multiplicity="?"> .K </location>
                </crntObj>
              </control>
              <env> .Map </env>
              <store> .Map </store>
              <nextLoc> 0 </nextLoc>
              <classes>
                  <classData multiplicity="*" type="Map" >
                    <className > Main </className>
                    <baseClass > Object </baseClass>
                    <declarations > .K </declarations>
                  </classData>
              </classes>
              </T>
```

CLASSのconfigurationは大きく3つに分けられる。1つめは、評価中のプログラムである。これは`k`セルで表される。

2つめは、評価中のプログラムのための環境である。これは`control`セル、`env`セル、`store`セル、`nextLoc`セルで表される。`env`セルは変数名とその変数のstore内の位置を対応付けるセルである。`control`セルはメソッドが呼び出された場合に、のちの処理を退避しておくためのスタックである`fstack`セルと、現在のオブジェクトのクラスをあらわす`crntClass`セルと、次にオブジェクトが生成され

る場所を示す nextLoc セルを持つセルである。オブジェクトが生成されるたびに nextLoc セルはインクリメントされる。CLASS では、オブジェクト指向の機能を持つため、現在実行しているプログラムがどのオブジェクトのコンテキストで実行されているかを示す crntObj セルが必要である。

3 つめは、定義されたクラスの情報である。これは classes セルで表される。classes セルは、クラスの名前 (className、継承元のクラス (baseClass)、クラス内部の宣言 (declaration) を持つ classData セルのリストである。classData セルの multiplicity = "*"という設定は、classData セルが複数存在可能なことを示している。これは CLASS のプログラム内で複数のクラスを定義できることを示している。

5.2 CLASS の構文と書き換え規則

5.2.1 クラスの定義

CLASS においてクラスは以下のように定義される。

```
class A {
  var x;

  method A() {
      x = "A Var";
  }
}
```

class <クラス名> <宣言> という形式でクラスを定義する。クラス内部では変数宣言とメソッド定義が可能である。クラス名と同じ名前のメソッドはコンストラクタである。コンストラクタについては後で説明する。

この例では、クラス A を宣言し、クラスのメンバ変数 x を宣言し、コンストラクタ A を定義している。メンバ変数はクラス内のメソッドで参照することができる。

class 構文及び書き換え規則は以下のように定義される。

```
syntax Stmt ::= "class" Id Block
              | "class" Id "extends" Id Block

rule class C:Id S => class C extends Object S

rule <k> class Class1 extends Class2 { S } => .K ...</k>
      <classes>... (.Bag => <classData>
                              <className> Class1 </className>
                              <baseClass> Class2 </baseClass>
                              <declarations> S </declarations>
                            </classData>)
      ...</classes>
```

class 構文は明示的に extends を用いない場合、すべてのクラスは Object クラスを継承する。Object クラスは CLASS の組み込みクラスであり、すべてのクラスは Object クラスを継承する。class C extends Object S は、クラス C が Object クラスを継承し、クラス内部の宣言 S を持つことを示している。class C extends

Object S が書き換えられると、classes セルにクラス情報が追加される。.Bag はリスト内の要素にセル構造を追加するための K framework の構文である。

5.2.2 オブジェクトの生成

CLASS においてオブジェクトは以下のように生成される。

```
var obj;
obj = new A();
```

new <クラス名>() という形式でオブジェクトを生成する。オブジェクト生成構文及び書き換え規則は以下のように定義される。

```
syntax Exp ::= "new" Id "(" Exps ")"   [strict(2)]

rule <k> new Class:Id(Vs:Vals) ~> K
          => create(Class) ~> storeObj ~> Class(Vs); return this; </k>
     <env> Env => .Map </env>
     <nextLoc> L:Int => L +Int 1 </nextLoc>
     <control>
        <crntObj> Obj
                  => <crntClass> Object </crntClass>
                     <envStack> ListItem(envStackFrame(Object, .Map)) </envStack>
                     <location> L </location>
        </crntObj>
        <fstack> .List =>
         ListItem(fstackFrame(Env, K, <crntObj> Obj </crntObj>))
         ...</fstack>
     </control>
```

new Class:Id(Vs:Vals) は、コンストラクタを呼び出すための構文である。この構文は create(Class) ~> storeObj ~> Class(Vs); return this; に書き換えられる。それぞれの書き換え規則の内容は後述する。
また new Class:Id(Vs:Vals) が評価されると、現在のオブジェクトを表す crntObj セルが、Object クラスに設定され、環境スタックに Object クラスの環境が追加される。Object クラスの環境は空であるため、.Map が設定される。この操作はオブジェクトクラスのインスタンスを生成するための処理であり、インスタンスの生成であるので、nextLoc セルはインクリメントされ、location セルには nextLoc セルのインクリメント前の値が設定される。fstack セルには、コンストラクタが呼び出される処理が追加される。

create(Class) の書き換え規則は以下のように定義される。

```
rule <k> create(Class:Id)
          => create(Class1) ~> setCrntClass(Class) ~> S ~> addEnvLayer ...</k>
     <className> Class </className>
     <baseClass> Class1:Id </baseClass>
     <declarations> S </declarations>

rule <k> setCrntClass(C) => .K ...</k>
      <crntClass> _ => C </crntClass>

rule <k> addEnvLayer => .K ...</k>
     <env> Env => .Map </env>
     <crntClass> Class:Id </crntClass>
     <envStack> .List => ListItem(envStackFrame(Class, Env)) ...</envStack>
```

`create(Class)` は、crntClass を Class に変更したのち、Class の宣言された内容を評価する。そして環境に現在のクラスに関する情報を Stack しておく。Class の宣言された内容を評価することで、クラス内の変数やメソッドが envStack に追加される。envStack はプログラム全体の環境ではなく、オブジェクトの環境を保持するためのスタックであることに注意する。また `create(Class)` は、クラスの継承元のクラスに対して再帰的に実行される。これにより、Class が継承しているクラスすべての objectClosure が envStack に追加される。

storeObj は crntObj を store に追加する書き換え規則である。storeObj の書き換え規則は以下のように定義される。

```
rule <k> storeObj => .K ...</k>
     <crntObj>
       <crntClass> CC </crntClass>
       <envStack> ES </envStack>
      (<location> L:Int </location> => .Bag)
    </crntObj>
     <store>... .Map => L |-> objectClosure(CC, ES) ...</store>
```

`new Class:Id(Vs:Vals)` の時点で設定していた Location に対して objectClosure を store に追加する。objectClosure はクラス名と環境スタックを持つ。この環境スタックは先述の `create(Class)` で追加されたものである。クラス内の変数やメソッドを保持するための環境スタックである。

this は crntObj を objectClosure に変換する書き換え規則である。this の書き換え規則は以下のように定義される。

```
rule <k> this => objectClosure(CC, ES) ...</k>
      <crntObj> <crntClass> CC </crntClass> <envStack> ES </envStack> </crntObj>
```

this は crntObj を objectClosure に変換する。ここまでの処理により、
`new Class:Id(Vs:Vals)` が評価されると、継承されたクラスそれぞれの objectClosure が store に追加され、プログラム上では objectClosure が返されることがわかる。

5.2.3 メソッドとメンバの定義と呼び出し

CLASS においてメソッドとメンバは以下のように定義される。

```
class A {
  var x;
  method A() {
      x = "A Var";
  }

  method add1(n) {
      return n + 1;
  }
}
```

`method` <メソッド名>(引数) <宣言> という形式でメソッドを定義する。メソッド内部では return 文を用いて値を返すことができる。構文規則と書き換え規則は以下のように定義される。

```
syntax Stmt ::= "method" Id "(" Ids ")" Block

rule <k> method F:Id(Xs:Ids) S => .K ...</k>
     <crntClass> Class:Id </crntClass>
     <location> OL:Int </location>
     <env> Env => Env[F <- L] </env>
     <store>... .Map => L |-> methodClosure(Class,OL,Xs,S) ...</store>
     <nextLoc> L => L +Int 1 </nextLoc>
```

method F:Id(Xs:Ids) S は、メソッド F は引数 Xs を持つとと共に、メソッド本体 S を持つことを示している。これが書き換えられると、store に methodClosure が追加される。methodClosure は定義されたクラス名、オブジェクトの Location、引数、メソッド本体を持つ。またプログラム全体の次の store への Location がインクリメントされる。

メンバの定義を扱う規則は以下のように定義される。

```
syntax Stmt ::= "var" Id

rule <k> var X:Id; => .K ...</k>
     <env> Env => Env[X <- L] </env>
     <store>... .Map => L |-> undefined ...</store>
     <nextLoc> L => L +Int 1 </nextLoc>
```

メンバの場合は store に値を undefined として保存される。また環境に store 内の場所を保存する。オブジェクトを生成する際には、store に objectClosure を保存するが、その objectClosure 内の環境にはメンバの名前と store 内の場所が保存されている。

メソッドの呼び出しを扱う規則は以下のように定義される。

```
syntax Exp ::= Exp "." Id

rule (objectClosure(_, EStack) . X
  => lookupMember(EStack, X:Id))(_:Exps)

rule lookupMember(ListItem(envStackFrame(_, X|->L _)) _, X)
  => lookup(L)

rule <k> lookup(L) => V ...</k> <store>... L |-> V:Val ...</store>
rule <k> (lookup(L) => V)(_:Exps) ...</k>  <store>... L |-> V:Val ...</store>
```

objectClosure のメンバを呼び出す際には、objectClosure 内の環境スタックからメンバの場所を取得する。メンバの場合は値がそのまま store に保存されているため、store から値を取得することでメンバの値を取得することができる。メンバの呼び出しに () がついており、store に methodClosure が保存されている場合、その methodClosure が評価される。

メソッドに引数を渡して呼び出す規則は以下のように定義される。

```
rule <k> methodClosure(Class,OL,Xs,S)(Vs:Vals) ~> K
          => mkDecls(Xs,Vs) S return; </k>
      <env> Env => .Map </env>
      <store>... OL |-> objectClosure(_, EnvStack)...</store>
      <control>
        <fstack> .List => ListItem(fstackFrame(Env, K, <crntObj> Obj' </crntObj>)
            )
        ...</fstack>
        <crntObj> Obj' => <crntClass> Class </crntClass> <envStack> EnvStack </
            envStack> </crntObj>
      </control>
```

methodClosure が評価されると、引数を変数として宣言し、メソッド本体を評価する。メソッド本体が評価されると return 文が評価され、メソッドの実行が終了する。この際関数のスタックにメソッド実行前の環境が保存される。メソッドを実行すると変数のスコープが切り替わることになる。その後現在のオブジェクトを methodClosure が保存していたオブジェクトに戻す。

5.2.4 実行例と視覚化

CLASS の視覚化ツールを用いて、オブジェクト生成、メソッド呼び出しを確認する。

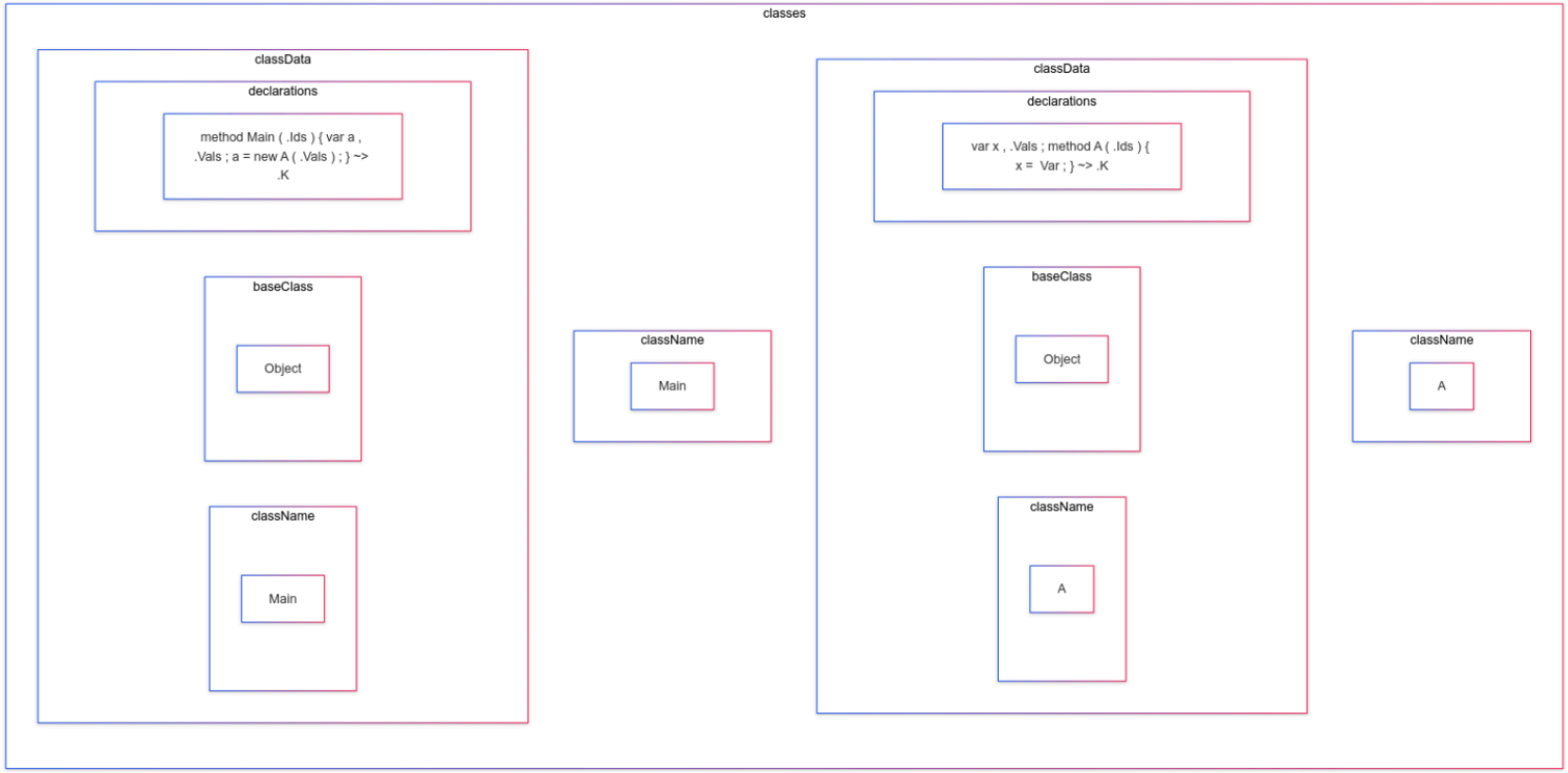
```
  class A {
    var x;
    method A() {
        x = "A Var";
    }

