遅延評価に基づく関数型言語における メモリ割当量の削減

高野 保真

電気通信大学大学院情報理工学研究科
博士（工学）の学位申請論文

2015 年 9 月

遅延評価に基づく関数型言語における メモリ割当量の削減

博士論文審査委員会

主査　岩崎 英哉　教授
委員　沼尾 雅之　教授
委員　小花 貞夫　教授
委員　寺田 実　　准教授
委員　中山 泰一　准教授
委員　大山 恵弘　准教授

著作権所有者

© 高野 保真 2015

Reducing Memory Allocations in Lazy Functional Programs

Yasunao Takano

Abstract

Lazy evaluation is an evaluation strategy in programming languages, especially in functional programming languages, which delays the evaluation of an expression until its value is demanded. Lazy evaluation brings advantages. One of the important advantages is to enable the program to avoid unnecessary evaluations. Another advantage is to help programmers write clear programs. For example, in case of linked list, programmers can divide a function into generation of a list and consumption of the generated list. Thus, lazy evaluation significantly contributes to the modularity of a program.

However, lazy evaluation has significant run-time overheads for building many as-yet unevaluated expressions, or thunks. Because thunk allocation is a space-consuming task, it is important to reduce the number of thunks in order to improve the performance of a lazy functional program. Previous researches have been devoted to static analysis algorithms to make functional programming language systems efficient. One of major static analysis algorithms is strictness analysis, which searches for expressions that need not to delay. If the algorithm finds such an expression, the compiler can generate code that does not allocate a thunk for this expression. Although strictness analysis algorithms effectively reduce thunk allocations in many cases, some thunks are unable to be reduced. For example, thunks for linked list cannot be decided statically whether they are necessary or not because we do not know how long the list is demanded. Thus, we need a dynamic approach.

To eliminate thunks dynamically, this paper proposes a mechanism named *thunk recycling*. Thunk recycling suppresses thunk generation by reusing and updating an already allocated thunk at the tail of a list, on the condition that the thunk is singly referred. This mechanism guarantees that reused thunks definitely satisfy this singly referred condition on the basis of a static analysis with program transformations.

This paper addresses a series of the research of thunk recycling.

First, we present the formal account of thunk recycling. We present a small-step operational semantics of the thunk recycling and show the correctness of this mechanism

on the basis of bisimulation. Intuitively speaking, the correctness means that internal states in which a program is executed are regarded as being the same except for the difference induced by thunk recycling.

Second, we present an sophisticated implementation of thunk recycling in Glasgow Haskell Compiler (GHC), which is a state-of-the-art and de facto standard compiler and runtime system for Haskell. On top of this implementation, we explain various design choices and implementation techniques. We also describe the knowledge of the implementation obtained from our experience that used GHC for the base of a programming language research. In particular, the fact that GHC is written by Haskell contributes to its extensibility.

Third, we present experimental result of our implementation of thunk recycling. We can see that there were plenty of eliminations for the total memory allocations. the results also show that the effect of thunk recycling on the execution times heavily depended on the target programs.

遅延評価に基づく関数型言語におけるメモリ割当量の削減

高野 保真

論文要旨

遅延評価は，値が実際に必要になるまで計算を遅らせる評価戦略である．必要になった値から計算するため，最終結果を求めるのに不要な計算を除去し，計算の最適化を目指すことができる．それと同時に，どの値が必要となるかという判断を処理系に任せることによって，プログラムに計算の進め方を記述する必要がなくなり，宣言的で簡潔なプログラムの記述につながる．たとえば，リストなどの再帰的データ構造を扱う際には，そのデータ構造を生成する処理と読み進める処理を分けて記述することができるため，遅延評価により得られる記述面での恩恵は大きい．

記述面での利点が多い一方で，遅延評価を行う言語処理系を実装するには，計算の遅延に必要となるオブジェクト（遅延オブジェクト，以下サンクと呼ぶ）について，十分考慮して処理系を設計する必要がある．特に，サンクをメモリ上に割り当てる時間的・空間的コストが問題となり，遅延評価によって不要な計算を除去できるとしても，プログラムの実行時のオーバヘッドが大きくなってしまうという問題点がある．そのため，効率的な遅延評価機構の実現を目指して，サンクの生成を抑制する静的解析手法について今まで多くの研究がなされてきた．たとえば，正格性解析は，プログラムの最終結果を求めるために必要となる計算を，プログラムから静的に解析する．値が必要となる式は遅延させずに済ませることができるため，その式に対するサンクを生成しない効率的なコードを生成することができる．多くのプログラムにおいて，正格性解析によりサンクの生成を抑えられることは，すでに確認されているが，プログラムの文面から得られる静的な情報のみを用いるため，動的なふるまいを考慮すれば削減可能と判断できるようなサンクは，正格性解析による削減の対象ではない．たとえば，リストをどれだけの長さ読み進めるかというような実行時に決定する要素があると，リストの遅延に必要となるサンクの生成を正格性解析のみで抑制することは難しい．

本論文は，サンクの削減という目的を達成するため，リストのような線形再帰的に定義される代数データ構造に注目し，既存のサンクを再利用する手法 Thunk Recycling を提案する．Thunk Recycling は，すでに割り当てられているサンクを破壊的に更新して再利用し，新たなサンクの生成を抑える．たとえば，リストであれば，後続のリストの生成を遅延するサンクを再利用できる．

本論文では，まず，Thunk Recycling の動作について述べ，その実現に必要となる機構についてまとめる．再利用を可能とするために，再利用が可能であるサンクを既存のサンクと区別して扱う．再利用機構は，破壊的な更新により矛盾が起こらないようにするコンパイル時の変換機構と，実行時に再利用を行う機構から構成される．プログラム変換の基本的な方針は，再利用可能なサンクへの参照を単一にすることである．また，実行時の再利用機構は，既存のサンクの生成・評価という仕組みの多くの部分を流用する．

次に，Thunk Recycling の形式的な定義と，その正しさの証明について述べる．簡単な関数型言語を定義し，その言語に対する Thunk Recycling のプログラム変換を定義した．さらに，サンクを再利用する操作的意味論を定義した．その意味論を用いて，Thunk Recycling の適用の有無により，プログラムのふるまいが変わらないことを証明した．

次に，関数型プログラミング言語 Haskell の処理系である Glasgow Haskell Compiler（GHC）への Thunk Recycling の実装について述べる．GHC は，Haskell の標準的な処理系であり，多くの研究の基盤として用いられ，新しい言語概念など先進的な研究成果が取り入れられている．本論文では，Thunk Recycling の機構の GHC における実装について，考えうる各種の設計方針と，それぞれの設計方針を選択するに至った設計上の得失に関して論じる．GHC は，その大部分が関数型言語である Haskell で記述されており，関数型言語による大規模で洗練されたシステムであるという面を持つ．そのため，Thunk Recycling の実装は，関数型言語による大規模なソフトウェアに対する開発事例の一例となっている．そこで，本論文では，関数型言語で書かれたプログラミング言語処理系に変更を加えるという観点から，遅延評価を行う関数型言語処理系の実装に関して得られた知見を論じる．

最後に，GHC 上の実装について，ベンチマークプログラムを用いた実験について述べる．実行時間に関しては，適用するプログラムを選ぶものの，再利用によって総メモリ割当量を削減できた．

目 次

図 目 次

表 目 次

第1章

序論

1.1 研究の背景

計算機は，機械語で書かれた一連の命令を順に実行していくが，その命令列を人間が機械語で記述するのは，労力を要する作業であり，誤りを生みやすい．そのような負担を軽減するため，人が理解しやすい形でプログラムを作ることができるように，プログラミング言語が開発されてきた．プログラミング言語によって記述されたプログラムは，プログラミング言語処理系によって，計算機が理解できる命令列に変換して実行される．このことで，プログラミング言語をインターフェースとして，実際に計算機で動く命令列からプログラムを切り離して，プログラムによって解くべき問題を抽象化してとらえることができる．

複雑で大規模なソフトウェアを開発するために，プログラミング言語が提供する抽象化機構は，ますます重要になってきた．その要求に応えるため，プログラムのモジュール化や記述性を重視した関数型プログラミング言語の研究が進み，関数型プログラミング言語を用いてプログラミングを記述する**関数プログラミング**が盛んになっている．関数プログラミングでは，処理を関数という単位に分けて定義し，関数を組み合わせることで，より高い抽象度でプログラムを記述する．

以下に，関数プログラミングの主な特徴を挙げる．

宣言的な記述 数式のように，関数を宣言的に記述する．つまり，関数プログラミングによるプログラムでは，計算機が何をするかという手続きを記述するのではなく，プログラムで解く問題の性質を記述することとなる．

副作用のない関数 変数への代入のような副作用がなく，関数は参照透明（referencially

transparent）である．ここで参照透明であるとは，関数の結果は，引数のみによって定まる性質のことを言う．この性質により，プログラマは関数の呼び出し順など，実行順を気にすることなくプログラムを記述することができる．

抽象データ型による処理 処理の対象を抽象データ型として，計算機上での実際の表現を隠蔽して扱う．特に，ユーザ定義のデータ型を代数的データ型（algebraic data type）として定義し，プログラム全体を代数的データ構造を扱う関数により構成することで，プログラムの抽象度を高めている．ここで，代数的データ型とは，データ構成子（constructor）によりタグ付けされて構成されるデータ型であり，付けられたタグに応じて処理を振り分けるパターンマッチングを利用できる．

上のような特徴から，関数プログラミングは，**遅延評価（lazy evaluation）**と相性がよい．遅延評価とは，値が実際に必要になるまで計算を遅らせる評価戦略である．基本的な遅延の対象は，関数やデータ構成子の引数であり，遅延を表す構文を導入する場合もある．たとえば，引数が遅延の対象であるとき，ある関数適用の引数に与えられる式の計算は遅延され，遅延された計算は，その値が実際に必要になったときに初めて処理される．必要になった値から計算するため，最終結果を求めるためには不要な計算を除去することができる．どの値が必要となるかという判断を処理系に任せることができるため，プログラマが計算の順序に頼ったプログラムを記述する必要がなくなるという利点がある．このことは，特にリストなどの再帰的データ構造を扱う際に，そのデータ構造の生成と消費という関数を分け，モジュール化された見通しのよいプログラムの記述につながる [18]．さらに，データ構造を扱う関数においても，役割ごとに関数を定義できるため，宣言的な記述により簡潔なプログラムを記述でき，関数プログラミングの効果を高めることができる．

遅延評価に基づく関数型プログラミング言語の歴史は古く，70 年代後半から 80 年代にかけて研究が始まった [17]．その中で 1985 年に，Research Software 社の販売するプログラミング言語 Miranda$^{\mathrm{TM}}$ [49] が出現した．Miranda は，商用目的で作られた最初の関数型プログラミング言語で，副作用のない関数，静的型システム，高階関数など，現在の関数プログラミングの基礎となる特徴を持った言語であった．特に，Miranda の文法は，等式による関数の定義，パターンマッチング，リスト内包表記（list comprehensions）などの要素により，記述の簡潔さを考えたものであった．1990 年代前半には，教育用途と商業用途の両面において広く使われるようになっていたが，処理系の実装や使用ライセンスなどを Research Software 社が 1 社のみで維持することができず，後続の関数型プログラミング言語である Haskell [23] に取って代わられることとなった．Haskell は，Miranda

の多くの利点を取り入れて，大学の研究者グループによって開発が始まったが，研究者のみでない幅広いユーザ層によるコミュニティによって維持されている点が，Miranda と異なっている．また，Haskell のプログラミング言語としての特徴は，遅延評価に加えて，代数的データ構造を用いた高い記述性，静的型付けと型推論による高い信頼性など，多くの理論的背景を持った機能を積極的に採用していることである．これらの機能を積極的に採用してきたのは，Haskell が当初からプログラミング言語研究の基盤となることを目指してきたことによる点が大きい．たとえば，Haskell は言語仕様 [23] において，以下の 5 つの項目を設計目標としてかかげている．

- 教育・研究・実用プログラムの開発において，大規模なものまで作るのに適していること
- 形式的な定義ができること
- 誰もが Haskell の処理系を実装できること
- 合意のとれたアイデアから構成すること
- 関数型プログラミング言語の無意味な多様化を避けること

このような目標に向けて，言語自体の発展と処理系の開発が急速に進められてきたことから，実用的な場面でも Haskell が用いられるようになってきた．たとえば，金融取引に Haskell を用いた例 [11]，システムの管理に Haskell を用いた例 [41]，Linux ディストリビューションの開発に用いた例 [4] などがあり，幅広い分野に渡って Haskell が使われている．

このようにして，遅延評価や Haskell は注目を集めているが，効率的に式を遅延するには，言語処理系において様々な工夫が必要となる．つまり，問題を抽象化して記述することが，記述性において大きな利点がある一方で，言語処理系にとってみれば，抽象的な記述と，計算機の理解できる命令列の間のギャップが大きくなるため，効率のよい実現が難しくなるという面がある．よって，遅延評価に基づく関数型プログラミング言語処理系に関する研究の主な課題は，プログラマが記述した抽象度の高いプログラムを，いかに最適化し，効率よく実行できる命令列に変換するかということにあるといっても過言ではない．

遅延評価においては，計算の遅延に必要となるオブジェクト（遅延オブジェクト，以下 **サンク** と呼ぶ）について，十分考慮して処理系を設計する必要がある．処理系内では，式を評価する代わりにサンクを割り当てるが，このことは時間的・空間的にコストがか

かる操作であり，遅延評価により結果に寄与しない計算を除去できるとしても，プログラムの実行時のオーバヘッドが大きくなってしまうという問題点がある．そのため，効率的に遅延評価を行う言語処理系においては，サンクの生成を抑制するために，様々な静的解析手法が開発されてきた．それらの中には，たとえば，正格性解析 [53] や Cheap eagerness analysis [8,9] といった解析があげられる．正格性解析は，必ず値が必要となるような式を静的に解析し，値が必要となる式に対しては，サンクを生成せずに済ませる．Cheap eagerness analysis は遅延させるまでもない軽い計算を解析し，遅延させずに，すなわちサンクを生成せずに前もって計算してしまう．これらの解析によって，遅延評価を前提としたプログラムの実行時間，使用メモリ領域ともに削減できることが確認されている．しかし，静的解析によって，すべての不要なサンクを除去できるわけではない．たとえば，長いリストをどれだけ読み進めるかというような，実行時に決定する要素が含まれる場合には，プログラムを動かしてみないと分からず，これらの解析のみを用いてサンクの生成を抑制することはできない．

1.2 本論文の目的と貢献点

本論文では，サンクの削減という問題に対し，リストのように線形に定義される再帰的データ構造に注目し，既存のサンクを再利用してサンクの生成を抑える手法（以下，**Thunk Recycling** と呼ぶ）を提案する．Thunk Recycling の目的は，すでに割り当てられているサンクの内容を破壊的に更新して再利用し，新たな遅延オブジェクトの生成を抑えることである．Thunk Recycling を用いれば，たとえば，代表的な再帰的データ構造である線形リストでは，tail 部の遅延に必要となるサンクを再利用することができる．また，Thunk Recycling は，サンクの内容を破壊的に書き換えることにより，矛盾が生じないようにする機構も合わせ持つ．具体的には，コンパイル時の静的なプログラム変換により，再利用するサンクへの参照が単一であるようにする．

Thunk Recycling の再利用対象のサンクは，1.1 節で述べた正格性解析などの静的な解析手法が削減の対象とするサンクと異なっており，削減できることが動的に判断できるようなサンクである．したがって，正格性解析を適用済みのプログラムについても Thunk Recycling を用いてサンクを削減可能である．

本論文は，Thunk Recycling における一連の研究をまとめたものであり，以下の貢献点がある．

- Thunk Recycling を提案した．

- Thunk Recycling の形式的な定義を与え，その正しさを証明した．
- Glasgow Haskell Compiler（GHC）[25] 上に Thunk Recycling を設計し，実装した．
- Thunk Recycling の効果をベンチマークプログラムにより確認した．
- 遅延評価を行う関数型プログラミング言語処理系の構成法に関して議論した．

形式的な定義では，Sestoft の抽象機械 [44] を元にした small-step の操作的意味論を定義した．small-step の操作的意味論では，ある言語で記述された計算の状態間での 1 ステップの簡約を，複数の規則からなる抽象機械として定義する．さらに，その定義を用いて Thunk Recycling の “正しさ” を**双模倣（bisimulation）**[42] の考え方と帰納法を用いて証明した．ここでいう正しさとは，Thunk Recycilng を適用したプログラムが，適用前のプログラムと同じふるまいをして同じ結果を導くことをいう．

GHC は，Haskell の標準的な処理系であり，関数型プログラミング言語に関する多くの研究の基盤となっている．また，コンパイラの多くの部分が Haskell で記述されているため，関数型プログラミング言語で記述された先進的で巨大なソフトウェアであるという面を合わせもつ．洗練された実装と，遅延評価に基づく関数プログラミングの基礎を築いた功績が評価され，2011 年の ACM/SIGPLAN Programming Languages Software Awards[1] に選ばれている．GHC が高度なソフトウェアであることにより，Thunk Recycling の実現には，多くの設計上の選択を行う必要があった．そこで本論文では，Thunk Recycling の実現に必要であった様々な設計上の判断や選択を，GHC の実装を考慮した上で考え得る利害得失の観点から評価し，遅延評価を行う言語処理系の実装に関して得られた知見も論じる．

ベンチマークプログラムによる実験により，再利用のおよぼす影響を確認した．実行時間の観点から見ると，適用するプログラムを選ぶものであるが，総メモリ割当量について，十分な削減効果が得られた．

1.3 本論文の構成

本論文の構成は以下のとおりである．2 章で，遅延評価を行うプログラミング言語処理系について説明する．そこでは，遅延評価における再帰的データ構造について説明し，既存の実装で必要となるサンクについて概観する．また，GHC の内部構造について説明

[1] `http://www.sigplan.org/Awards/Software/`

し，遅延評価を行う言語処理系の実例を示す．3 章では，Thunk Recycling の概要について述べる．4 章では，Thunk Recycling の形式的な定義に関して述べる．単純な関数型プログラミング言語を定義し，その言語上で，Thunk Recycling のプログラム変換に関して述べる．さらに，操作的意味論を定義し，その定義に基づいて Thunk Recycling の正しさを証明する．5 章では，GHC のリストに Thunk Recycling を組み込む実装の詳細を述べる．特に，GHC 上でサンクを再利用する際に，必要となる設計上の選択について詳細を述べる．6 章では，ベンチマークプログラムを用いた実験結果を報告する．7 章では，Thunk Recycling の実現により得られた知見について述べる．8 章では，関連研究について述べる．関連研究には，再利用に関連する研究とともに，GHC を基盤として実現された様々な手法についても述べる．最後に 9 章でまとめる．

Thunk Recycling は一般的な再帰的データ構造で必要となるサンクを削減することを可能とするが，本論文は線形リストを用いて説明する．また，本論文を通して，プログラミング言語 Haskell [5] の記法に従い，プログラム等を記述する．

第2章

遅延評価と遅延に基づく処理

本論文が提案する Thunk Recycling は，サンクの生成を抑えることで，遅延にかかる実行コストの削減を目指す．本章では，その背景となる遅延評価について，概要と実際の処理について述べる．遅延評価には，多くの利点がある一方で，効率のよいプログラミング言語処理系を実現するには，遅延にかかる実行コストを十分考慮する必要がある．そのことを，プログラミング言語 Haskell の処理系である Glasgow Haskell Compiler を例として，本章で説明する．

2.1 遅延評価

遅延評価は，式の評価をその値が必要になるまで遅らせるプログラミング言語の評価戦略である．関数呼び出しの引数やデータ構成子の引数の評価が遅延され，それらの値が必要になったときに，はじめて計算が進む．そのようにして，プログラムの実行を要求駆動的に進めることができるため，主に以下のような利点が生じる．

- 無駄な計算を除去することによる実行効率の向上
- 宣言的な記述を用いることによる記述性の向上

まず，一つ目の項目に関して，最終結果に寄与しない無駄な計算を省くことができれば，C などの手続き型言語に代表される先行評価の言語に比べて，潜在的には実行効率が良い．たとえば，以下の単純な関数 `first` を考える．

```
first :: Int -> Int -> Int
first a b = a
```

この関数は与えられた 2 つの引数のうち第一引数を返す．遅延評価に基づく言語では，関数の実引数は遅延の対象であるので，first 0 (heavy 1) という呼び出しにおいて，実引数の 0 と heavy 1 はともに遅延される．ここで，heavy 1 の計算が非常に時間がかかるとすると，遅延評価を採用したことによる効果が大きくなる．上で潜在的と限定していたのは，次節以降で述べるように計算の遅延にかかるコストを考慮する必要があるためである．

次に，二つ目の項目に関して，プログラムの実行順序を処理系に任せることができることにより，プログラムには処理の手順を記述するのではなく，そのプログラムの性質を記述することが可能となる．そのため，プログラムは宣言的になり，モジュール化を促進することができる [18]．特に，この特徴は抽象データ構造を扱う際に有効である．つまり，あるデータ構造に対して，そのデータを生成するプログラム記述と，そのデータを読み進めるプログラム記述を分離でき，プログラムの見通しがよくなる．たとえば，リストを生成する関数 enumFromTo とリストを読み進める関数 sum の例を考える．まず，Haskell 標準ライブラリの関数 enumFromTo は以下のように定義できる．

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo n m | m < n     = []
               | otherwise = n : enumFromTo (n+1) m
```

enumFromTo は引数に Int 型の値 n と m を受けとる関数である．m が n より小さいならば結果は空リストであり，そうでなければ，n を先頭とし，後続が enumFromTo の再帰呼び出しにより生成されるリストを構成する．ここで，: はリストのコンスセルを構成する中置のデータ構成子であり，その引数 (n と enumFromTo (n+1) m) は遅延される．そのようにして，引数に与えられた整数 n からはじまり，m までを要素とするリストを生成する．つまり，Haskell で一般的に使われる [n..m] という記法は，enumFromTo n m の構文糖衣である．

次に，整数のリストを読み進めて，要素の和を求める関数 sum は以下のように定義できる．

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

sum は，第一引数に Int 型のリストを受けとる．そのリストが空リストであれば，0 を返す．そうでなければ，引数のリストを先頭の Int 型の値 x と後続のリスト xs に分解

して（この分解を以下**パターンマッチ**と言う），x と sum xs を加えたものを値とする．sum の再帰呼び出しが後続のリスト xs に対してなされるため，リストが空になるまで和を求めることとなる．関数 sum において，引数の分岐が必要となった際に引数の評価が進むことに注意されたい．

たとえば，関数 enumFromTo と関数 sum を組み合わせ，1 から 10 までの数の合計を求める呼び出し sum（enumFromTo 1 10）は次のように評価が進む．

1. 関数 sum の定義にもとづいて，引数のリストが空リストであることを確認する．
2. そこで，遅延された enumFromTo 1 10 という式の評価が進む．
3. enumFromTo 1 10 の呼び出しで，まず 10 < 1 を評価する（厳密には，enumFromTo の引数である 1 も 10 も遅延されている）．
4. 10 < 1 が偽であるため，新たにリストを構成する．
5. よって，関数 sum の一つ目のパターンで，引数に与えられたリストが空でないことが確認できる．
6. 構成したリストを sum のパターンマッチにより分解し，1 と後続 sum xs の和を求める．

ここで，関数 enumFromTo がリストを生成するのに対して，関数 sum がリストを読み進めていることになる．そのため，関数 enumFromTo をリストの**生産者**，関数 sum をリストの**消費者**とみなして，リストを処理の中心とした機能にプログラムをモジュール化できる．たとえば，関数 enumFromTo が生成するリストを，以下のように定義する別の消費者の関数 map と組み合わせて利用することができる．

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

map（+1）（enumFromTo 1 10）とすれば，関数 enumFromTo が生成する 1 から 10 までのリストに対して，各要素に 1 を足したリストを得ることができる．ここで，関数 map は，リストを消費した結果，新たなリストを生成するので，消費者でありながら生産者でもある関数である．以上のように，関数を組み合わせてプログラムを構成することで，モジュール化により簡潔な記述が可能となる．

遅延評価には多くの利点がある一方で，以下のような欠点も挙げられる．

- 処理を手続き的に追うことは難しい
- 代入などの副作用のあるプログラミング言語と相性が悪い

一つ目の項目について，上の関数 enumFromTo と関数 sum を組み合わせた呼び出しの例から分かるように，プログラムはその性質を定義するものであって，順に動作を追うようなことには向かない．上の例では，手順 4 で enumFromTo によりリストを構成し，次の手順 5 では sum の定義に基づいて空リストであるか確認を行っており，リストの生産者を定義したプログラム上の場所と，消費者により実際に評価が進むプログラム上の場所が離れてしまう．そのため，関数の定義から，手続き的にプログラムの動作を手続き的に追うのは難しい．また，二つ目の項目も，副作用を伴う操作は計算の順序を考えざるをえないため，遅延評価には向かない．本論文では，副作用のないプログラムを対象とし，プログラマは明示的に副作用のある操作を記述できないものとする．

2.2　サンク

遅延評価を行う言語においては，式の評価を遅延するために処理系内にサンクを生成する．サンクは遅延する式とその式を評価するのに必要な環境からなるオブジェクトとして表現される．ここで，環境とは，変数と値の対応の集合である．本論文は，以下のような記法により，サンクを表記する．

$$T_{label}\{exp\}\{env\}$$

ここで，exp が遅延している式，env がその環境を示し，$label$ によりそれぞれのサンクを区別する．また，文脈より明らかな場合には，$\{exp\}\{env\}$ の部分を省略することもある．たとえば，2.1 節で示した関数 enumFromTo に対する enumFromTo 1 10 という呼び出しで得られるコンスセルは，以下のようになる．

$$T_0\{\texttt{n}\}\{\texttt{n} = 1\} : T_1\{\texttt{enumFromTo (n+1) m}\}\{\texttt{n} = 1, \texttt{m} = 10\}$$

コンスセルの head 部と tail 部それぞれに T_0，T_1 というサンクが生成される．T_0 が保持する遅延した式は n で，環境中では n が 1 に対応している．また，T_1 が保持する遅延した式は enumFromTo (n+1) m で，環境中では n が 1，m が 10 にそれぞれ対応している．サンクが保持する遅延した式は，関数 enumFromTo の定義本体ではなく，enumFromTo (n+1) m という関数呼び出しの式だということに注意されたい．また，サ

ンクの環境は，式の評価に必要となる変数の値を求めればよいため，遅延した式に含まれる自由変数に関する値を保持している．

もし，評価を遅延した式の値が実際に必要になれば，サンクに記憶しておいた式を，やはりサンクに記憶しておいた環境において評価する．この操作をサンクの**強制**（**force**）と呼ぶ．サンクの強制については，以下の点に注意が必要である．

- 副作用がない場合には，あるサンクの強制は高々一回で済ませる評価戦略（call-by-need）を採ることで，サンクが保持する式を繰り返し評価せずに済ませることができる．call-by-need を実現するには，あるサンクを参照していたオブジェクトが複数ある場合には，そのオブジェクト間で強制結果を共有しなければならない．
- サンクの強制中に，同じサンクが強制の対象となると無限ループである．したがって，ある強制中のサンクの強制結果が得られる前に，再びそのサンクを強制の対象とすることは，正しく動作するプログラムでは起きてはならない．
- サンクを強制せずに，プログラム中からサンクそのものを参照することはできない．つまり，サンクは処理系内に暗黙に作られるデータ構造であるため，プログラムから直接操作されることはない．

2.3 正格性解析

遅延評価には 2.1 節で説明したような利点があるが，同時に，遅延評価には式の遅延のためにサンクの割当て・強制という処理が必要である．サンクを割り当てるためには，メモリ空間が必要となるという空間的オーバヘッドに加えて，サンクの割当て処理を行うための時間的オーバヘッドを考慮しなくてはならない．また，サンクの強制に関しても，サンクから式を取り出してきて評価を開始し，強制結果を共有できるようにするなど，オーバヘッドがかかってしまう．これらのオーバヘッドの軽減を目指すには，正格性解析（strictness analysis）に代表される静的なプログラム解析により，サンクの量を減らすのが一般的である．

ある関数 f が第一引数について正格（strict）であるとは，停止しない計算を表す $\perp$ を用いて，以下の関係が成り立つときを言う．

$$f \perp = \perp$$

つまり，第一引数が停止しなければ，関数 f の呼び出しも停止しないことを意味する．そのため，関数 f が第一引数について正格であれば，関数 f の呼び出し時には第一引数に

ついて遅延する必要はなく，サンクを生成しなくてよい．正格性解析は，プログラム中の関数について，関数の正格性情報を収集し，遅延の必要のない式を解析する．たとえば，2.1 節で上げた enumFromTo であれば，enumFromTo が呼ばれたときには，必ず m < n という比較がなされるため，enumFromTo は第一引数，第二引数の両者に対して正格な関数であると分かる．そのため，enumFromTo が生成するリストの tail 部における自己再帰的呼び出し（enumFromTo（n+1）m）について，(n+1) や m という式を遅らせるサンクを生成する必要はない．同様にして，2.1 節の関数 sum でも，第一引数に与えられるリストは，空リストかどうかという判断に必ず用いられるため，遅延する必要がない．以上のようにして，それぞれの関数の正格性の情報を伝播していくことで，プログラム全体で遅延する必要のない式を求めることができる．

しかし，正格性解析のみで，すべての不要なサンクを取り除くことができるわけではない．たとえば，リストがどれだけ進むかということをもとに，サンクの生成を抑えるような解析を行うわけではない．そのため，生産者 enumFromTo と消費者 sum を組み合わせた式 sum（enumFromTo 1 10）に対して，生産者 enumFromTo の定義だけを見て enumFromTo が生成するリストの後続を遅延する必要がないという解析は，正格性解析のみではできない．

2.4　再帰的データ構造でのサンク

再帰的データ構造とは，線形リストや木などに代表されるように，その一部を取り出したとき，再びそのデータ構造が現れるようなデータ構造である．たとえば，各要素の型が a であるような線形リストは次のように定義することができる．