    method add1(n) {
        return n + 1;
    }
}

class Main {
    method Main() {
        var a;
        a = new A();
        print(a.x);
        print(a.add1(2));
  }
}
```

指定する書き換え規則は以下のとおりである。

```
  rule [new]: <k> new Class:Id(Vs:Vals) ~> K
  => create(Class) ~> storeObj ~> Class(Vs); return this; </k>
<env> Env => .Map </env>
<nextLoc> L:Int => L +Int 1 </nextLoc>
<control>
<crntObj> Obj
          => <crntClass> Object </crntClass>
            <envStack> ListItem(envStackFrame(Object, .Map)) </envStack>
            <location> L </location>
</crntObj>
<fstack> .List =>
 ListItem(fstackFrame(Env, K, <crntObj> Obj </crntObj>))
 ...</fstack>
</control>
```

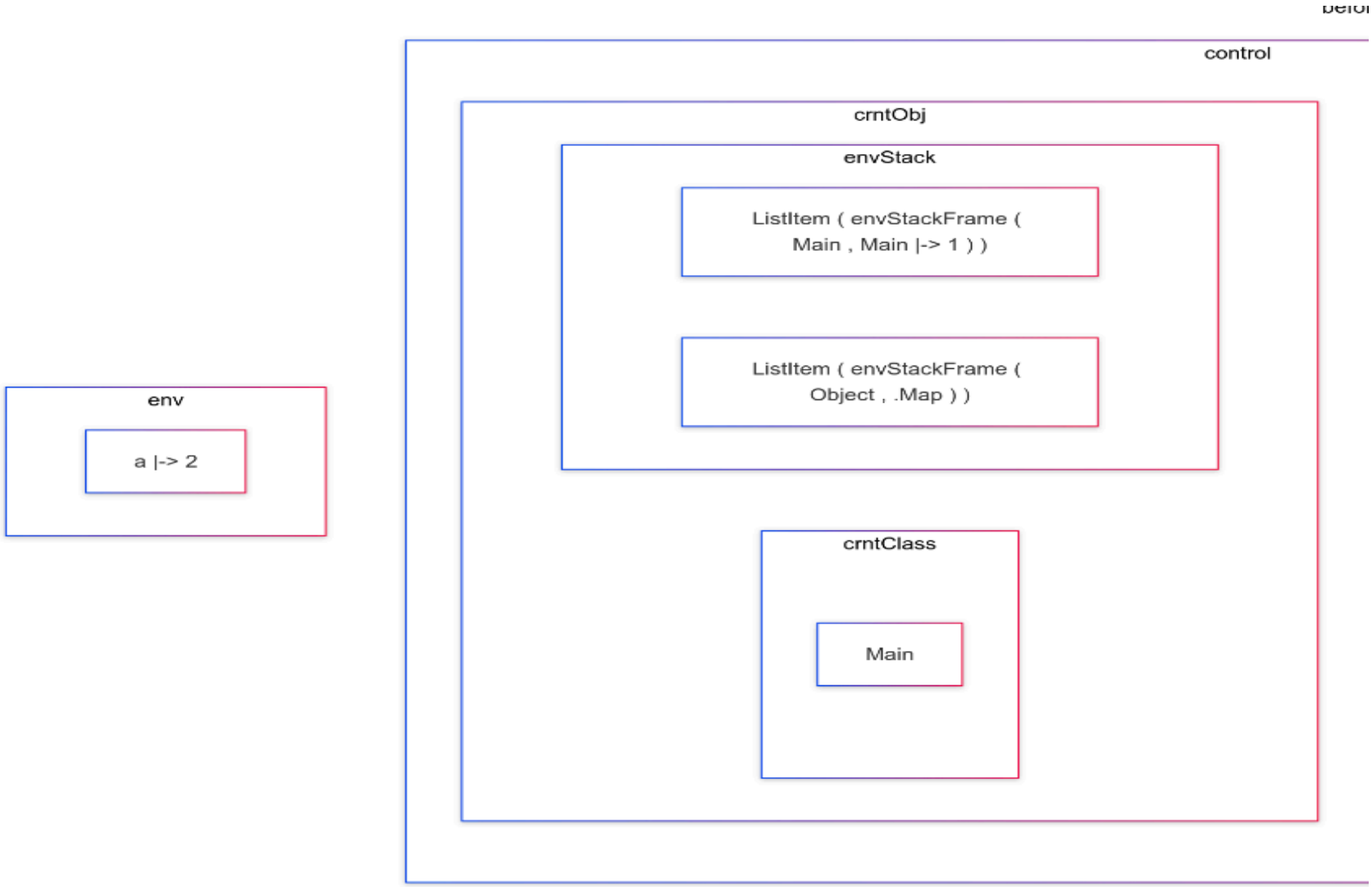
このプログラムを実行すると、Main クラスのコンストラクタが実行され、その内部で A クラスのオブジェクトを生成し、メソッドを呼び出す。視覚化ツールを用いると、new A() が評価され crntObj が一度 Object、つまりプレーンな状態のクラスに変更されていることが確認できる。CLASS については configuration の要素が多く、紙面掲載の範囲では全てを示すことができないため、一部のみを示し説明する。

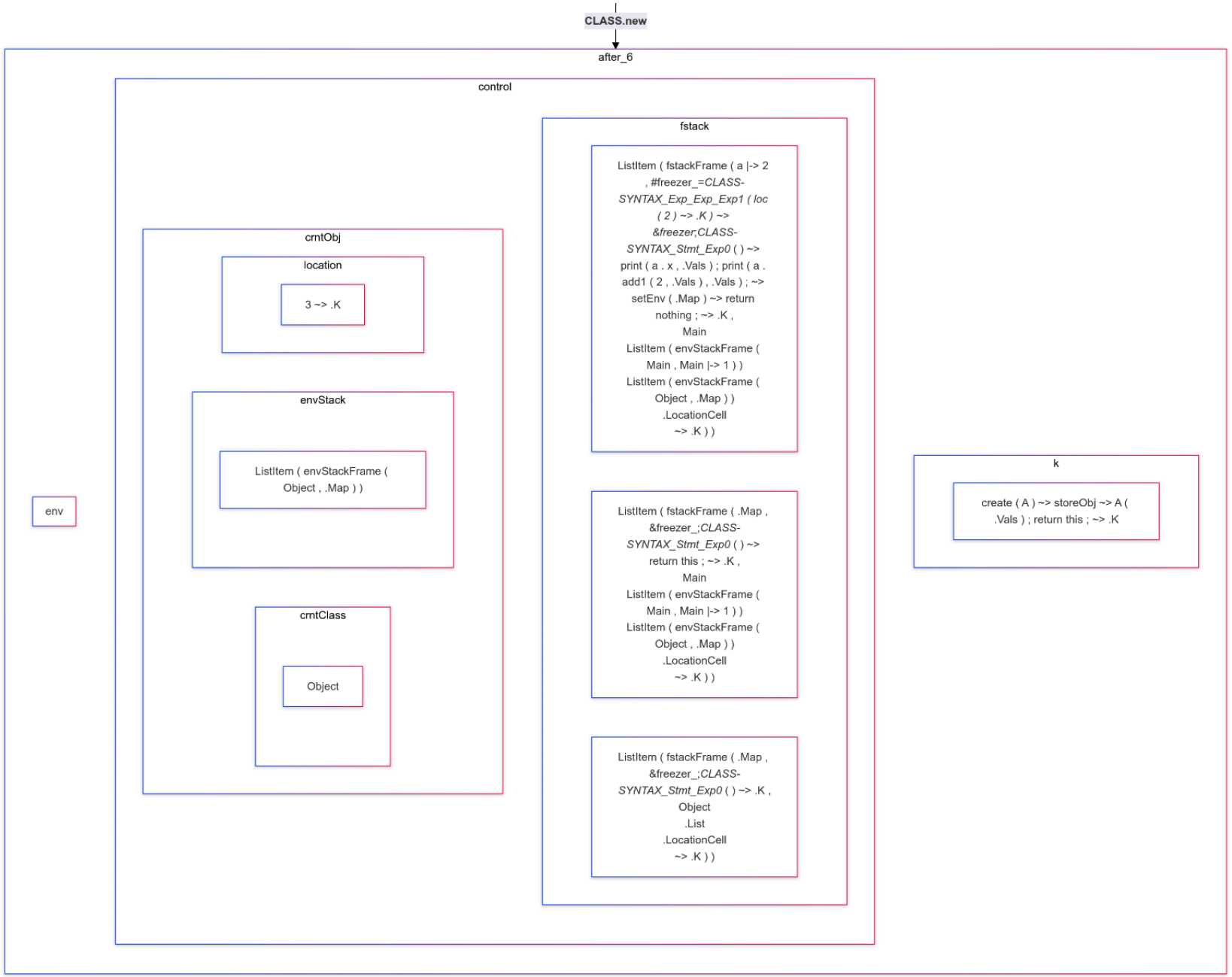
まず、`new A()` に対する書き換え規則が適用される前の classes セルの状態を視覚化したものが以下の図である。


classes
classData
declarations
method Main (.Ids) { var a ,
.Vals ; a = new A (.Vals) ; } ~>
.K
baseClass
Object
className
Main
className
Main
classData
declarations
var x , .Vals ; method A (.Ids) {
x = Var ; } ~> .K
baseClass
Object
className
A
className
A

この画像は、クラスの定義部分のみを視覚化したものである。Main クラスと A クラスが定義されていることがわかる。Main クラスは CLASS におけるプログラムのエントリーポイントであり、CLASS は Main クラスのコンストラクタを実行することからプログラムが実行される。また、A クラス、Main クラスどちらも `baseClass` が Object クラスであることがわかる。これにより、CLASS のクラスは特別な指定がない限り Object クラスを継承することがわかる。

次に、`new A()` に対する書き換え規則の適用を視覚化する。この視覚化した図についても、紙面掲載のため、書き換え規則によって変化するセルのみを示していることに留意する。


control
crntObj
envStack
ListItem (envStackFrame (
Main , Main |-> 1))
ListItem (envStackFrame (
Object , .Map))
crntClass
Main
env
a |-> 2


CLASS.new
after_6
control
fstack
ListItem (fstackFrame (a |-> 2 , #freezer_=CLASS-SYNTAX_Exp_Exp_Exp1 (loc (2) ~> .K) ~> &freezer;CLASS-SYNTAX_Stmt_Exp0 ( ) ~> print (a . x , .Vals) ; print (a . add1 (2 , .Vals) , .Vals) ; ~> setEnv (.Map) ~> return nothing ; ~> .K , Main ListItem (envStackFrame (Main , Main |-> 1)) ListItem (envStackFrame (Object , .Map)) .LocationCell ~> .K))
ListItem (fstackFrame (.Map , &freezer_;CLASS-SYNTAX_Stmt_Exp0 ( ) ~> return this ; ~> .K , Main ListItem (envStackFrame (Main , Main |-> 1)) ListItem (envStackFrame (Object , .Map)) .LocationCell ~> .K))
ListItem (fstackFrame (.Map , &freezer_;CLASS-SYNTAX_Stmt_Exp0 ( ) ~> .K , Object .List .LocationCell ~> .K))
crntObj
location
3 ~> .K
envStack
ListItem (envStackFrame (Object , .Map))
crntClass
Object
env
k
create (A) ~> storeObj ~> A (.Vals) ; return this ; ~> .K

この図は、new A() の書き換え規則が適用される前後の状態を示している。この図は一つの一つの文字の内容を確認しなくても、

- fstack セルに新しい fstackFrame が追加されていること
- env セルが初期化されていること
- crntObj セルがコンストラクタを呼び出していた Main クラスから Object クラスに変更されていること

がわかるようになっている。このように、視覚化ツールを用いることで、書き換え規則によって複雑なオブジェクトの生成過程を視覚的に理解することができる。

5.3 まとめ

この章では、K framework を用いて CLASS という言語を定義した。CLASS は、IMP のスーパーセットであり、IMP の機能に加えてクラスとオブジェクト指向の機能を持つ。最初に CLASS の状態について説明した。CLASS は IMP の configration に加えて、クラスの情報を保持する classes セルや関数呼び出しのスタックを保持する fstack セル、実行中のオブジェクトの情報を保持する crntObj セルを持つ。CLASS の構文と書き換え規則について説明した。クラスの定義、オブジェクトの生成、メソッドとメンバの定義と呼び出しについて説明し、実行例を示した。また、CLASS の視覚化ツールを用いて、オブジェクトの生成を確認することができることを示した。特にセルの変化を視覚化することで、書き換え規則によってどのような変化が生じるのかを視覚的に理解することができることを示した。

第6章　THREAD

この章ではK frameworkを用いてTHREADという言語を定義する。THREADはCLASSのSuperSetであり、CLASSの機能に加えてマルチスレッドの機能を持つ。

6.1　THREADの構文と書き換え規則

6.1.1　THREADの初期状態

THREADのconfigrationは以下のように定義される。