```
data List a = Nil | Cons a (List a)
```

ここで，データ構成子 Cons の第二引数が再帰的に List 型のデータを生成するため，後続も同様のデータ構造となり，List a は線形に連なるデータ構造を表現している．簡潔のため，enumFromTo の定義に用いていたように，List a を [a] と，Nil を [] と，Cons を中置記法の : を用いて記述する．

遅延評価を行う言語においては，データ構造も遅延性を持ち，その構成要素はサンクとして保持され必要になるまで評価されない．たとえば，前節で定義した関数 enumFromTo を用いた呼び出しを，サンクを明示して書くと次のようになる．ただしここでは，環境中で n や m に対応する値は，正格性解析の情報を用いて正格に評価するものとし，一方

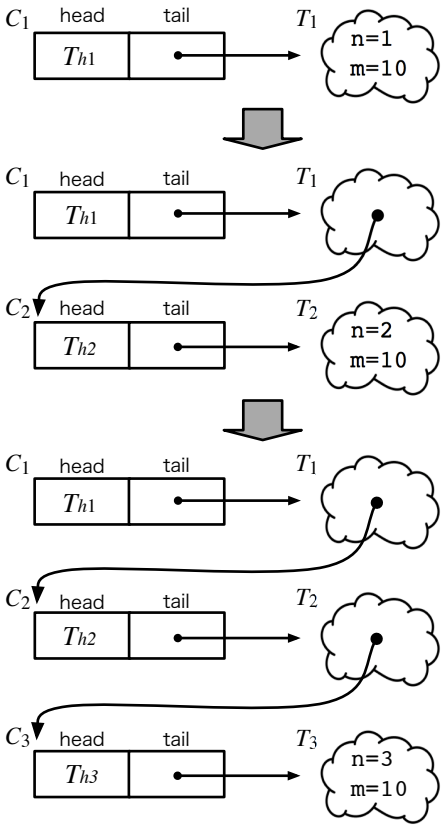

図 2.1: 遅延線形リストの評価

で，リストの後続は正格に評価せずに，リストを要求駆動的に読み進めるものとする．

$$\begin{aligned}
\texttt{enumFromTo 1 10} &\Rightarrow T_{h1} : T_1\{\texttt{enumFromTo (n + 1) m}\}\{\texttt{n} = 1, \texttt{m} = 10\} \\
&\Rightarrow T_{h1} : T_{h2} : T_2\{\texttt{enumFromTo (n + 1) m}\}\{\texttt{n} = 2, \texttt{m} = 10\} \\
&\Rightarrow T_{h1} : T_{h2} : T_{h3} : T_3\{\texttt{enumFromTo (n + 1) m}\}\{\texttt{n} = 3, \texttt{m} = 10\} \\
&\Rightarrow \cdots
\end{aligned}$$

まず，2.2 節で述べたように，コンスセルが構成される．ここでは，後続のサンクに注目するため head 部のサンク T_{h1} は略記している．リストを読み進めるときには，T_1 を強制し，T_{h2} と T_2 からなる後続のコンスセルが結果として得られる．以下同様に後続のサンクを強制していけば，要求駆動的にリストを生成することができる．

この様子を図 2.1 に示す．まず，初期状態として，head 部にサンク T_{h1}，tail 部にサンク T_1 を持ったコンスセル C_1 が割り当てられる．ここで，サンクは，環境だけ明示する雲型のオブジェクトとして表現し，コンスセルと区別している．このリストを読み進めるため，サンク T_1 を強制し，サンク T_2 を tail 部に保持するコンスセル C_2 が強制結果と

して得られる．このとき，call-by-need を実現するために，評価済みのサンクを，強制結果のオブジェクト（この例では C_2）へのポインタ（間接ポインタ，indirection pointer）で上書きするのが一般的である [21,22]．さらにリストの次を読み進め，T_2 を強制し，結果として C_3 が得られる．

ここで注目すべきは，リストを読み進めるごとに，T_1，T_2，T_3 といった新たなサンクをヒープ内に割り当てている点である．このような処理過程では，後続を遅延するためのサンクがリストの長さに応じて必要となる．

2.5　遅延評価処理系の例

本節は，遅延評価を行う言語処理系の例として，Haskell の標準的な処理系である Glasgow Haskell Compiler（GHC）について，遅延評価機構に関する部分と，Thunk Recycling の実装の説明に必要である項目に絞って説明する．具体的には，サンクの表現方法，強制の際の手順，ごみ集めについてなどである．説明の主眼としては，Thunk Recycling の実装を理解するのに最低限の知識を説明することにある．それに加えて，現実に利用されている遅延評価機構の動作を述べることは，サンクのふるまいを概観することにもつながる．なお，Thunk Recycling の実装時に変更を加えた GHC のバージョンは 7.0.3 であり，本論文の記述は基本的に GHC 7.0.3 に従う．

2.5.1　概要

GHC は 1989 年に開発が始まり，これまで多くの研究成果を導入しながら発展し，実用レベルのプログラムも Haskell を用いて作られるようになってきた．遅延評価を行う処理系は計算の遅延に必要となるコストが大きくなるため，実用プログラムまで動かすような高性能な処理系の実現は一般に難しいとされているが，GHC は設計段階から多くの手法・技術を取り入れることで，大規模なプログラムの動作環境に用いることを可能とした．それと同時に，GHC は拡張性を考えた構造となっているため，GHC を基盤として実現された研究が多くなされている [2,7,24,35,36]．たとえば，Optimistic Evaluation [7] は，遅延評価に対する投機的な評価を行う手法を提案し，GHC 上に実装している．また，ポインタタギングを用いたパターンマッチの実現法 [36] や Call-pattern specialization [24] は GHC の効率化を行うことにより，関数型言語処理系の実装に関する知見を得ている．

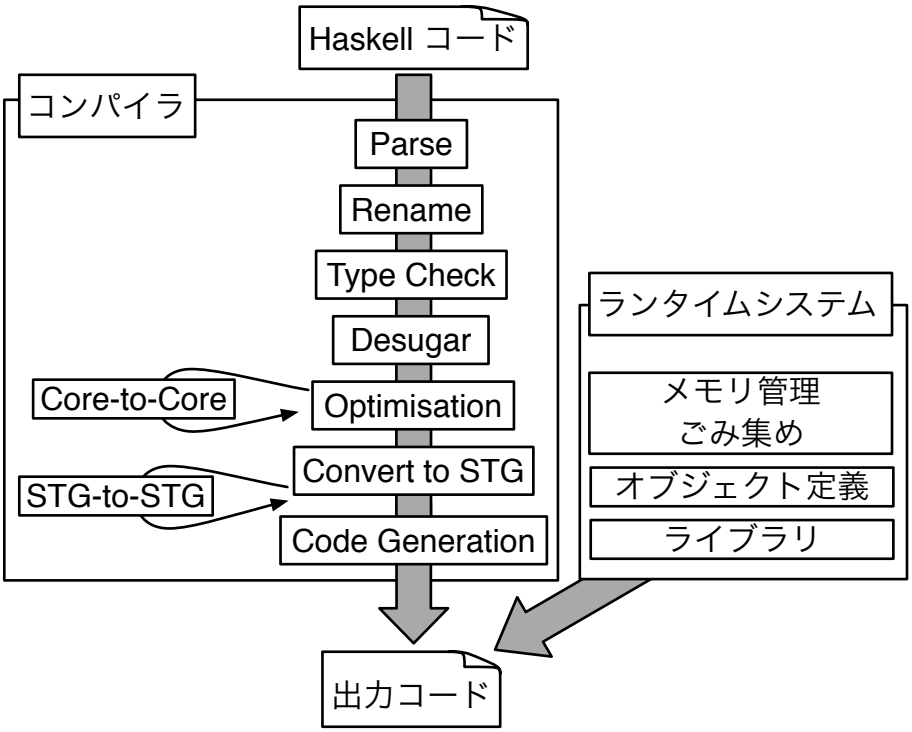

図 2.2: GHC の全体像

GHC の実装の大きな特徴は，コンパイラの大部分が Haskell 自身で書かれていることである．2011 年時のバージョン 7.0.3 は，23 万行の Haskell コードからなり，Haskell で記述されたプログラムとして，最大級のものである．一方，ランタイムシステムは，大部分が C 言語，一部について C−− [27] という内部言語で書かれており，合計で約 7 万行ある．

2.5.2 GHC の全体像

GHC は，大きく分けてコンパイラとランタイムシステムからなる．図 2.2 に GHC の全体像を示す．まず，Haskell のソースコードは，次節で述べるコンパイラのパイプラインに従ってコンパイルされる．このとき，コンパイルされたコード中には，ランタイムシステム中のコードの呼び出しが含まれており，出力コードとランタイムシステムがリンクされることによって実行可能バイナリが生成される．ランタイムシステムは，ごみ集めやサンクの強制結果を共有する組み込み処理などのコードを提供している．

2.5.3　コンパイラの内部構造

GHC への入力は Haskell コードであり，図 2.2 に示した段階を経て出力コードを得る．出力コードは，ネイティブコード・C 言語のコード・LLVM [30] コードから選択することができる．

各段階の処理内容は以下のとおりである．各段階で用いる中間言語については，2.5.4 節で後述する．

Parse Haskell ソースコードを構文解析し，抽象構文木にする．Alex と Happy というツールを用いている．

Rename 抽象構文木中のすべての識別子を完全修飾された名前に変更し，スコープの解決を行う．解決できない識別子があれば，コンパイルエラーとする．

Type Check 抽象構文木に対して，型検査を行い，型情報付きの抽象構文木にする．Desugar の前に型検査することが特徴である．これは，型エラーが起きた場合のメッセージを考えての設計である．

Desugar 型情報付きの抽象構文木を GHC の中間言語の一つである Core 言語 [45, 48] にする．

Optimisation インライン展開，デッドコード除去など種々の最適化を Core 言語から Core 言語への変換として行う．また，正格性解析もここで行う．

Convert to STG Core 言語を遅延評価機構の動作モデルに合わせた言語である STG 言語 [22] に変換する．また，STG 言語から STG 言語への変換というパスが用意されており，プロファイリング情報の収集などの処理が挿入されている．

Code Generation STG 言語のコードを別の内部言語である C−− 言語を経由して，ターゲットコードを出力する．

GHC が変更を想定しているのは，基本的に，これらのパイプライン中の各処理にまたがったものではなく，たとえば，Core 言語から Core 言語の変換や，STG 言語から STG 言語の変換などを想定している．これは，新たな最適化手法の実装には，それで十分であるという考えからである．図 2.2 中で Core-to-Core と STG-to-STG としてループで表しているパスがそれである

一方で，本論文で扱う Thunk Recycling の導入のためには，GHC の複数箇所に渡ってソースコードを見る必要があった．詳しくは 5 章で述べるが，たとえば，Convert to STG フェーズで再利用することが可能なサンクを特定し，それに対応するコードを Code Generation フェーズまで伝え，再利用のための C−− コードを生成する．

2.5.4 GHC の内部言語

GHC の内部言語には，Core 言語，STG 言語，C−− 言語の三種類がある．

Core 言語は，型付きラムダ計算を基盤とする System F_C [45] という体系に基づいた関数型の言語である．定義は非常にコンパクトで，変数，リテラル，関数適用，データコンストラクタ適用，ラムダ式，let 式，case 式からなる式に，型に関する情報を付加したものである．構文上は，関数やコンストラクタの引数に任意の式を指定できるが，STG 言語に変換される前の最終段階（CorePrep と呼ばれる）では，変数とリテラルに限定されるよう変換される．

STG 言語は，Core 言語に対して文法上の制限を加え，情報を付加した関数型言語である．STG 言語は Core 言語とは異なる以下のような主な特徴を持つ．

- 関数の引数は変数とリテラルに限定される．
- let 式で変数に束縛するのは，関数定義，コンストラクタ適用，サンクのいずれかである．ここで，Core 言語の let 式における変数への束縛値に関してサンクを作る必要のあるものは，STG 言語の let 式においてはサンクを表現するデータ構造を束縛値とする．
- 自由変数の計算や変数の生存解析など，様々な静的解析の結果を式中に保持している．
- Haskell ソースの型情報の大部分は捨てられている．

関数の引数が変数とリテラルに限定されるため，その変数が let 式で束縛されたものであれば非正格評価となることが明示される．

C−− 言語は，C 言語に近い文法を持つ手続き型言語である．STG 言語に変換されたコンパイル対象のプログラムを，内部のレジスタと呼ばれる領域に対するロード・ストア命令列からなる C−− のプログラムに変換し，最終的にアセンブリ言語のような低レベルの表現まで落とす．

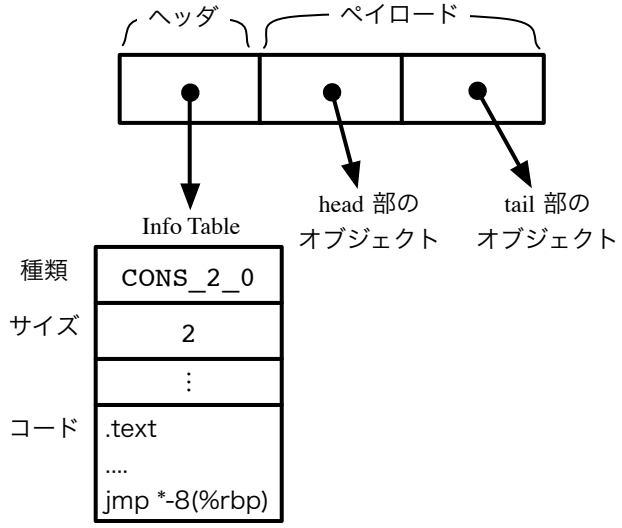

図 2.3: GHC 上でのコンスセル

2.5.5　オブジェクトの表現

GHC 中のオブジェクト（処理系内のデータ）は，**ヘッダ**情報と**ペイロード**と呼ばれるデータ領域からなる[1]．ヘッダ情報は，**Info Table** と呼ばれる構造により，オブジェクトの種類，コンパイル済みコード，ごみ集め時に用いる情報などを保持する．ペイロードは，そのオブジェクトの評価に必要となる他のオブジェクトへの参照を保持する．保持する情報は，オブジェクトの種類によって異なる．

たとえば，2.4 節で述べた Haskell のリストデータ構造のオブジェクトは，図 2.3 のように表現される．ペイロードに head 部と tail 部の要素を持ち，Info Table に以下のような情報を保持する．

種類　（図中 `CONS_2_0`）

二つのポインタを持つ代数的データ構造であることを示す．保持するペイロードの情報をオブジェクトの種類に含めたものであり，たとえば，`CONS_`m`_`n はペイロードにポインタを m 個，即値（非ポインタ）を n 個持つサンクを表す．この他にサイズの大きいデータ構造のための `CONSTR`，関数のための `FUN`，サンクのための `THUNK` など，全部で 59 種類のオブジェクトがある．

サイズ　（図中 2）

[1] ヘッダ部とペイロードの間にプロファイル用の情報を保持したオブジェクト表現になる場合もある．プロファイルの有無は，コンパイル時に指定される．

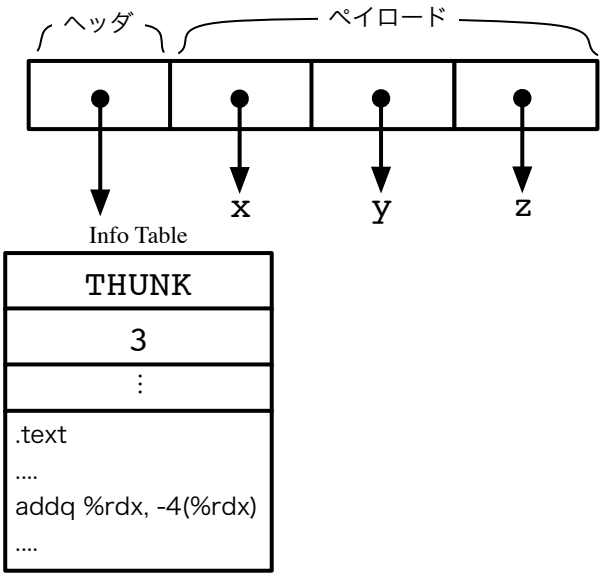

図 2.4: GHC 上でのサンク

ごみ集め時に必要となるペイロードの大きさを保持する．コンスセルの場合は head 部と tail 部への参照があるため 2 である．

コード 代数的データ構造の場合には，このオブジェクトをただ return するコンパイル済みコード（図 2.3 では `jmp *-8(%rbp)` 命令）を保持する．図中ではコンパイル済みコードを埋め込んだ形で示している．コンパイル時に指定することで，Info Table にコードへのポインタのみを保持させることもできる．

2.5.6 GHC の遅延評価機構

サンクの表現も，基本的には 2.5.5 節で述べたコンスセルなどの通常オブジェクトと同様である．サンクの環境へのポインタをペイロードに保持し，サンクのコードはコンパイルされた形で Info Table に保持する．たとえば，GHC は `x + y + z` という式を遅延するサンクを図 2.4 のように表現する．遅延する式のコンパイル済みのコードを Info Table に保持し，環境はペイロードに保持する．サンクに対応するオブジェクトの種類は `THUNK` である．使用頻度の高い小さなサイズのサンクに対しては，2.5.5 節の `CONS_2_0` のときと同様に `THUNK_1_0` や `THUNK_0_1` などと特殊化されたものを，効率化のために用意している．図 2.4 に示したオブジェクト構造を採用することにより，ペイロードのオフセットを用いて環境にアクセスするようにコンパイルすることが可能である．ここでは，コンパイル済みコード中の `x` はこのサンクの先頭から 1 ワード先，`y` は 2 ワード

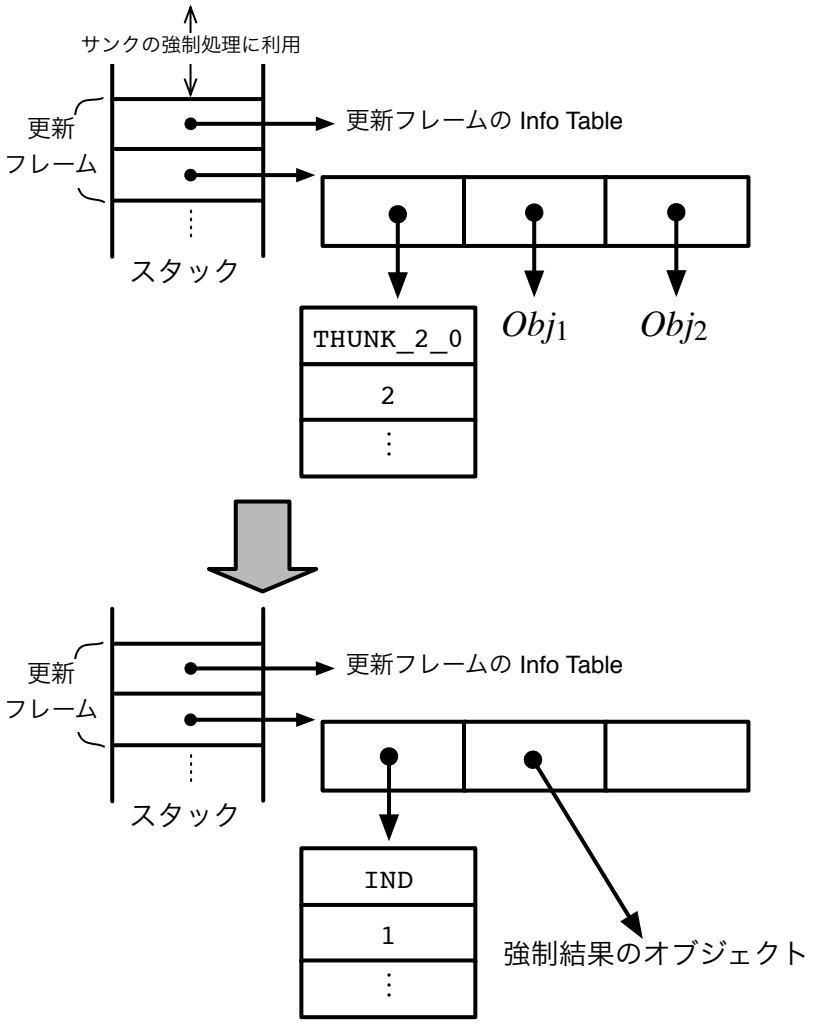

図 2.5: サンク強制におけるスタック

先，z は 3 ワード先というように，先頭からのオフセットを用いて参照することで，環境へのアクセスを効率化する．サンクのペイロードのサイズは，環境の大きさ，すなわち遅延する式中の自由変数の数となることに注意されたい．

サンクの強制は，**スタック**を利用することで実現される．GHC のスタックは，コンパイル済みコードを保持する Info Table への参照とそのコード中で必要となるオブジェクトへの参照を合わせて実行フレームとして順番に保持するような，処理系内の領域である．スタックは，処理系内に一つだけ確保されており，関数呼び出しと同様にサンクの強制にも使われる．

以下にサンクの強制手順を示す．そのときの様子は図 2.5 のようになる．

1. サンクの強制開始時に，強制後の処理を行う Info Table への参照と強制するサンクへのポインタからなる実行フレームをスタックに積む．この実行フレームはサンクの更新を行うため，特別に**更新フレーム**（update frame）と呼ばれる．

2. Info Table 内にある遅延した式に相当するコンパイルコードへジャンプすることでサンクの強制処理を始める．強制処理は，更新フレームより上方のスタックを用いて行う．

3. 強制結果が求まったら，強制結果を **レジスタ**と呼ばれる GHC 内のメモリ領域から参照できるようにする．その後，スタックの一番上にある更新フレームを参照し，更新フレームに含まれている強制後の処理を行う Info Table のコードにジャンプする．

4. 強制後の処理は，まず，更新フレームに保持するサンク（強制結果が求まったサンク）の Info Table を間接参照を示す `IND` とする．続いて，そのサンクのペイロードの先頭を強制結果のオブジェクトへの参照を持つように破壊的に書き換える．もし，これ以降の計算でこのサンクの値が再び必要となったときには，Info Table が `IND` となっているため，ペイロードの先頭にある強制結果への参照により，サンクを繰り返し評価することなく値を得ることができる．

5. 更新フレームをスタックから取り除き，強制を完了する．

2.5.7 メモリ管理

Haskell は評価戦略として遅延評価を採用し，副作用を許さない言語であるため，メモリ管理に関しても他の言語とは異なる特徴がある．

遅延評価を効率的に行うためには，メモリ管理において，特にサンクを十分に考慮しなくてはならない．メモリ管理に注目するとき，サンクには以下のような特徴があることが知られている．

- サンクは頻繁に割り当てられる．
- 多くのサンクは生成されて短い時間で強制される．
- 強制後のサンクの内容はごみになりやすい．

まず，1 つ目について，関数の実引数など多くの式が遅延の対象であるため，サンクのヒープへの割り当てが頻繁に起こる．2 つ目の項目について，多くのサンクは短命であると実験的に分かっている [7,43]．しかし，正格性解析などの静的解析では，プログラムの意味を変えずに短命のサンクの生成を抑制することは難しい．3 つ目の項目について，サンクの強制結果を共有することにより，一度強制されたサンクの中身は不要になる．つまり，サンクがペイロードに保持する環境は，強制が始まるとすぐにごみとなりやすい．

また，Haskell の純関数型言語としての特性から，明示的な代入操作はほとんどないこともメモリ管理の設計に大きく影響する．代入はなくても強制結果の共有のためにサンクの書き換えが処理系内で必要であり，参照の書き換えは頻繁に起こることには注意しなければならない．

以上のような Haskell の特性に基づき，GHC は次のようなメモリ管理をするよう設計されている．

- Bump アロケータ
- 世代別ごみ集め（generational GC）[33, 43]
- コピー式ごみ集め（copying GC）[10]

Bump アロケータとは，連続するまとまったメモリ領域をあらかじめ確保した上で，未使用部分の先頭を示すポインタ（フリーポインタ）を保持し，そこから順にオブジェクトを割り当てていくメモリ割り当て方式である．連続する領域を確保することで，フリーポインタを必要な分だけ進めればよく，非常に高速にオブジェクトを割り当てられる．この手法は，オブジェクト割当てが頻繁である Haskell において有効である．

世代別ごみ集めは，大部分のオブジェクトの生存期間は短く，長く生存し続けるオブジェクトはその後も生存する可能性が高いという経験則 [33] に着目した方式である．結果として，長く生存しているオブジェクトから参照されているものも，長く生存しつづけることとなる．このような経験則に基づいて，オブジェクトの生成時期に応じた世代（新世代と旧世代）にヒープを分割し，世代ごとにごみを回収することで，ごみ集めの効率化を図る．新世代で長く生存していると判断したオブジェクトは，旧世代に昇格（promotion）させる．新世代に対するごみ集め（**マイナーごみ集め**）を行えば，短い寿命のオブジェクトを回収できるため，ヒープ領域全体のオブジェクトを走査する必要がなくなる．さらに，適切な頻度で旧世代に対するごみ集め（**メジャーごみ集め**）を行うことで，旧世代に残ったごみを回収することができる．そのため，上で述べたように，サンクが比較的早くごみになる Haskell の特性と合っている．また，オブジェクトが保持している参照

を新世代のオブジェクトへ書き換えることがなければ，長く生存しているオブジェクトから参照されているものも，長く生存しつづけるという性質がある．この性質と Haskell で副作用を伴う処理が基本的にないことを考慮し，旧世代にあるオブジェクトから参照されているオブジェクトは，即座に昇格し，旧世代に置くようにしている．このことにより，マイナーごみ集めの対象となるオブジェクトを削減している．

コピー式ごみ集めは，ヒープ上の生きているオブジェクトだけを別の領域にコピーし，新たにそのコピー先の領域を使用するごみ集め方式である．その後，コピー元となった領域はまとめて捨てることで，ごみ集めができる．生きているオブジェクトが少なければ，ヒープ全体を走査する必要がないため，サンクが比較的短命であるという特性と合わせて考え，Haskell と相性がよい．GHC はマイナーごみ集めにおいて，コピー式ごみ集めを採用し，**Evacuation** と **Scavenge** という二段階に分けて実現している．Evacuation は，生きているオブジェクトをその移動先の世代へとコピーする処理であり，Scavenge はコピーされたオブジェクトのペイロードを走査し，Evacuation する処理である．

2.6 本章のまとめ

本章では，Thunk Recycling の背景となる遅延評価について述べた．遅延評価は，式の評価をその値が必要になるまで遅らせるプログラミング言語の評価戦略である．サンクと呼ばれるオブジェクトを言語処理系内に生成することで式の遅延を実現するが，サンクの割当て・強制という処理が必要となるため，遅延にかかわるオーバヘッドは大きい．これまで，オーバヘッドの削減を目指して，正格性解析に代表されるプログラム解析を導入し，必要のないサンクを静的に判別する方法が既存の処理系で用いられてきた．正格性解析によって多くの無駄なサンクが削減できることが広く知られているが，データ構造に必要となるサンクに関しては，削減の余地がある．

遅延評価を行う言語処理系の例として，Haskell の処理系である GHC の遅延評価機構についても説明した．GHC の遅延評価機構は，サンクを十分に考慮して設計されているため，GHC を基盤として実現された研究が多くなされており，Thunk Recycling も GHC を基盤とする．本章では Thunk Recycling の実装の理解に必要な項目に絞って説明した．

第3章

Thunk Recycling によるサンクの削減

本章は，本論文で提案するサンクの再利用機構である Thunk Recycling について説明する．まず，2 章で用いた enumFromTo の例を用いて，削減可能なサンクについて概観する．それに基づいて，非形式的ではあるが，直観的な形で Thunk Recycling を提案する．再利用のための破壊的な書き換えによって矛盾した状態になってはならないため，再利用できるサンクには満たすべき条件がある．それらの条件について，本章で述べる．

3.1 enumFromTo に対する Thunk Recycling の適用

本節では，前節の関数 enumFromTo に対して Thunk Recycling を適用する例を用いて，その動作を概観する．まず，リストを読み進めるごとにサンクを新たに生成する必要があったことに注目する．実際 enumFromTo の例では，図 2.1 で示したように T_1，T_2，T_3 を生成しており，ともに enumFromTo (n+1) m という同じ式を遅延するサンクであった．このことに注目し，Thunk Recycling は再帰的データ構造の後続にあるサンクの保持している式や環境を更新し，再帰的データ構造の後続を構成する際に再利用し，サンクの生成を抑制する．

再び Haskell の標準ライブラリ関数 enumFromTo をとりあげ，Thunk Recycling により抑制できるサンクの割り当てについて説明する．enummFromTo の定義を再掲する．

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo n m  | m < n     = []
                | otherwise = n : enumFromTo (n+1) m
```

Thunk Recycling を導入しなければ，リストを読み進めるごとに，コンスセルの後続の式を遅延するために新たなサンクをヒープ内に割り当てる必要があった．一方 Thunk Recycling では，`enumFromTo (n+1) m` を遅延するためのサンクを毎回割り当てるのではなく，一回割り当てたらそれを再利用して使いまわす．再利用の様子を `enumFromTo 1 10` という式により生成されるリストを読み進める場合を例に概観する．

$$
\begin{aligned}
\texttt{enumFromTo 1 10} &\Rightarrow T_{h1} : RT_1\{\texttt{enumFromTo (n + 1) m}\}\{\texttt{n} = 1, \texttt{m} = 10\} \\
&\Rightarrow T_{h1} : T_{h2} : RT_1\{\texttt{enumFromTo (n + 1) m}\}\{\underline{\texttt{n} = 2}, \texttt{m} = 10\} \\
&\Rightarrow T_{h1} : T_{h2} : T_{h3} : RT_1\{\texttt{enumFromTo (n + 1) m}\}\{\underline{\texttt{n} = 3}, \texttt{m} = 10\} \\
&\Rightarrow \cdots
\end{aligned}
$$

ここで RT_1 は再利用可能なサンクを示しており，T のときと同様に以下の記法とする．

$$RT_{label}\{exp\}\{env\}$$

`enumFromTo 1 10` を読み進める例では，RT_1 が保持する環境の中の `n` の値を更新し，関数 `enumFromTo` の初回の呼び出し時に生成される RT_1 を再利用している．Thunk Recycling では，リストを読み進めるごとに後続のサンクを生成しないので，2.4 節のように T_2，T_3 と次々にサンクを生成する場合と比べ，サンクの生成を抑えることができる．この様子は，図 2.1 に示した再利用のない場合の遅延線形リストの評価過程に従えば，図 3.1 のようになる．サンク RT_1 の内容を破壊的に書き換えて再利用し，サンクの生成を抑制する．図中の RT_1 のように，内容が破壊的に書き換えられて再利用されるサンクには，リサイクルの印を付けて図示している．再利用しない場合（図 2.1）は，サンクを間接参照で上書きすることによりリストを接続するが，再利用する場合（図 3.1）は，コンスセルの tail 部を直接破壊的に書き換えてリストを接続する．

ここで，サンクの持つ式や環境を更新することにより矛盾が生じないようにするため，Thunk Recycling では，対象となるサンクへの参照がただ一つであるような場合しか再利用を行なわない．たとえば，図 3.2 のように RT_1 が C_1' からのみでなく，別のオブジェクト O_1 からも参照されている場合は，矛盾が生じる．O_1 は {`n`=1, `m`=10} という環境を持つ RT_1 を参照していたにもかかわらず，リストを読み進めると，RT_1 の環境は {`n`=2, `m`=10} と破壊的に書き換えられてしまう．ここで，環境が {`n`=2, `m`=10} となった RT_1 を O_1 が参照して計算が進むと，元のプログラムとは異なる誤った結果を求めてしまう．このような事態を避けるため，Thunk Recycling は，コンパイル時のプログラム変換により，再利用対象のサンクがただ一つの参照によってしか指されないようにする．詳細

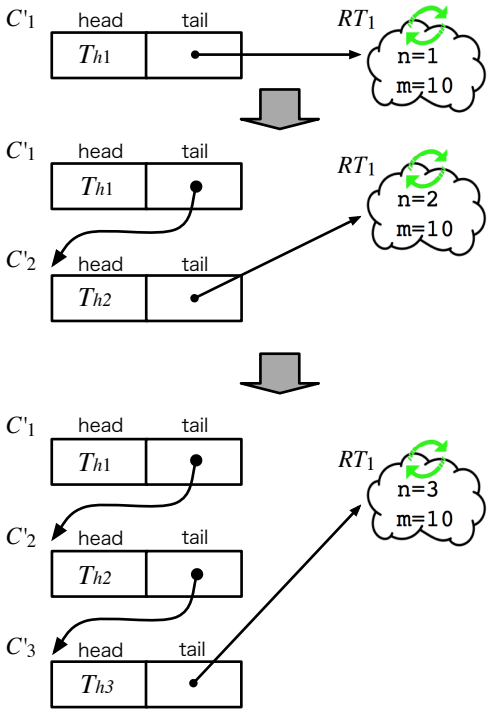

図 3.1: 遅延線形リストにおけるサンク再利用

については，3.3 節で述べる．なお，以降，本論文ではただ一つの参照で指されていることを，**単一参照性**が保証されているという．また，図 2.1 に示した既存の処理系で用いられている再利用されないサンクを**通常サンク**と，図 3.1 に示した再利用されるサンクを**再利用サンク**と呼ぶ．

3.2 Thunk Recycling の提案

Thunk Recycling を一般化してまとめると，以下のとおりである．

- コンスセルの tail 部の遅延に必要なサンクを再利用する．
- 再利用は，強制中のサンクの内容を破壊的に書き換えることで実現する．ここで，書き換えるサンクの内容は環境のみに限定されず，式を書き換えて用いる場合も考えられる．
- 再利用するサンクには，再利用後の内容を保持できるだけの十分なサイズが必要である．

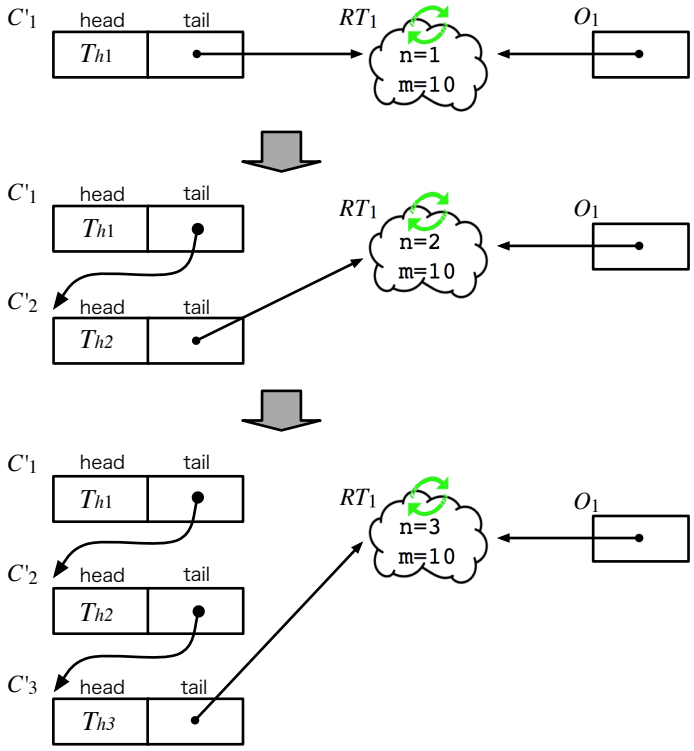

図 3.2: サンクを破壊的に書き換えることにより生じる矛盾した状態

- 破壊的な書き換えにより矛盾した状態になってはならない．

以上を踏まえて，Thunk Recycling の実装は，いくつかの機構に分けて考えることができる．

(a) 再利用サンクおよび，コンスセルの tail 部を破壊的に書き換える機構

(b) 再利用可能なサンクを判別する機構

(c) サンクを破壊的に書き換えた結果，矛盾が生じないようにする機構

(d) ランタイムシステムで再利用サンクを扱うための機構

(a) は，再利用する時に，内容を書き換えるべき再利用サンクやコンスセルを発見し，破壊的な書き換えを行う機構である．つまり，現在強制中の再利用サンクと，そのサンクを tail 部から参照しているコンスセルを発見できなければならない．(b) は，コンパイル時に判別を行う機構と，実行時に判別を行う機構の 2 つで構成される．コンパイル時は，以下の 2 点を満たすサンクであることを静的に判別し，そのようなサンクを再利用するコードを生成する．

- 再利用可能なサンクはコンスセルの第二引数を遅延するものでなくてはならない.
- すべての再利用サンクは，強制される前に別の式を遅延するために再利用されてはならない.

実行時に判別するのは，再利用可能なサンクが実際に再利用するのに十分なサイズかということである．(c) は書き換え前の状態のサンクを参照しているものが 2 個以上存在しないようにすることが基本方針である．上で述べたように，再利用に際しては，再利用サンクの単一参照性が保証されていなければならない．(d) は，再利用サンクのオブジェクトの種類を追加し，ごみ集めなどにおいて再利用サンクを適切に扱うために必要である．

GHC への実装については，(a) を 5.3 節，(b) を 5.4 節，(c) を 5.5 節でそれぞれ説明する．それらを合わせて 5.6 節で再利用機構の全体を説明する．(d) に関しては，追加したオブジェクトの種類については 5.2 節で，ごみ集めについては 5.8 節で説明する．

3.3 単一参照されるサンク

サンクを再利用することにより矛盾が生じないようにするため，再利用対象となるサンクへの参照がただ一つであるような場合だけに再利用を限定する．そのため，コンパイル時のプログラム変換により，再利用対象のサンクがただ一つの参照によってしか指されないようにする．Thunk Recycling は，対象となるサンクが単一参照されていることに基づいて再利用を行う．このような前提を置くことで，サンクの再利用時に破壊的に書き換えるべき参照を，コンスセルの tail 部だけに限定することができる．本節では，単一参照性について考える．

対象の再利用サンクへの参照が増える場合は，

- リストの消費
- リストの生産

の二通りに分けて考えられる．注意すべきなのは，ある関数がリストの消費者・生産者のどちらか一方だけに属すとは限らず，両者になりうることである．たとえば，2.1 節の関数 `enumFromTo` はリストを生産し，関数 `sum` はリストを消費するが，2.1 節で示した Haskell の標準関数 `map` はリストを消費しつつ生産する関数である．

3.3.1 リストの消費

リストの内容を使用する関数で，サンクへの参照が増える場合について考える．Haskell においては，パターンマッチがデータ構造を分解し，その要素への参照を増やす．たとえば，以下の関数 f を考える．

```
f :: [a] -> (a, [a])
f (x:xs) = (x, xs)
```

関数 f はパターンマッチで取り出された xs をペアの要素とする．変数 xs に束縛されているサンクは，すでにコンスセル x:xs の tail 部から参照されているため，関数 f はこのサンクに対する参照を増やしてしまう．しかし，静的なプログラム変換を用いて単一参照性を保証できる．

ここでのプログラム変換は，次のようなものである．

> リストへのパターンマッチによって導入された変数による，コンスセルの tail 部への直接的な参照を，コンスセルからの間接的な参照へと置換する．

つまり，コンスセルを通してのみ tail 部の参照が可能であるように，プログラム全体を変換する．たとえば，上で示した関数 f を以下のような関数 f’ に変換する．

```
f’ :: [a] -> (a, [a])
f’ xxs@(x:xs) = (x, xxs*)
```

ここで，* は tail 部での間接参照を意味する．xxs* の値が必要となれば，コンスセル xxs の tail 部をたどって値を返すこととする．もし，tail 部をたどった先が再利用サンクであれば，強制しその結果が xxs* の値である．この変換により，関数 f’ では，再利用サンクとなりうるコンスセルの tail 部のオブジェクトは，コンスセルを通した間接的な参照のみでアクセスされることとなる．xxs* の強制が必要となったということは，弱頭部正規形（Weak Head Normal Form）まで簡約を行うことが要求されているので，後続のサンクそのものを強制せずに結果として返すことはない．また，サンクは第一級のオブジェクトではないため，プログラム中で明示的に後続のサンクを参照することはできない．つまり，すべてのパターンマッチにおいて，* を用いた間接参照に置き換えてしまえば，リストの後続として指定されている再利用サンクを直に参照することは不可能である．

上の例で，tail 部の再利用サンクが直接指されることがないことは，図 3.3 のように，tail 部以外からの参照がコンスセルを通して行われるようになることから分かる．プログ

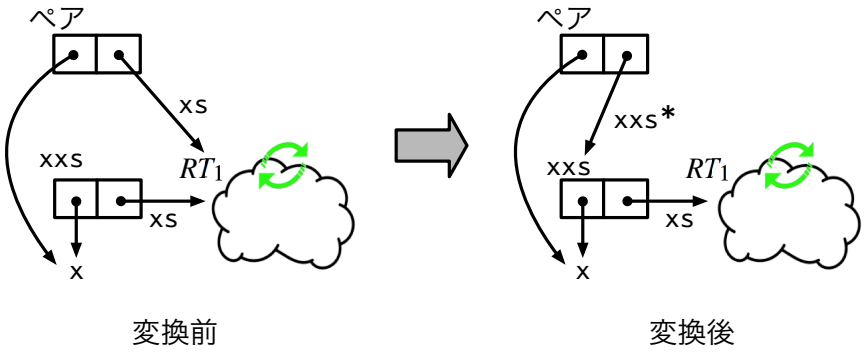

図 3.3: ∗ による間接参照

ラム変換前では，関数 f の返り値であるペアの第二要素が RT_1 を直接参照しているが，変換後では，ペアの第二要素は，xxs の tail 部を通して RT_1 を間接参照している．

3.3.2 リストの生産

リストを生成する際，tail 部に指定されるサンクが共有される場合を考慮する必要がある．

たとえば，以下の関数 g1 は，tail 部に指定されるサンクを束縛する xs が共有されていない．

```
g1 :: a -> [a]
g1 x = let xs = [x] in x:xs
```

[x] を遅延するサンクは関数 g1 の返り値のコンスセルの tail 部からのみ参照されている．そのため，[x] を遅延するためには，再利用サンクを用いることができる．

これに対し，以下のようなリストを生成する関数 g2 は，再利用サンクを使用できない．

```
g2 :: a -> ([a], [a])
g2 x = let xs = [x] in (x:xs, xs)
```

let 式により xs に束縛される [x] を遅延するサンクが，ペアの第一要素と第二要素の間で共有されているため，g2 はサンクへの参照を明らかに増やしてしまう．したがって，g2 では [x] を遅延するのに再利用可能なサンクを割り当てることはできない．

これらの例から分かるように，サンクを割り当てる際には，それを束縛する変数への参照が一箇所からであることを確認しなければならない．一般的には，リストの生産時

はコンスセルの共有関係を変えずに単一参照性を保つことができず，g2 のような関数では本再利用機構の適用をあきらめ，通常サンクを用いる必要がある．

以上より，サンクの生産時に対しては，

> 複数で共有されないと静的に解析できる式の遅延のみに，再利用サンクを使う

こととする．このことにより注意すべきなのは，Thunk Recycling を適用したプログラムの実行時には，コンスセルの tail 部には，通常サンクと再利用サンクのいずれも出現し得ることである．静的に解析できる範囲にのみ再利用サンクを用いることは，厳密に言えば，再利用可能であるサンクを見逃すこととなる．つまり，再利用サンクの内容を書き換えた時点で複数から参照されていなければ，矛盾した状態にはならないが，Thunk Recycling の静的解析は，そこまでは踏み込まない．

3.4　Thunk Recycling のリスト以外への適用について

以上の議論は，線形リストに対するものであるが，木などの複数箇所で再帰定義されるデータ構造に関しても Thunk Recycling を適用可能である．線形再帰的データ構造であれば，再利用できる可能性のあるサンクは一意に定まるため，リストと同様にして適用できる．しかし，二分木などの複数箇所で再帰的に定義されるデータ構造に関しては，どの式でサンクを再利用するかを決定しなくてはならない．たとえば，Haskell の単純な二分木は，以下のように定義できる．

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

二分木であるので，データ構成子 `Node` の第二引数と第三引数に `Tree a` 型を用いて再帰定義されている．このとき，たとえば，左部分木（あるいは右部分木）にあたる第二引数（あるいは第三引数）の再帰定義に対してサンクを再利用して用いると静的に決めれば，本論文の内容をそのまま適用できる．基本的な考え方はリストの場合と同じで，単一参照性を木の生産者と消費者において保証すればよい．

ただし，複数箇所で再帰的に定義されるデータ構造に対して，再利用する箇所を静的に決めることにより，再利用の機会を逃す場合もある．つまり，上のように左部分木に対してのみ再利用すると決めてしまったときに，右部分木が深い木では，ほとんどの部分で再利用できない．そのため，木の形により再利用箇所を静的に判別するか，再利用する箇所を動的に決定するなど，線形再帰的データ構造ではなかった問題が存在する．

本論文では，簡単のため，GHC への実装も含めて線形リストのみを対象としている．木など複数箇所で再帰的に定義されるデータ構造への対応については，今後の課題とする．

3.5 本章のまとめ

本章では，Thunk Recycling の概要を直観的な形でまとめた．Thunk Recycling は，すでに割り当てられているサンクを再利用し，新たなサンクの生成を抑える．再利用の対象とするのは線形リストの後続のリストの生成を遅延するサンク，つまり，コンスセルの tail 部の遅延に必要なサンクであり，そのサンクの内容を破壊的に更新して再利用する．破壊的な更新により矛盾が生じないようにするため，リストの生産と消費のそれぞれにおいて，再利用対象のサンクが単一参照であることを保証する変換をプログラムのコンパイル時に適用する．それと合わせて，実行時には，再利用サンクに十分なサイズがあるか判別した上で再利用する．

Thunk Recycling は，線形リスト以外の再帰的データ構造にも適用することができる．二分木など複数の箇所で再帰的に定義されるデータ構造に関しては，どの式でサンクを再利用するかを決定しなければならない．本論文では，簡単のため，線形リストに絞って議論する．

第4章

Thunk Recycling の形式化と正当性

3 章のアイデアを元に，本章では Thunk Recycling の形式化について扱う．まず，単純な関数型プログラミング言語 SF を定義し，SF のプログラムに対して単一参照性が保証されたプログラムに変換するアルゴリズムを定義する．その上で，言語 SF の動作を与える意味論を定義し，その意味論のもとで簡約を進めた場合に，再利用の有無によってプログラムの意味が変わらないことを証明する．

4.1 言語 SF

Thunk Recycling の形式化に用いる言語 SF を，図 4.1 に示す．この文法は，GHC の内部言語である Core 言語を簡素化したものであるが，関数適用の引数が変数に限定されている点において STG 言語と似ている．引数を変数に限定しているのは，サンクの生成を明示できることに加えて，後で示す変換を簡潔に定義するためである．サンクの生成は let 式によって表現するが，STG 言語のようにサンクを表現するデータ構造を明示的に記述することはない．

まず，識別子は x と x^* を用い，必要な場合には x_t や x_t^* のように下付き添字を付けて区別する．* は，3.3 節で述べた再利用サンクの間接参照に用いる構文である．簡単のため，プログラム中のすべての識別子は異なるものが指定されているとする．

Thunk Recycling を利用しない場合の式を Expr^- とする．式が Expr^- の要素であることを陽に表す場合は e^- のように右上に $-$ を付けて，後述の Expr の式と区別する．Expr^- の要素は，ラムダ式（$\backslash x \to e^-$），コンストラクタ（Nil，$\mathsf{Cons}\, v_h^-\, v_t^-$），変数（$v^-$），関数適用式（$e^-\, v^-$），case 式，let 式からなる．特に以下の点に注意されたい．

識別子

$$
\begin{aligned}
x &\in \mathit{Identifier} \\
x^* &\in \mathit{Marked\ Identifier}
\end{aligned}
$$

式（サンクの再利用なし）

$$
\begin{aligned}
e^- \in \mathit{Expr}^- &::= \backslash x \to e^- \mid \mathsf{Nil} \mid \mathsf{Cons}\, v_h^-\, v_t^- \mid v^- \mid e^-\, v^- \mid \mathsf{case}\, e^-\, \mathsf{of}\, x\, p^- \\
&\mid \mathsf{let}\, x = e_1^-\, \mathsf{in}\, e_2^- \\
v^- \in \mathit{Variable}^- &::= x \\
p^- \in \mathit{Pattern}^- &::= \{\, \mathsf{Nil} \to e_n^-;\ \mathsf{Cons}\, x_h\, x_t \to e_c^- \,\}
\end{aligned}
$$

式（サンクの再利用あり）

$$
\begin{aligned}
e \in \mathit{Expr} &::= \backslash x \to e \mid \mathsf{Nil} \mid \mathsf{Cons}\, v_h\, v_t \mid v \mid e\, v \mid \mathsf{case}\, e\, \mathsf{of}\, x\, p \\
&\mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid \mathsf{letreuse}\, x = e_1\, \mathsf{in}\, e_2 \\
v \in \mathit{Variable} &::= x \mid x^* \\
p \in \mathit{Pattern} &::= \{\, \mathsf{Nil} \to e_n;\ \mathsf{Cons}\, x_h\, x_t^* \to e_c \,\}
\end{aligned}
$$

図 4.1: 言語 SF

(1) 変数には v^- を用い，識別子の x と区別する．これは，後述の Thunk Recycling を利用する場合に，識別子の x と間接参照の印のついた識別子 x^* をまとめて，変数 v と扱うためである．

(2) 関数およびコンストラクタ Cons への引数は変数に制限されている．そのため，ヒープへのサンクの割り当ては，let 式における局所変数への束縛値の式を遅延させる時だけに限定される．ここでのサンクは，まだ通常サンクである．

(3) 強制は $\mathsf{case}\, e^-\, \mathsf{of}\, x\, p$ の e^- の部分でのみ起こる．この式は，e^- の強制結果を識別子 x に束縛した上で，p に基づくパターンマッチ処理を行う．なお，データ構造はリストだけなので，パターンマッチ処理では Nil の場合と Cons の場合の分岐を行う．

(4) let 式で，変数に束縛される値は遅延され通常サンクが生成される．(1) より，通常サンクが生成されるのは，let 式においてのみである．

(5) ラムダ式，case 式，let 式，パターンにおける新しい変数の導入時には，一意となるような変数名が割り当てられる．

Thunk Recycling を利用する場合の式 *Expr* は，$Expr^-$ に letreuse 式とパターンにおいて間接参照を示す識別子への * 印を追加したものである．*Expr* の式は，3.3 節で基本的なアイデアを述べた変換を，$Expr^-$ の式に適用した結果得られる．変換の形式的な定義は 4.2 節で述べる．$\mathsf{letreuse}\, x = e_1 \,\mathsf{in}\, e_2$ は，再利用して用いるサンクを明示する構文である．これは let 式に似ているが，e_1 の計算を遅延させるのに，再利用サンクが生成されていなければ新たに生成し，すでに生成されていればそれを再利用する点が let 式と異なる．また，パターンマッチの結果の変数に間接参照を指定するため，case 式のパターンが $Expr^-$ の場合と異なる．3.3.1 節で述べたように，プログラム中のすべてのパターンで間接参照を用いることによって，再利用サンクの単一参照性を崩さないようにできる．識別子 x と間接参照を示す印のついた x^* を合わせて変数 v として扱う．let 式で導入する x やパターンの x_t などの識別子が指定可能な式中の箇所と，関数適用 $e\,v$ などの変数を指定可能な箇所が区別されていることに注意されたい．

4.2 変換アルゴリズム

3 章では，Thunk Recycling の動作を概観した．そのアイデアに基づいた単一参照性を保証するプログラム変換は，

リストの消費者に関する変換 $\mathcal{C}$: パターンマッチにおける間接参照の導入

リストの生産者に関する変換 $\mathcal{P}$: 再利用するサンクが明示された式への変換

の 2 つからなる．変換 $\mathcal{C}$ は，3.3 節における `f` から `f'` への変換すなわち，コンスセルのパターンマッチにおいて，束縛される変数に印をつける．また，変換 $\mathcal{P}$ では，どのサンクを再利用サンクにすることができるかを解析し，特定する．なお，3.3 節で説明した，リストを生成する側での単一参照性については，変換 $\mathcal{P}$ で考慮する必要がある．はじめに変換 $\mathcal{C}$ を行って，その後に変換 $\mathcal{P}$ を行う．つまり，以下の手順により，Thunk Recycling によりサンクの再利用が可能なプログラムを得ることができる．

$$expr^- \xrightarrow{\text{変換 } \mathcal{C}} expr' \xrightarrow{\text{変換 } \mathcal{P}} expr$$

ここで，$expr^-$ が $Expr^-$ で書かれた元となるプログラム，$expr'$ は letreuse 式を含まない *Expr* のプログラムで，*expr* が単一参照性が保証された再利用サンクを用いる *Expr* のプログラムとなる．

$$S \in \mathit{Set} \;\; ::= \;\; \phi \mid S,\, x$$

$$\begin{array}{rcl}
\mathcal{C} & :: & \mathit{Expr}^{-} \to \mathit{Set} \to \mathit{Expr} \\
\mathcal{C}[\![\backslash x \to e^{-}]\!]\; S & = & \backslash x \to \mathcal{C}[\![e^{-}]\!]\; S \\
\mathcal{C}[\![\mathsf{Nil}]\!]\; S & = & \mathsf{Nil} \\
\mathcal{C}[\![\mathsf{Cons}\; v_h^{-}\; v_t^{-}]\!]\; S & = & \mathsf{Cons}\; (\mathit{replace}\; v_h^{-}\; S)\; (\mathit{replace}\; v_t^{-}\; S) \\
\mathcal{C}[\![v^{-}]\!]\; S & = & \mathit{replace}\; v^{-}\; S \\
\mathcal{C}[\![e^{-}\; v^{-}]\!]\; S & = & (\mathcal{C}[\![e^{-}]\!]\; S)\; (\mathit{replace}\; v^{-}\; S) \\
\mathcal{C}[\![\mathsf{case}\; e^{-}\; \mathsf{of}\; x\{ & & \mathsf{case}\; (\mathcal{C}[\![e^{-}]\!]\; S)\; \mathsf{of}\; x\{ \\
\quad \mathsf{Nil} \to e_n^{-}; & = & \quad\; \mathsf{Nil} \to \mathcal{C}[\![e_n^{-}]\!]\; S; \\
\quad \mathsf{Cons}\; x_h\; x_t \to e_c^{-}\; \}]\!]\; S & & \quad\; \mathsf{Cons}\; x_h\; x_t^{*} \to \mathcal{C}[\![e_c^{-}]\!]\; (S,\, x_t) \\
\mathcal{C}[\![\mathsf{let}\; x = e_1^{-}\; \mathsf{in}\; e_2^{-}]\!]\; S & = & \mathsf{let}\; x = \mathcal{C}[\![e_1^{-}]\!]\; S\; \mathsf{in}\; \mathcal{C}[\![e_2^{-}]\!]\; S
\end{array}$$

$$\begin{array}{rcl}
\mathit{replace} & :: & \mathit{Variable}^{-} \to \mathit{Set} \to \mathit{Variable} \\
\mathit{replace}\; x_t\; S & = & \mathsf{if}\; x_t \in S\; \mathsf{then}\; x_t^{*}\; \mathsf{else}\; x_t
\end{array}$$

図 4.2: プログラム変換 $\mathcal{C}$ と 補助関数 *replace*

以降，それぞれのプログラム変換の定義を示す．

4.2.1　プログラム変換 $\mathcal{C}$

図 4.2 に変換 $\mathcal{C}$ と補助関数の *replace* を示す．変換 $\mathcal{C}$ は，変換対象の Expr^{-} と Set を受けとり，*Expr* を返す．前述したように，パターンマッチにより分解されたコンスセルの tail 部に相当する変数に $*$ 印をつけることで，リストの消費時の再利用サンクの単一参照性を実現する．引数に受けとる Set は，$*$ 印をつけるべき変数の集合を保持する．そのため，$\mathsf{case}\; e^{-}\; \mathsf{of}\; x\; \{ \cdots ;\; \mathsf{Cons}\; x_h\; x_t \to e_c^{-}\; \}$ を変換するとき，識別子 x_t を Set に入れた上で，e_c^{-} について $\mathcal{C}$ を再帰的に呼び出す．補助関数 *replace* は，Set に変数が入って

いるか判定し，必要があれば実際に変数を置き換える．

4.2.2 プログラム変換 $\mathcal{P}$

変換 $\mathcal{P}$ は，let 式で束縛される変数について，束縛値に再利用サンクを使うことができれば letreuse に置換する．ここで，let の局所変数 x が次の条件をすべて満足する場合に，x の束縛値のサンクは再利用可能であると判断する．

(1) Cons の第二引数を遅延するものである．

(2) 実行時に再利用サンクへの参照を増やさない．

(3) すべての再利用サンクは，強制される前に別の式を遅延するために再利用されてはならない．

(2) の再利用サンクの単一参照性について，前述の変換 $\mathcal{C}$ で対処したが，変換 $\mathcal{P}$ でも考慮しなければならない．(3) の条件は，あるサンクを破壊的に書き換えることで，矛盾した状態となることを避けるために必要である．ここで，矛盾した状態になり得る間違った（let から letreuse への）置き換え例を示す．まず，置き換え前のプログラムを以下に示す．

$$
\begin{aligned}
&\mathsf{let}\ x_0 = e_0\ \mathsf{in}\\
&\mathsf{let}\ x_1 = e_1\ \mathsf{in}\\
&\mathsf{let}\ x_2 = e_2\ \mathsf{in}\\
&\mathsf{case}\ e_3\ \mathsf{of}\ x_3\ \{\ \mathsf{Nil} \to \mathsf{Cons}\ x_0\ x_1;\\
&\qquad\qquad \mathsf{Cons}\ x_h\ x_t \to \mathsf{Cons}\ x_0\ x_2\}
\end{aligned}
$$

このプログラムにおいて，x_1 と x_2 はともに Cons の第二引数を遅延するものであり，実行時に参照が増えることもないため，上の再利用を可能とする条件（1）と（2）を同時に満たす．そこで，let を letreuse に置換すると，以下のようなプログラムとなる．

$$
\begin{aligned}
&\mathsf{let}\ x_0 = e_0\ \mathsf{in}\\
&\mathsf{letreuse}\ x_1 = e_1\ \mathsf{in}\\
&\mathsf{letreuse}\ x_2 = e_2\ \mathsf{in}\\
&\mathsf{case}\ e_3\ \mathsf{of}\ x_3\ \{\ \mathsf{Nil} \to \mathsf{Cons}\ x_0\ x_1;\\
&\qquad\qquad \mathsf{Cons}\ x_h\ x_t \to \mathsf{Cons}\ x_0\ x_2\}
\end{aligned}
$$

$$
\begin{array}{rcl}
\mathcal{P} & :: & \mathit{Expr} \to \mathit{Expr} \\
\mathcal{P}[\![\backslash x \to e]\!] & = & \backslash x \to \mathcal{P}[\![e]\!] \\
\mathcal{P}[\![\mathsf{Nil}]\!] & = & \mathsf{Nil} \\
\mathcal{P}[\![\mathsf{Cons}\ v_1\ v_2]\!] & = & \mathsf{Cons}\ v_1\ v_2 \\
\mathcal{P}[\![v]\!] & = & v \\
\mathcal{P}[\![e\ v]\!] & = & \mathcal{P}[\![e]\!]\ v \\
\mathcal{P}[\![\mathsf{case}\ e\ \mathsf{of}\ x\ \{ & & \mathsf{case}\ \mathcal{P}[\![e]\!]\ \mathsf{of}\ x\ \{ \\
\quad \mathsf{Nil} \to e_n; & = & \quad \mathsf{Nil} \to \mathcal{P}[\![e_n]\!]; \\
\quad \mathsf{Cons}\ x_h\ x_t^* \to e_c\ \}]\!] & & \quad \mathsf{Cons}\ x_h\ x_t^* \to \mathcal{P}[\![e_c]\!]\} \\
\mathcal{P}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] & = & \begin{array}{l}
\mathsf{if}\ \ \mathit{spine}\, e_1\, x = \mathit{None}\ \ \&\&\ \ \mathit{spine}\, \mathcal{P}[\![e_2]\!]\, x = \mathit{Once} \\
\qquad \&\&\ \ \mathit{occ}\, e_1\, x = \mathsf{T}\ \ \&\&\ \ \mathit{occ}\, e_2\, x = \mathsf{T} \\
\mathsf{then}\ \ \mathsf{letreuse}\ x = \mathcal{P}[\![e_1]\!]\ \mathsf{in}\ \mathcal{P}[\![e_2]\!] \\
\mathsf{else}\ \ \mathsf{let}\ x = \mathcal{P}[\![e_1]\!]\ \mathsf{in}\ \mathcal{P}[\![e_2]\!]
\end{array}
\end{array}
$$

図 4.3: プログラム変換 $\mathcal{P}$

もし，e_1 と e_2 の遅延に，同一の再利用サンクを破壊的に書き換えて用いると，x_1 と x_2 が同じ再利用サンクを参照することとなり，強制されていない再利用サンクを破壊的に書き換えてしまうこととなる．この場合は，矛盾した状態となり得るため，連続した letreuse 式を導入しないように変換 $\mathcal{P}$ を設計した．

プログラム変換 $\mathcal{P}$ を図 4.3 に示す．また，補助関数 *occ* と *spine* の定義を図 4.4 に示す．変換 $\mathcal{P}$ の入力は変換 $\mathcal{C}$ を適用済みのプログラムなので，入力には letreuse 式が与えられることはない．そのため，変換 $\mathcal{P}$ は *Expr* から *Expr* への部分関数である．let 式について，letreuse に置換できるかどうかを，補助関数 *occ* と *spine* を用いて解析する．*occ* は，式と変数を受けとり，その変数が式中の Cons の第二引数以外に出現しないことから (1) を確認する．また，*spine* は (2) と (3) を確認する関数である．*spine* で，letreuse が連続していないことも同時に確かめ，連続していた場合は *NoMore* を返し，(3) に対応した判定を行う．ここで，$\uparrow$ は $\mathit{None} < \mathit{Once} < \mathit{NoMore}$ という順序で値を比較し，大きいほうを返す関数である．

$$
\begin{array}{rcl}
\mathit{Reuse} & = & \mathit{None} \mid \mathit{Once} \mid \mathit{NoMore} \\
\mathit{spine} & :: & \mathit{Expr} \to \mathit{Variable} \to \mathit{Reuse} \\
\mathit{spine}\,(\backslash x_1 \to e)\,v & = & \mathit{None} \\
\mathit{spine}\,\mathsf{Nil}\,v & = & \mathit{None} \\
\mathit{spine}\,(\mathsf{Cons}\,v_1\,v_2)\,v & = & \mathsf{if}\ v_2 = v\ \mathsf{then}\ \mathit{Once}\ \mathsf{else}\ \mathit{None} \\
\mathit{spine}\,v_1\,v & = & \mathit{None} \\
\mathit{spine}\,(e\,v_1)\,v & = & \mathit{None} \\
\mathit{spine}\,(\mathsf{case}\ e\ \mathsf{of}\ x_s\ \{ & & \mathsf{if}\ \mathit{spine}\,e\,v = \mathit{None} \\
\quad \mathsf{Nil} \to e_n; & = & \mathsf{then}\ \mathit{spine}\,e_n\,v \uparrow \mathit{spine}\,e_c\,v \\
\quad \mathsf{Cons}\,x_h\,x_t^* \to e_c\})\,v & & \mathsf{else}\ \mathit{NoMore} \\
 & & \mathsf{if}\ \mathit{spine}\,e_1\,v = \mathit{None} \\
\mathit{spine}\,(\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2)\,v & = & \mathsf{then}\ \mathit{spine}\,e_2\,v \\
 & & \mathsf{else}\ \mathit{NoMore} \\
\mathit{spine}\,(\mathsf{letreuse}\ x_1 = e_1\ \mathsf{in}\ e_2)\,v & = & \mathit{NoMore}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{occ} & :: & \mathit{Expr} \to \mathit{Variable} \to \mathit{Bool} \\
\mathit{occ}\,(\backslash x_1 \to e)\,v & = & \mathit{occ}\,e\,v \\
\mathit{occ}\,\mathsf{Nil}\,v & = & \mathsf{T} \\
\mathit{occ}\,(\mathsf{Cons}\,v_1\,v_2)\,v & = & v_1 \neq v \\
\mathit{occ}\,v_1\,v & = & v_1 \neq v \\
\mathit{occ}\,(e\,v_1)\,v & = & \mathit{occ}\,e\,v\ \&\&\ v_1 \neq v \\
\mathit{occ}\,(\mathsf{case}\ e\ \mathsf{of}\ x_s\ \{ & & \\
\quad \mathsf{Nil} \to e_n; & = & \mathit{occ}\,e\,v\ \&\&\ \mathit{occ}\,e_n\,v\ \&\&\ \mathit{occ}\,e_c\,v \\
\quad \mathsf{Cons}\,x_h\,x_t^* \to e_c\})\,v & & \\
\mathit{occ}\,(\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2)\,v & = & \mathit{occ}\,e_1\,v\ \&\&\ \mathit{occ}\,e_2\,v
\end{array}
$$

図 4.4: プログラム変換 $\mathcal{P}$ の補助関数

4.2.3　変換の例

以下の $expr^- \in Expr^-$ というプログラムを用いて，変換 $\mathcal{C}$ と $\mathcal{P}$ の適用例を示す．

$$
\begin{aligned}
expr^- = {} & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathsf{let}\ x_1 = \mathsf{let}\ x_2 = \mathsf{Cons}\ x_{nil}\ x_{nil} \\
& \qquad\qquad \mathsf{in\ Cons}\ x_{nil}\ x_2 \\
& \quad \mathsf{in\ Cons}\ x_{nil}\ x_1)\ \mathsf{of}\ x_{p1}\ \{ \\
& \mathsf{Nil} \to \mathsf{Nil} \\
& \mathsf{Cons}\ x_{h1}\ x_{t1} \to \mathsf{case}\ x_{t1}\ \mathsf{of}\ x_{p2}\ \{ \\
& \qquad\qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad\qquad \mathsf{Cons}\ x_{h2}\ x_{t2} \to x_{t2}\}\}
\end{aligned}
$$

図 4.5 に変換 $\mathcal{C}$ に $expr^-$ を与えた場合の変換過程を示す．図中では，適用可能な場合は $\mathcal{C}$ を並列に適用し，変換過程を適宜省略している．注目するのは，以下の 3 点である．

- case 式が変換対象となったときにはパターンを変換する（図 4.5 中（2）の $\mathsf{Cons} x_{h1} x^*_{t1}$ と（3）の $\mathsf{Cons}\ x_{h2}\ x^*_{t2}$）．
- パターンを変換したときには，その部分式の変換時の Set に tail 部の識別子を追加する（図 4.5 中（2）の (x_{t1}) と（3）の (x_{t2}, x_{t1})）．
- Set にある識別子が変数として出現したときには，$*$ 印の付いた識別子で置き換える（図 4.5 中（4）の x^*_{t1} と x^*_{t2}）．

変換 $\mathcal{C}$ によりパターンマッチの変数に間接参照を示す $*$ が付加されたので，続いて，変換 $\mathcal{P}$ を適用する．変換 $\mathcal{C}$ を適用した結果の式に対して，図 4.6 のように変換 $\mathcal{P}$ を適用する．