```
configuration <T>
                <threads>
                  <thread multiplicity="*" type="Set" initial="">
                    <k> $PGM:Stmt </k>
                    <env> .Map </env>
                    <holds> .Map </holds>
                    <id> 0 </id>
                  </thread>
                </threads>

                <store> .Map </store>
                <busy> .Set </busy>
                <terminated> .Set </terminated>
                <nextLoc> 0 </nextLoc>
            </T>
```

THREADはIMPを並行プログラミングが可能になるように拡張した言語である。IMPの`configration`に加えて、`<threads>`, `<thread>`, `<holds>`, `<id>`, `<busy>`, `<terminated>`のセルが追加されている。プログラムの初期状態ではスレッドは一つだけだが、プログラム内でスレッドを生成することができる。また、値をキーとしてロックを獲得できる機能も持つ。K frameworkでは書き換え可能な項が複数ある場合、どの項を書き換えるかは非決定的である。そのため、書き換え可能なスレッドが複数ある場合、どのスレッドを書き換えるかは非決定的である。これにより、スレッドの競合状態を再現することができる。

`<threads>`はスレッドの集合を表すセルであり、`<thread>`はスレッドを表すセルである。`<holds>`はスレッドが保持するstore内の要素を表すセルである。`<id>`はスレッドのIDを表すセルである。`<busy>`は実行中のスレッドを表すセルであり、`<terminated>`は終了したスレッドを表すセルである。

6.1.2 スレッドの生成

THREAD では最初に一つのスレッド上で評価が始まる。スレッドを生成するには spawn 式を用いる。

```
spawn {
  var obj;
  obj = new A();
}
```

spawn S という形式でスレッドを生成する。

スレッド生成構文及び書き換え規則は以下のように定義される。

```
syntax Exp ::= "spawn" Block

rule <thread>...
       <k> spawn S => !T:Int ...</k>
       <env> Env </env>
     ...</thread>
     (.Bag => <thread>...
              <k> S </k>
              <env> Env </env>
              <id> !T </id>
            ...</thread>)
```

spawn S は、新しいスレッドを生成し、スレッド内で S を評価することを示している。スレッドを生成する際には、新しいスレッドの ID がインクリメントされ、環境も引き継がれる。store は thread 間で共有される。

6.1.3 スレッドの終了

スレッドの終了は以下のように定義される。

```
rule (<thread>... <k>.K</k> <holds>H</holds> <id>T</id> ...</thread> => .Bag)
     <busy> Busy => Busy -Set keys(H) </busy>
     <terminated>... .Set => SetItem(T) ...</terminated>
```

thread 内のプログラムに項がなくなると、スレッドが保持していた値が削除され、terminated に終了したスレッドの id が追加される。

6.1.4 join 文

K framework では書き換え可能な項が複数ある場合、どの項を書き換えるかは非決定的である。そのため、以下のようなプログラムは作成者の意図通りに動作しない可能性がある。

```
var x;
x = 1;

spawn {
  x = x + 1;
}

if (x > 1) {
  x = x + 2;
}
```

このプログラムを評価すると、最終的な store 内の x の値は 2 もしくは 4 になる。これは x = x + 1; が評価される前に `if (x > 2)` が評価される可能性があるためである。この問題を解決するために、join 式を用いる。

```
var x, id;
x = 1;

id = spawn {
  x = x + 1;
}

join id;

if (x > 2) {
  x = x + 2;
}
```

join 式は引数にスレッド ID を取り、そのスレッドが終了するまで待機する。このプログラムを評価すると、x の store 内の値は必ず 4 になる。join 式の構文及び書き換え規則は以下のように定義される。

```
  syntax Stmt ::= "join" Exp ; [strict]

  rule <k> join T:Int; => .K ...</k>
       <terminated>... SetItem(T) ...</terminated>
```

join T:Int; は、スレッド T が terminated セルに追加されるまで待機することを示している。これにより、join 式は terminated セルにスレッド ID が追加されるまで書き換えられず、対象のスレッドが終了するまで待機することができる。

6.1.5 競合状態が発生する実行の視覚化

非同期処理では、スレッド間でのデータの共有が問題となる。以下のようなプログラムで説明する。これは x の値が 1 以上であれば、- 1 を行う操作を、spawn 式によって生成されたスレッドとメインスレッドで行うプログラムである。つまり、２つのスレッドが同じ変数 x を参照、変更するプログラムである。

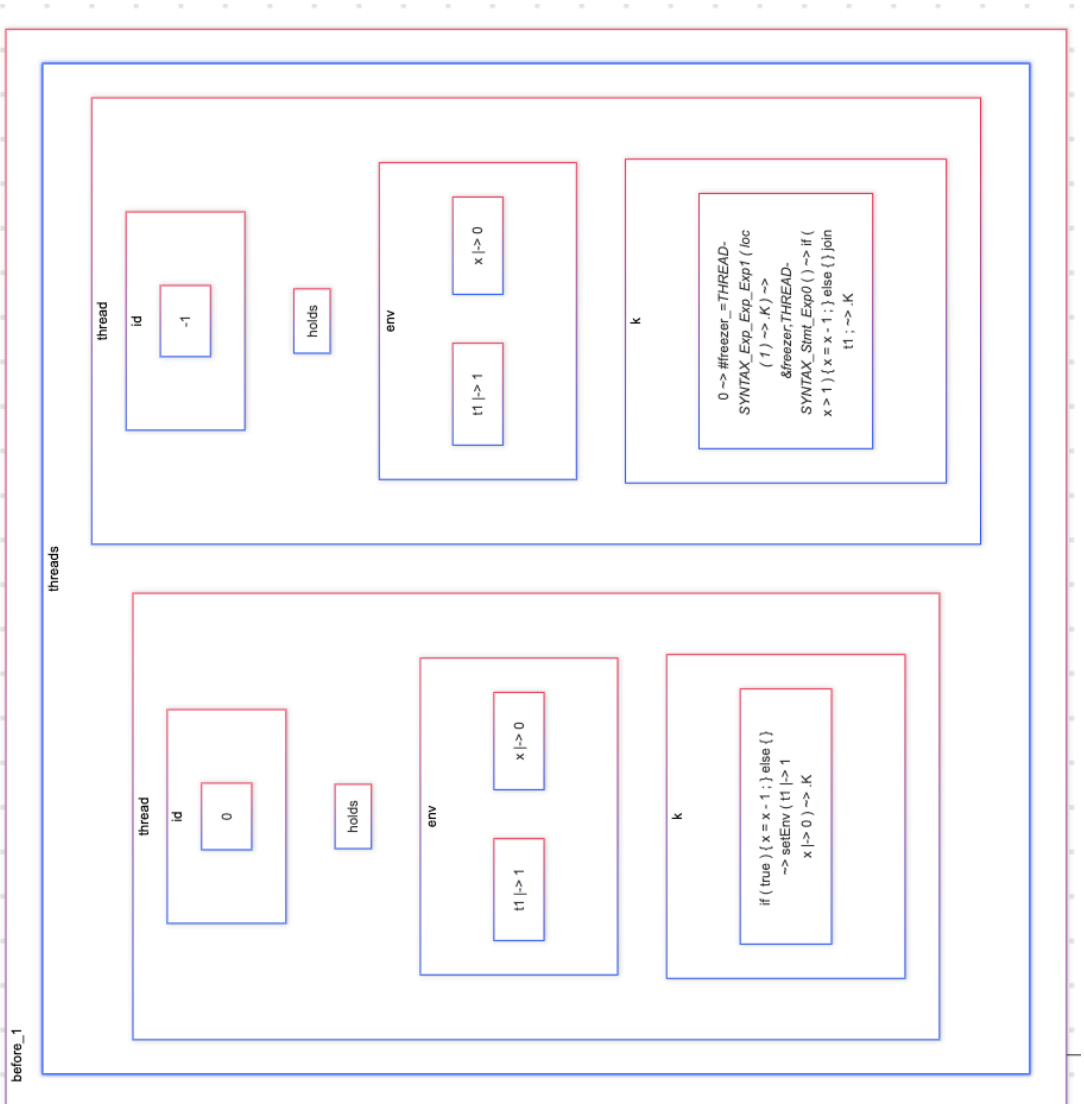
```
var x;
x = 2;

var t1;
t1 = spawn {
  if (x > 1) {
    x = x - 1;
  }
};

if (x > 1) {
  x = x - 1;
}

join t1;
```

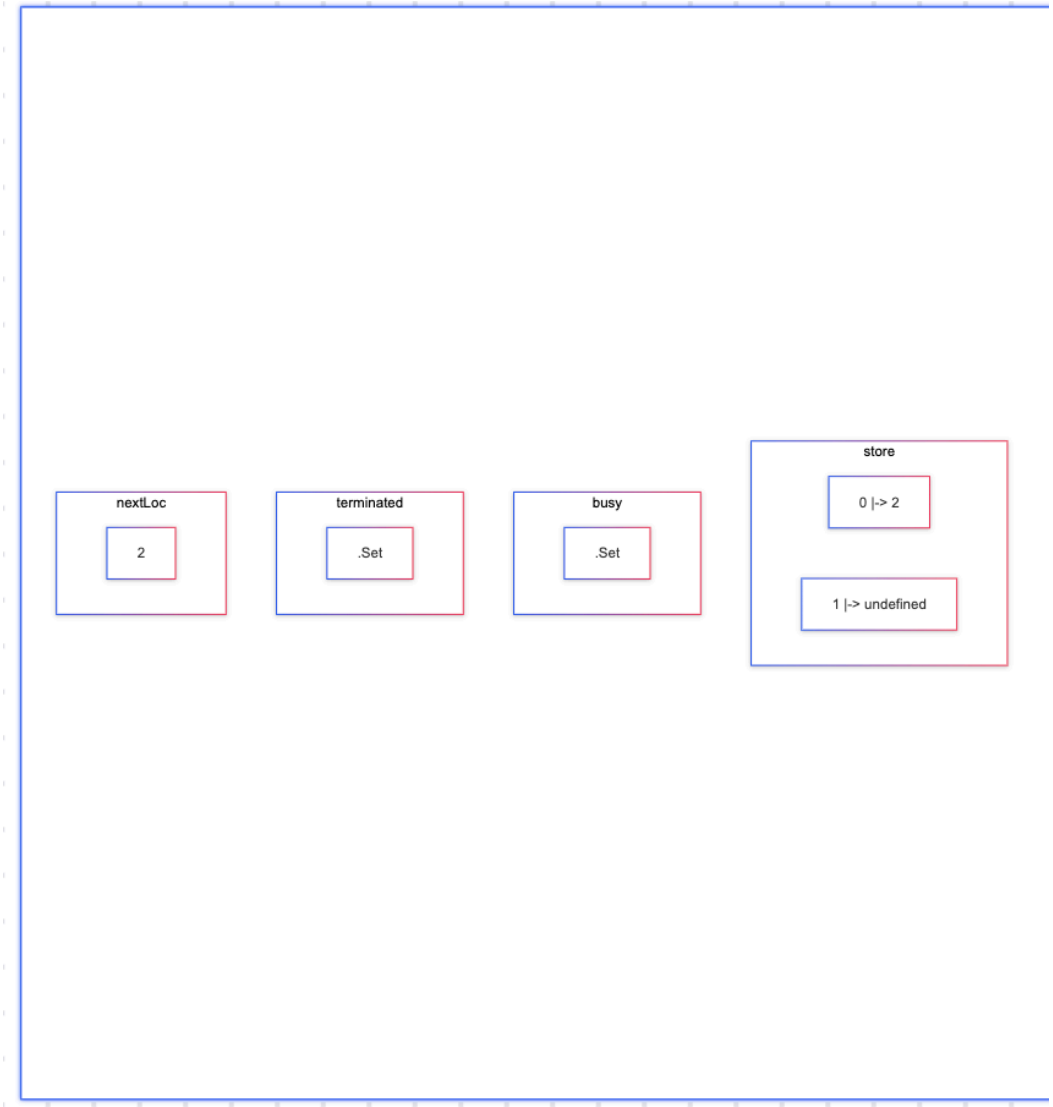
この時、x の store 内の値は 0 もしくは 1 になる。`x > 1` の条件分岐が評価された後、別のスレッドが x の値を変更する可能性があるためである。

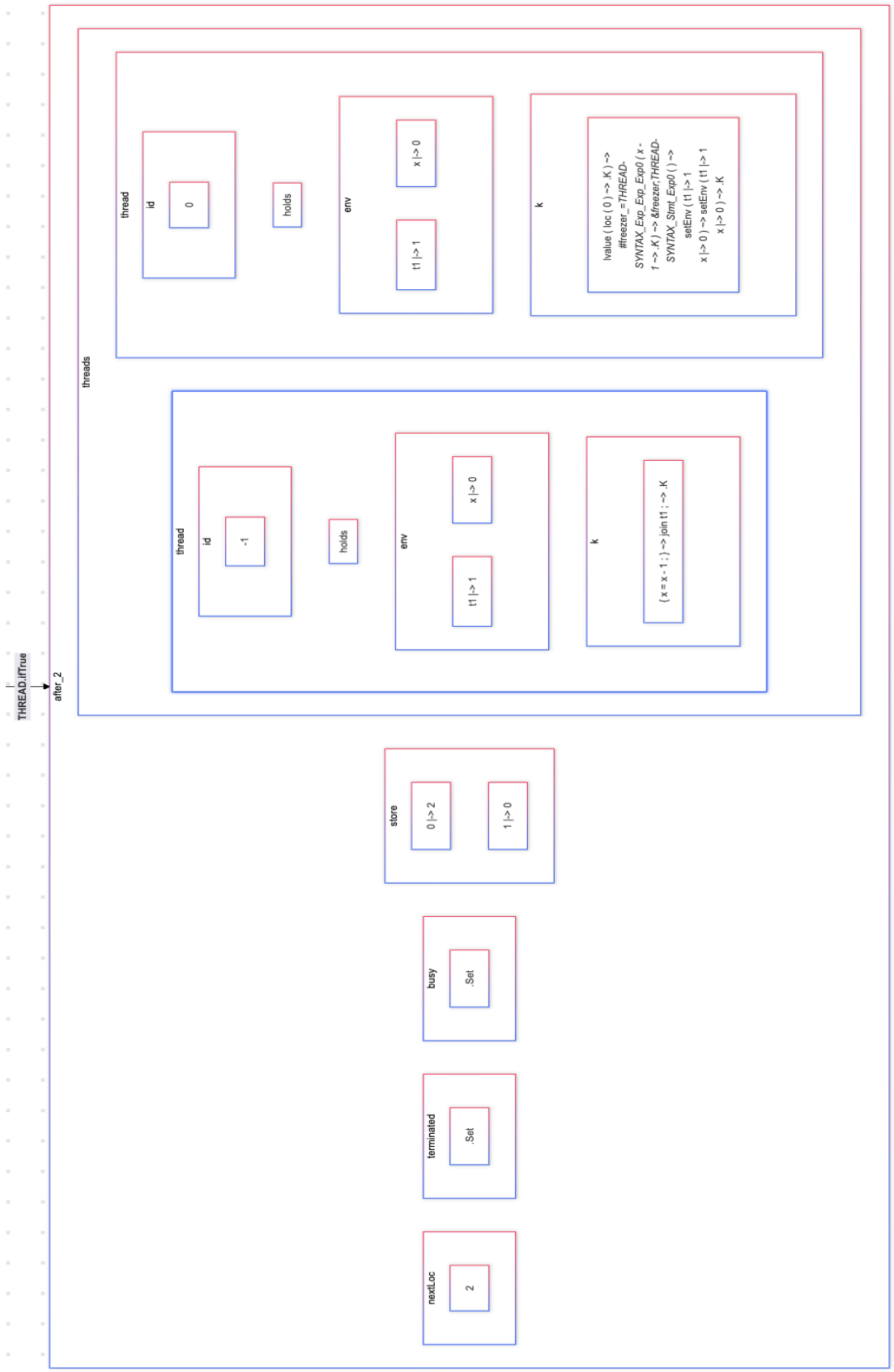
このプログラムに対し、以下のように THREAD 内で if 文の書き換え規則を指定して視覚化ツールを用いて確認する。

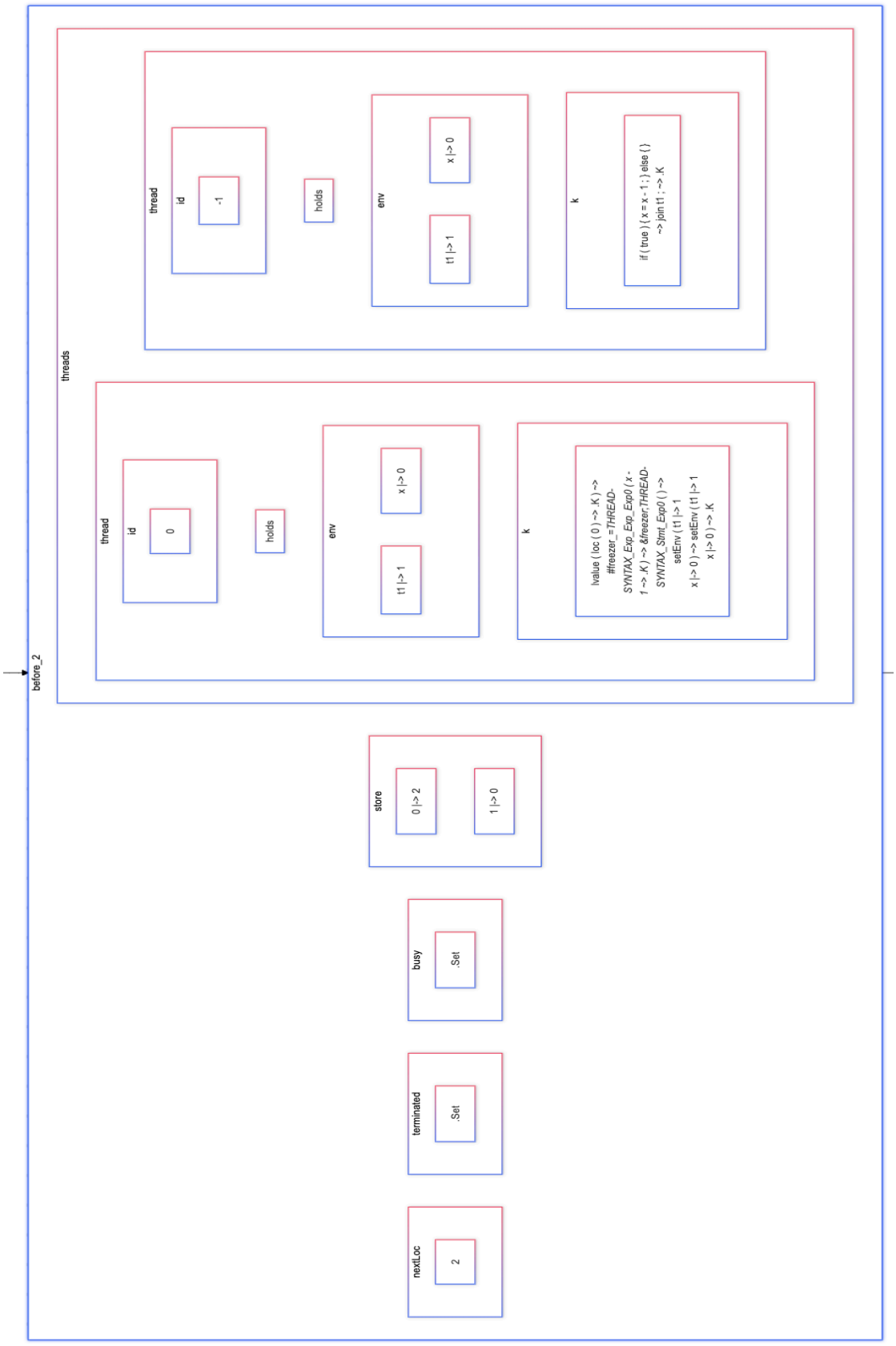
```
rule [ifTrue]: if (true) S else _ => S
```

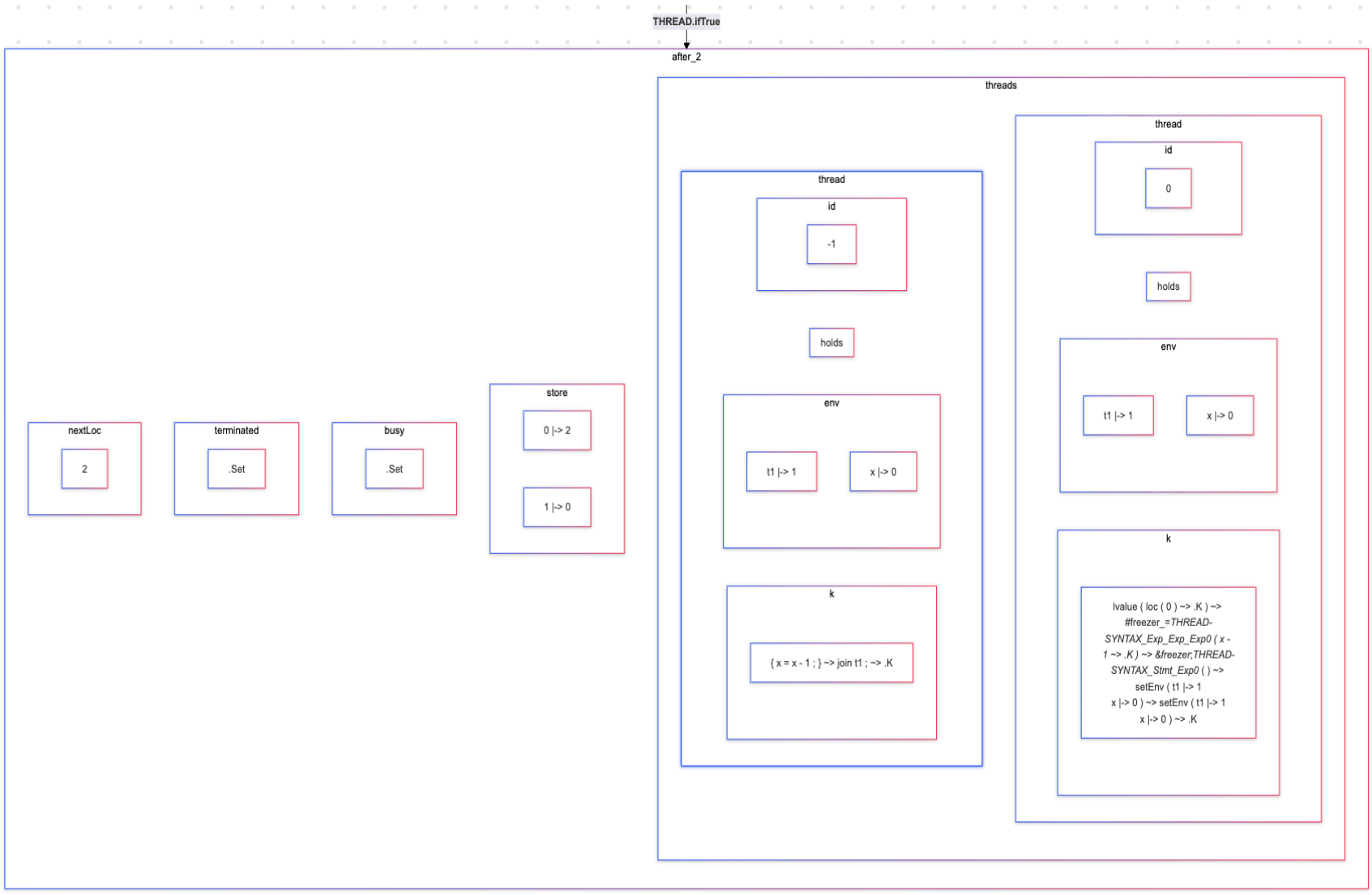
この状態で視覚化ツールを通してプログラムを実行すると、if 文の条件式が true の場合の書き換えが 2 回行われることが確認できる。つまり、あるスレッドが `x - 1` を行う前に、別のスレッドが `x > 1` の評価を行っている。

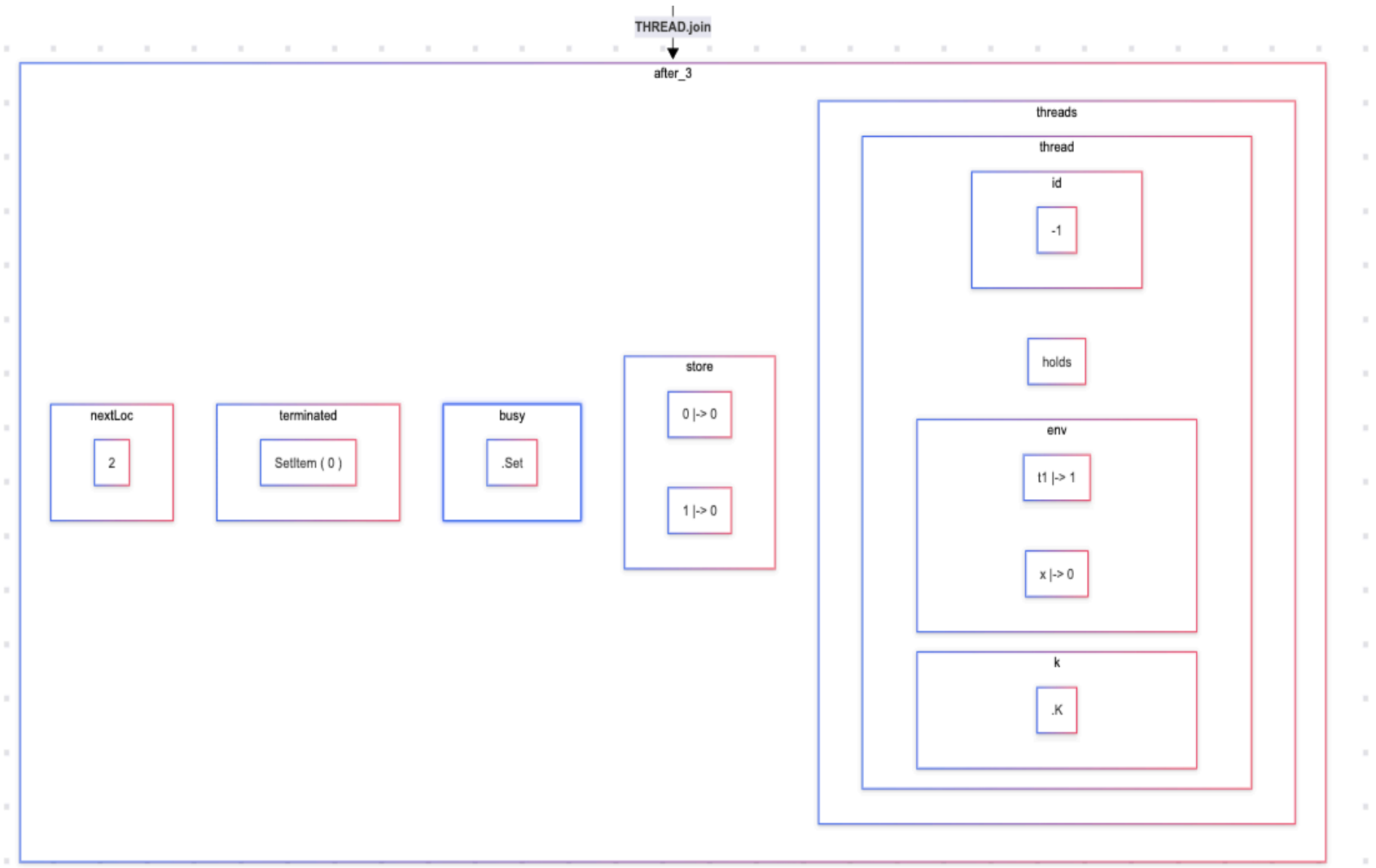



nextLoc
2
terminated
.Set
busy
.Set
store
0 |-> 2
1 |-> undefined


THREAD.ifTrue
after_2
threads
thread
id
0
holds
env
x |-> 0
t1 |-> 1
k
lvalue (loc (0) ~> .K) ~>
#freezer_=THREAD-
SYNTAX_Exp_Exp_Exp0 (x -
1 ~> .K) ~> &freezer;THREAD-
SYNTAX_Stmt_Exp0 ( ) ~>
setEnv (t1 |-> 1
x |-> 0) ~> setEnv (t1 |-> 1
x |-> 0) ~> .K
thread
id
-1
holds
env
x |-> 0
t1 |-> 1
k
{ x = x - 1 ; } ~> join t1 ; ~> .K
store
0 |-> 2
1 |-> 0
busy
.Set
terminated
.Set
nextLoc
2


before_2
threads
thread