まず，(1) で変換対象となる let 式は，識別子 x_{nil} がコンスセルの tail 部のみで用いられておらず，かつ参照が単一でないため，補助関数は，それぞれ以下の値となり，letreuse 式への置き換え条件を満たさない．

$$
\begin{aligned}
& spine\ \mathsf{Nil}\ x_{nil} = None \\
& spine\ (\mathcal{P}[\![\mathsf{case}\ (\mathsf{let} \ldots) \ldots]\!])\ x_{nil} = NoMore \\
& occ\ \mathsf{Nil}\ x_{nil} = \mathsf{T} \\
& occ\ (\mathsf{case\ let} \ldots)\ x_{nil} = \mathsf{F}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{C}[\![expr^{-}]\!]\ \phi \ \Rightarrow\ & \mathsf{let}\ x_{nil} = \mathcal{C}[\![\mathsf{Nil}]\!]\ \phi \\
& \mathsf{in}\ \mathcal{C}[\![\mathsf{case}\ (\mathsf{let}\ldots)\ \mathsf{of}\ x_{p1}\ \{ \\
& \qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad \mathsf{Cons}\ x_{h1}\ x_{t1} \to \mathsf{case}\ x_{t1}\ \mathsf{of}\ x_{p2}\ \{\ldots\}]\!]\ \phi
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
\Rightarrow\ & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathcal{C}[\![\mathsf{let}\ldots]\!]\ \phi)\ \mathsf{of}\ x_{p1}\ \{ \\
& \qquad \mathsf{Nil} \to \mathcal{C}[\![\mathsf{Nil}]\!]\ \phi \\
& \qquad \mathsf{Cons}\ x_{h1}\ x^{*}_{t1} \to \mathcal{C}[\![\mathsf{case}\ x_{t1}\ \mathsf{of}\ x_{p2}\ \{\ldots\}]\!]\ (x_{t1})
\end{aligned}
\tag{2}
$$

$$
\begin{aligned}
\Rightarrow\ & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathsf{let}\ldots)\ \mathsf{of}\ x_{p1}\ \{ \\
& \qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad \mathsf{Cons}\ x_{h1}\ x^{*}_{t1} \to \mathsf{case}\ (\mathcal{C}[\![x_{t1}]\!]\ (x_{t1}))\ \mathsf{of}\ x_{p2}\ \{ \\
& \qquad\qquad\qquad \mathsf{Nil} \to \mathcal{C}[\![\mathsf{Nil}]\!]\ (x_{t1}) \\
& \qquad\qquad\qquad \mathsf{Cons}\ x_{h2}\ x^{*}_{t2} \to \mathcal{C}[\![x_{t2}]\!]\ (x_{t2}, x_{t1})\}\}
\end{aligned}
\tag{3}
$$

$$
\begin{aligned}
\Rightarrow\ & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathsf{let}\ldots)\ \mathsf{of}\ x_{p1}\ \{ \\
& \qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad \mathsf{Cons}\ x_{h1}\ x^{*}_{t1} \to \mathsf{case}\ x^{*}_{t1}\ \mathsf{of}\ x_{p2}\ \{ \\
& \qquad\qquad\qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad\qquad\qquad \mathsf{Cons}\ x_{h2}\ x^{*}_{t2} \to x^{*}_{t2}\}\}
\end{aligned}
\tag{4}
$$

図 4.5: プログラム変換 $\mathcal{C}$ の適用例

$$
\begin{array}{l}
\mathcal{P}[\![\mathsf{let}\ x_{nil} = \mathsf{Nil} \\
\quad \mathsf{in\ case}\ (\mathsf{let}\ldots)\ \mathsf{of}\ x_{p1}\ \{ \\
\qquad \mathsf{Nil} \to \mathsf{Nil} \\
\qquad \mathsf{Cons}\ x_{h1}\ x^*_{t1} \to \mathsf{case}\ x^*_{t1}\ \mathsf{of}\ x_{p2}\ \{\ldots\}]\!]
\end{array}
\tag{1}
$$

$$
\begin{array}{ll}
\Rightarrow & \mathsf{let}\ x_{nil} = \mathcal{P}[\![\mathsf{Nil}]\!] \\
& \mathsf{in\ case}\ (\mathcal{P}[\![\mathsf{let}\ x_1 = \mathsf{let}\ x_2 = \mathsf{Cons}\ x_{nil}\ x_{nil} \\
& \qquad\qquad\qquad\qquad \mathsf{in\ Cons}\ x_{nil}\ x_2 \\
& \qquad\qquad \mathsf{in\ Cons}\ x_{nil}\ x_1]\!])\ \mathsf{of}\ x_{p1}\ \{ \\
& \quad \mathsf{Nil} \to \mathcal{P}[\![\mathsf{Nil}]\!] \\
& \quad \mathsf{Cons}\ x_{h1}\ x^*_{t1} \to \mathcal{P}[\![\mathsf{case}\ x^*_{t1}\ \mathsf{of}\ x_{p2}\ \{\ldots\}]\!]
\end{array}
\tag{2}
$$

$$
\begin{array}{ll}
\Rightarrow & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathsf{letreuse}\ x_1 = \mathcal{P}[\![\mathsf{let}\ x_2 = \mathsf{Cons}\ x_{nil}\ x_{nil} \\
& \qquad\qquad\qquad\qquad \mathsf{in\ Cons}\ x_{nil}\ x_2]\!] \\
& \qquad \mathsf{in}\ \mathcal{P}[\![\mathsf{Cons}\ x_{nil}\ x_1]\!])\ \mathsf{of}\ x_{p1}\ \{ \\
& \quad \mathsf{Nil} \to \mathsf{Nil} \\
& \quad \mathsf{Cons}\ x_{h1}\ x^*_{t1} \to \mathsf{case}\ \mathcal{P}[\![x^*_{t1}]\!]\ \mathsf{of}\ x_{p2}\ \{\ldots\}
\end{array}
\tag{3}
$$

$$
\begin{array}{ll}
\Rightarrow & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathsf{letreuse}\ x_1 = \mathsf{letreuse}\ x_2 = \mathcal{P}[\![\mathsf{Cons}\ x_{nil}\ x_{nil}]\!] \\
& \qquad\qquad\qquad\qquad \mathsf{in}\ \mathcal{P}[\![\mathsf{Cons}\ x_{nil}\ x_2]\!] \\
& \qquad \mathsf{in\ Cons}\ x_{nil}\ x_1)\ \mathsf{of}\ x_{p1}\ \{ \\
& \quad \mathsf{Nil} \to \mathsf{Nil} \\
& \quad \mathsf{Cons}\ x_{h1}\ x^*_{t1} \to \mathsf{case}\ x^*_{t1}\ \mathsf{of}\ x_{p2}\ \{\ldots\}
\end{array}
\tag{4}
$$

図 4.6: プログラム変換 $\mathcal{P}$ の適用例

次に，(2) で変換対象となる let 式は，識別子 x_1 について，それぞれ以下のようになり，letreuse 式への置き換え条件を満たす．

$$
\begin{aligned}
&\mathit{spine}\,(\mathsf{let}\ x_2 \ldots)\ x_1 = \mathit{None}\\
&\mathit{spine}\,(\mathcal{P}[\![\mathsf{Cons}\ x_{nil}\ x_1]\!])\ x_1 = \mathit{Once}\\
&\mathit{occ}\,(\mathsf{let}\ x_2 \ldots)\ x_1 = \mathsf{T}\\
&\mathit{occ}\,(\mathsf{Cons}\ x_{nil}\ x_1)\ x_1 = \mathsf{T}
\end{aligned}
$$

そこで，(3) のように letreuse 式へと置き換える．

続いて，(3) で変換対象となる let 式も (2) の場合と同様に，letreuse 式への置き換え条件を満たす．

$$
\begin{aligned}
&\mathit{spine}\,(\mathsf{Cons}\ x_{nil}\ x_{nil})\ x_2 = \mathit{None}\\
&\mathit{spine}\,(\mathcal{P}[\![\mathsf{Cons}\ x_{nil}\ x_2]\!])\ x_2 = \mathit{Once}\\
&\mathit{occ}\,(\mathsf{Cons}\ x_{nil}\ x_{nil})\ x_2 = \mathsf{T}\\
&\mathit{occ}\,(\mathsf{Cons}\ x_{nil}\ x_2)\ x_2 = \mathsf{T}
\end{aligned}
$$

最終的に，間接参照を示す * のついた識別子と，letreuse 式が導入された以下の expr が得られた．

$$
\begin{aligned}
\mathit{expr} = {}& \mathsf{let}\ x_{nil} = \mathsf{Nil}\\
&\mathsf{in\ case}\ (\mathsf{letreuse}\ x_1 = \mathsf{letreuse}\ x_2 = \mathsf{Cons}\ x_{nil}\ x_{nil}\\
&\qquad\qquad\qquad\qquad \mathsf{in\ Cons}\ x_{nil}\ x_2\\
&\qquad\quad \mathsf{in\ Cons}\ x_{nil}\ x_1)\ \mathsf{of}\ x_{p1}\ \{\\
&\quad \mathsf{Nil} \to \mathsf{Nil}\\
&\quad \mathsf{Cons}\ x_{h1}\ x^*_{t1} \to \mathsf{case}\ x^*_{t1}\ \mathsf{of}\ x_{p2}\ \{\\
&\qquad\qquad\qquad \mathsf{Nil} \to \mathsf{Nil}\\
&\qquad\qquad\quad \mathsf{Cons}\ x_{h2}\ x^*_{t2} \to x^*_{t2}\}\}
\end{aligned}
$$

4.3　操作的意味論

Expr と $Expr^-$ のそれぞれに対して，Sestoft の abstract machine [44] を元とした操作的意味論を定義する．*Expr* の場合には，ある状態 Q_1 に対して，操作的意味論の規則 μ で一段階簡約し，状態 Q_2 になることを以下のように記述する．

$$Q_1 \stackrel{\mu}{\Longrightarrow} Q_2$$

$Expr^-$ の場合にも同様に，以下のように記述する．

$$Q_1^- \stackrel{\mu^-}{\Longrightarrow} Q_2^-$$

ここで，Q_1 や Q_1^- などの状態をヒープ，環境，簡約対象，スタックからなる四つ組 $H;\ t;\ \Gamma;\ S$ あるいは $H^-;\ t^-;\ \Gamma^-;\ S^-$ で表現する．再利用のない場合の状態の要素は，式の場合と同様に右肩に $-$ をつけることで再利用のある場合の要素と区別する．

もし，Q_1 の簡約対象である式が，Q_1^- の簡約対象の式をプログラム変換した結果得られた式であれば，それぞれの簡約で新たに変数・識別子を導入する際には，同じ名前で導入されるものとする．つまり，その場合には，Q_2 と Q_2^- の環境に含まれる識別子の集合は等しくなる．

Sestoft による abstract machine との主な違いは，以下の 2 点である．

ヒープと環境の関係 4.3.1 節で詳しく述べるが，環境の表現が変数からヒープの場所となっている点で異なる．Sestoft のものでは，変数とアドレスの組であった．これに対し本研究では，ヒープの場所という考え方を入れることで，$*$ 印の付いた変数に必要となる間接参照の表現が簡潔になる．ある変数の値を得る際に，ヒープから直接アドレスを引くのではなく，変数ごとに付加された間接参照であることを示すフラグを確認したうえでヒープの内容をとる．

サンクの表現法 変数名の置き換えを適宜行い，グローバルな環境を 1 つ持つようにすることによって，サンク毎に環境を保持しない．このことで，サンクの表現が簡潔になり，サンクの再利用の記述の見通しがよくなる．

4.3.1　四つ組による状態の表現

四つ組の要素を図 4.7 に示すように定義する．

$$
\begin{array}{rcl}
a & \in & \mathit{Address} \\
f & \in & \mathit{Flag} ::= \mathsf{T} \mid \mathsf{F}
\end{array}
$$

$$
\begin{array}{rcl}
w^- \in \mathit{Whnf}^- & ::= & \lambda x \,.\, e^- \mid [\,] \mid h_h^- : h_t^- \\
h^- \in \mathit{Location}^- & ::= & \langle a \rangle \\
t^- \in \mathit{ExprOrWhnf}^- & ::= & e^- \mid w^- \mathbin{@} a \\
H^- \in \mathit{Heap}^- & ::= & \phi \mid H^-, a \mapsto C^- \\
C^- \in \mathit{Content}^- & ::= & w^- \mid a \mid (e^-, f)_N \\
\Gamma^- \in \mathit{Environment}^- & ::= & \phi \mid \Gamma^-, x \mapsto h^- \\
S^- \in \mathit{Stack}^- & ::= & \phi \mid [\mathsf{P} p^-] +\!\!+ S^- \mid [\mathsf{A} h^-] +\!\!+ S^- \mid [\mathsf{B} x] +\!\!+ S^- \mid [\mathsf{N} a] +\!\!+ S^-
\end{array}
$$

$$
\begin{array}{rcl}
w \in \mathit{Whnf} & ::= & \lambda x \,.\, e \mid [\,] \mid h_h : h_t \\
h \in \mathit{Location} & ::= & \langle a, f \rangle \\
t \in \mathit{ExprOrWhnf} & ::= & e \mid w \mathbin{@} a \\
H \in \mathit{Heap} & ::= & \phi \mid H, a \mapsto C \\
C \in \mathit{Content} & ::= & w \mid a \mid (e, f)_N \mid (e, f)_R \\
\Gamma \in \mathit{Environment} & ::= & \phi \mid \Gamma, x \mapsto h \\
S \in \mathit{Stack} & ::= & \phi \mid [\mathsf{P} p] +\!\!+ S \mid [\mathsf{A} h] +\!\!+ S \mid [\mathsf{B} x] +\!\!+ S \mid [\mathsf{N} a] +\!\!+ S \mid [\mathsf{R} a] +\!\!+ S
\end{array}
$$

図 4.7: 状態の要素

Whnf^- は弱頭部正規形を表現する．その要素は，関数，空リスト，コンスセルのいずれかである．ヒープ上に置かれたオブジェクトとして，式と区別するため，Expr^- のラムダ式，空リスト，コンスセルの表現をそのまま用いることはない．コンスセルは，head 部と tail 部のヒープの場所のペア $h_h^- : h_t^-$ で表現する．Whnf は，e^- と h^- の代わりに e と h を使うことを除いて，Whnf^- と同じである．

ヒープ上の場所 h は，アドレス a と tail 部での間接参照を示すフラグの組で $\langle a, f \rangle$ のように表現する．もし，フラグ f が T であれば，アドレス a にはコンスセルがあり，そのコンスセルの tail 部が，本来のそのヒープ上の場所の内容であることを意味する．なお，再利用のない場合においては，tail 部の間接参照が必要ないため，フラグは必要なく $\mathit{Location}^-$ は $\langle a \rangle$ と表現する．ヒープ上の場所 h とアドレス a が区別されていることに注意されたい．

$$\mathcal{A}^- :: \mathit{Location}^- \to \mathit{Content}^-\texttt{@}\mathit{Address}$$

$$\begin{aligned}
\mathcal{A}^-\,\langle a\rangle &= w^-\texttt{@}a && \text{if } h^-(a) = w^-\\
\mathcal{A}^-\,\langle a\rangle &= (e^-, f)_N\texttt{@}a && \text{if } h^-(a) = (e^-, f)_N\\
\mathcal{A}^-\,\langle a\rangle &= H^-(a')\texttt{@}a' && \text{if } H^-(a) = a'
\end{aligned}$$

$$\mathcal{A} :: \mathit{Location} \to \mathit{Content}\texttt{@}\mathit{Address}$$

$$\begin{aligned}
\mathcal{A}\,\langle a, \mathsf{F}\rangle &= w\texttt{@}a && \text{if } H(a) = w\\
\mathcal{A}\,\langle a, \mathsf{F}\rangle &= (e, f)_N\texttt{@}a && \text{if } H(a) = (e, f)_N\\
\mathcal{A}\,\langle a, \mathsf{F}\rangle &= H(a')\texttt{@}a' && \text{if } H(a) = a'\\
\mathcal{A}\,\langle a, \mathsf{F}\rangle &= (e, f)_R\texttt{@}a && \text{if } H(a) = (e, f)_R\\
\mathcal{A}\,\langle a, \mathsf{T}\rangle &= \mathcal{A}\,h_t && \text{if } H(a) = h_h : h_t
\end{aligned}$$

図 4.8: 補助関数 $\mathcal{A}^-$ と $\mathcal{A}$

四つ組の 1 つ目の要素はヒープで，アドレスとそのアドレスにある内容の組を集めたものである．ヒープ H にあるアドレス a に対して，$H(a)$ という記法を用いて，そのアドレスの内容を取得することができることとする．ヒープへの操作は，要素の追加と内容の書き換えの 2 つがある．まず，要素の追加は，$H, a \mapsto C$ と書き，これはヒープ H の新たな a というアドレスに内容 C を登録することを意味する．同様にして，内容の書き換えは $H[a \mapsto C]$ と書き，これはヒープ H にある a というアドレスにある内容を C で破壊的に書き換えることを意味する．

ヒープに置かれる内容 C^- は弱頭部正規形 w^-，別のアドレス a，通常サンク $(e^-, f)_N$ のいずれかである．再利用のある場合の内容である C には，これに再利用サンク $(e, f)_R$ が追加される．別のアドレス a がヒープの内容に指定されている場合，それは間接参照を意味する．間接参照は，2.2 節で示したように，強制後の通常サンクの結果を共有する際に利用する．サンクにおいては，e^- と e が遅延する式，フラグ f は，そのサンクが強制済みであるかを示す．$f = \mathsf{T}$ のとき，そのサンクは強制中または強制済みであり，$f = \mathsf{F}$ のとき，そのサンクは未強制である．なお，通常サンクと再利用サンクは，N と R を右下に付加することで，区別して表現する．本節のはじめにも述べたが，サンク中の遅延する式 e^- や e 内の変数を重複しない新たな変数に置き換えることで，サンク毎に環境を保持しない．

あるヒープ H にあるアドレス a から内容を取る $H(a)$ という記法に加えて，ヒープの場所 h から内容を取る補助関数を図 4.8 に定義する．補助関数 $\mathcal{A}^-$ と $\mathcal{A}$ は，ヒープの場所を受けとり，必要に応じて間接参照をたどった上で，その場所にある内容を返す．こ

のとき，内容の置かれたアドレスを明確にするため，$C^-@a$ または $C@a$ というように，アドレスを明示する．$\mathcal{A}^-$ は，あるアドレスに別のアドレスが指定されるような間接参照がある場合，$\mathcal{A}^-$ を再帰呼び出しして参照をたどる．それに加えて，$\mathcal{A}$ の場合には，tail 部への間接参照をたどり得る．ヒープの場所が $\langle a, f\rangle$ というように，tail 部での間接参照を示すフラグを保持していたことに注意されたい．$\mathcal{A}$ の動作例を次に示す．

$$\begin{aligned}
&\mathcal{A}\,\langle a, \mathsf{F}\rangle = w@a && \text{if } H(a) = w\\
&\mathcal{A}\,\langle a, \mathsf{F}\rangle = w@a_1 && \text{if } H(a) = a_1,\ H(a_1) = w\\
&\mathcal{A}\,\langle a, \mathsf{T}\rangle = (e, f)_R@a' && \text{if } H(a) = h_h : \langle a', \mathsf{F}\rangle,\ H(a') = (e, f)_R
\end{aligned}$$

はじめの場合，$H(a)$ により単純に内容を得る．2 つ目の場合，別のアドレスへの間接参照をたどった上で内容を得る．このような間接参照は，通常サンクの強制結果として生成されるため，たどった先がまた別の間接参照となっていることはない．そのため，$H(a)$ により間接参照をたどる必要があるのは，高々 1 回である．最後の場合，$\mathcal{A}$ に与えられるヒープの場所にフラグが設定されているため，$H(a)$ にあるコンスセルの tail 部をたどった上で返す．

四つ組の 2 つ目の要素は，簡約の対象となる t^- と t で，式か弱頭部正規形である．ある式から始めて，簡約を繰り返して弱頭部正規形に至る．$\mathcal{A}$ のときと同様に，$w^- @ a$ と $w @ a$ というように，弱頭部正規形の置かれたヒープのアドレス a を明示する．

四つ組の 3 つ目の要素は，環境である．環境は，識別子とヒープの場所の組を集めたもので，$\Gamma(x)$ という記法により，環境 Γ において x と組となっているヒープの場所を得る．ヒープの時と同様に，環境への値の追加を $\Gamma, x \mapsto h$，環境の更新を $\Gamma[x \mapsto h]$ と表現する．ヒープのアドレスを環境の組に直接は持たず，ヒープの場所との組とすることに注意されたい．このことにより，$*$ 付きの識別子を簡潔に表現できる．たとえば，$x \mapsto h$ という組が環境中にあり，$h = \langle a, \mathsf{T}\rangle$ かつ $H(a) = h_h : h_t$ となっているとき，x^* の意味するところはヒープの場所 h_t にある内容である．

四つ組の 4 つ目の要素は，スタックであり，後続の簡約ステップに情報を渡すのに用いられる．スタックの要素になり得るのは以下の 5 つである．

$\mathsf{P}p^-$, $\mathsf{P}p$ – 空リストかコンスセルかによって，パターン p^- あるいは p にもとづいたパターンマッチにより分岐する．

$\mathsf{A}h^-$, $\mathsf{A}h$ – h^- あるいは h を実引数として，評価結果の値である関数を適用する．

$\mathsf{B}x$ – 評価結果の値を x に束縛する．

Na – アドレス a にある通常サンクを強制中であることを意味する．強制結果が求まったら，その値への間接参照としてアドレス a の通常サンクを破壊的に更新する．

Ra – アドレス a にあるコンスセルの tail 部から参照されている再利用サンクを強制中であることを意味する．強制結果が求まったら，アドレス a にあるコンスセルの tail 部を破壊的に更新する．

先で述べたように，環境をグローバルにするため，式 e 中の変数 x の自由な出現を新たな変数 x' に置き換える必要がある．このことを $e[x'/x]$ と書く．この記法は，e^-，p^-，p にも用いる．

四つ組の簡約時に，他と重複しない変数やアドレスを新たに確保することを，下線を引くことで簡潔に表現する．たとえば，$\underline{x}$ は名前の衝突がないように新たな変数 x を用意することを，また，$\underline{a}$ は新たなアドレスをヒープ上に確保することを意味する．

4.3.2 *Expr*$^-$ の簡約規則

図 4.9 に *Expr*$^-$ に関する操作的意味論を示す．これまでの命名規則に従い，簡約規則名にも上付きの $-$ によって，再利用のない場合の規則であることを示す．

WhnfFunc$^-$ と WhnfNil$^-$ と WhnfCons$^-$ が，該当する弱頭部正規形をヒープに割り当てる規則である．ある変数が弱頭部正規形に束縛されていれば，VarWhnf$^-$ によりその弱頭部正規形を次の簡約対象とする．ある変数が通常サンクに束縛されていれば，VarNormalThunk$^-$ により，その通常サンクを強制しはじめる．強制結果が求まった際には，UpdateNormalThunk$^-$ により，その通常サンクの内容を間接参照へと破壊的に書き換える．関数適用 $e^-\ x$ に対しては，Apply$^-$ で，その関数部分である e^- を簡約を始める．その際，関数適用の引数 x をスタックに積んでおき，e^- がラムダ式に簡約されると Apply2$^-$ で参照する．case 式 case e^- of $x\ p^-$ に対しては，まず，Case$^-$ で対象となる式 e^- を評価する．その式の評価結果が求まれば，CaseBind$^-$ で x にその値を束縛した上で，CaseNil$^-$ と CaseCons$^-$ によってパターンマッチする．最後に，Let$^-$ でヒープに通常サンクを割り当てる．

どの規則にも当てはまらない状態に至ると，プログラムは停止する．正常なプログラムの場合は，そのときの四つ組の 2 つ目の要素（*ExprOrWhnf*$^-$）が簡約結果である．プログラムによっては，実行エラーによりプログラムが停止する場合もある．たとえば，強制中のサンクを再び強制しようとすると，実行時エラーであるので該当する規則はない．

これは，VarNormalThunk^{-} において，通常サンクのフラグを F に限定していることによって表現されている．

	$Heap^-$	$ExprOrWhnf^-$	$Environment^-$	$Stack^-$	$rule$
	H^-	$\backslash x \to e^-$	Γ^-	S^-	
$\Rightarrow$	$H^-,\ \underline{a} \mapsto \lambda\, x\,.\, e^-$	$\lambda\, x\,.\, e^-$ @ $\underline{a}$	Γ^-	S^-	WhnfFunc$^-$
	H^-	Nil	Γ^-	S^-	
$\Rightarrow$	$H^-,\ \underline{a} \mapsto [\,]$	$[\,]$ @ $\underline{a}$	Γ^-	S^-	WhnfNil$^-$
	H^-	Cons $x_h\ x_t$	Γ^-	S^-	
	where $\Gamma^-(x_h) = h_h^-,\ \Gamma^-(x_t) = h_t^-$				
$\Rightarrow$	$H^-,\ \underline{a} \mapsto h_h^- : h_t^-$	$h_h^- : h_t^-$ @ $\underline{a}$	Γ^-	S^-	WhnfCons$^-$
	H^-	x	Γ^-	S^-	
	where $\Gamma^-(x) = h_v^-,\ \mathcal{A}^-(h_v^-) = w^-$ @ a_w				
$\Rightarrow$	H^-	w^- @ a_w	Γ^-	S^-	VarWhnf$^-$
	H^-	x	Γ^-	S^-	
	where $\Gamma^-(x) = h_v^-,\ \mathcal{A}^-(h_v^-) = (e^-, \mathsf{F})_N$ @ a				VarNormalThunk$^-$
$\Rightarrow$	$H^-[a \mapsto (e^-, \mathsf{T})_N]$	e^-	Γ^-	$[\mathsf{N}a] +\!\!+ S^-$	
	H^-	w^- @ a_w	Γ^-	$[\mathsf{N}a] +\!\!+ S^-$	
$\Rightarrow$	$H^-[a \mapsto a_w]$	w^- @ a_w	Γ^-	S^-	UpdateNormalThunk$^-$
	H^-	$e^-\ x$	Γ^-	S^-	
	where $\Gamma^-(x) = h^-$				
$\Rightarrow$	H^-	e^-	Γ^-	$[\mathsf{A}h^-] +\!\!+ S^-$	Apply$^-$
	H^-	$\lambda\, x\,.\, e^-$ @ a_l	Γ^-	$[\mathsf{A}h^-] +\!\!+ S^-$	
$\Rightarrow$	H^-	$e^-[\underline{x'}/x]$	$\Gamma^-,\ \underline{x'} \mapsto h^-$	S^-	Apply2$^-$
	H^-	case e^- of $x\ p^-$	Γ^-	S^-	Case$^-$
$\Rightarrow$	H^-	e^-	Γ^-	$[\mathsf{B}\underline{x'}] +\!\!+ [\mathsf{P}p^-[\underline{x'}/x]] +\!\!+ S^-$	
	H^-	w^- @ a	Γ^-	$[\mathsf{B}x] +\!\!+ S^-$	
$\Rightarrow$	H^-	w^- @ a	$\Gamma^-,\ x \mapsto \langle a \rangle$	S^-	CaseBind$^-$
	H^-	$[\,]$ @ a	Γ^-	$[\mathsf{P}p^-] +\!\!+ S^-$	
	where $p^- = \{\ \mathsf{Nil} \to e_n^-;\ \cdots\ \}$				
$\Rightarrow$	H^-	e_n^-	Γ^-	S^-	CaseNil$^-$
	H^-	$h_h^- : h_t^-$ @ a	Γ^-	$[\mathsf{P}p^-] +\!\!+ S^-$	
	where $p^- = \{\ \cdots;\ \mathsf{Cons}\ x_h\ x_t \to e_c^-\ \}$				
$\Rightarrow$	H^-	$e_c^-[\underline{x'_h}/x_h][\underline{x'_t}/x_t]$	$\Gamma^-,\ \underline{x'_h} \mapsto h_h^-,\ \underline{x'_t} \mapsto h_t^-$	S^-	CaseCons$^-$
	H^-	let $x = e_1^-$ in e_2^-	Γ^-	S^-	
$\Rightarrow$	$H^-,\ \underline{a} \mapsto (e_1^-[\underline{x'}/x], \mathsf{F})_N$	$e_2^-[\underline{x'}/x]$	$\Gamma^-,\ \underline{x'} \mapsto \langle \underline{a} \rangle$	S^-	Let$^-$

図 4.9: $Expr^-$ に対する操作的意味論

4.3.3 *Expr* の簡約規則

図 4.10 に *Expr* に関する操作的意味論を示す.

CaseCons において，tail 部での間接参照を導入する．具体的には，コンスセルの tail 部を指すために新たに確保する変数 x'_t のヒープの場所について，フラグを T とする．ここで，e_c 内のすべての x^*_t の出現を x'_t に置き換えるため，$*$ の付いた変数が簡約の対象となることはないことに注意されたい．そのため，変数を簡約の対象とする規則（VarWhnf, VarNormalThunk，VarReuseThunk）で $*$ を考慮せずに済ませている．

letreuse 式は，以下のとおりに簡約される．

- もし，再利用可能な再利用サンクがまだ割り当てられていなければ，letreuse 式により再利用サンクをヒープに割り当てる（LetreuseAlloc）．このことは，スタックの最上部に Ra がないことで確認できる．

- もし，再利用可能な再利用サンクがすでに割り当てられていれば，アドレス a_t にある再利用サンクが保持する式を，letreuse 式で新たに遅延する式 e_1 で置き換える（LetreuseReuse）．このことは，スタックの最上部に Ra があることから判定できる．なお，アドレス a には tail 部に再利用サンクを持つコンスセルがある．

Let が新たにヒープのアドレス $\underline{a}$ を確保しているのに対して，LetreuseReuse では新たなアドレスが必要となっていないことにより，Thunk Recycling によって新たなメモリを割り当てる必要がなくなる．なお，変換 $\mathcal{P}$ によって letreuse 式が導入されるため，e_1 中で x が出現しないことは保証されている．そのため，LetreuseReuse で $e_1[x'/x]$ という置き換えは必要ない.

VarReuseThunk は再利用サンクを強制する規則で，強制中に tail 部に再利用サンクを持つコンスセルを参照できるように，アドレス a を Ra という形でスタック上に積む．UpdateTail は再利用サンクの結果が求まったところでコンスセルの tail 部を破壊的に書き換える規則である.