id
0
holds
env
x |-> 0
t1 |-> 1
k
lvalue (loc (0) ~> .K) ~>
#freezer_=THREAD-
SYNTAX_Exp_Exp0 (x -
1 ~> .K) ~> &freezer;THREAD-
SYNTAX_Stmt_Exp0 ( ) ~>
setEnv (t1 |-> 1
x |-> 0) ~> setEnv (t1 |-> 1
x |-> 0) ~> .K
thread
id
-1
holds
env
x |-> 0
t1 |-> 1
k
if (true) { x = x - 1 ; } else { }
~> join t1 ; ~> .K
store
0 |-> 2
1 |-> 0
busy
.Set
terminated
.Set
nextLoc
2


THREAD.ifTrue
after_2
threads
thread
id
-1
holds
env
t1 |-> 1
x |-> 0
k
{ x = x - 1 ; } ~> join t1 ; ~> .K
thread
id
0
holds
env
t1 |-> 1
x |-> 0
k
lvalue (loc (0) ~> .K) ~>
#freezer_=THREAD-SYNTAX_Exp_Exp_Exp0 (x - 1 ~> .K) ~> &freezer,THREAD-SYNTAX_Stmt_Exp0 ( ) ~>
setEnv (t1 |-> 1
x |-> 0) ~> setEnv (t1 |-> 1
x |-> 0) ~> .K
store
0 |-> 2
1 |-> 0
nextLoc
2
terminated
.Set
busy
.Set


THREAD.join
after_3
threads
thread
id
-1
holds
env
t1 |-> 1
x |-> 0
k
.K
store
0 |-> 0
1 |-> 0
nextLoc
2
terminated
SetItem (0)
busy
.Set

参考のため、最終的な join の書き換え規則適用後の configration を視覚化している。最終的に x の store 内の値は 0 になっている。このように、非同期処理ではスレッド間でのデータの共有が問題となる。

この問題を解決するために、THREAD ではロックを取得できる。

```
var x, lock_key;
x = 1;
lock_key = true;

var t1, t2;
t1 = spawn {
  acquire lock_key;
  if (x > 1) {
    x = x - 1;
  }
  release lock_key;
};

t2 = spawn {
  acquire lock_key;
  if (x > 1) {
    x = x - 1;
  }
  release lock_key;
};

join t1;
join t2;
```

ロックの取得と開放は以下のように定義される。

```
syntax Stmt ::=  "acquire" Exp ";" [strict]
              | "release" Exp ";" [strict]

rule <k> acquire V:Val; => .K ...</k>
       <holds>... .Map => V |-> 0 ...</holds>
       <busy> Busy (.Set => SetItem(V)) </busy>
  requires (notBool(V in Busy))

rule <k> acquire V; => .K ...</k>
  <holds>... V:Val |-> (N => N +Int 1) ...</holds>

rule <k> release V:Val; => .K ...</k>
  <holds>... V |-> (N => N -Int 1) ...</holds>
  requires N >Int 0

rule <k> release V; => .K ...</k>
  <holds>... V:Val |-> 0 => .Map ...</holds>
  <busy>... SetItem(V) => .Set ...</busy>
```

`acquire V:Val;` は、ロックを取得することを示している。ロックを取得する際には、holds セルにロックのキーが追加され、busy セルにロックを取得していることが追加される。この書き換え規則は Busy 内の同じ値のロックがない場合にのみ書き換えられる。もし Busy 内に Val がある場合、つまり他のスレッドがロックを取得している場合、acquire は書き換えられず、他のスレッドがロックを開放するまでそのスレッドの評価は止まる。もし、holds にすでに存在する、つまり自スレッドがロックを取得している場合、holds 内の値をインクリメントする。

`release V:Val;` は、ロックを開放することを示している。もし、holds 内の値が 1 以上の場合、holds 内の値をデクリメントする。もし、holds 内の値が 0 の場

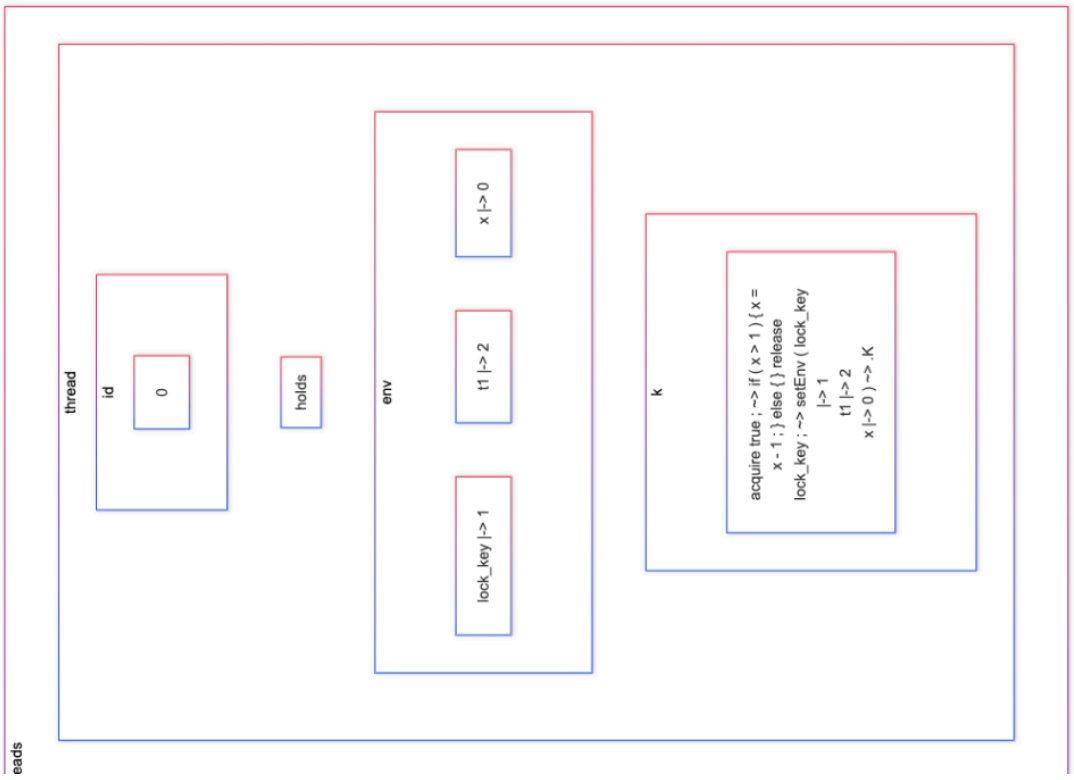
合、holds からロックのキーが削除され、busy セルからロックを取得していることが削除される。

6.1.6 実行例

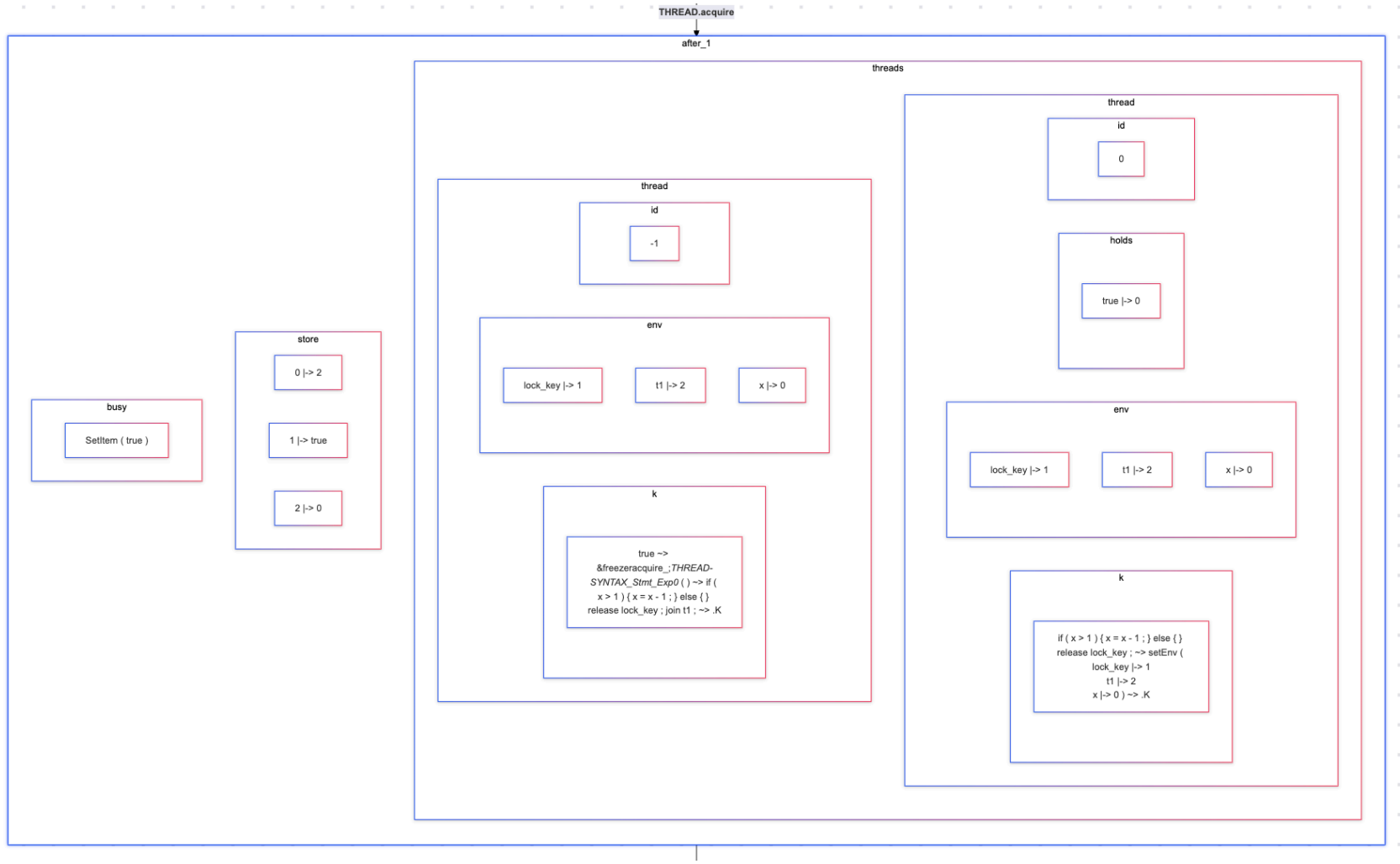
視覚化ツールを用いて、Thread のロック機能を確認する。以下のようにロックとリリースの書き換え規則を指定し、プログラムを実行する。

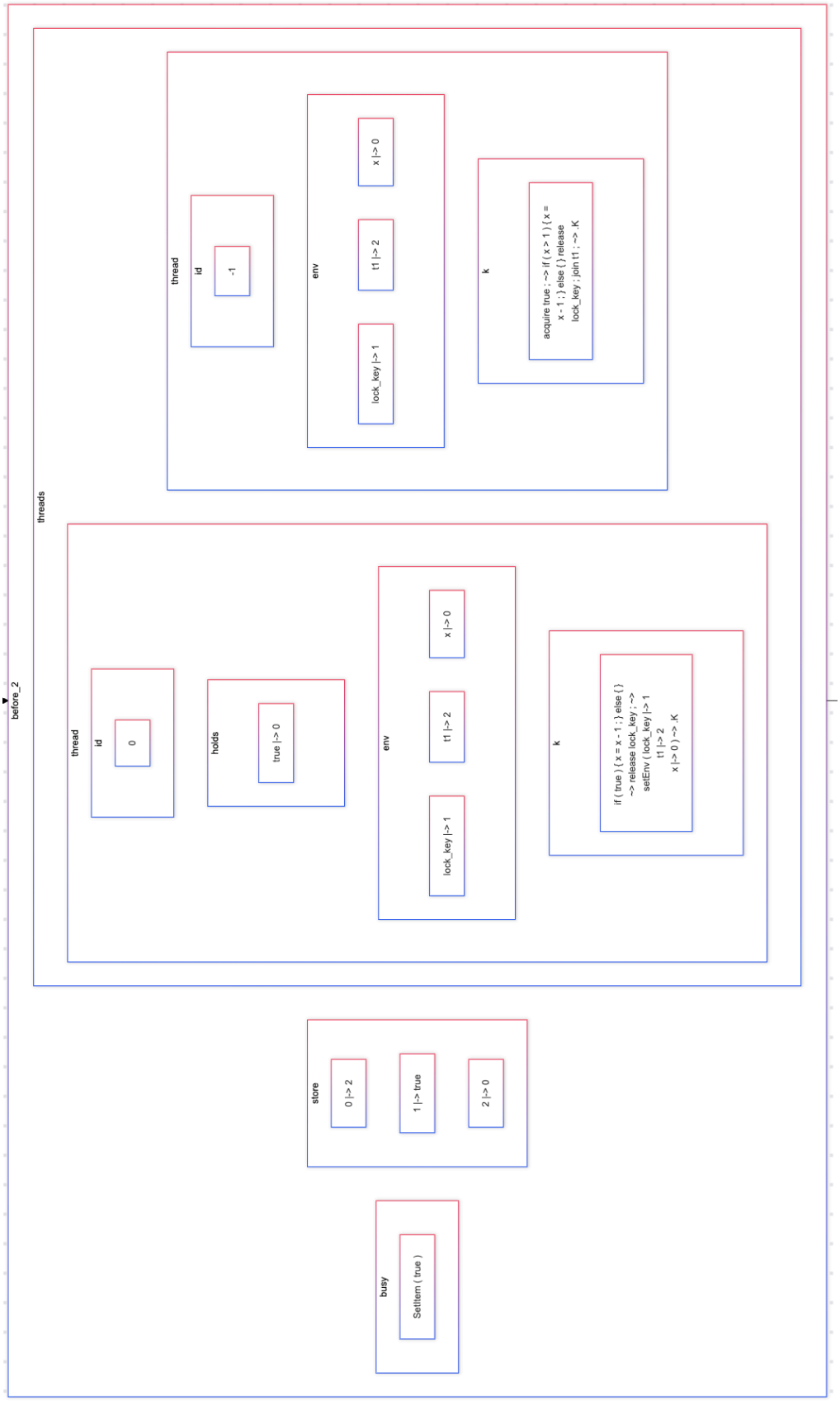
```
rule [acquire]: <k> acquire V:Val; => .K ...</k>
      <holds>... .Map => V |-> 0 ...</holds>
      <busy> Busy (.Set => SetItem(V)) </busy>
  requires (notBool(V in Busy))