	Heap	*ExprOrWhnf*	*Environment*	*Stack*	*rule*
	H	$\backslash x \to e$	Γ	S	
$\Rightarrow$	$H,\ \underline{a} \mapsto \lambda\, x\, .\, e$	$\lambda\, x\, .\, e \mathbin{@} \underline{a}$	Γ	S	WhnfFunc
	H	Nil	Γ	S	
$\Rightarrow$	$H,\ \underline{a} \mapsto [\,]$	$[\,] \mathbin{@} \underline{a}$	Γ	S	WhnfNil
	H	$\mathsf{Cons}\ x_h\ x_t$	Γ	S	
	where $\Gamma(x_h) = h_h,\ \Gamma(x_t) = h_t$				
$\Rightarrow$	$H,\ \underline{a} \mapsto h_h : h_t$	$h_h : h_t \mathbin{@} \underline{a}$	Γ	S	WhnfCons
	H	x	Γ	S	
	where $\Gamma(x) = h,\ \mathcal{A}(h) = w \mathbin{@} a$				
$\Rightarrow$	H	$w \mathbin{@} a$	Γ	S	VarWhnf
	H	x	Γ	S	
	where $\Gamma(x) = h,\ \mathcal{A}(h) = (e, \mathsf{F})_N \mathbin{@} a$				VarNormalThunk
$\Rightarrow$	$H[a \mapsto (e, \mathsf{T})_N]$	e	Γ	$[\mathsf{N}a] +\!\!+ S$	
	H	x	Γ	S	
	where $\Gamma(x) = h_t,\ \mathcal{A}(h_t) = (e, \mathsf{F})_R \mathbin{@} a_t,\ H(a) = h_h : \langle a_t, \mathsf{F}\rangle$				VarReuseThunk
$\Rightarrow$	$H[a_t \mapsto (e, \mathsf{T})_R]$	e	Γ	$[\mathsf{R}a] +\!\!+ S$	
	H	$w \mathbin{@} a_w$	Γ	$[\mathsf{N}a] +\!\!+ S$	
$\Rightarrow$	$H[a \mapsto a_w]$	$w \mathbin{@} a_w$	Γ	S	
					UpdateNormalThunk
	H	$w \mathbin{@} a_w$	Γ	$[\mathsf{R}a] +\!\!+ S$	
	where $H(a) = h_h : h_t$				
$\Rightarrow$	$H[a \mapsto h_h : \langle a_w, \mathsf{F}\rangle]$	$w \mathbin{@} a_w$	Γ	S	UpdateTail
	H	$e\ x$	Γ	S	
	where $\Gamma(x) = h$				
$\Rightarrow$	H	e	Γ	$[\mathsf{A}h] +\!\!+ S$	Apply
	H	$\lambda\, x\, .\, e \mathbin{@} a$	Γ	$[\mathsf{A}h] +\!\!+ S$	
$\Rightarrow$	H	$e[\underline{x'}/x]$	$\Gamma,\ \underline{x'} \mapsto h$	S	Apply2
	H	$\mathsf{case}\ e\ \mathsf{of}\ x\ p$	Γ	S	Case
$\Rightarrow$	H	e	Γ	$[\mathsf{B}\underline{x'}] +\!\!+ [\mathsf{P}p[\underline{x'}/x]] +\!\!+ S$	
	H	$w \mathbin{@} a$	Γ	$[\mathsf{B}x] +\!\!+ S$	
$\Rightarrow$	H	$w \mathbin{@} a$	$\Gamma,\ x \mapsto \langle a, \mathsf{F}\rangle$	S	CaseBind
	H	$[\,] \mathbin{@} a$	Γ	$[\mathsf{P}p] +\!\!+ S$	
	where $p = \{\ \mathsf{Nil} \to e_n;\ \cdots\ \}$				
$\Rightarrow$	H	e_n	Γ	S	CaseNil
	H	$h_h : h_t \mathbin{@} a$	Γ	$[\mathsf{P}p] +\!\!+ S$	
	where $p = \{\ \cdots\ ;\ \mathsf{Cons}\ x_h\ x_t^* \to e_c\ \}$				
$\Rightarrow$	H	$e_c[\underline{x'_h}/x_h][\underline{x'_t}/x_t^*]$	$\Gamma,\ \underline{x'_h} \mapsto h_h,\ \underline{x'_t} \mapsto \langle a, \mathsf{T}\rangle$	S	CaseCons
	H	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	Γ	S	
$\Rightarrow$	$H,\ \underline{a} \mapsto (e_1[\underline{x'}/x], \mathsf{F})_N$	$e_2[\underline{x'}/x]$	$\Gamma,\ \underline{x'} \mapsto \langle \underline{a}, \mathsf{F}\rangle$	S	Let
	H	$\mathsf{letreuse}\ x = e_1\ \mathsf{in}\ e_2$	Γ	S	
	where $S = \phi \vee \text{top of } S \neq \mathsf{R}a'$				
$\Rightarrow$	$H,\ \underline{a} \mapsto (e_1, \mathsf{F})_R$	$e_2[\underline{x'}/x]$	$\Gamma,\ \underline{x'} \mapsto \langle \underline{a}, \mathsf{F}\rangle$	S	LetreuseAlloc
	H	$\mathsf{letreuse}\ x = e_1\ \mathsf{in}\ e_2$	Γ	$[\mathsf{R}a] +\!\!+ S$	
	where $H(a) = h_h : \langle a_t, \mathsf{F}\rangle,\ H(a_t) = (e, \mathsf{T})_R$				
$\Rightarrow$	$H[a_t \mapsto (e_1, \mathsf{F})_R]$	$e_2[\underline{x'}/x]$	$\Gamma,\ \underline{x'} \mapsto \langle a_t, \mathsf{F}\rangle$	S	LetreuseReuse

図 4.10: *Expr* に対する操作的意味論

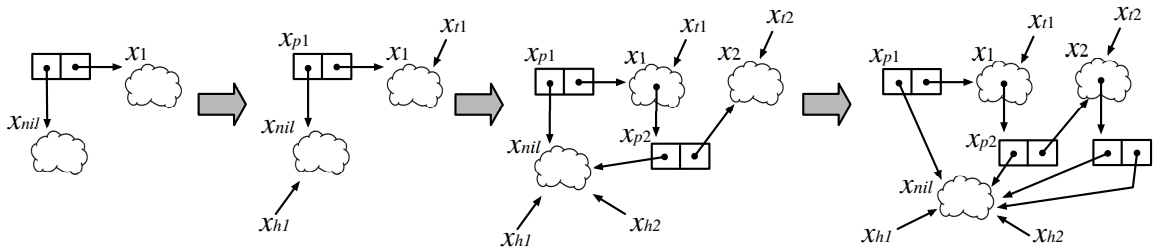

図 4.11: $expr^-$ の簡約

4.4 簡約の例

4.2.3 節で用いた $expr^-$ と $expr$ を用いて，操作的意味論の動作例を示す．
$expr^-$ は以下のとおりであった．

$$
\begin{aligned}
expr^- = {} & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathsf{let}\ x_1 = \mathsf{let}\ x_2 = \mathsf{Cons}\ x_{nil}\ x_{nil} \\
& \qquad\qquad \mathsf{in\ Cons}\ x_{nil}\ x_2 \\
& \quad \mathsf{in\ Cons}\ x_{nil}\ x_1)\ \mathsf{of}\ x_{p1}\ \{ \\
& \mathsf{Nil} \to \mathsf{Nil} \\
& \mathsf{Cons}\ x_{h1}\ x_{t1} \to \mathsf{case}\ x_{t1}\ \mathsf{of}\ x_{p2}\ \{ \\
& \qquad\qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad\qquad \mathsf{Cons}\ x_{h2}\ x_{t2} \to x_{t2}\}\}
\end{aligned}
$$

このプログラムは，最終的に 3 つのコンスセルからなるリストを生成するように，サンクを要求駆動で強制していき，最後尾のコンスセルを返す．その様子を図 4.11 に示す．この図は，パターンマッチにより生成される識別子を明示して描いている．まず，case 式によって，let 式の評価を進め，$\mathsf{Cons}\ x_{nil}\ x_1$ というコンスセルを生成する．次に，そのコンスセルに対して，パターンマッチにより，x_{h1} と x_{t1} という参照を作る．それに続いて，その x_{t1} で参照するサンクを強制することでリストを読み進め，$\mathsf{Cons}\ x_{nil}\ x_2$ というコンスセルを生成する．このとき，そのコンスセルにパターンマッチし，x_{h2} と x_{t2} という参照を作る過程もまとめて描かれている．最後に，x_{t2} で参照するサンクを強制し，簡約結果の $\mathsf{Cons}\ x_{nil}\ x_{nil}$ というコンスセルを生成する．

$expr^-$ に対して，Thunk Recycling のプログラム変換を適用したプログラム $expr$ も以

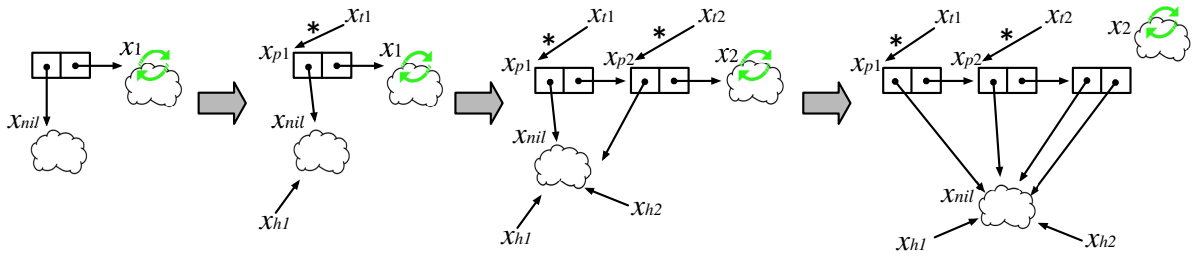

図 4.12: $expr$ の簡約

下に再掲する．

$$
\begin{aligned}
expr = {} & \mathsf{let}\ x_{nil} = \mathsf{Nil} \\
& \mathsf{in\ case}\ (\mathsf{letreuse}\ x_1 = \mathsf{letreuse}\ x_2 = \mathsf{Cons}\ x_{nil}\ x_{nil} \\
& \qquad \mathsf{in\ Cons}\ x_{nil}\ x_2 \\
& \quad \mathsf{in\ Cons}\ x_{nil}\ x_1)\ \mathsf{of}\ x_{p1}\ \{ \\
& \mathsf{Nil} \to \mathsf{Nil} \\
& \mathsf{Cons}\ x_{h1}\ x^*_{t1} \to \mathsf{case}\ x^*_{t1}\ \mathsf{of}\ x_{p2}\ \{ \\
& \qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad \mathsf{Cons}\ x_{h2}\ x^*_{t2} \to x^*_{t2}\}\}
\end{aligned}
$$

$expr$ の簡約過程を図 4.12 に示す．パターンマッチで生成される tail 部への参照が間接参照である点と，初めに割り当てた再利用サンクを使いまわす点で，図 4.11 の場合と異なっている．結果のコンスセルには後続がなく Nil で終端しているため，最終的には再利用サンクはどこからも参照されなくなる．

本節では，$expr^-$ と $expr$ のそれぞれについて，四つ組の遷移を順に追うことで，簡約規則の動作を示す．そこでは，ある状態に対して，適合する規則が複数ないように操作的意味論が定義されていることに注意されたい．さらに，両者の簡約過程を比較する．

4.4.1　$expr^-$ の簡約

初期状態は，四つ組の $ExprOrWhnf^-$ 以外の要素は ϕ である．このとき，簡約の対象が let 式であるので，

$$\phi \qquad \mathsf{let}\ x_{nil} = \mathsf{Nil}\ \mathsf{in} \ldots \qquad \phi \qquad \phi$$
$$\xRightarrow{\textsc{Let}^-}$$
$$\underline{a_{nil}} \mapsto (\mathsf{Nil}, \mathsf{F})_N \quad \mathsf{case}\ (\mathsf{let} \ldots)\ \mathsf{of}\ x_{p1}\{\ldots\} \quad \underline{x'_{nil}} \mapsto \langle \underline{a_{nil}} \rangle \quad \phi$$

というように，$\textsc{Let}^-$ により，ヒープのアドレス a_{nil} に通常サンクを割り当てる．続いて，case 式が簡約の対象となり，以下のように $\textsc{Case}^-$ で簡約を進める．

$$\xRightarrow{\textsc{Case}^-}$$
$$a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \quad \mathsf{let}\ x_1 = \mathsf{let}\ x_2 \ldots\ \mathsf{in} \ldots \quad x'_{nil} \mapsto \langle a_{nil} \rangle \quad [\mathsf{B}\underline{x'_{p1}}] +\!\!+ [\mathsf{P}p_0^-]$$

$\textsc{Case}^-$ は，スタック上に後のパターンマッチのための準備を行い，対象となる式を簡約する．ここでは，簡約対象は let 式で，その本体は Cons によりコンスセルを構成するので，以下のように簡約が進む．

$$\xRightarrow{\textsc{Let}^-}$$
$$\begin{array}{l} a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\ \underline{a_1} \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{F})_N \end{array} \quad \mathsf{Cons}\ x'_{nil}\ x'_1 \quad \begin{array}{l} x'_{nil} \mapsto \langle a_{nil} \rangle \\ \underline{x'_1} \mapsto \langle \underline{a_1} \rangle \end{array} \quad [\mathsf{B}x'_{p1}] +\!\!+ [\mathsf{P}p_0^-]$$

$$\xRightarrow{\textsc{WhnfCons}^-}$$
$$\begin{array}{l} a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\ a_1 \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{F})_N \\ \underline{a_{c1}} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle \end{array} \quad \langle a_{nil} \rangle : \langle a_1 \rangle \mathbin{@} \underline{a_{c1}} \quad \begin{array}{l} x'_{nil} \mapsto \langle a_{nil} \rangle \\ x'_1 \mapsto \langle a_1 \rangle \end{array} \quad [\mathsf{B}x'_{p1}] +\!\!+ [\mathsf{P}p_0^-]$$

ここで，弱頭部正規形が求まったので，スタックに用意しておいた情報をもとに，x'_{p1} にここまで求めたコンスセルを束縛し，パターンマッチを行う．

$$\xRightarrow{\textsc{CaseBind}^-}$$
$$\begin{array}{l} a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\ a_1 \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{F})_N \\ a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle \end{array} \quad \langle a_{nil} \rangle : \langle a_1 \rangle \mathbin{@} a_{c1} \quad \begin{array}{l} x'_{nil} \mapsto \langle a_{nil} \rangle \\ x'_1 \mapsto \langle a_1 \rangle \\ x'_{p1} \mapsto \langle a_{c1} \rangle \end{array} \quad [\mathsf{P}p_0^-]$$

$\xrightarrow{\textsc{CaseCons}^-}$

$$
\begin{array}{lll}
 & & x'_{nil} \mapsto \langle a_{nil} \rangle \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_1 \mapsto \langle a_1 \rangle \\
a_1 \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{F})_N & \mathsf{case}\ x'_{t1}\ of\ x_{p2}\ \{\ldots\} & x'_{p1} \mapsto \langle a_{c1} \rangle \qquad \phi \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle & & \underline{x'_{h1}} \mapsto \langle a_{nil} \rangle \\
 & & \underline{x'_{t1}} \mapsto \langle a_1 \rangle
\end{array}
$$

コンスセルにマッチしたため，head 部と tail 部を参照する変数を環境に追加した．次に，Cons のパターンの本体である case 式を簡約する．この case 式の簡約対象は，変数 x'_{t1} であり，この変数は先のパターンマッチによってコンスセルの tail 部の通常サンクを束縛している．そのため，tail 部の通常サンクを強制し，リストを読み進める．

$\xrightarrow{\textsc{Case}^-}$

$$
\begin{array}{lll}
 & & x'_{nil} \mapsto \langle a_{nil} \rangle \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_1 \mapsto \langle a_1 \rangle \\
a_1 \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{F})_N & x'_{t1} & x'_{p1} \mapsto \langle a_{c1} \rangle \quad \underline{[\mathsf{B}x'_{p2}]} \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle & & x'_{h1} \mapsto \langle a_{nil} \rangle \quad +\!\!+[\mathsf{P}p_1^-] \\
 & & x'_{t1} \mapsto \langle a_1 \rangle
\end{array}
$$

$\xrightarrow{\textsc{VarNormalThunk}^-}$

$$
\begin{array}{lll}
 & & x'_{nil} \mapsto \langle a_{nil} \rangle \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_1 \mapsto \langle a_1 \rangle \quad [\mathsf{N}a_1] \\
a_1 \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{T})_N & \mathsf{let}\ x_2 = \mathsf{Cons}\ x'_{nil}\ x'_{nil} & x'_{p1} \mapsto \langle a_{c1} \rangle \quad +\!\!+[\mathsf{B}x'_{p2}] \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle & \mathsf{in}\ \mathsf{Cons}\ x'_{nil}\ x_2 & x'_{h1} \mapsto \langle a_{nil} \rangle \quad +\!\!+[\mathsf{P}p_1^-] \\
 & & x'_{t1} \mapsto \langle a_1 \rangle
\end{array}
$$

通常サンクの簡約は，$\textsc{VarNormalThunk}^-$ により，アドレス a_1 から遅延した式をとりだし，フラグを T にした上で，その通常サンクを強制しはじめる．このとき，スタックにはその通常サンクの強制後の操作によって，値が共有するための準備をしておく．遅延した式は let 式であり，その let 式の本体はデータ構成子 Cons の適用である．

$\xrightarrow{\textsc{Let}^-}$

$$
\begin{array}{l}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\
a_1 \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{T})_N \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle \\
\underline{a_2} \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{F})_N
\end{array}
\quad
\mathsf{Cons}\ x'_{nil}\ x'_2
\quad
\begin{array}{l}
x'_{nil} \mapsto \langle a_{nil} \rangle \\
x'_1 \mapsto \langle a_1 \rangle \\
x'_{p1} \mapsto \langle a_{c1} \rangle \\
x'_{h1} \mapsto \langle a_{nil} \rangle \\
x'_{t1} \mapsto \langle a_1 \rangle \\
\underline{x'_2} \mapsto \langle \underline{a_2} \rangle
\end{array}
\quad
\begin{array}{l}
[\mathsf{N}a_1] \\
+\!\!+ [\mathsf{B}x'_{p2}] \\
+\!\!+ [\mathsf{P}p_1^-]
\end{array}
$$

$\xrightarrow{\textsc{WhnfCons}^-}$

$$
\begin{array}{l}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\
a_1 \mapsto (\mathsf{let}\ x_2 = \ldots, \mathsf{T})_N \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle \\
a_2 \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{F})_N \\
\underline{a_{c2}} \mapsto \langle a_{nil} \rangle : \langle a_2 \rangle
\end{array}
\quad
\langle a_{nil} \rangle : \langle a_2 \rangle \mathbin{@} \underline{a_{c2}}
\quad
\begin{array}{l}
x'_{nil} \mapsto \langle a_{nil} \rangle \\
x'_1 \mapsto \langle a_1 \rangle \\
x'_{p1} \mapsto \langle a_{c1} \rangle \\
x'_{h1} \mapsto \langle a_{nil} \rangle \\
x'_{t1} \mapsto \langle a_1 \rangle \\
x'_2 \mapsto \langle a_2 \rangle
\end{array}
\quad
\begin{array}{l}
[\mathsf{N}a_1] \\
+\!\!+ [\mathsf{B}x'_{p2}] \\
+\!\!+ [\mathsf{P}p_1^-]
\end{array}
$$

アドレス a_1 に割り当てられた通常サンクの強制結果は，コンスセルであった．強制結果が求まったので，強制結果を共有する UpdateNormalThunk により簡約を進める．

$\xrightarrow{\textsc{UpdateNormalThunk}^-}$

$$
\begin{array}{l}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\
a_1 \mapsto a_{c2} \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle \\
a_2 \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{F})_N \\
a_{c2} \mapsto \langle a_{nil} \rangle : \langle a_2 \rangle
\end{array}
\quad
\langle a_{nil} \rangle : \langle a_2 \rangle \mathbin{@} a_{c2}
\quad
\begin{array}{l}
x'_{nil} \mapsto \langle a_{nil} \rangle \\
x'_1 \mapsto \langle a_1 \rangle \\
x'_{p1} \mapsto \langle a_{c1} \rangle \\
x'_{h1} \mapsto \langle a_{nil} \rangle \\
x'_{t1} \mapsto \langle a_1 \rangle \\
x'_2 \mapsto \langle a_2 \rangle
\end{array}
\quad
[\mathsf{B}x'_{p2}] +\!\!+ [\mathsf{P}p_1^-]
$$

スタックに積んでおいた情報をもとに，サンクの強制結果を，ヒープ内のアドレス a_1 から参照できるように破壊的に書き換える．続いて，スタックに準備しておいた情報をもとに，パターンマッチを行う．

$\xrightarrow{\textsc{CaseBind}^-}$

		$x'_{nil} \mapsto \langle a_{nil} \rangle$	
$a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N$		$x'_1 \mapsto \langle a_1 \rangle$	
$a_1 \mapsto a_{c2}$		$x'_{p1} \mapsto \langle a_{c1} \rangle$	
$a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle$	$\langle a_{nil} \rangle : \langle a_2 \rangle \mathbin{@} a_{c2}$	$x'_{h1} \mapsto \langle a_{nil} \rangle$	$[\mathsf{P}p_1^-]$
$a_2 \mapsto (\mathsf{Cons}\, x'_{nil}\, x'_{nil}, \mathsf{F})_N$		$x'_{t1} \mapsto \langle a_1 \rangle$	
$a_{c2} \mapsto \langle a_{nil} \rangle : \langle a_2 \rangle$		$x'_2 \mapsto \langle a_2 \rangle$	
		$x'_{p2} \mapsto \langle a_{c2} \rangle$	

$\xrightarrow{\textsc{CaseCons}^-}$

		$x'_{nil} \mapsto \langle a_{nil} \rangle$	
		$x'_1 \mapsto \langle a_1 \rangle$	
$a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N$		$x'_{p1} \mapsto \langle a_{c1} \rangle$	
$a_1 \mapsto a_{c2}$		$x'_{h1} \mapsto \langle a_{nil} \rangle$	
$a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle$	x'_{t2}	$x'_{t1} \mapsto \langle a_1 \rangle$	ϕ
$a_2 \mapsto (\mathsf{Cons}\, x'_{nil}\, x'_{nil}, \mathsf{F})_N$		$x'_2 \mapsto \langle a_2 \rangle$	
$a_{c2} \mapsto \langle a_{nil} \rangle : \langle a_2 \rangle$		$x'_{p2} \mapsto \langle a_{c2} \rangle$	
		$\underline{x'_{h2}} \mapsto \langle a_{nil}$	
		$\underline{x'_{t2}} \mapsto \langle a_2 \rangle$	

case 式によりパターンマッチすると，次の簡約対象は変数 x'_{t2} である．この変数は，パターンマッチによりコンスセルの tail 部を取り出したもので，未強制の通常サンクを参照している．そのため，先の x'_{t1} を強制した場合と同様に簡約を進める．

$\xrightarrow{\textsc{VarNormalThunk}^-}$

$$
\begin{array}{lll}
 & & x'_{nil} \mapsto \langle a_{nil} \rangle \\
 & & x'_1 \mapsto \langle a_1 \rangle \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_{p1} \mapsto \langle a_{c1} \rangle \\
a_1 \mapsto a_{c2} & & x'_{h1} \mapsto \langle a_{nil} \rangle \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle & \mathsf{Cons}\ x'_{nil}\ x'_{nil} & x'_{t1} \mapsto \langle a_1 \rangle \qquad [\mathsf{N}a_2] \\
a_2 \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{T})_N & & x'_2 \mapsto \langle a_2 \rangle \\
a_{c2} \mapsto \langle a_{nil} \rangle : \langle a_2 \rangle & & x'_{p2} \mapsto \langle a_{c2} \rangle \\
 & & x'_{h2} \mapsto \langle a_{nil} \\
 & & x'_{t2} \mapsto \langle a_2 \rangle
\end{array}
$$

$\xrightarrow{\textsc{WhnfCons}^-}$

$$
\begin{array}{lll}
 & & x'_{nil} \mapsto \langle a_{nil} \rangle \\
 & & x'_1 \mapsto \langle a_1 \rangle \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_{p1} \mapsto \langle a_{c1} \rangle \\
a_1 \mapsto a_{c2} & & x'_{h1} \mapsto \langle a_{nil} \rangle \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle & \langle a_{nil} \rangle : \langle a_{nil} \rangle \mathbin{@} \underline{a_{c3}} & x'_{t1} \mapsto \langle a_1 \rangle \qquad [\mathsf{N}a_2] \\
a_2 \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{T})_N & & x'_2 \mapsto \langle a_2 \rangle \\
a_{c2} \mapsto \langle a_{nil} \rangle : \langle a_2 \rangle & & x'_{p2} \mapsto \langle a_{c2} \rangle \\
\underline{a_{c3}} \mapsto \langle a_{nil} \rangle : \langle a_{nil} \rangle & & x'_{h2} \mapsto \langle a_{nil} \\
 & & x'_{t2} \mapsto \langle a_2 \rangle
\end{array}
$$

$\xrightarrow{\textsc{UpdateNormalThunk}^{-}}$

$$
\begin{array}{l}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\
a_1 \mapsto a_{c2} \\
a_{c1} \mapsto \langle a_{nil} \rangle : \langle a_1 \rangle \\
a_2 \mapsto a_{c3} \\
a_{c2} \mapsto \langle a_{nil} \rangle : \langle a_2 \rangle \\
a_{c3} \mapsto \langle a_{nil} \rangle : \langle a_{nil} \rangle
\end{array}
\quad
\langle a_{nil} \rangle : \langle a_{nil} \rangle \mathbin{@} a_{c3}
\quad
\begin{array}{l}
x'_{nil} \mapsto \langle a_{nil} \rangle \\
x'_1 \mapsto \langle a_1 \rangle \\
x'_{p1} \mapsto \langle a_{c1} \rangle \\
x'_{h1} \mapsto \langle a_{nil} \rangle \\
x'_{t1} \mapsto \langle a_1 \rangle \\
x'_2 \mapsto \langle a_2 \rangle \\
x'_{p2} \mapsto \langle a_{c2} \rangle \\
x'_{h2} \mapsto \langle a_{nil} \\
x'_{t2} \mapsto \langle a_2 \rangle
\end{array}
\quad \phi
$$

最後に，アドレス a_2 の内容を，強制結果への参照で上書きする．

最終的に，このプログラムは $\langle a_{nil} \rangle : \langle a_{nil} \rangle \mathbin{@} a_{c3}$ という結果に至り，停止した．

4.4.2 *expr* の簡約

expr の簡約も，初期状態の四つ組では *ExprOrWhnf* 以外の要素は ϕ である．

$$
\phi \qquad \mathsf{let}\ x_{nil} = \mathsf{Nil}\ \mathsf{in} \ldots \qquad \phi \qquad \phi
$$

$\xrightarrow{\textsc{Let}}$

$$
\underline{a_{nil}} \mapsto (\mathsf{Nil}, \mathsf{F})_N \quad \mathsf{case}\ (\mathsf{letreuse} \ldots)\ \mathsf{of}\ x_{p1}\{\ldots\} \quad \underline{x'_{nil}} \mapsto \langle \underline{a_{nil}}, \mathsf{F} \rangle \quad \phi
$$

$\xrightarrow{\textsc{Case}}$

$$
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \quad \mathsf{let}\ x_1 = \mathsf{letreuse}\ x_2 \ldots\ \mathsf{in} \ldots \quad x'_{nil} \mapsto \langle a_{nil}, \mathsf{F} \rangle \quad [\mathsf{B}\underline{x'_{p1}}] +\!\!+ [\mathsf{P}p_0]
$$

ここで，letreuse 式の簡約が必要となり，

$\xrightarrow{\textsc{LetreuseAlloc}}$

$$
\begin{array}{l}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\
\underline{a_1} \mapsto (\mathsf{letreuse}\ x_2 = \ldots, \mathsf{F})_R
\end{array}
\quad \mathsf{Cons}\ x'_{nil}\ x'_1 \quad
\begin{array}{l}
x'_{nil} \mapsto \langle a_{nil}, \mathsf{F} \rangle \\
\underline{x'_1} \mapsto \langle a_1, \mathsf{F} \rangle
\end{array}
\quad [\mathsf{B}x'_{p1}] +\!\!+ [\mathsf{P}p_0]
$$

というように再利用サンクを割り当てる．ここでは，まだ再利用サンクがヒープ内に存在しないため，新たな再利用サンクを割り当てる必要がある．続いて，$expr^{-}$ のときと同様に，パターンマッチを処理する．

$\xrightarrow{\textsc{WhnfCons}}$

$$
\begin{array}{lll}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \quad [\mathsf{B}x'_{p1}] \\
a_1 \mapsto (\mathsf{letreuse}\ x_2 = \ldots, \mathsf{F})_R & \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle \mathbin{@} \underline{a_{c1}} & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \quad +\!\!+[\mathsf{P}p_0] \\
\underline{a_{c1}} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle & &
\end{array}
$$

$\xrightarrow{\textsc{CaseBind}}$

$$
\begin{array}{lll}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
a_1 \mapsto (\mathsf{letreuse}\ x_2 = \ldots, \mathsf{F})_R & \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle \mathbin{@} a_{c1} & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \quad [\mathsf{P}p_0] \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle & & x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle
\end{array}
$$

$\xrightarrow{\textsc{CaseCons}}$

$$
\begin{array}{lll}
 & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \\
a_1 \mapsto (\mathsf{letreuse}\ x_2 = \ldots, \mathsf{F})_R & \mathsf{case}\ x'_{t1}\ of\ x_{p2}\ \{\ldots\} & x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle \quad \phi \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle & & \underline{x'_{h1}} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
 & & \underline{x'_{t1}} \mapsto \langle a_{c1}, \mathsf{T}\rangle
\end{array}
$$

ここで，コンスセルの tail 部を参照する x'_{t1} について，環境中のフラグが T となり，tail 部の間接参照となっていることに注意されたい．続いて case 式の x'_{t1} を簡約する．

$\xrightarrow{\textsc{Case}}$

$$
\begin{array}{lll}
 & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \\
a_1 \mapsto (\mathsf{letreuse}\ x_2 = \ldots, \mathsf{F})_R \quad x'_{t1} & & x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle \quad [\mathsf{B}\underline{x'_{p2}}] +\!\!+ [\mathsf{P}p_1] \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle & & x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
 & & x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle
\end{array}
$$

ここで，変数 x'_{t1} の参照する内容を取るとき，ヒープの場所に設定されたフラグを参照し，コンスセルの tail 部にあるアドレス a_1 の再利用サンクを参照する．再利用サンク

は，VarReuseThunk を適用することで強制する．

$\xrightarrow{\textsc{VarReuseThunk}}$

$$
\begin{array}{lll|ll}
 & & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & \mathsf{letreuse} & & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle & [\mathsf{R}a_{c1}] \\
a_1 \mapsto (\mathsf{letreuse}\ x_2 = \ldots, \mathsf{T})_R & x_2 = \mathsf{Cons}\ x'_{nil}\ x'_{nil} & & x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle & +\!\!+[\mathsf{B}x'_{p2}] \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle & \mathsf{in}\ \ldots & & x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle & +\!\!+[\mathsf{P}p_1] \\
 & & & x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle &
\end{array}
$$

アドレスから遅延した式をとりだし，フラグを T にした上で，その再利用サンクを強制しはじめる．このとき，スタックにはその再利用サンクの強制後の操作の準備のため，a_1 を tail 部に持つ a_{c1} を積む．x'_{t1} を参照する際に，a_1 は a_{c1} の tail 部の間接参照であるという関係が分かるため，このような対応付けが可能となる．

次に，再利用サンクを強制中に，別の letreuse 式が出現するため，そこで再利用が起こる．再利用サンクを強制中ということは，スタックのトップに $\mathsf{R}a_{c1}$ があるということから判断できるため，LetreuseAlloc ではなく LetreuseReuse で簡約を進める．

$\xrightarrow{\textsc{LetreuseReuse}}$

$$
\begin{array}{lll|ll}
 & & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle & [\mathsf{R}a_{c1}] \\
a_1 \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{F})_R & \mathsf{Cons}\ x'_{nil}\ x'_2 & & x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle & +\!\!+[\mathsf{B}x'_{p2}] \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle & & & x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle & +\!\!+[\mathsf{P}p_1] \\
 & & & x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle & \\
 & & & x'_2 \mapsto \langle a_1, \mathsf{F}\rangle &
\end{array}
$$

ここで a_1 にあった再利用サンクを破壊的に書き換えて用いる．環境中には，a_1 を参照する変数 x'_1 が存在するが，この変数は今後の簡約に現れないことが変換 $\mathcal{P}$ により保証されているため，破壊的な書き換えで全体の計算が矛盾した状態になることはない．

$\xrightarrow{\textsc{WhnfCons}}$

$$\begin{array}{l} a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\ a_1 \mapsto (\mathsf{Cons}\, x'_{nil}\, x'_{nil}, \mathsf{F})_R \\ a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_1, \mathsf{F}\rangle \\ \underline{a_{c2}} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle \end{array} \quad \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle \mathbin{@} \underline{a_{c2}} \quad \begin{array}{l} x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\ x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \\ x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle \\ x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\ x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle \\ x'_2 \mapsto \langle a_1, \mathsf{F}\rangle \end{array} \quad \begin{array}{l} [\mathsf{R}a_{c1}] \\ +\!\!+[\mathsf{B}x'_{p2}] \\ +\!\!+[\mathsf{P}p_1] \end{array}$$

もともと a_1 にあった再利用サンクの強制結果は，コンスセルであった．強制結果が求まったので，UpdateTail により後処理をする．

$\xrightarrow{\textsc{UpdateTail}}$

$$\begin{array}{l} a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\ a_1 \mapsto (\mathsf{Cons}\, x'_{nil}\, x'_{nil}, \mathsf{F})_R \\ a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{c2}, \mathsf{F}\rangle \\ a_{c2} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle \end{array} \quad \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle \mathbin{@} a_{c2} \quad \begin{array}{l} x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\ x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \\ x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle \\ x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\ x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle \\ x'_2 \mapsto \langle a_1, \mathsf{F}\rangle \end{array} \quad \begin{array}{l} [\mathsf{B}x'_{p2}] \\ +\!\!+[\mathsf{P}p_1] \end{array}$$