rule [release]: <k> release V; => .K ...</k>
  <holds>... V:Val |-> 0 => .Map ...</holds>
  <busy>... SetItem(V) => .Set ...</busy>
```

この状態で視覚化ツールを通してプログラムを実行すると、ロックが取得されたスレッドが他のスレッドによってロックが取得されるまで待機することが確認できる。



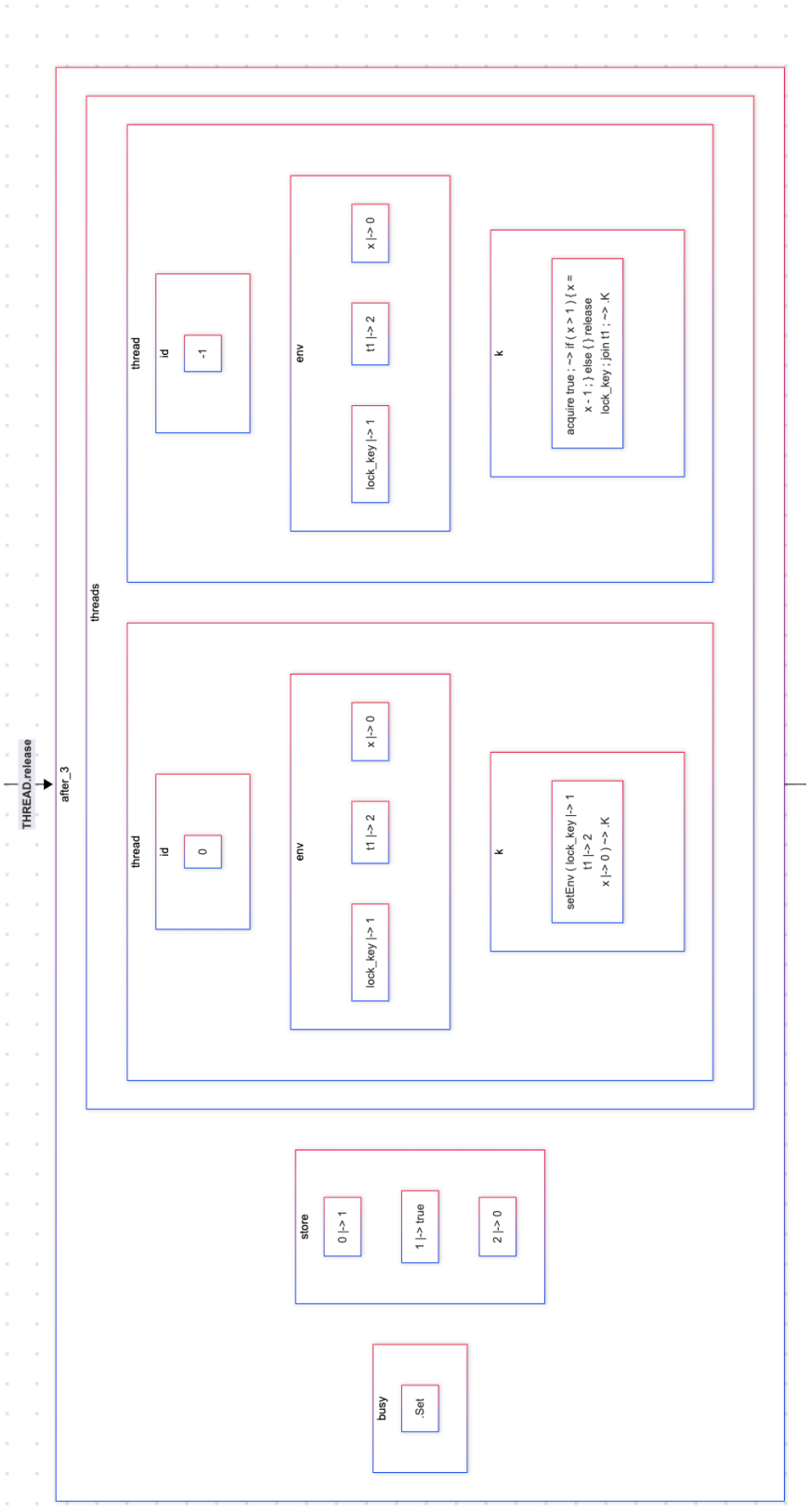
thread
id
-1
holds
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
true ~>
&freezeracquire_;THREAD-
SYNTAX_Stmt_Exp0 ( ) ~> if (
x > 1) { x = x - 1 ; } else { }
release lock_key ; join t1 ; ~> .K
store
0 |-> 2
1 |-> true
2 |-> 0
busy
.Set

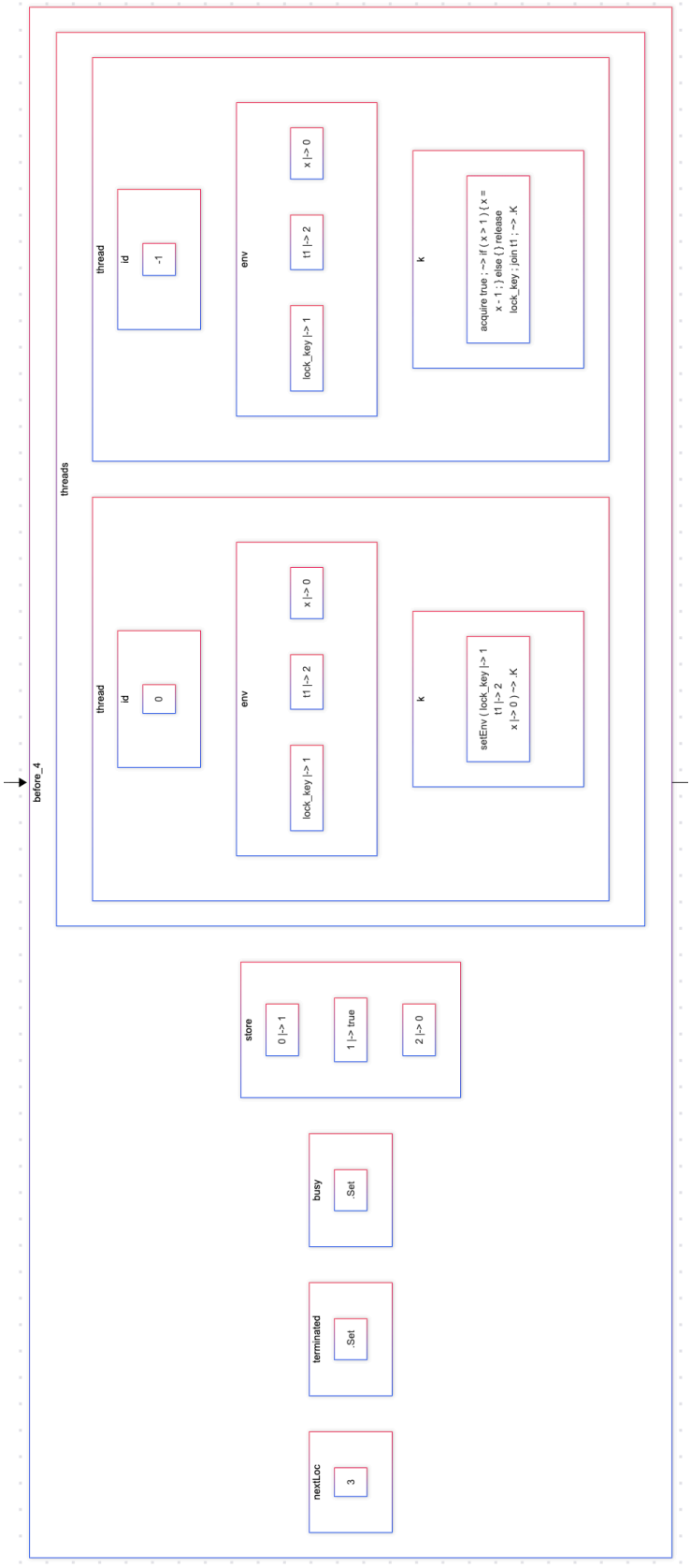

THREAD.acquire
after_1
threads
thread
id
0
holds
true |-> 0
env
lock_key |-> 1
t1 |-> 2
x |-> 0
k
if (x > 1) { x = x - 1 ; } else { }
release lock_key ; ~> setEnv (
lock_key |-> 1
t1 |-> 2
x |-> 0) ~> .K
thread
id
-1
env
lock_key |-> 1
t1 |-> 2
x |-> 0
k
true ~>
&freezeracquire_;THREAD-SYNTAX_Stmt_Exp0 ( ) ~> if (
x > 1) { x = x - 1 ; } else { }
release lock_key ; join t1 ; ~> .K
store
0 |-> 2
1 |-> true
2 |-> 0
busy
SetItem (true)


before_2
threads
thread
id
-1
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
acquire true ; ~> if (x > 1) { x = x - 1 ; } else { } release lock_key ; join t1 ; ~> .K
thread
id
0
holds
true |-> 0
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
if (true) { x = x - 1 ; } else { } ~> release lock_key ; ~> setItem (lock_key |-> 1 t1 |-> 2 x |-> 0) ~> .K
store
0 |-> 2
1 |-> true
2 |-> 0
busy
SetItem (true)

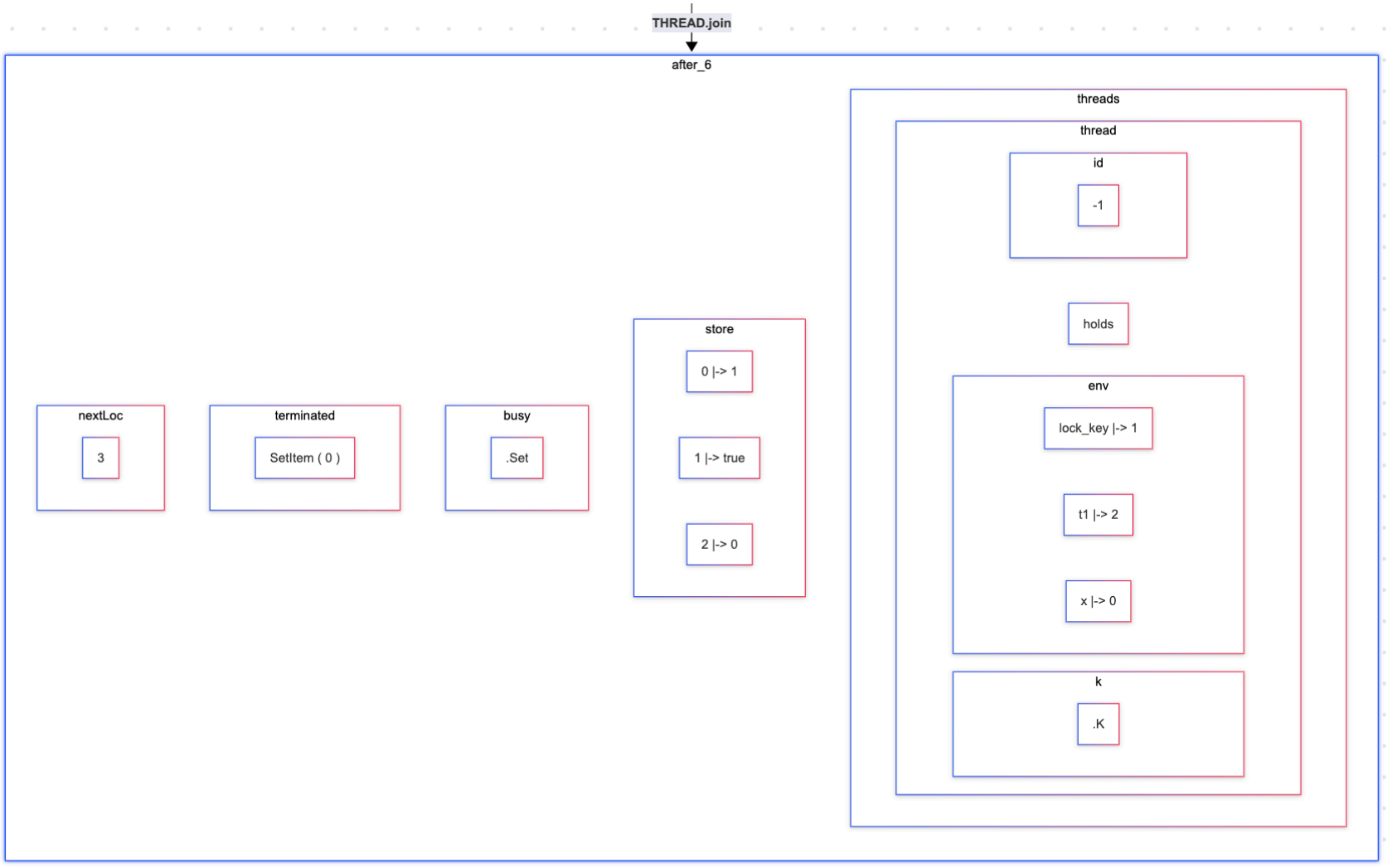
THREAD_ifTrue
after_2
threads
thread
id
0
holds
true |-> 0
env
lock_key |-> 1
t1 |-> 2
x |-> 0
k
{ x = x - 1 ; } ~> release lock_key ; ~> setEnv (lock_key |-> 1 t1 |-> 2 x |-> 0) ~> .K
thread
id
-1
env
lock_key |-> 1
t1 |-> 2
x |-> 0
k
acquire true ; ~> if (x > 1) { x = x - 1 ; } else { } release lock_key ; join t1 ; ~> .K
store
0 |-> 2
1 |-> true
2 |-> 0
busy
SetItem (true)

before_3
threads
thread
id
0
holds
true |-> 0
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
release true ; ~> setEnv (
lock_key |-> 1
t1 |-> 2
x |-> 0) ~> .K
thread
id
-1
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
acquire true ; ~> if (x > 1) { x =
x - 1 ; } else ( ) release
lock_key ; join t1 ; ~> .K
store
0 |-> 1
1 |-> true
2 |-> 0
busy
SetItem (true)


THREAD.release
after_3
threads
thread
id
-1
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
acquire true ; ~> if (x > 1) { x = x - 1 ; } else { } release lock_key ; join t1 ; ~> .K
thread
id
0
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
setEnv (lock_key |-> 1 t1 |-> 2 x |-> 0) ~> .K
store
0 |-> 1
1 |-> true
2 |-> 0
busy
.Set


before_4
threads
thread
id
-1
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
acquire true ; ~> if (x > 1) { x =
x - 1 ; } else { } release
lock_key ; join t1 ; ~> .K
thread
id
0
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
setEnv (lock_key |-> 1
t1 |-> 2
x |-> 0) ~> .K
store
0 |-> 1
1 |-> true
2 |-> 0
busy
.Set
terminated
.Set
nextLoc
3

THREAD.acquire
after_4
threads
thread
id
-1
holds
true |-> 0
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
if (x > 1) { x = x - 1 ; } else { }
release lock_key ; join t1 ; ~> .K
thread
id
0
env
x |-> 0
t1 |-> 2
lock_key |-> 1
k
setEnv (lock_key |-> 1
t1 |-> 2
x |-> 0) ~> .K
store
0 |-> 1
1 |-> true
2 |-> 0
busy
SetItem (true)


THREAD.join
after_6
threads
thread
id
-1
holds
env
lock_key |-> 1
t1 |-> 2
x |-> 0
k
.K
store
0 |-> 1
1 |-> true
2 |-> 0
nextLoc
3
terminated
SetItem (0)
busy
.Set

以上の視覚化では、id が 0 のスレッドが、変数 `lock_key` に入れた true という値をキーとしてロックを獲得し、他のスレッドはそのロックがリリースされるまで、`acquire true` の書き換えが行われないことが確認できる。id が 0 のスレッドが `acquire true` を行った後に、busy セルに true が追加され、スレッド全体に対してロックを獲得していることがわかる。その後、id が 1 のスレッドが `acquire true` まで到達するが、busy セルに true があるため、書き換えが行われず、スレッドが待機していることがわかる。id が 0 のスレッドが `release true` を行った後、busy セルから true が削除され、id が 1 のスレッドが `acquire true` を行い、busy セルに true が追加されることがわかる。

結果として id が 0 のスレッドが `x = x - 1` を行った後に、値をリリースするまで、id が 1 のスレッドは `acquire true` の書き換えが行われないことが確認できる。

参考のため、最終的な join の書き換え規則適用後の configration を視覚化している。最終的に x の store 内の値は 1 になっている。このように、ロックを取得することで、スレッド間でのデータの共有が問題となることを解決することができる。

6.2 まとめ

この章では、K framework を用いて THREAD という言語を定義した。THREAD は並行プログラミングの機能を持つ。最初に THREAD の configration について説明した。THREAD は IMP の configration に加えて、スレッドの情報を保持する threads セルや、スレッドの ID を保持する id セル、ロックの情報を保持する holds セル、実行中のスレッドを保持する busy セル、終了したスレッドを保持する terminated セルを持つ。これらの状態を保持することで、スレッド間でのデータの共有や競合状態を再現することができる。THREAD の構文と書き換え規則について説明した。スレッドの生成、終了、join 式、ロックの取得と開放について説明し、実行例を示した。また、THREAD に視覚化ツールを用いて、スレッドのロック機能を確認することができることを示した。特にセルの変化を視覚化することで、ロックの取得と開放の過程を視覚的に理解することができることを示した。

第7章　まとめと今後の課題

ここでは様々なパラダイムのプログラミング言語を K framework を用いて定義し、書き換え規則を指定し視覚化してきたまとめと今後の展望について述べる。

7.1　まとめ

本課題研究では、まず K framework の概要と基本的な構文と書き換え規則の記述方法を紹介した。

K framework は、プログラミング言語の意味論を操作的意味論を用いて定義するためのフレームワークである。K framework を用いることで、定義した意味論から直接実行可能なインタプリタや、モデル検査器を生成することができる。
K framework におけるプログラミング言語の意味論は、構文、状態、書き換え規則からなる。構文はプログラムの構造を定義し、状態はプログラムの実行中の状態を表す。書き換え規則は、状態の変化を定義する。書き換え規則は、左辺にマッチした状態を右辺に書き換える。

手続きプログラミング言語の意味論の紹介では、変数の宣言、代入、参照、算術式、ブール式、条件分岐、繰り返し、文の逐次合成、ブロックの評価を定義した。また、K framework が生成したモデル検査器を用いて、プログラムの検証が行えることを示した。

次に、K framework には、プログラミング言語設計者が直接記述した書き換え規則以外にも多くの書き換え規則が生成されることを示した。例として、手続きプログラミング言語の意味論において、引数の評価戦略を設定する属性の指定によって、4 つの書き換え規則が生成されることを示した。そうした K framework が自動的に生成する書き換え規則や、直接記述された書き換え規則は、K framework 内の書き換え規則の適用のステップにおいて、区別なく適用されてしまい、プログラミング言語の意味論を理解しようとする際に、重要な書き換え規則が埋もれてしまうという課題があることを示した。

この課題を解決するために、プログラミング言語設計者が、プログラムの意味論を定義する際に、重要な規則のみに名前を付けて指定し、視覚化することで、プログラムの意味論を理解することができるツールを提案した。これは、プログラミング言語設計者が記述した意味論から、K framework が生成した実行可能なプログラミング言語インタプリタを実行する際に、指定された重要な書き換え規則が

適用されたステップだけを抽出し、適用前後のプログラムの状態 (configration) を視覚化することで、大量に適用される書き換え規則の中から重要な部分のみを抽出し、理解することができるツールである。手続きプログラミング言語の意味論において、繰り返し規則を指定し、視覚化した結果を示した。繰り返し規則のみを指定し、視覚化することで、本来、229 回の書き換え規則について観察する必要があるところを、ツールを用いて、7 回の書き換え規則について観察するだけでよくなることを示した。手続きプログラミング言語に加え、関数プログラミング言語、オブジェクト指向プログラミング言語、並行プログラミング言語を K framework を用いて定義し、書き換え規則を指定し、視覚化した。それぞれの題材の中で、言語の特徴を表すために重要な書き換え規則のみを指定し、視覚化することで大量の書き換え規則の中でも重要な部分を抽出することができた。

関数プログラミング言語においては、式と値、条件分岐、変数の束縛、クロージャ、関数適用、リスト、再帰関数を扱う規則を定義し、通常の関数の定義及び関数適用と再帰関数の定義と関数適用について視覚化した。通常の関数定義については、let 式が lambda 式に変換され、引数が let 式の右辺に束縛される様子を確認した。クロージャが生成される様子を視覚化することで、クロージャ自身が環境を保持していることを確認した。再帰関数の定義については、letrec 式についての動作を確認した。letrec 式は mu 式に変換される。let 式とは異なり、letrec 式は store セルにクロージャを保存するような書き換え規則が適用されることについて、視覚化ツールを通して比較することができた。

オブジェクト指向プログラミングにおいては、クラスの定義、オブジェクトの生成、メソッドとメンバの定義と呼び出しを定義し、オブジェクトの生成について視覚化した。オブジェクトの生成については、クラスのコンストラクタが実行され、オブジェクトが生成される様子を確認した。オブジェクト生成時の状態を視覚化することで、オブジェクトの生成過程において、実行されているオブジェクトの変遷やどういった状態が初期化されているかを確認することができた。

並行プログラミング言語においては、スレッドの生成、スレッドの終了、join 文、ロックの取得と開放を定義した。特に join 文とロックについては、それがないとどのような問題が発生するかを示し、それを解決するために join 文とロックを導入した。また、ロックを使用していない場合を視覚化することで、ロックを取得しない場合一方のスレッドが共有資源に変更を加える前に、本来参照してほしくない状態のまま、別のスレッドが資源を参照してしまう様を示した。そして、ロックを取得した場合のプログラムに対して視覚化ツールを用いることで、ロックを取得したスレッドが他のスレッドによってロックが取得されるまで待機する様子を視覚化した。

7.2 今後の課題

最後に、今後の課題について述べる。

この課題研究では、K framework で記述される状態 (configration) 自体が多くの要素をもち、視覚化した場合に、多くの情報が表示されるため、状態の情報を把握するのが難しいという問題がある。本課題研究報告書内でも、オブジェクト指向プログラミング言語や並行プログラミング言語は言語機能の達成のために、状態内に多くの項目を持ち、ただ視覚化しただけでは状態の把握を行うことが難しいという問題があった。

この問題を解決するためには、インタラクティブな操作を実装するのが効果的だと考えられる。インタラクティブな操作を実装することで、ユーザーが状態の情報を整理しやすくなると考えられる。例えば、必要のない情報を非表示にする、特定の情報を強調する、複数の状態の遷移を比較するなどの操作が考えられる。関連研究の節で紹介したように、SMGA はインタラクティブな操作を実装しており、状態の情報を整理しやすくなっている。このようなインタラクティブな操作を実装することで、K framework を用いたプログラムの意味論の理解をより効果的に行うことができると考えられる。

また、アニメーションについても検討したい。アニメーションを用いることで、状態の遷移を視覚的に理解しやすくなると考えられる。例えば、状態の遷移を時間軸に沿って表示することで、プログラムの実行過程を視覚的に理解しやすくなると考えられる。

また、プログラミング言語の重要な書き換え規則とは何であるかという問題がある。本研究で解説した、手続きプログラミング言語、関数プログラミング言語、オブジェクト指向プログラミング言語、並行プログラミング言語の書き換え規則は、それぞれの言語の特徴を表すために重要な書き換え規則を指定し視覚化を行った。しかし、本研究で指定した書き換え規則が本当に重要な書き換え規則だったかというのは検証ができていない。プログラミング言語の意味論設計者自身であれば、どの書き換え規則が重要なのかを指定することは可能かもしれないが、その意味論を理解しなくてはならない側の人間については、どの書き換え規則が重要であるかを事前に指定するのは容易ではない。本研究の視覚化ツールを用いて重要な書き換え規則を指定する際に、どのような基準で重要な書き換え規則を選択すればよいかについては、今後の課題として残されている。

また、モデル検査器との統合についても今後の課題である。本研究では K framework によって記述された意味論から直接モデル検査器を生成し、プログラムの検証を行うことができることを示した。しかし、K framework によって生成されたモデル検査器によって判明した反例について、その反例から重要な書き換え規則を抽出し、視覚化することで、プログラムの意味論を理解することができるかどうかについては検証ができていない。モデル検査器によって判明した反例から、モデル検査の条件に記載した条件を満たしていない状態までの実行経路には、本研究で示した通り、反例が起きてしまう直接の条件となる書き換え規則以外にも、多くの書き換え規則が適用されてしまう。このような多くの書き換え規則の中から、重要な書き換え規則を抽出し、視覚化することで、モデル検査の反例からプログ

ラムの意味論を理解することができるかどうかについては、今後の課題として残されている。今回の研究では K framework でのモデル検査が現実的な時間で完了できず、K framework のモデル検査の調査が難しかったため、この課題に取り組むことができなかった。

またより多くのパラダイムの言語や現実に広く使われているプログラミング言語に対し、本研究の視覚化ツールを用いて視覚化を行い、それらの定義について調査することも今後の課題である。本研究では、手続きプログラミング言語、関数プログラミング言語、オブジェクト指向プログラミング言語、並行プログラミング言語について、それぞれの言語の特徴を表すために重要な書き換え規則を指定し、視覚化を行った。しかし、これらの言語以外にも、K framework では現実に広く使われている C 言語や Java などのプログラミング言語に対して形式意味論が定義されている。これらのプログラミング言語の意味論に対しても、視覚化ツールを適用し、状態が書き換えられる様子を視覚化することで、より現実のソフトウェアエンジニアリングにおいて、形式意味論が有効に活用できる範囲を広げることができると考えられる。

そして、この視覚化ツール自体の客観的な検証も今後の課題である。本研究では、手続きプログラミング言語、関数プログラミング言語、オブジェクト指向プログラミング言語、並行プログラミング言語において、それぞれの言語の特徴を表すために重要な書き換え規則を指定し、視覚化を行った。しかし、この視覚化ツールがプログラミング言語の意味論を理解するために有効であるかどうかについては、客観的な検証が必要である。客観的な検証を行うには、人間の主観による評価ではなく、SMGA に対する評価においてゲシュタルトの原理が用いられた様に、本研究の視覚化ツールによって生成された画像に対してゲシュタルトの原理を使用して評価を行い、その評価結果をもとに、視覚化ツールの有効性を検証することが必要である。

謝辞

本課題研究を執筆するにあたりご支援いただいた皆様に感謝申し上げます。特に、指導教員である緒方和博教授には本当に根気強く指導していただき、本研究を進めることができました。心より感謝します。緒方先生がいなければ執筆はできませんでした。平石邦彦教授、青木利晃教授、石井大輔准教授には審査において貴重な時間を頂き、多くのアドバイスをいただきました。ありがとうございました。

関連図書

[1] Artho, C., Pande, M., and Tang, Q. Visual analytics for concurrent Java executions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)* (2019), IEEE, pp. 1102–1105.

[2] Beschastnikh, I., Liu, P., Xing, A., Wang, P., Brun, Y., and Ernst, M. D. Visualizing Distributed System Executions. *ACM Transactions on Software Engineering and Methodology (TOSEM) 29*, 2 (Mar. 2020), 9:1–9:38.

[3] Bogdanas, D., and Rosu, G. K-java: A complete semantics of java. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)* (2015), ACM, pp. 445–456.

[4] Bui, D. D., and Ogata, K. Better state pictures facilitating state machine characteristic conjecture. *Multimedia Tools and Applications 81* (2022), 237–272.

[5] Bui, D. D., Tran, D. D., Ogata, K., et al. Integration of state machine graphical animation and Maude to facilitate characteristic conjecture: an approach to lemma discovery in theorem proving. *Multimedia Tools and Applications 83* (2024), 36865–36898.

[6] Chen, X., and Rosu, G. $\mathbb{K}$—A semantic framework for programming languages and formal analysis. In *Engineering Trustworthy Software Systems - 5th International School, SETSS 2019, Chongqing, China, April 21-27, 2019, Tutorial Lectures* (2019), J. P. Bowen, Z. Liu, and Z. Zhang, Eds., vol. 12154 of *Lecture Notes in Computer Science*, Springer, pp. 122–158.

[7] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. *All About Maude - A High-Performance Logical Framework*, 1 ed., vol. 4350 of *Lecture Notes in Computer Science*. Springer Berlin, Heidelberg, 2007. Published: 19 July 2007 (eBook), 20 July 2007 (Softcover).

[8] Ellison, C., and Rosu, G. An executable formal semantics of c with applications. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)* (2012), ACM, pp. 533–544.

[9] Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., and Rosu, G. Kevm: A complete semantics of the ethereum virtual machine. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF'18)* (2018).

[10] Kale, A., Guo, Z., Qiao, X. L., Heer, J., and Hullman, J. Evm: Incorporating model checking into exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics 30*, 1 (Oct. 2023), 208–218.

[11] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. How amazon web services uses formal methods. *Communications of the ACM 58*, 4 (2015), 66–73.

[12] Park, D., Stefanescu, A., and Rosu, G. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)* (2015), ACM, pp. 346–356.

[13] Rosu, G. K - a semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering* (2017), D. Peled and A. Pretschner, Eds., NATO Science for Peace and Security, IOS Press.

[14] Runtime Verification Inc. K framework: Programming languages tutorial, 2025. `https://kframework.org/k-distribution/pl-tutorial`.

[15] Sewell, P., Nardelli, F. Z., Owens, S., Peskine, G., Ridge, T., Sheard, T., and Vytiniotis, D. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming 20*, 1 (2010), 71–122.

[16] Suzuki, I., and Kasami, T. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems 3*, 4 (1985), 344–349.

[17] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. Model checking programs. *Automated Software Engineering 10* (2003), 203–232.

[18] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 2014.