ここでは，スタックに積んでおいた情報を用いて，a_1 を参照していたコンスセル a_{c1} の tail 部を強制結果のコンスセルに書き換える．このことによって，リストを正しく接続する．($expr^-$ の場合には，a_1 を間接参照としてリストを接続しなければならなかった．) 再利用サンクを強制し終わったので，スタックの情報をもとにパターンマッチを行う．

$\xrightarrow{\textsc{CaseBind}}$

$$\begin{array}{l} a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\ a_1 \mapsto (\mathsf{Cons}\, x'_{nil}\, x'_{nil}, \mathsf{F})_R \\ a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{c2}, \mathsf{F}\rangle \\ a_{c2} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle \end{array} \quad \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle \mathbin{@} a_{c2} \quad \begin{array}{l} x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\ x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \\ x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle \\ x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\ x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle \\ x'_2 \mapsto \langle a_1, \mathsf{F}\rangle \\ x'_{p2} \mapsto \langle a_{c2}, \mathsf{F}\rangle \end{array} \quad [\mathsf{P}p_1]$$

$\xRightarrow{\textsc{CaseCons}}$

$$
\begin{array}{l}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\
a_1 \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{F})_R \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{c2}, \mathsf{F}\rangle \\
a_{c2} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle
\end{array}
\quad
x'_{t2}
\quad
\begin{array}{l}
x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \\
x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle \\
x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
x'_{t1} \mapsto \langle a_1, \mathsf{F}\rangle \\
x'_2 \mapsto \langle a_2, \mathsf{F}\rangle \\
x'_{p2} \mapsto \langle a_{c2}, \mathsf{F}\rangle \\
\underline{x'_{h2}} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
\underline{x'_{t2}} \mapsto \langle a_{c2}, \mathsf{T}\rangle
\end{array}
\quad \phi
$$

今度の case 式によるパターンマッチは，変数 x'_{t2} の値を簡約対象としている．この変数は，パターンマッチによりコンスセルの tail 部を取り出したもので，環境中のフラグが T であり，a_{c2} の tail 部を間接参照している．そのため，先の x'_{t1} を強制した場合と同様にして強制を進める．

$\xRightarrow{\textsc{VarReuseThunk}}$

$$
\begin{array}{l}
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N \\
a_1 \mapsto (\mathsf{Cons}\ x'_{nil}\ x'_{nil}, \mathsf{T})_R \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{c2}, \mathsf{F}\rangle \\
a_{c2} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle
\end{array}
\quad
\mathsf{Cons}\ x'_{nil}\ x'_{nil}
\quad
\begin{array}{l}
x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
x'_1 \mapsto \langle a_1, \mathsf{F}\rangle \\
x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle \\
x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle \\
x'_2 \mapsto \langle a_2, \mathsf{F}\rangle \\
x'_{p2} \mapsto \langle a_{c2}, \mathsf{F}\rangle \\
x'_{h2} \mapsto \langle a_{nil}, \mathsf{F}\rangle \\
x'_{t2} \mapsto \langle a_{c2}, \mathsf{T}\rangle
\end{array}
\quad [\mathsf{R}a_{c2}]
$$

$\xrightarrow{\textsc{WhnfCons}}$

$$
\begin{array}{lll|l|l}
 & & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
 & & & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle & \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & & x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle & \\
a_1 \mapsto (\mathsf{Cons}\, x'_{nil}\, x'_{nil}, \mathsf{T})_R & & & x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{c2}, \mathsf{F}\rangle & \langle a_{nil}, \mathsf{F}\rangle : \langle a_{nil}, \mathsf{F}\rangle \mathbin{@} \underline{a_{c3}} & & x'_{t1} \mapsto \langle a_{c1}, \mathsf{T}\rangle & [\mathsf{R}a_{c2}] \\
a_{c2} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_2, \mathsf{F}\rangle & & & x'_2 \mapsto \langle a_2, \mathsf{F}\rangle & \\
\underline{a_{c3}} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{nil}, \mathsf{F}\rangle & & & x'_{p2} \mapsto \langle a_{c2}, \mathsf{F}\rangle & \\
 & & & x'_{h2} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
 & & & x'_{t2} \mapsto \langle a_{c2}, \mathsf{T}\rangle &
\end{array}
$$

$\xrightarrow{\textsc{UpdateTail}}$

$$
\begin{array}{lll|l|l}
 & & & x'_{nil} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
 & & & x'_1 \mapsto \langle a_1, \mathsf{F}\rangle & \\
a_{nil} \mapsto (\mathsf{Nil}, \mathsf{F})_N & & & x'_{p1} \mapsto \langle a_{c1}, \mathsf{F}\rangle & \\
a_1 \mapsto (\mathsf{Cons}\, x'_{nil}\, x'_{nil}, \mathsf{T})_R & & & x'_{h1} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
a_{c1} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{c2}, \mathsf{F}\rangle & \langle a_{nil}, \mathsf{F}\rangle : \langle a_{nil}, \mathsf{F}\rangle \mathbin{@} a_{c3} & & x'_{t1} \mapsto \langle a_1, \mathsf{F}\rangle & \phi \\
a_{c2} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{c3}, \mathsf{F}\rangle & & & x'_2 \mapsto \langle a_2, \mathsf{F}\rangle & \\
a_{c3} \mapsto \langle a_{nil}, \mathsf{F}\rangle : \langle a_{nil}, \mathsf{F}\rangle & & & x'_{p2} \mapsto \langle a_{c2}, \mathsf{F}\rangle & \\
 & & & x'_{h2} \mapsto \langle a_{nil}, \mathsf{F}\rangle & \\
 & & & x'_{t2} \mapsto \langle a_2, \mathsf{F}\rangle &
\end{array}
$$

最後に，a_{c2} の tail 部を，強制結果である a_{c3} への参照へと上書きする．

最終的に，このプログラムは $\langle a_{nil}, \mathsf{F}\rangle : \langle a_{nil}, \mathsf{F}\rangle \mathbin{@} a_{c3}$ という結果に至り停止した．$expr^-$ の場合との違いは，ヒープ内に a_2 というアドレスを新たに確保することなく，a_1 を再利用することで，同じ結果が得られたことである．

表 4.1: $expr^-$ と $expr$ の簡約過程の対応

ステップ	$expr^-$ の簡約過程	$expr$ の簡約過程
1	Let$^-$	Let
2	Case$^-$	Case
3	Let$^-$	LetreuseAlloc
4	WhnfCons$^-$	WhnfCons
5	CaseBind$^-$	CaseBind
6	CaseCons$^-$	CaseCons
7	Case$^-$	Case
8	VarNormalThunk$^-$	VarReuseThunk
9	Let$^-$	LetreuseReuse
10	WhnfCons$^-$	WhnfCons
11	UpdateNormalThunk$^-$	UpdateTail
12	CaseBind$^-$	CaseBind
13	CaseCons$^-$	CaseCons
14	VarNormalThunk$^-$	VarReuseThunk
15	WhnfCons$^-$	WhnfCons
16	UpdateNormalThunk$^-$	UpdateTail

4.4.3　$expr^-$ と $expr$ の簡約過程の比較

$expr^-$ と $expr$ は，ともに head 部と tail 部に `Nil` への参照を持つコンスセルという結果が求まった．両者は，大部分が同様の規則を用いて簡約されて，両者が同じステップ数で簡約結果を得ていた．両者の簡約規則の適用過程を表 4.1 にまとめる．

$expr^-$ の簡約は，初期状態から始めて，Let$^-$，Case$^-$，Let$^-$ と進み，UpdateNormalThunk$^-$ の後で簡約が止まり結果が求まった．それに対して，$expr$ の簡約は初期状態から始めて Let，Case，LetreuseAlloc と順に簡約が進み，UpdateTail の後で簡約が止まり，$expr^-$ と同様の結果が求まった．同じステップ数を経た際のそれぞれの四つ組は，対応付けて同じとみなすことができる状態にある．このことは，次節で示す Thunk Recycling の正当性の証明の基礎となっている．

4.5 Thunk Recycling の正当性

前節の意味論を用いて，Thunk Recycling の正当性の証明を行った．ここでいう正当性とは，ある $expr^- \in Expr^-$ に対して，プログラム変換 $\mathcal{C}$ と $\mathcal{P}$ を適用した結果である $expr \in Expr$ があるとき，それぞれを簡約すると，“対等” な状態に至ることをいう．

まず，ある $expr^-$ の簡約中に出現する四つ組 $Q^- = H^-;\ t^-;\ \Gamma^-;\ S^-$ と，$expr$ の簡約中に出現する四つ組 $Q = H;\ t;\ \Gamma;\ S$ の間の対等関係（$\cong$ で示す）を定義する．直感的に説明すれば，状態が対等であるとは，一方では通常サンクであったものが再利用サンクであるという違いを除き，環境の中に含まれる有効な変数，結果の弱頭部正規形，スタックの要素が，同じであることを言う．次に，再利用の有無で簡約が足並みを揃えて進むことに基づいて，双模倣 [42] の考え方で正当性の証明を行う．

証明を始めるまえに，式についての以下の点に注意されたい．

注 1 (単一参照) $expr$ 中の letreuse $x = e_1$ in e_2 について，x は e_1 の中に出現せず，かつ，e_2 の中であるコンスセルの tail 部に一度のみ出現する．

これは，変換 $\mathcal{P}$ の結果として letreuse 式が導入されることから成立する．

双模倣は，以下のように定義される [42].

定義 1 (双模倣) 簡約規則 μ^- を図 4.9 で定義した規則のひとつとし，簡約規則 μ を図 4.10 で定義した規則のいずれかの規則とする．また，$expr^-$ の簡約中に出現する状態を Q_1^-，$expr$ の簡約中に出現する状態を Q_1 とし，$Q_1^- \cong Q_1$ という関係が成り立つとする．このとき，以下の両者が成り立つとき，対等関係 $\cong$ は 双模倣 であるという．

1. $Q_1^- \xRightarrow{\mu^-} Q_2^-$ となるすべての μ^- と Q_2^- に対し，$Q_1 \xRightarrow{\mu} Q_2$ となる μ と Q_2 が存在するならば $Q_2^- \cong Q_2$ である．

2. $Q_1 \xRightarrow{\mu} Q_2$ となるすべての μ と Q_2 に対し，$Q_1^- \xRightarrow{\mu^-} Q_2^-$ となる μ^- と Q_2^- が存在するならば $Q_2^- \cong Q_2$ である．

以下，四つ組の状態の対等性を定義する．そのため，四つ組の各要素に関して，順に対等性を定義する．

定義 2 (式の対等性) 以下の条件のいずれかが成立するとき，式 e^- と e は対等であるといい，$e^- \cong e$ と表す．

- $e^- = \backslash x \to e_1^- \ \wedge\ e = \backslash x \to e_1 \ \wedge\ e_1^- \cong e_1$.

- $e^- = \mathsf{Nil} \wedge e = \mathsf{Nil}$.
- $e^- = \mathsf{Cons}\, v_h^-\, v_t^- \wedge e = \mathsf{Cons}\, v_h\, v_t \wedge v_h^- \cong v_h \wedge v_t^- \cong v_t$.
- $e^- = v^- \wedge e = v \wedge v^- \cong v$.
- $e^- = e_1^-\, v^- \wedge e = e_1\, v \wedge e_1^- \cong e_1 \wedge v^- \cong v$.
- $e^- = \mathsf{case}\, e_1^-\, \mathsf{of}\, x\, p^- \wedge e = \mathsf{case}\, e_1\, \mathsf{of}\, x\, p \wedge e_1^- \cong e_1 \wedge p^- \cong p$.
- $e^- = \mathsf{let}\, x = e_1^-\, \mathsf{in}\, e_2^- \wedge e = \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \wedge e_1^- \cong e_1 \wedge e_2^- \cong e_2$.
- $e^- = \mathsf{let}\, x = e_1^-\, \mathsf{in}\, e_2^- \wedge e = \mathsf{letreuse}\, x = e_1\, \mathsf{in}\, e_2 \wedge e_1^- \cong e_1 \wedge e_2^- \cong e_2$.

定義 3 (パターンの対等性) パターン p^- を $\{\, \mathsf{Nil} \to e_n^-;\ \mathsf{Cons}\, x_h\, x_t \to e_c^- \,\}$ とし，パターン p を $\{\, \mathsf{Nil} \to e_n;\ \mathsf{Cons}\, x_h\, x_t^* \to e_c \,\}$ とするとき，$e_n^- \cong e_n$ かつ $e_c^- \cong e_c$ ならば，p^- と p は対等であるといい，$p^- \cong p$ と表す.

定義 4 (変数の対等性) $v^- = x \wedge v = x$ または $v^- = x \wedge v = x^*$ のとき, v^- と v は対等であるといい，$v^- \cong v$ と表す.

定義 5 (ヒープの場所の対等性) 以下の条件のいずれかが成り立つとき，あるヒープの場所 h^- と h は対等であるといい，$h^- \cong h$ と表す.

- $\mathcal{A}^-(h^-) = w^- \mathbin{@} a_1 \wedge \mathcal{A}(h) = w \mathbin{@} a_2 \wedge w^- \cong w$.
- $\mathcal{A}^-(h^-) = (e^-, \mathsf{F})_N \mathbin{@} a_1 \wedge \mathcal{A}(h) = (e, \mathsf{F})_N \mathbin{@} a_2 \wedge e^- \cong e$.
- $\mathcal{A}^-(h^-) = (e^-, \mathsf{T})_N \mathbin{@} a_1 \wedge \mathcal{A}(h) = (e, \mathsf{T})_N \mathbin{@} a_2 \wedge e^- \cong e$.
- $\mathcal{A}^-(h^-) = (e^-, \mathsf{F})_N \mathbin{@} a_1 \wedge \mathcal{A}(h) = (e, \mathsf{F})_R \mathbin{@} a_2 \wedge e^- \cong e$.
- $\mathcal{A}^-(h^-) = (e^-, \mathsf{T})_N \mathbin{@} a_1 \wedge \mathcal{A}(h) = (e, f)_R \mathbin{@} a_2$.

この定義において，一番最後の条件は注意が必要である．あるヒープの場所 h^- には通常サンクがあり，別のヒープの場所 h には再利用サンクがある．このとき，注 1 より h は単一参照であり，そこにある再利用サンクが評価中であれば，サンクの内容（サンクの式 e とフラグ f）を再び参照することはない．そのため，他の条件とは異なり，サンクの保持する式 e^- と e の対等性を調べる必要はない.

定義 6 (弱頭部正規形の対等性) 以下の条件のいずれかが成り立つとき，ある弱頭部正規形 w^- と w は対等であるといい，$w^- \cong w$ と表す.

- $w^- = \lambda\, x\,.\, e^- \ \wedge\ w = \lambda\, x\,.\, e \ \wedge\ e \cong e^-$
- $w^- = [\,] \ \wedge\ w = [\,]$
- $w^- = h_h^- : h_t^- \ \wedge\ w = h_h : h_t \ \wedge\ h_h^- \cong h_h \ \wedge\ h_t^- \cong h_t$

環状のコンスセルが存在すると，ヒープの場所の対等性と弱頭部正規形の対等性が相互再帰的に定義されることに注意する必要がある．これらの定義には明示されていないが，一度確認した対等性が再び現われたとき，その先の対等性は調べずに対等であるものとする．

定義 7 (*ExprOrWhnf* の対等性) 以下の条件のいずれかが成り立つとき，t^- と t が対等であるといい，$t^- \cong t$ と表す．

- $t^- = e^- \ \wedge\ t = e \ \wedge\ e^- \cong e$
- $t^- = w^- \mathbin{@} a_1 \ \wedge\ t = w \mathbin{@} a_2 \ \wedge\ w^- \cong w$

定義 8 (スタックの対等性) 以下の条件のいずれかが成り立つとき，スタック S^- と S は対等であるといい，$S^- \cong S$ と表す．

- $S^- = \phi \ \wedge\ S = \phi$
- $S^- = [\mathsf{P}p^-] +\!\!+ S_1^- \ \wedge\ S = [\mathsf{P}p] +\!\!+ S_1 \ \wedge\ p^- \cong p \ \wedge\ S_1^- \cong S_1$
- $S^- = [\mathsf{A}h^-] +\!\!+ S_1^- \ \wedge\ S = [\mathsf{A}h] +\!\!+ S_1 \ \wedge\ h^- \cong h \ \wedge\ S_1^- \cong S_1$
- $S^- = [\mathsf{B}x] +\!\!+ S_1^- \ \wedge\ S = [\mathsf{B}x] +\!\!+ S_1 \ \wedge\ S_1^- \cong S_1$
- $S^- = [\mathsf{N}a_1] +\!\!+ S_1^- \wedge S = [\mathsf{N}a_2] +\!\!+ S_1 \wedge H^-(a_1) = (e^-, \mathsf{T})_N \wedge H(a_2) = (e, \mathsf{T})_N \wedge e^- \cong e \ \wedge\ S_1^- \cong S_1$
- $S^- = [\mathsf{N}a_1] +\!\!+ S_1^- \ \wedge\ S = [\mathsf{R}a_2] +\!\!+ S_1 \ \wedge\ H^-(a_1) = (e^-, \mathsf{T})_N \ \wedge\ H(a_2) = h_h : \langle a_t, \mathsf{F}\rangle \wedge H(a_t) = (e, f)_R \ \wedge\ S_1^- \cong S_1$

定義 9 (状態の対等性) 以下のすべてが成り立つとき，ある状態 $H^-;\ t^-;\ \Gamma^-;\ S^-$ と $H;\ t;\ \Gamma;\ S$ は対等であるといい，$H^-;\ t^-;\ \Gamma^-;\ S^- \cong H;\ t;\ \Gamma;\ S$ と表す．

- $t^- \cong t$
- $S^- \cong S$
- $\forall v_1^- \in V^-\,.\,\exists v_1 \in V\,.\,\Gamma^-(v_1^-) \cong \Gamma(v_1)$

表 4.2: 規則の対応関係

	μ^- (*Expr*$^-$ の簡約規則)	μ (*Expr* の簡約規則)
(1)	WhnfFunc$^-$	WhnfFunc
(2)	WhnfNil$^-$	WhnfNil
(3)	WhnfCons$^-$	WhnfCons
(4)	VarWhnf$^-$	VarWhnf
(5)	VarNormalThunk$^-$	VarNormalThunk
(6)	VarNormalThunk$^-$	VarReuseThunk
(7)	UpdateNormalThunk$^-$	UpdateNormalThunk
(8)	UpdateNormalThunk$^-$	UpdateTail
(9)	Apply$^-$	Apply
(10)	Apply2$^-$	Apply2
(11)	Case$^-$	Case
(12)	CaseBind$^-$	CaseBind
(13)	CaseNil$^-$	CaseNil
(14)	CaseCons$^-$	CaseCons
(15)	Let$^-$	Let
(16)	Let$^-$	LetreuseAlloc
(17)	Let$^-$	LetreuseReuse

- $\forall v_2 \in V\,.\,\exists v_2^- \in V^-\,.\,\Gamma(v_2) \cong \Gamma^-(v_2^-)$

ここで，V^- は Γ^- に含まれる変数のうち参照され得る変数の集合，V は Γ に含まれる変数のうち参照され得る変数の集合であるとする．

定理 1 (Thunk Recycling の正当性) Thunk Recycling はプログラムの結果の対等性を崩さない．

証明. 証明は，双模倣と帰納法に基づく．4.3 節で定義した規則には，表 4.2 に挙げる一対一の対応関係があることを利用する．つまり，Thunk Recycling のない状態 Q^- が表中のある規則で簡約されるとき，Thunk Recycling のある状態 Q はそれに対応するとした規則で簡約される．簡約時にある変数・識別子を新たに導入する際には，同じ名前で導入されるため，定義 9 における V^- と V は同じとなる．

対応関係 (15) から (17) については，Let$^-$ が複数の規則と対応しているが，ここにあいまい性がないことに注意されたい．図 4.9 において，ある状態が，複数の規則にマッチすることはないよう定義されている．

一対一の対応において，定義 1 の条件 1 が成り立つことを確認する．表 4.2 の (1)–(5), (7)，(9) – (15) について成り立つことは明らかである．そこで，残りの (6)，(8)，(14), (16)，(17) について，個別に確認する．簡約規則を対応関係ごとにまとめたものを，図 4.13 に再掲する．説明のため，図 4.13 では，図 4.9 と図 4.10 の定義から名前の付け替えを適宜行っている．

対応関係 (6) VarNormalThunk$^-$ は通常サンクのフラグを書き換えた上で，スタックに $\mathsf{N}a$ を積む．それに対して，VarReuseThunk は再利用サンクのフラグを書き換えた上で，スタックに $\mathsf{R}a$ を積む．よって，それぞれの定義より，

$$H^-(a_1) = (e^-, \mathsf{T})_N \ \wedge \ H(a_2) = h_h : \langle a_t, \mathsf{F} \rangle \ \wedge \ H(a_t) = (e, f)_R$$

となり，$S^- \cong S$ であるとき，定義 8 より $[\mathsf{N}a_1] +\!\!+ S^- \cong [\mathsf{R}a_2] +\!\!+ S$ となり，スタックの対等性を崩さない．

対応関係（8） 定義 8 より，$[\mathsf{N}a_1] +\!\!+ S^- \cong [\mathsf{R}a_2] +\!\!+ S$ であれば $S^- \cong S$ であり，スタックの対等性を崩さない．UpdateNormalThunk$^-$ は通常サンクの内容を強制結果の弱頭部正規形 w^- へと書き換える．同様にして，UpdateTail はアドレス a にあるコンスセルの tail 部を強制結果の弱頭部正規形 w へと書き換える．対応関係 (6) より，UpdateNormalThunk$^-$ では，

$$\Gamma^-(x) = h_v^- \ \wedge \ \mathcal{A}^-(h_v^-) = (e^-, \mathsf{F})_N \mathbin{@} a_1$$

となっており，UpdateTail では，

$$\Gamma(x) = h_t \ \wedge \ \mathcal{A}(h_t) = (e, \mathsf{F})_R \mathbin{@} a_t \ \wedge \ H(a_2) = h_h : \langle a_t, \mathsf{F} \rangle$$

となっている．よって，a_1 と a_2 を書き換えた後のヒープの状態で，

$$H^-(a_1) = a_w \ \wedge \ H(a_2) = h_h : \langle a_w, \mathsf{F} \rangle \ \wedge \ H(a_t) = (e, f)_R$$

となっていても，対等性を崩さない．

対応関係（14） CaseCons$^-$ の x_t' はあるコンスセルの tail 部を参照しており，CaseCons の x_t' は tail 部への間接参照である．よって，CaseCons において $h = \langle a, \mathsf{T} \rangle$

とすると，$\mathcal{A}(h)$ は h_t の内容を示す．$h_t^- \cong h_t$ であるので，簡約後の状態の対等性も保たれている．ヒープの場所の対等性が，補助関数 $\mathcal{A}^-$ と $\mathcal{A}$ を用いて定義されていることに注意されたい．

対応関係（16）それぞれ，Let$^-$ が通常サンクを割り当て，LetreuseAlloc が再利用サンクを割り当てる．簡約規則より，

$$\mathcal{A}^-(\langle a_1 \rangle) = (e_1^-[x'/x], \mathsf{F})_N \ \wedge \ \mathcal{A}(\langle a_2, \mathsf{F} \rangle) = (e_1, \mathsf{F})_R$$

であり，定義 7 と 定義 2 より

$$\mathsf{let}\ x = e_1^-\ \mathsf{in}\ e_2^- \cong \mathsf{letreuse}\ x = e_1\ \mathsf{in}\ e_2 \ \wedge \ e_1^- \cong e_1$$

である．よって，定義 5 より，x' で参照されるヒープの場所の対等性が保証される．

対応関係（17）Let$^-$ が通常サンクを割り当てているのに対し，LetreuseReuse はアドレス a_t にある再利用サンクを再利用している．対応関係（16）の場合と同様にして，

$$\mathcal{A}^-(\langle a \rangle) = (e_1^-[x'/x], \mathsf{F})_N \ \wedge \ \mathcal{A}(\langle a_t, \mathsf{F} \rangle) = (e_1, \mathsf{F})_R \ \wedge \ e_1^- \cong e_1$$

であるため，定義 5 より，x' で参照されるヒープの場所の対等性が保証される．アドレス a_t に書き換え前にあった再利用サンク $(e, \mathsf{T})_R$ は注 1 より，この簡約ステップ以後に，他から参照されることはなく，対等性を崩さない．

以上のように，表 4.2 が，すべての *Expr* に対する規則について対応関係があり，簡約後の状態が対等関係にあることが確認できた．つまり，Thunk Recycling の操作的意味論において，定義 1 の条件 1 が成り立つ．同様にして，定義 1 の条件 2 についても証明できる．よって，対等関係 $\cong$ は双模倣である．

最終的に，Thunk Recycling の正当性を帰納法に基づいて証明する．ある式 $expr^- \in Exprm$ と $expr \in Expr$ を考える．初期状態 $\phi;\ expr^-;\ \phi;\ \phi$ と $\phi;\ expr;\ \phi;\ \phi$ は定義 9 より対等である $Q_1^- \cong Q_1$ が成り立つと仮定する．Q_1^- の 1 ステップ後の状態 Q_2^- と Q_1 の 1 ステップ後の状態 Q_2 について，$Q_2^- \cong Q_2$ が成り立つ．これは，Q_1^- と Q_1 が双模倣であることから明らかである．

よって，Thunk Recycling はプログラムの結果の対等性を崩さないことが証明された．

4.6 本章のまとめ

本章では，Thunk Recycling の形式化について扱った．単純な関数型プログラミング言語 SF を定義し，SF のプログラムに対して単一参照性を保証するプログラミング変換を定義した．そのプログラム変換は，リストの消費者に関する変換 $\mathcal{C}$ とリストの生産者に関する変換 $\mathcal{P}$ である．変換 $\mathcal{C}$ ではパターンマッチにおいて間接参照を導入し，変換 $\mathcal{P}$ では再利用可能な単一参照であるサンクを letreuse 式により明示する．

その上で，Thunk Recycling を用いる場合と Thunk Recycling を用いない場合のそれぞれに対して，別の small-step の操作的意味論として定義した．それらの定義した意味論に基づいて簡約すれば，Thunk Recycling の適用の有無によりプログラムのふるまいが変わらないことを証明した．証明のために，まず，ふるまいが変わらないということを，状態間の対等関係として定義した．その対等関係が保たれたままで，再利用の有無で簡約が足並みを揃えて進むことを利用して，双模倣の考え方に基づいて，Thunk Recycling はプログラムの結果の対等性を崩さないことを証明した．

	Heap	*ExprOrWhnf*	*Environment*	*Stack*	*rule*
対応関係（6）					
	H^-	x	Γ^-	S^-	
	where $\Gamma^-(x) = h_v^-$, $\mathcal{A}^-(h_v^-) = (e^-, \mathsf{F})_N \mathbin{@} a_1$				VarNormalThunk$^-$
$\Rightarrow$	$H^-[a_1 \mapsto (e^-, \mathsf{T})_N]$	e^-	Γ^-	$[\mathsf{N}a_1] +\!\!+ S^-$	
	H	x	Γ	S	
	where $\Gamma(x) = h_t$, $\mathcal{A}(h_t) = (e, \mathsf{F})_R \mathbin{@} a_t$, $H(a_2) = h_h : \langle a_t, \mathsf{F}\rangle$				VarReuseThunk
$\Rightarrow$	$H[a_t \mapsto (e, \mathsf{T})_R]$	e	Γ	$[\mathsf{R}a_2] +\!\!+ S$	
対応関係（8）					
	H^-	$w^- \mathbin{@} a_w$	Γ^-	$[\mathsf{N}a_1] +\!\!+ S^-$	
$\Rightarrow$	$H^-[a_1 \mapsto a_w]$	$w^- \mathbin{@} a_w$	Γ^-	S^-	
					UpdateNormalThunk$^-$
	H	$w \mathbin{@} a_w$	Γ	$[\mathsf{R}a_2] +\!\!+ S$	
	where $H(a_2) = h_h : h_t$				UpdateTail
$\Rightarrow$	$H[a_2 \mapsto h_h : \langle a_w, \mathsf{F}\rangle]$	$w \mathbin{@} a_w$	Γ	S	
対応関係（14）					
	H^-	$h_h^- : h_t^- \mathbin{@} a$	Γ^-	$[\mathsf{P}p^-] +\!\!+ S^-$	
	where $p^- = \{ \cdots ; \mathsf{Cons}\, x_h\, x_t \to e_c^- \}$				CaseCons$^-$
$\Rightarrow$	H^-	$e_c^-[\underline{x'_h}/x_h][\underline{x'_t}/x_t]$	$\Gamma^-, \underline{x'_h} \mapsto h_h^-, \underline{x'_t} \mapsto h_t^-$	S^-	
	H	$h_h : h_t \mathbin{@} a$	Γ	$[\mathsf{P}p] +\!\!+ S$	
	where $p = \{ \cdots ; \mathsf{Cons}\, x_h\, x_t^* \to e_c \}$				CaseCons
$\Rightarrow$	H	$e_c[\underline{x'_h}/x_h][\underline{x'_t}/x_t^*]$	$\Gamma, \underline{x'_h} \mapsto h_h, \underline{x'_t} \mapsto \langle a, \mathsf{T}\rangle$	S	
対応関係（16）					
	H^-	$\mathsf{let}\ x = e_1^-\ \mathsf{in}\ e_2^-$	Γ^-	S^-	
$\Rightarrow$	$H^-, \underline{a_1} \mapsto (e_1^-[\underline{x'}/x], \mathsf{F})_N$	$e_2^-[\underline{x'}/x]$	$\Gamma^-, \underline{x'} \mapsto \langle \underline{a_1} \rangle$	S^-	Let$^-$
	H	$\mathsf{letreuse}\ x = e_1\ \mathsf{in}\ e_2$	Γ	S	
	where $S = \phi \vee \text{top of } S \neq \mathsf{R}a'$				LetreuseAlloc
$\Rightarrow$	$H, \underline{a_2} \mapsto (e_1, \mathsf{F})_R$	$e_2[\underline{x'}/x]$	$\Gamma, \underline{x'} \mapsto \langle \underline{a_2}, \mathsf{F}\rangle$	S	
対応関係（17）					
	H^-	$\mathsf{let}\ x = e_1^-\ \mathsf{in}\ e_2^-$	Γ^-	S^-	
$\Rightarrow$	$H^-, \underline{a} \mapsto (e_1^-[\underline{x'}/x], \mathsf{F})_N$	$e_2^-[\underline{x'}/x]$	$\Gamma^-, \underline{x'} \mapsto \langle \underline{a} \rangle$	S^-	Let$^-$
	H	$\mathsf{letreuse}\ x = e_1\ \mathsf{in}\ e_2$	Γ	$[\mathsf{R}a] +\!\!+ S$	
	where $H(a) = h_h : \langle a_t, \mathsf{F}\rangle$, $H(a_t) = (e, \mathsf{T})_R$				LetreuseReuse
$\Rightarrow$	$H[a_t \mapsto (e_1, \mathsf{F})_R]$	$e_2[\underline{x'}/x]$	$\Gamma, \underline{x'} \mapsto \langle a_t, \mathsf{F}\rangle$	S	

図 4.13: 対応関係の規則

第5章

GHC への Thunk Recycling の実装

純関数型言語 Haskell の代表的な処理系である GHC のバージョン 7.0.3 に，Thunk Recycling を実装した．本章では，その実装の詳細について述べる．いくつかの項目では，複数の実装方針が考えられ，採るべき方法の取捨選択が必要であるため，それらの利害得失を中心に論じる．基本的な方針は，4.3 節で示した操作的意味論を実装するものであるが，GHC の現状の実装に合わせる際に，検討が必要となる点がある．さらに，実行時間の削減を目指して導入した手法についても述べる．

5.1 GHC の概要

GHC の全体像は，2.5.2 節の図 2.2 に示した構造となっている．Thunk Recycling は，コンパイラとランタイムシステムに変更を加えることで実装した．コンパイラへは，変換 $\mathcal{C}$ と $\mathcal{P}$ を追加し，コード生成系にも修正を加えた．変換 $\mathcal{C}$ と $\mathcal{P}$ はともに，サンクが明示された形の言語である STG 言語の段階の変換パス（STG-to-STG パス）で導入した．また，STG 言語から C−− のコード生成時において，letreuse に対するコードを生成するための変更に加えて，コンスセル（データコンストラクタ）に関するコード生成系にも修正を加えた．

現在は，変換 $\mathcal{C}$ と 変換 $\mathcal{P}$ は図 4.2，4.3，4.4 で示したものをほぼそのまま実装しており，*occ* などの補助関数が同じ式に対して繰り返し呼ばれる可能性がある．コンパイル時間を短くするためには，式ごとに対応表を用意し，同じ式に対して繰り返し同じ処理が行われないようにすることが望ましい．しかし，コンパイル時間の削減は今回の主目的でないため，現状では対応表を用意せずに実装することとした．なお，`nofib` ベンチマー

ク [39] の real サブセットにあるコード量が比較的大きい anna というプログラム（全 31 ファイル，9520 行）のコンパイル時間は，修正前の GHC で 16.4 秒であったのに対して Thunk Recycling を実装したコンパイラでは 18.7 秒となり，約 14.1 % の増加にとどまっている．

ランタイムシステムには，再利用サンク用のオブジェクトタイプとコードを追加し，さらに再利用サンクの更新を行う再利用機構を追加した．また，ごみ集めも，再利用サンクに対応するように変更した．

なお，関数 `map` や `take` などの Haskell の標準関数については，Thunk Recycling を適用するようコンパイルし直して，置き換える必要がある．

5.2　再利用サンクに対応するオブジェクトの種類の追加

2.5.6 節で触れた既存の 59 種類のオブジェクトの種類に加えて，再利用サンクに対応するオブジェクトの種類 `THUNK_REUSE` を追加した．これに加えて，使用頻度の高い小さなサイズについて特殊なオブジェクトの種類 `THUNK_REUSE_`n（サイズ n の再利用サンク）を用意した．たとえば，`THUNK_REUSE_2` はサイズ 2 の再利用サンクに対応するオブジェクトの種類である．オブジェクトのタイプからサイズが分かっていれば，Info Table を重複して参照する必要がなくなるため，効率面での恩恵がある．また，典型的なプログラムにおいては，サンクのサイズは高々 10 程度であると考え，現在では `THUNK_REUSE_10` まで用意している．

これらのオブジェクトの種類を，既存のランタイムで適切に扱うため変更を加えた．具体的には，C 言語と C−− で書かれているランタイムの処理に対して，オブジェクトの種類によって振り分けられている箇所に，`THUNK_REUSE` など追加したオブジェクトの種類を扱えるようにした．大部分は通常サンクと同じように扱うが，以降で述べる項目については，再利用が起こることを考えて動作を変える必要があった．

5.3　再利用サンクのオブジェクト表現の検討

プログラム変換 $\mathcal{P}$ によって挿入された letreuse 式によって，再利用が起こり得る箇所を特定した．実行時に再利用サンクを強制して値が得られた後は，以下の 2 点が可能でなければならない．

- 現在強制中の（これから再利用する）再利用サンクへの参照が得られること．

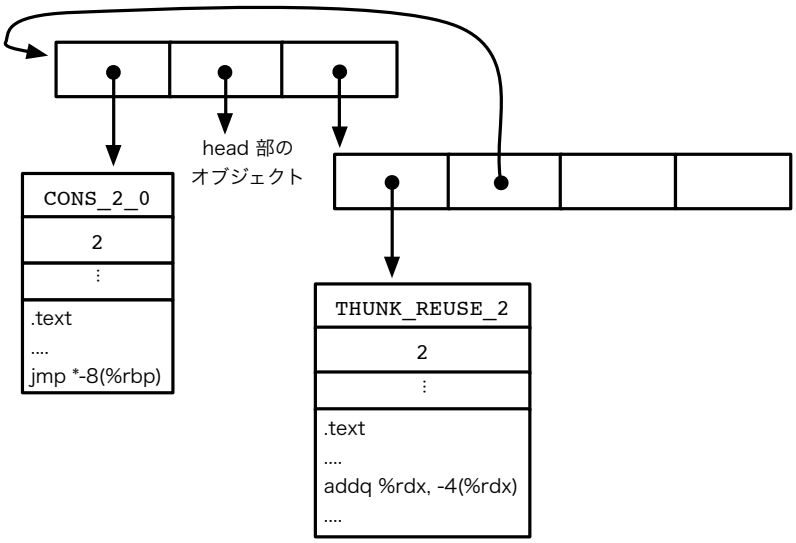

図 5.1: コンスセルへの参照を持つ再利用サンクの表現

- その再利用サンクを tail 部から参照しているコンスセルへの参照が得られること.

これらにより，コンスセルの tail 部を再利用サンクの強制結果で破壊的に書き換えることができる．1 つ目の項目について，再利用サンクを強制し始めるときに，その再利用サンクを実行時スタックに積む．そうすれば，強制結果を返すコード片においてスタックを参照して，再利用サンクを得ることができる．この時，積んでおいた再利用サンクへの参照は，スタックのトップにある.

2 つ目を可能とするには，以下の 2 つの方法が考えられる.

方法 1 再利用サンク内部に，その再利用サンクを tail 部から参照するコンスセルへの参照（**双方向ポインタ**）を持つ.

方法 2 スタックを用いて，再利用サンクとその再利用サンクを tail 部から参照するコンスセルとの対応づけを可能にする.

方法 1 では，図 5.1 のように再利用サンクを表現する．再利用サンクを起点として双方向ポインタをたどることにより，tail 部を書き換えるべきコンスセルを参照できる．この方法を採った際の利点と欠点は以下のとおりである.

利点 再利用サンクとコンスセルの対応が明解である.

欠点 1 サンクの環境を保持するペイロード部に，コンスセルへの参照という，本来の環境とは意味の異なる参照を保持するようなオブジェクト表現となるため，GHC への変更コストが増える．

欠点 2 再利用サンクのサイズが 1 ワード大きくなるため，ヒープスペースのオーバヘッドが生じる．特に長さの短いリストではメモリ削減の効果が薄くなる．

一方，方法 2 は，再利用サンクの強制を始める際に，スタックにコンスセルへの参照を積む．強制の開始時にコンスセルを積むことができるのは，$*$ の付けられた変数が，コンスセルのアドレスに対しフラグが付けられているためである．この方法の利点と欠点は次のようになる．

利点 通常サンクと再利用サンクの内部表現が同様であるため，GHC への変更コストが小さい．

欠点 4.2.1 節で示した変換 $\mathcal{C}$ で示した通り，パターンマッチで tail 部を参照するために導入する変数すべてに間接参照の印 $*$ を付ける必要がある．間接参照をたどる操作は，実行時のオーバヘッドになり得る．

本論文では，方法 2 を採用した．そのような選択を行った設計上の理由は，次の通りである．

- Thunk Recycling によるメモリ削減の効果を重視した．
- 方法 1 は，再利用サンクのサイズが 1 ワード大きくなってしまうことにより，短いリストを多用するプログラムに対して再利用の効果が薄まることを許容できないと考えた．
- 方法 1 では，再利用サンクに 1 ワード追加するための変更コストが小さくないと判断した．

なお，サイズが 2 の再利用サンクは，通常サンクと同じ内部表現であるため，図 5.2 のようなオブジェクトとなる．

5.4　再利用対象となるサンクの選定

コンスセルの tail 部を遅延するサンクのうち，再利用の対象となるサンクの候補を特定し，let 式から letreuse を含む式に変換することが，Thunk Recycling の実現に必要で

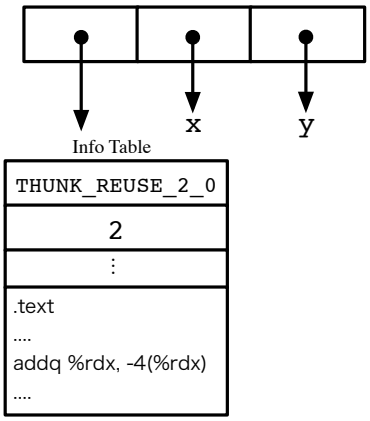

図 5.2: 再利用サンクの表現

ある．4.2.1 節で述べたように，変換 $\mathcal{P}$ は，単一参照性を保ちつつ，破壊的な更新が続けて起こらないという条件が必要であるが，それらの条件を満たせば，以下のような複数の方針が考えられる．

方針 1 式の返り値となるコンスセルについて，tail 部の遅延に用いる let を letreuse への置換対象とする．

方針 2 let で束縛される変数が，プログラムの文面上で一度しか現れないサンクだけを letreuse への置換対象とする．

4.2.1 節で述べたプログラム変換 $\mathcal{P}$ は，方針 1 を採っており，返り値となるリストのみを対象とすることで，関数の返り値となるリストの背骨[1] を構成するコンスセルの遅延のみに再利用サンクの利用を限定する．そこでは，letreuse 式の入れ子を許容しないことで，用いる再利用サンクを変換時に特定している．そのため，letreuse 式の実行時は，再利用するサンクがすでに更新されたものであるかチェックする必要がなく，単純な動作となる．これに対して，方法 2 を採った場合には，一度しか現れないサンクを生成する let 式を letreuse 式に変換することになるため，方法 1 と比べると，入れ子の letreuse 式を許容するなど積極的に letreuse 式へ変換する．その代わり，letreuse 式の実行時には，再利用サンクに対する破壊的な更新がすでになされたかどうか判別し，まだ更新がなされていない時だけ再利用することとなる．結果として，再利用が起きる機会は増える可能性があるが，実行時の処理が複雑となる．たとえば，4.2.2 節で挙げた letreuse 式の入れ

[1] リストの背骨とは，コンスセルの tail 部から次のコンスセルが接続されている一連のコンスセルをいう．

子により矛盾した状態になり得るとした以下のプログラムを考える．

$$
\begin{aligned}
&\mathsf{let}\ x_0 = e_0\ \mathsf{in}\\
&\mathsf{letreuse}\ x_1 = e_1\ \mathsf{in}\\
&\mathsf{letreuse}\ x_2 = e_2\ \mathsf{in}\\
&\mathsf{case}\ e_3\ \mathsf{of}\ x_3\ \{\ \mathsf{Nil} \to \mathsf{Cons}\ x_0\ x_1;\\
&\qquad\qquad \mathsf{Cons}\ x_h\ x_t \to \mathsf{Cons}\ x_0\ x_2\}
\end{aligned}
$$

方法 2 では，このような入れ子を許容することで，たとえば，x_1 では再利用サンクを再利用し，x_2 では新たな再利用サンクを割り当てるといったように，letreuse 式の動作を実行時の状況によって切り替えることとする．

いずれの方針を採るかを決定する際，再利用すべきサンクがまだ割り当てられていなければ，letreuse は再利用サンクを割り当てなければならないということに注意する必要がある．つまり，再利用できる可能性が低い場合には，letreuse に変換せずに let のままにしておき通常サンクを用いるほうが，再利用サンクが割当て済みかどうかを調べる必要がないため，実行速度の面から考えれば，オーバヘッドが少なく実装ができる可能性が高い．ここで，`map` や `take` などの多くの標準関数が再帰的にリストを生成することを考えると，方針 1 のように背骨を構成するサンクに再利用を限定し，再利用の可能性が高い部分のみ letreuse に置換することが妥当である．

以上の方針を採った上で，GHC への実装を考える．変換 $\mathcal{P}$ への入力 *Expr* が let と letruse でサンクの生成を明示して扱っていたように，GHC の実装でも STG 言語はサンクを明示している．一方，Core 言語にはサンクが明示されていない．そのため，STG 言語におけるパスに $\mathcal{P}$ に相当する変換を挿入する．具体的には，

```
transformP :: UniqSupply -> [StgBinding] -> [StgBinding]
```

というシグネチャを持つ関数 `transformP` を定義し，STG 言語の最適化パスに加える．

なお，letreuse 式に対して生成するコードは，再利用サンクがすでに割り当てられているか，また，それが本当に再利用可能であるか実行時にチェックを行う必要がある．つまり，コンパイラは以下のようなチェックを含んだコードを生成する．

- 再利用すべき再利用サンクがすでにあるか
- その再利用サンクに十分なペイロードサイズがあるか

2 つ目の項目について，サンクが遅延する式中の自由変数の数がペイロードのサイズとなっており，書き換えるのに十分なサイズがなければ再利用できないため，実行時にチェックしている．

5.5 単一参照の実現

本節では，再利用サンクへの単一参照性を保証する機構を，GHC に実装する際の設計方針について検討する．

5.5.1 tail 部での間接参照方法の検討

3.3.1 節で述べたように，パターンマッチにおいて，リストの消費時の単一参照性を考慮する．コンスセルのすべてのパターンマッチに再利用サンクが出現しうるため，GHC 処理系全体に改変が必要であり，慎重に設計しなければならない．本節では以降，実装のコストと実行効率の両面から，以下に示す 2 種類の方法を検討する．

方法 1 4.3 節の操作的意味論に従い，tail 部を間接参照することを示すフラグをポインタに持たせる．

方法 2 リストに対する標準関数の `tail` を用いて間接参照を実現する．

方法 1 について，4.3 節の操作的意味論は，4.2.1 節で述べたプログラム変換 $\mathcal{C}$ を基本としたもので，再利用サンクを直接参照し得るパターンマッチの変数に印をつけることで間接参照へ置き換えて，再利用サンクの単一参照性を保つ．変数に印をつけるということは，実装においては，tail 部の間接参照のポインタを他のポインタと区別できるようにすることに相当する．すべてのポインタ参照時に，そのような区別が必要となるため，既存の処理系への実装を考えた場合，すべてのポインタ操作への変更が必要になり，実装コストが大きい．

そこで，実装コストを抑える方法として，方法 2 を検討した．標準関数 `tail` は，コンスセルの tail 部を取る関数として以下のように定義される．

$$
\begin{aligned}
\mathsf{tail} \;=\; & \backslash x \to \mathsf{case}\; x\; \mathsf{of}\; x' \\
& \qquad \mathsf{Nil} \to \mathsf{Nil} \\
& \qquad \mathsf{Cons}\; x_h\; x_t^* \to x_t^*
\end{aligned}
$$

$$
\begin{array}{rcl}
S \in \mathit{Set} & ::= & \phi \mid S,\, x \\
\mathcal{C}_{tail} & :: & \mathit{Expr}^{-} \to \mathit{Set} \to \mathit{Expr} \\
\mathcal{C}_{tail}[\![\backslash x \to e^{-}]\!]\, S & = & \backslash x \to \mathcal{C}_{tail}[\![e^{-}]\!]\, S \\
\mathcal{C}_{tail}[\![\mathsf{Nil}]\!]\, S & = & \mathsf{Nil} \\
\mathcal{C}_{tail}[\![\mathsf{Cons}\ v_h^{-}\ v_t^{-}]\!]\, S & = & \mathsf{Cons}\ (\mathit{replace}\ v_h^{-}\ S)\ (\mathit{replace}\ v_t^{-}\ S) \\
\mathcal{C}_{tail}[\![v^{-}]\!]\, S & = & \mathit{replace}\ v^{-}\ S \\
\mathcal{C}_{tail}[\![e^{-}\ v^{-}]\!]\, S & = & (\mathcal{C}[\![e^{-}]\!]\, S)\ (\mathit{replace}\ v^{-}\ S) \\
\mathcal{C}'_{tail}[\![\mathsf{case}\ e^{-}\ \mathsf{of}\ x\{ & & \mathsf{case}\ (\mathcal{C}_{tail}[\![e^{-}]\!]\, S)\ \mathsf{of}\ x\{ \\
\quad \mathsf{Nil} \to e_n^{-}; & = & \quad \mathsf{Nil} \to \mathcal{C}_{tail}[\![e_n^{-}]\!]\, S; \\
\quad \mathsf{Cons}\ x_h\ x_t \to e_c^{-}\ \}]\!]\, S & & \quad \mathsf{Cons}\ x_h\ x_t^{*} \to \mathsf{let}\ x_t^{*} = \mathsf{tail}\ x\ \mathsf{in}\ \mathcal{C}_{tail}[\![e_c^{-}]\!]\, (S,\, x_t) \\
\mathcal{C}_{tail}[\![\mathsf{let}\ x = e_1^{-}\ \mathsf{in}\ e_2^{-}]\!]\, S & = & \mathsf{let}\ x = \mathcal{C}_{tail}[\![e_1^{-}]\!]\, S\ \mathsf{in}\ \mathcal{C}_{tail}[\![e_2^{-}]\!]\, S
\end{array}
$$

図 5.3: プログラム変換 $\mathcal{C}_{tail}$

方法 2 では，この関数 tail を用いて，図 4.2 のプログラム変換 $\mathcal{C}$ のかわりに，図 5.3 に示すプログラム変換 $\mathcal{C}_{tail}$ を用いる．

図 5.3 のアルゴリズムが図 4.2 のアルゴリズムと異なるのは，case 式で Cons のパターンにマッチするときに，let 式を挿入する箇所である．この let 式を挿入することで，通常サンクを用いて間接参照を実現できる．ヒープのイメージで言えば，図 5.4 のようになる．変換前に再利用サンクを直接参照していた変数 x_t が，変換後には，新たに割り当てた通常サンクを通して参照することとなる．この通常サンクの保持する遅延した式は $\mathsf{tail}\, x$ であるので，値が必要になったときには，x の tail 部の再利用サンクを強制した上で結果を返すこととなる．なお，間接参照への方針が異なるため，変換 $\mathcal{C}$ に代わり変換 $\mathcal{C}_{tail}$ を適用した場合には，4.3 節で議論した操作的意味論に変更が必要となる．

以降，方法 1 と方法 2 のそれぞれについて，GHC への具体的な実装を示した上で，比較を行う．

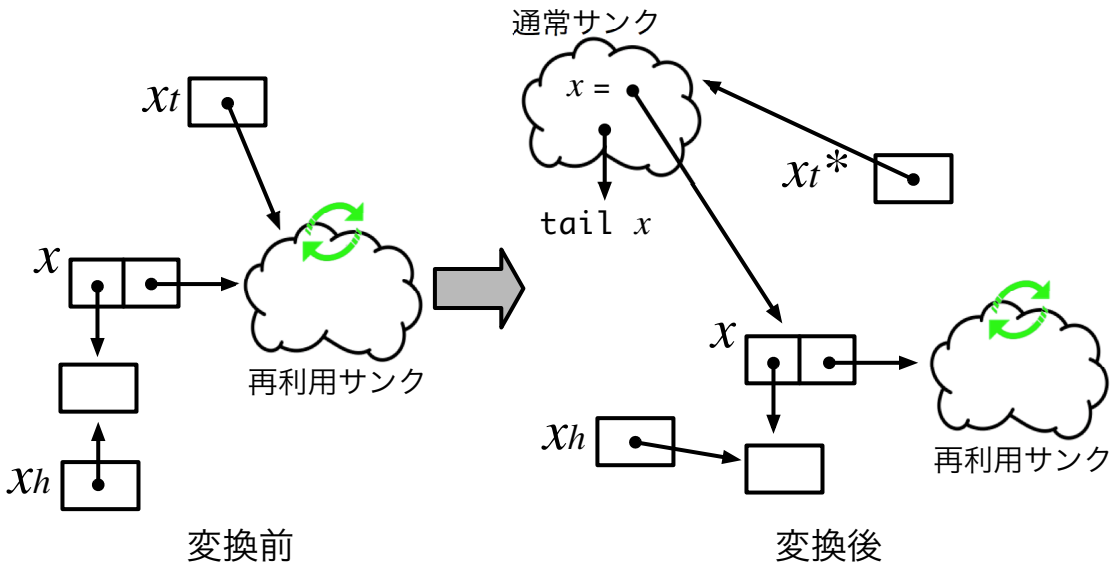

図 5.4: tail による間接参照

5.5.2 方法 1 : フラグの付加による間接参照

ポインタにフラグを付ける手法として，ポインタの未使用ビットに情報を保持する実装テクニックである**ポインタタギング**がある．たとえば，ポインタが 8 バイト境界に揃えられるならば，すべてのポインタの下位 3 ビットは 0 となる．その 3 ビットをタグとして利用し，ポインタが指す先のものを区別することができる．ただし，すべてのポインタの参照において，適宜タグの着脱が必要であるため，実装に必要となるコストは一般には大きい．しかし，GHC のバージョン 6.8.1 以降においては，ポインタタギングを用いてパターンマッチを行う手法 [36] が導入されている．その機構を応用し，Thunk Recycling を実装ができたため，実装コストが大幅に下がった．

GHC のポインタタギングによるパターンマッチは，下位ビットをタグとして，case 式におけるデータ構成子の振り分けを行う手法である．たとえば，以下のような List 型のデータに対する処理を振り分ける Haskell の case 式を考える．

```
case xxs of
  Nil -> ...
  Cons x xs -> ...
```

xxs のデータ構成子によって，処理を分岐することとなるが，遅延評価であるため xxs はサンクの状態である可能性もある．もし，xxs がサンクであれば，そのサンクを強制

```
caseEnt() {
  tag = xxs & 0x7;           // タグを取得
  if (tag == 0) {            // サンクの場合，後で caseRet を実行するよう準備する
    Sp[1] = caseRet;
    Sp++;
    jump thunkcode;          // サンクを強制
  } else {                   // サンクでない場合，データ構成子により振り分ける
    R1 = xxs;
    caseRet();
  }
}
caseRet() {
  tag = R1 & 0x7;            // タグを取得
  if (tag == 1) {            // タグが 1 ならば，Nil の場合の処理を行う
    ...
  } else if (tag == 2) {     // タグが 2 ならば，Cons の場合の処理を行う
    cons = R1 - tag;         // R1 から tag を除去
    x = cons->payload[0];    // head 部
    xs = cons->payload[1];   // tail 部
    ...
  }
}
```

図 5.5: ポインタタギングによるパターンマッチ

し，結果のデータ構造を得たうえで，データ構成子を見て振り分けなければならない．素朴に実装すると，xxs の Info Table を参照して，空リストかコンスセルかサンクかを見分けて処理を分岐することとなる．そのような分岐の手間を軽減するため，データ構成子の情報を，ポインタの下位ビットにもたせる．GHC では，オブジェクトを 32 ビットマシンであれば 4 バイト，64 ビットマシンであれば 8 バイト境界に揃えることとなっているため，0 となっている下位ビットに，そのポインタの参照先がデータ構成子かサンクかという情報を持たせる．たとえば，List 型であれば，サンクを 0，空リストを 1，コンスセルを 2 とタグ付けると決め，オブジェクトの生成時にタグを付け，パターンマッチではタグによる判別を行う．そのようにタグ付けした場合の List 型の処理の振り分けの擬似コードを，図 5.5 に示す．タグは，下位をマスクするビット演算により取得する．この処理は xxs の Info Table を参照するのに比べて非常に軽量であるため，ポインタタ

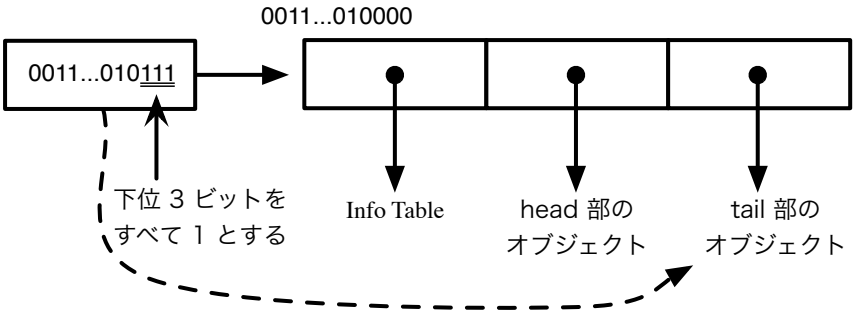

図 5.6: ポインタタギングによる間接参照

ギングを導入すれば，case 式の実行時間の面での恩恵が大きい．

case 式の評価を開始すると，まず，caseEnt に処理を移す．caseEnt は，タグを用いてサンクかどうか判定した後，サンクであればそれを強制する．サンクのコードがどのようなものかは実行時でないと分からないため，サンクを強制する場合には，強制後に caseRet を実行するように，スタックに caseRet のアドレスをあらかじめプッシュしておく．サンクの強制が終了し値が求まると（値は R1 に入っているものとする），caseRet に処理が移る．caseRet では，振り分ける対象のデータ構造が R1 に入っているので，タグに基づき case 式で処理を振り分ける．この擬似コードは，64 ビットマシンを想定しており，タグは下位 3 ビットとして 0x7 によってマスクしている．そのため，データ構成子が 7 つまでのデータ型であれば，ポインタタギングによってパターンマッチの振り分けが可能である．注意が必要なのは，ポインタタギングではすべてのポインタにタグが付くため，GHC 処理系全体に変更が及ぶ点である．たとえば，ごみ集めにおいては，タグを取り除いた上でコピーなどを行い，コピー後に元のタグを付けるなどの処理が必要となる．

以上の GHC のポインタタギングを利用して，方法 1 による間接参照の実装コストを抑える．すなわち，オリジナルの GHC がデータ構成子とサンクを判別するタグをポインタの下位ビットに保持していたのに対して，間接参照であることを表すタグを新たに追加する．具体的には，ポインタの下位 3 ビットをタグ付けに用いた場合，List 型のポインタはサンクを 0，空リストを 1，コンスセルを 2 とタグ付けしていたものに加え，7 であれば間接参照とする．7 でタグ付けされたポインタの指す先にはコンスセルがあり，そのコンスセルを介した tail 部への間接参照を意味するものとする．図 5.6 にその様子を示す．0011...010000 というアドレスにあるコンスセルに対して，図中の破線で示したようにそのコンスセルの tail 部のオブジェクトを間接参照する場合，コンスセルのア

ドレスの下位 3 ビットすべてを 1 としてオブジェクトに保持する（0011...010111）．この間接参照ポインタの指す先の値が必要となった際には，タグを取り除いてタグのないポインタ（0011...010000）を得て，その参照先オブジェクトであるコンスセルのペイロードの二番目（tail 部）を返す．

ポインタのタグは，4.3 節の操作的意味論では，ヒープの場所におけるフラグが T であることに対応する．つまり，図 4.10 の CaseCons において，パターンマッチの束縛を生成する際に，ヒープの場所のフラグを T としていたのに対応し，GHC 上の実装においては，パターンマッチの束縛を生成する際に，該当のコンスセルのアドレスに 7 というタグを加えて間接参照を表現する．また，変数を読み出す箇所で，タグ値 7 の付いた間接参照であるかを判定する（図 4.8 に示した補助関数 $\mathcal{A}$ を使用することに相当）．具体的には，プログラムの実行中に間接参照の値が必要となる箇所に，以下の擬似コードに相当するコードをコンパイル結果として出力する．

```
tag = xxs & 0x7;
if (tag == 7)
  xxs = I64[(xxs - 0x7) + 16];
```

ここで 1 ワードは 8 バイトとし，I64[*addr*] は *addr* の値を 64 ビットのアドレスとして値を参照することを表す．2.5.5 節で述べたように，コンスセルのオブジェクトの表現（ヘッダとペイロードのサイズ）はコンパイル時に決まるため，ヘッダ情報が 1 ワードあるとすれば，I64[(xxs - 0x7) + 16] は，ペイロードの二つ目の値を参照する．つまり，xxs からタグ 0x7 を取り除いたアドレスの 16 バイト先がペイロードの二つ目の値に相当する．参照をたどった先の値にも間接参照のタグが付いている場合があると，間接参照をたどり続ける必要があるが，コンスセルを構成する際に間接参照をたどる処理を入れることで，タグによる間接参照が続くのは高々一つとすることができる．

以上のようにポインタタギングを用いた場合の利点と欠点は，次のとおりである．

利点 1 間接参照をタグで確認するという軽量な操作で実現できる．

利点 2 通常サンクを割り当てることなく間接参照を実現できる．

欠点 1 GHC の広範囲に変更を加える必要がある．

欠点 2 元のポインタタギングで扱える構成子が一つ少なくなる．

欠点 1 について，GHC が元々ポインタタギングをデータ構造のパターンマッチに用いていたことで，ポインタタギングが全くない状態から導入するに比べれば，実装コスト

は軽減されている．欠点 2 について，ポインタタギングによるデータ構成子の振り分け方法が提案されている論文 [36] によれば，構成子の最大数が 7 であったものが 6 になった場合に影響を受けるプログラム数は約 0.9% と少なく，大きな問題とならないと考えられる．

5.5.3 方法 2 : 標準関数 tail による間接参照の実現

方法 2 で間接参照のために挿入する tail の処理を細かく見ると，tail x に必要な操作は以下のようになる．

1. x を弱頭部正規形が得られるまで評価する．
2. 結果の弱頭部正規形の値がコンスセルであるか（データ構成子が Cons であるか）をチェックする．
3. コンスセルであれば，その tail 部（ペイロードの二つ目の値）をとる．
4. その tail 部の値を弱頭部正規形が得られるまで評価する．

ここで，変換 $\mathcal{C}_{tail}$ で考えれば，x はパターンマッチの対象となったリストが Cons とパターンマッチした場合の値であるため，弱頭部正規形であるコンスセルに必ずなっている．したがって，上の手順のうち，1 と 2 は，間接参照に用いる際には実際には必要ない．そこで，以下の方法 2' が考えられる．

方法 2' tail を標準関数ではなく，必要な処理（上の 3 と 4）のみを行うプリミティブ関数として新たに処理を定義する．

この方法に基づき，tail の 3 と 4 だけを行うプリミティブ関数を実装する．3 の操作は，方法 1 の場合と同様にして，コンスセルからオフセットアクセスできるため，以下の擬似コードをインライン展開して埋め込む．ここで，x はコンスセルのアドレスとし，1 ワードを 8 バイトとしている．

```
I64[x + 16];
```

方法 1 の擬似コードと同様に，1 ワードのヘッダ情報にペイロードが続くオブジェクトの表現となっているため，コンスセルのアドレスから 2 ワード（16 バイト）先に tail 部のオブジェクトへの参照がある．この擬似コードは，方法 1 の場合とは，タグを取り除

く必要がない点が異なる．続いて 4 の操作は，参照した結果が通常サンクである場合も考えられるので，実際には上のコードに続いてサンクであれば強制するようなコードが続くこととなる．具体的な実装では，GHC のコード生成部の STG から C−− 言語への変換を記述する関数において，上の tail に相当するプリミティブ関数のコードを生成するように変更する．

以上の処理を実装することにより，方法 2 の効率化をはかることができるが，変換 $\mathcal{C}_{tail}$ には，間接参照に用いる通常サンクが，実際には割り当てる必要がない場合を考慮していないという潜在的な無駄がある．たとえば，プログラム変換 $\mathcal{C}_{tail}$ の case 式の変換で導入する通常サンク ${x_t}^*$ が e_c^- 中で使用されていない，またこれが，正格な文脈に現われれば，通常サンクを割り当てる必要がない．そこで，実際には，$\mathsf{let}\ x' = \mathsf{tail}\ x\ \mathsf{in}$ の挿入場所を変数と置き換えることとなる．たとえば，以下のプログラム

$$
\begin{array}{l}
\mathsf{case}\ e^-\ \mathsf{of}\ x\ \{ \\
\quad \mathsf{Nil} \to e_n^-; \\
\quad \mathsf{Cons}\ x_h\ x_t \to \mathsf{let}\ x_1 = g\ x_t\ \mathsf{in}\ f\ x_1\ \}
\end{array}
$$

に対して変換 $\mathcal{C}_{tail}$ を適用すると

$$
\begin{array}{l}
\mathsf{case}\ e\ \mathsf{of}\ x\ \{ \\
\quad \mathsf{Nil} \to e_n; \\
\quad \mathsf{Cons}\ x_h\ {x_t}^* \to \mathsf{let}\ {x_t}^* = \mathsf{tail}\ x \\
\qquad\qquad\qquad\qquad \mathsf{in\ let}\ x_1 = \ g\ {x_t}^*\ \mathsf{in}\ f\ x_1
\end{array}
$$

となっていた．しかし，これでは，変数 ${x_t}^*$ に束縛される通常サンクを必ず生成してしまう．よって，以下のように変換されるようにしたい．

$$
\begin{array}{l}
\mathsf{case}\ e\ \mathsf{of}\ x\ \{ \\
\quad \mathsf{Nil} \to e_n; \\
\quad \mathsf{Cons}\ x_h\ {x_t}^* \to \mathsf{let}\ x_1 = \mathsf{let}\ {x_t}^* = \mathsf{tail}\ x \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ g\ {x_t}^* \\
\qquad\qquad\qquad\qquad \mathsf{in}\ f\ x_1\ \}
\end{array}
$$

関数 g が引数に関して正格であれば，$\mathsf{tail}\ x$ のための通常サンクは，正格性解析により除去できる．このような変換を新たに変換 $\mathcal{C}'_{tail}$ と呼び，図 5.7 のように定義する．

$$
\begin{array}{rcl}
M \in \mathit{Map} & ::= & \phi \mid M, \langle x_t, x \rangle \\
\mathcal{C}'_{tail} & :: & \mathit{Expr}^- \to \mathit{Map} \to \mathit{Expr} \\
\mathcal{C}'_{tail}[\![\backslash x \to e^-]\!]\ M & = & \backslash x \to \mathcal{C}'_{tail}[\![e^-]\!]\ M \\
\mathcal{C}'_{tail}[\![\mathsf{Nil}]\!]\ M & = & \mathsf{Nil} \\
\mathcal{C}'_{tail}[\![\mathsf{Cons}\ x_h\ x_t]\!]\ (M, \langle x_h, x_1 \rangle, \langle x_t, x_2 \rangle) & = & \mathsf{let}\ x'_1 = \mathsf{tail}\ x_1\ \mathsf{in}\ \mathsf{let}\ x'_2 = \mathsf{tail}\ x_2\ \mathsf{in}\ \mathsf{Cons}\ x'_1\ x'_2 \\
 & & \quad \text{where } x'_1 \text{ and } x'_2 \text{ are fresh variables} \\
\mathcal{C}'_{tail}[\![\mathsf{Cons}\ x_h\ x_t]\!]\ (M, \langle x_h, x \rangle) & = & \mathsf{let}\ x' = \mathsf{tail}\ x\ \mathsf{in}\ \mathsf{Cons}\ x'\ x_t \\
 & & \quad \text{where } x' \text{ is a fresh variable} \\
\mathcal{C}'_{tail}[\![\mathsf{Cons}\ x_h\ x_t]\!]\ (M, \langle x_t, x \rangle) & = & \mathsf{let}\ x' = \mathsf{tail}\ x\ \mathsf{in}\ \mathsf{Cons}\ x_h\ x' \\
 & & \quad \text{where } x' \text{ is a fresh variable} \\
\mathcal{C}'_{tail}[\![\mathsf{Cons}\ x_h\ x_t]\!]\ M & = & \mathsf{Cons}\ x_h\ x_t \\
\mathcal{C}'_{tail}[\![x_t]\!]\ (M, \langle x_t, x \rangle) & = & \mathsf{tail}\ x \\
\mathcal{C}'_{tail}[\![x]\!]\ M & = & x \\
\mathcal{C}'_{tail}[\![e^-\ x_t]\!]\ (M, \langle x_t, x \rangle) & = & \mathsf{let}\ x' = \mathsf{tail}\ x\ \mathsf{in}\ (\mathcal{C}'_{tail}[\![e^-]\!]\ M)\ x' \\
 & & \quad \text{where } x' \text{ is a fresh variable} \\
\mathcal{C}'_{tail}[\![e^-\ x]\!]\ M & = & (\mathcal{C}'_{tail}[\![e^-]\!]\ M)\ (\mathcal{C}'_{tail}[\![x]\!]\ M) \\
\mathcal{C}'_{tail}[\![\mathsf{case}\ e^-\ \mathsf{of}\ x\{ & & \mathsf{case}\ (\mathcal{C}'_{tail}[\![e^-]\!]\ M)\ \mathsf{of}\ x\{ \\
\quad \mathsf{Nil} \to e_n^-; & = & \quad \mathsf{Nil} \to \mathcal{C}'_{tail}[\![e_n^-]\!]\ M; \\
\quad \mathsf{Cons}\ x_h\ x_t \to e_c^-\ \}]\!]\ M & & \quad \mathsf{Cons}\ x_h\ x_t \to\ \mathcal{C}'_{tail}[\![e_c^-]\!]\ (M, \langle x_t, x \rangle)\} \\
\mathcal{C}'_{tail}[\![\mathsf{let}\ x = e_1^-\ \mathsf{in}\ e_2^-]\!]\ M & = & \mathsf{let}\ x = \mathcal{C}'_{tail}[\![e_1^-]\!]\ M\ \mathsf{in}\ \mathcal{C}'_{tail}[\![e_2^-]\!]\ M
\end{array}
$$

図 5.7: プログラム変換 $\mathcal{C}'_{tail}$

表 5.1: 単一参照性の実現にかかる変更点

	C・C−−・ヘッダファイル		Haskell	
	ファイル数	行数	ファイル数	行数
方法 1				
ランタイムシステムへの変更	1	68	0	0
プログラム変換 $\mathcal{C}$	0	0	2	60
コード生成部への変更	0	0	11	201
方法 2				
ランタイムシステムへの変更	0	0	6	90
プログラム変換 $\mathcal{C}'_{tail}$	0	0	2	85
コード生成部への変更	0	0	0	0

変換 $\mathcal{C}'_{tail}$ は，変換対象の $Expr^{-}$ と Map を受けとり，$Expr$ を返す．ここで，Map は，$\langle x_t, x\rangle$ の形の変数の組の集合で，変数 x_t が Map に含まれていれば，新たな局所変数を導入して $\mathsf{tail}\, x$ へと置き換える．

方法 2 の利点と欠点は次のとおりである．

利点 1 変数のアクセスに際して処理系全体に渡る処理変更は少ない．

利点 2 GHC の変更箇所は，プリミティブ関数のコード生成部のみである．

欠点 $\mathsf{tail}\, x$ を遅延する通常サンクが必要となる場合がある．

5.5.4　方法 1 と方法 2 の比較

以上より，方法 1 と 2 ともに再利用サンクの単一参照性を実現するための利点と欠点があることが分かった．そこで，実際に GHC に両者を実装し，実装コストと実行効率の面から比較した．

まず，実装のコストについて比較する．実装にかかったファイル数・行数を，C・C−−・ヘッダファイルと，Haskell で記述された部分に分けて集計したものを，表 5.1 に示す．まず，C 言語等で書かれた部分については，方法 1 ではタグを処理するために，ランタイムのごみ集めにかかわる箇所に変更が必要であった．一方で，方法 2 は，変更は必要なかった．次に，Haskell で書かれた部分について，それぞれの変換を最適化パスに挿入

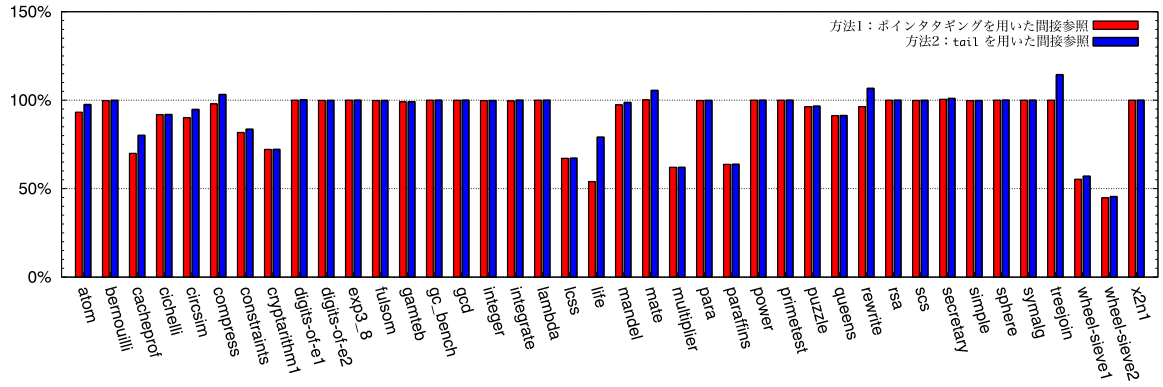

図 5.8: 間接参照の実装法の比較（総メモリ割当量）

するために変更が必要なファイル数・行数は，方法 1 で 60 行，方法 2 で 85 行であり，大きな違いはなかった．しかし，コード生成部への変更は，方法 2 では必要なかったのに対し，方法 1 でのコード生成部への変更は，11 ファイルで 201 行となった．2.5.1 節でも述べたが，GHC は 30 万行と非常に大きな言語処理系であり，その動作を理解し，適切な箇所に間違いなく変更を行うという観点から言うと，11 ファイルというのは，絶対的な数値はそれほど大きくなくても，変更コストは小さくない．なお，C 言語，Haskell での記述の両者において，5.5.2 節で述べたように，GHC がポインタタギングのための機構をすでに導入済みであったことは，方法 1 の実装コストの削減において非常に重要であった．

次に，実行時の総メモリ割当量について比較する．`nofib` ベンチマーク集を用いて，方法 1 と 2 を比較した結果を図 5.8 に示す．実験環境と `nofib` ベンチマーク集の詳細については，6 章を参照されたい．縦軸にオリジナルの GHC を 100 % としたときの割合をとった．全ベンチマークプログラムの平均で方法 1 が 88.9%，方法 2 が 91.2% であり，全ベンチマークプログラムにおいて，方法 1 が方法 2 より少ない総メモリ割当量で済んでいることが分かる．特に，treejoin や rewrite など方法 2 でオリジナルの GHC よりも割当量が増え 100% を越えたベンチマークにおいても，方法 1 を採用することで割当量を 100% 以下に減らすことができた．そのことで，方法 1 には，総メモリ割当量についてオーバヘッドがないことが分かる．方法 1 と方法 2 の総メモリ割当量における違いは，間接参照のために導入した通常サンクによるものである考えられる．

次に，総メモリ割当量の場合と同様，`nofib` ベンチマーク集を用いて，方法 1 と 2 の実行時間を比較したものを図 5.9 に示す．縦軸にオリジナルの GHC を 100 % としたときの割合をとった．平均で方法 1 が 107.8%，方法 2 が 109.6% であった．方法 1（ポイ

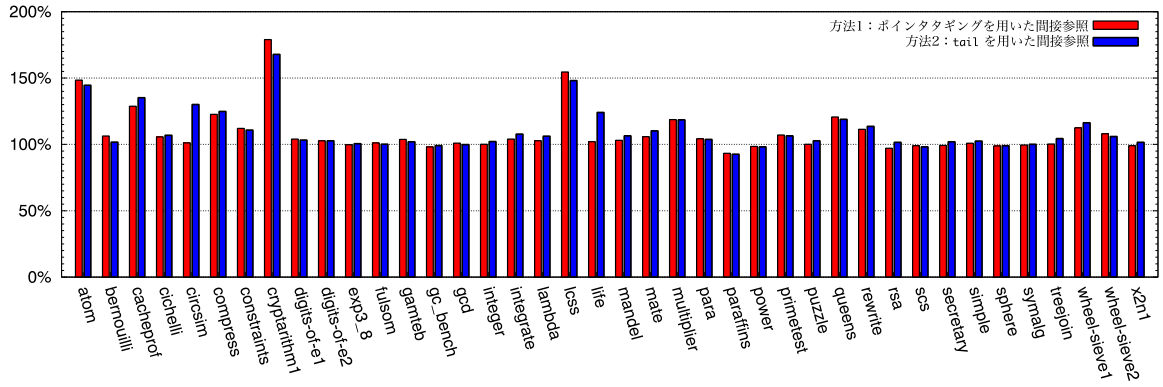

図 5.9: 間接参照の実装法の比較（実行時間）

ンタタギングの導入）によって，circsim や life において実行時間の改善がみられる．これは，間接参照に必要であった通常サンクの削減が，総メモリ割当量の削減につながり，実行時間も改善できたものと考えられる．一方で，方法 1 はすべてのベンチマークに有効なわけでなく，cryptarithm1 や lcss など，実行時間が増加する場合がある．考えられる理由は，間接参照を表すタグをたどる回数を高々 1 回で済ませるため，コンスセルの生成時にタグをたどる処理を入れたことによるものである可能性がある．

以上の結果より，GHC に再利用サンクの間接参照を実現するには，方法 1 を採用するのが妥当であると判断した．それは，実装に要するコストよりも総メモリ割当量と実行時間の結果を重視したためである．特に，総メモリ割当量の結果において，方法 2 ではオリジナルの GHC を越えてメモリ割当てをするプログラムが複数あったことが，方法 2 を採用しなかった最大の理由である．というのは，サンクを再利用することによって総メモリ割当量を減らすことが Thunk Recycling の本来の目的であり，複数のベンチマークで総メモリ割当量が増える場合があるのでは，その目的を完全には達成していないことを意味するからである．もし，ポインタタギングを実現していない他の処理系に Thunk Recycling を実装するのであれば，上で述べたような変更箇所だけでは済まなかったと考えられるため，実装効率の面においては再考する必要がある．

5.6　再利用機構の実現

5.3 節から 5.5 節の検討を踏まえて，再利用機構を次の手順で実現する．

1. 再利用サンクの強制開始に際して，**再利用フレーム**を GHC の実行スタックに積む．
   再利用フレームは，再利用後の処理のコードの Info Table と，その再利用サンクへ

の参照を持つコンスセルからなる．図 5.10 にその様子を示す．

2. 再利用サンクを強制する．強制結果がコンスセルである場合，その tail 部の遅延に再利用サンクを用いることができれば，スタック上の再利用フレームを参照して得た再利用サンクを用いる．リストの背骨を構成するコンスセルの tail 部の遅延のみに再利用サンクの利用を制限するので，強制結果のコンスセルを構成する際には，スタックトップには必ず再利用フレームがある．

3. 式の評価結果を保持するための，GHC が管理するレジスタ領域に強制結果を書き込み，スタックトップの再利用フレームにある再利用後の処理コードを実行する．

4. 再利用後の処理コードは，再利用フレーム中にあるコンスセルの tail 部に，レジスタに書き込まれた強制結果を書き込む．

この手順は，基本的に 4.3 節の操作的意味論と同様である．1 が VarReuseThunk，2 が LerreuseReuse，4 が UpdateTail に相当する．1 における，再利用サンクの強制開始は $*$ が付いた変数を強制することに相当することに注意されたい．つまり，再利用サンクを参照するコンスセルは，間接参照のフラグ（5.5.2 節でいうポインタのタグ）を取り除けば得られる．また，*Expr* に対する操作的意味論が $Expr^{-}$ に対応する操作的意味論に 1 対 1 の対応をしていたことからも分かるように，以上の手順は，元々の GHC が通常サンクを強制する際の手順に似ている．そのため，この手順を GHC に実現するのには，さほど大きな手間はかからなかった．

Thunk Recycling では，コンスセルの tail 部から指される再利用サンクを強制中に，そのコンスセルの後続のコンスセルの tail 部を遅延させるためのサンクを，新たに生成せずに再利用する．ただし，再利用サンクを間接参照オブジェクトに上書きしないので，再利用サンクの参照元が強制結果を直接参照するようにしなければならない．ここで，再利用サンクはコンスセルの tail 部から単一参照されているので，直接参照するように書き換えるのはその tail 部だけである．

図 5.10 のサンクについて，この操作で RT_1 を強制した結果，$Cons_2$ が構成された時のスタックとオブジェクトの様子を，図 5.11 に示す．再利用フレームの実行前には，$Cons_1$ の tail 部は RT_1 を参照していたが，再利用フレームの中に $Cons_1$ へのポインタを積んでおくことで，$Cons_1$ へのポインタをスタックから得ることができ，その tail 部を $Cons_2$ への参照に書き換えることができる．

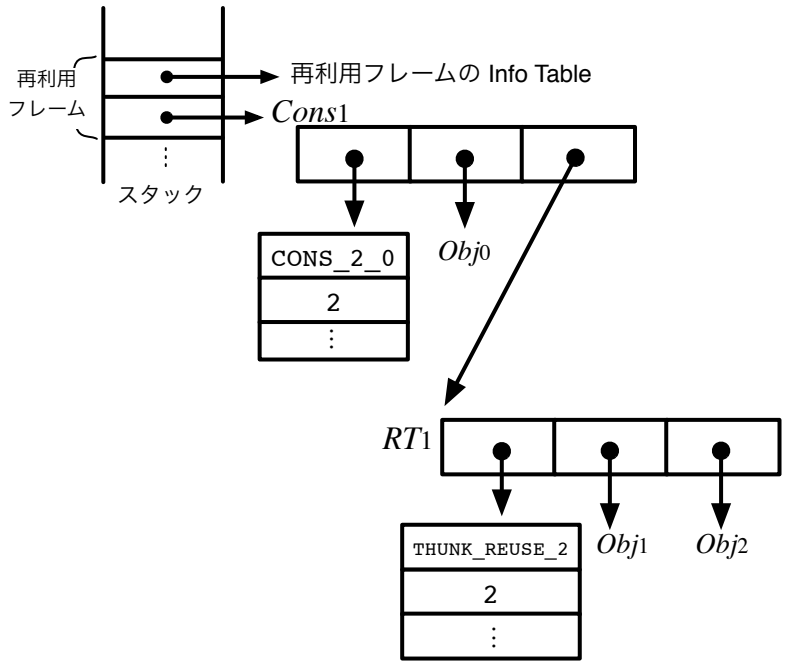

図 5.10: 再利用フレームと再利用サンク

5.7　Thunk Recycling のコード生成

続いて，以下のような式を例に，サンクの再利用処理に対して，どのようなコードがコンパイラより生成されるかを，擬似コードを用いて説明する．

$$f\ x\ y = \mathsf{letreuse}\ x_t = g\ x\ y\ \mathsf{in}\ \mathsf{Cons}\ x\ x_t$$

ここで，x_t への束縛値は，$g\ x\ y$ の評価を遅延する再利用サンクであり，自由変数の x と y はそのサンクの環境として保持される．f に対する関数適用式 $f\ a\ b$ の文脈として考慮すべきは，以下の場合である．

1. 通常サンクによって，$f\ a\ b$ が遅延されている場合
2. 再利用サンクによって，$f\ a\ b$ が遅延されている場合
3. `main` 関数から正格な文脈で呼ばれている場合

まず，(1) の場合は，たとえば

$$\mathsf{let}\ z = f\ a\ b\ \mathsf{in}\ \ldots\ z\ \ldots$$

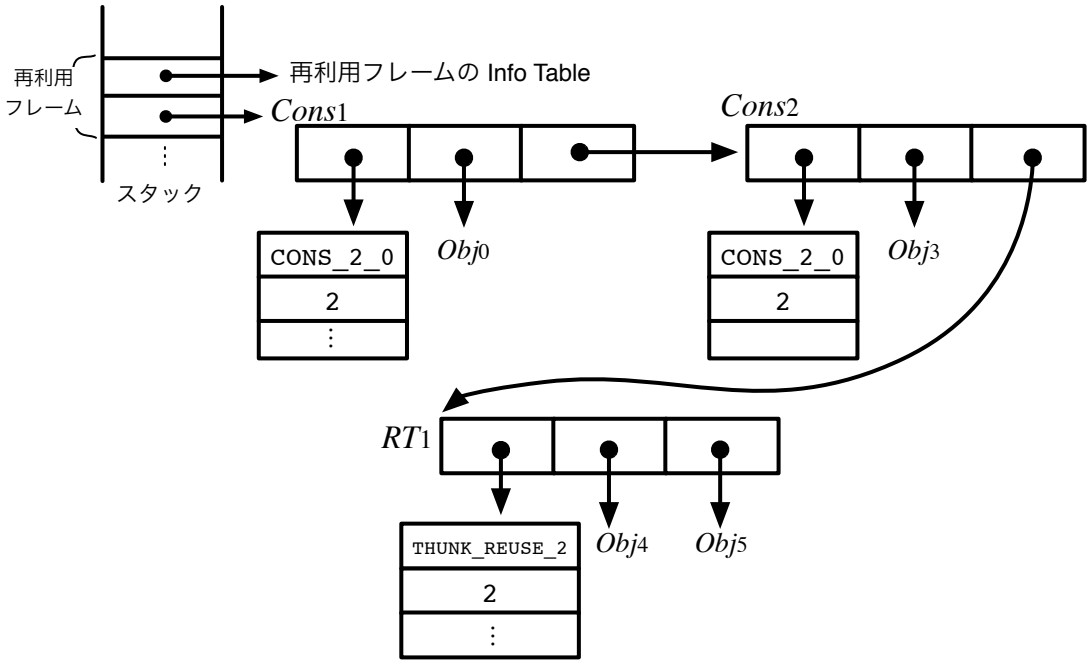

図 5.11: Thunk Recycling を導入済みの GHC 上でのサンクの評価

のようにして f が呼ばれており，$f\ a\ b$ を遅延させる通常サンクは割当てられているが，再利用サンクはまだ割り当てられていない．この通常サンクが強制されれば，図 5.12 (1) のように更新フレームがスタックへと積まれている．

次に，(2) の場合には，既に再利用サンクがヒープ上に確保されている．たとえば，以下のような y_t に束縛されている $f\ a\ b$ を遅延する再利用サンクを強制中ということが考えられる．

$$\mathsf{letreuse}\ y_t = f\ a\ b\ \mathsf{in}\ \mathsf{Cons}\ x\ y_t$$

この例では，強制中の再利用サンクを $g\ x\ y$ のために再利用する．図 5.12 (2) に，擬似コードに入る直前のスタックを示す．前節で述べたように，この再利用サンクの強制後にはスタックに積まれた再利用フレームの Info Table が実行されるようになっている．

また，(3) の場合にも，(1) と同様にまだ再利用可能なサンクが割り当てられていない．

f について，以上の（1）から（3）に対処した擬似コードは，図 5.13 のようになる．まず，`HP+=3` によって，コンスセルのための領域をヒープ上に確保する．次に，現在強制中のサンクが再利用可能であるかチェックする．再利用可能である条件は，以下のとおりである．

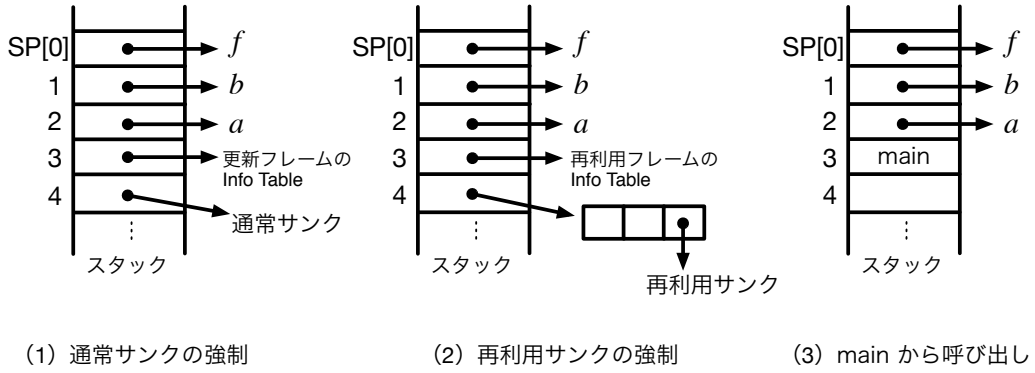

図 5.12: 擬似コードの実行前のスタック

- 再利用サンクがヒープに割り当て済みである．このことは，スタック上にある info table が，再利用後の処理コードの info table である `reuse_frame_info` へのポインタかどうか調べれば分かる．
- 再利用サンクが既にある場合には，そのサンクに今回再利用するために必要な領域がある．サンクのサイズは自由変数の数である．

再利用可能であれば，そのサンクの中を破壊的に更新する．擬似コード中では，`rt` が再利用サンクを参照している．また，再利用可能でなければ，`HP+=3` として，再利用サンクをヒープに新たに割り当てる．最後に，コンスセルの tail 部に再利用サンクを設定する．

5.8　再利用サンクとごみ集め

Thunk Recycling は，従来は使い捨てされていた使用済のサンクを，ある条件の下で次の遅延計算のために再利用することにより，ヒープの消費を抑制するものである．ヒープの消費を抑制することは，ごみ集めの起動回数の減少につながるものの，このことがプログラムの実行時間にどのような影響を及ぼすかは自明ではない．それは，GHC ランタイムのごみ集め機構は複雑であるため，再利用により繰り返し使用されるサンクのような，一般的な短命オブジェクトとは異なる長寿命のオブジェクトがヒープに存在すると，ごみ集めの挙動に影響を及ぼし，ごみ集め起動回数の減少が単純にプログラム実行時間の減少に直結するとは限らないからである．したがって，本節は，再利用サンクとごみ集めについて検討する．

```
// SP: スタックポインタ
// HP: ヒープポインタ
 HP += 3;                        // コンスセルのための領域をヒープに割り当てる
 a = SP[2];                      // f の実引数である a と b への参照はスタック上にある
 b = SP[1];

 if (SP[3] != &reuse_frame_info) { // 再利用サンクがまだ割り当てられていない
   goto ALLOCATE;
 }
 rt = SP[4].payload[1];          // 再利用サンクへの参照を取得する
 SP -= 3;                        // a と b をスタックから取り除く
 if (rt.payload_size < 2) {      // 再利用サンクが小さすぎる
   goto ALLOCATE;
 }

OVERWRITE:                       // 再利用サンクを用いる
 rt[0] = info table for g x y // 内容を上書きする
 rt[1] = a;
 rt[2] = b;
 tailobj = &rt[0];               // 再利用サンクへの参照を tailobj として保存する
 goto DONE;

ALLOCATE:                        // 再利用サンクを割り当てる
 HP += 3;                        // 割り当てる再利用サンクのサイズは 3 である
 HP[-5] = info table for g x y  // 再利用サンクの内容を書く
 HP[-4] = a;
 HP[-3] = b;
 tailobj = &HP[-5];              // 再利用サンクへの参照を tailobj として保存する

DONE:                            // コンスセルを構成する
 HP[-2] = info table of cons cell;
 HP[-1] = a;                     // head 部に a を指定する
 HP[0]  = tailobj;               // tail 部に再利用サンクを指定する
```

図 5.13: letreuse 式のコンパイル済みコード

5.8.1　再利用サンクを置く世代の検討

通常サンクは式の評価を遅延するために作られるので，強制された後の通常サンクはやがてごみ集めにより回収されるべきである．一方，Thunk Recycling では，再利用サンクを強制するとそのサンクを後続計算を遅延するのに用いる．つまり，ある再利用サンクがごみになる瞬間をとらえ，瞬時にそれを回収し再利用しているという見方もできる．再利用サンクは使い回されるため，その生存期間は長くなる傾向にあるという点で，通常サンクと異なるふるまいをする．そのような再利用サンクの性質を考えると，ごみ集めに関して通常サンクとは別の配慮をする必要がある．

再利用サンクの生存期間は長くなる傾向にある一方で，そこに保持される環境は再利用のたびに頻繁に書き換えられるため，書き換えられて再利用サンクから指されなくなった古い環境は，参照がなくなってごみとなる可能性が高い．以上のように再利用サンクそのものは長寿命だが，そこから指される環境中のオブジェクトは必ずしもそうではないことをふまえて，GHC で採用されている世代別ごみ集めにおける再利用サンクの世代の調整法を考察する．

再利用サンクを置く世代について，次の選択肢が考えられる．

- 常に旧世代領域に置く．
- 常に新世代領域に置く．
- 置く世代を固定しない．

まず，再利用サンクを常に旧世代領域に置く場合について考える．再利用サンクは，再利用されながらごみとならずに生き続けるであろうという予想があるためである．しかし，GHC では保持する参照の書き換えが起こらないことを前提としており，2.5.7 節で述べたように，旧世代にある再利用サンクから参照されている環境中のオブジェクトも再利用サンク本体と合わせて旧世代領域へ即座に昇格してしまうことに注意しなくてはならない．昇格した後，この環境はごみになっても，旧世代に置かれているため，次のメジャーごみ集めまで回収されない．そのため，旧世代に環境のごみが蓄積し，メジャーごみ集めに負担がかかる恐れがあるという問題がある．

図 5.14 に，その様子を示す．旧世代にある再利用サンク RT_1 から参照されるオブジェクトは旧世代に昇格してしまう．その後，再利用によって RT_1 が Obj_2 を参照するようになり，Obj_1 がごみになっても，メジャーごみ集めが起こるまで Obj_1 は回収されない．

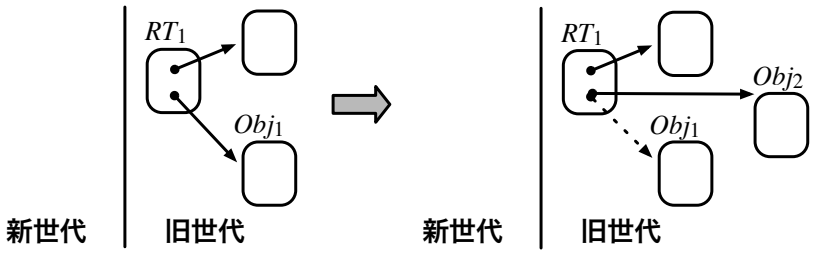

図 5.14: 再利用サンクを常に旧世代に配置する

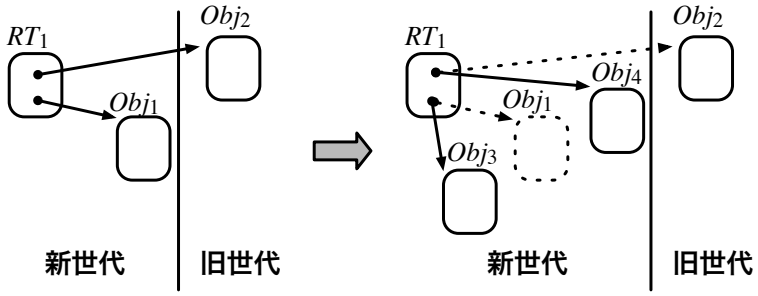

図 5.15: 再利用サンクを常に新世代に配置する

経験的に，再利用の頻度はメジャーごみ集めの頻度より高いため，旧世代が圧迫される可能性が高まる．

次に，再利用サンクを常に新世代に置く場合について考える．このようにすることで，再利用サンクの環境がごみとなると，マイナーごみ集めで即座に回収できる可能性が高くなるというメリットが生じる．その一方で，再利用サンクの数が多い場合，新世代の領域を圧迫し続けてしまうという問題がある．さらに，世代別ごみ集めにおいて頻度の多いマイナーごみ集めで，長寿命の再利用サンクが毎度走査の対象となってしまう．

図 5.15 に再利用サンクを常に新世代に配置する場合のイメージを示す．図 5.14 の場合と異なり，再利用サンク RT_1 が保持している Obj_1 への参照が Obj_3 へと変わり，Obj_1 がごみになっても，新世代を対象とするマイナーごみ集めによって Obj_1 を回収できる．一方で，長生きする可能性が高い再利用サンク RT_1 が，常にマイナーごみ集めにおける走査の対象となってしまう．

置く世代を固定しない場合には，通常のオブジェクトと同様にし，長生きの再利用サンクであれば自然に昇格し，旧世代に置かれることとなる．通常のオブジェクトと同じに扱うならば，環境は再利用サンクと同じ世代に置かれる．

以上より，次の 2 点について考察し，世代別ごみ集めにおける再利用サンクの配置方

表 5.2: 再利用サンクとその環境を配置する世代

	環境の世代を再利用サンクに	
再利用サンク	必ず合わせる	無理には合わせない
旧世代	方法 1	方法 2
新世代	方法 3	方法 4
固定しない	方法 5	方法 6

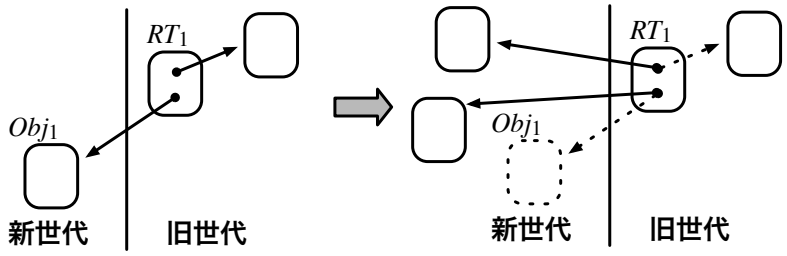

図 5.16: 世代を考慮した再利用サンクの配置

法を考える.

- 再利用サンク本体をどの世代に置くか.
- 再利用サンクの環境から参照されるオブジェクトを配置する世代を，再利用サンクと合わせるべきか.

取り得るすべて列挙すると，表 5.2 に示す 6 通りの選択肢がある．たとえば，方法 3 は以下のことを意味する.

> 再利用サンク本体は新世代に固定して置き，そこから参照されている環境中のオブジェクトも新世代に置く.

この方法 3 を取った場合，実際には，新世代が再利用サンクで埋まってしまう場合を配慮しなければならず，新世代に固定して置く再利用サンクの数を限定するなど工夫が必要である.

方法 2 を採れば，図 5.16 のように再利用サンク RT_1 は旧世代に配置しつつ，そこから参照されるオブジェクトを無理に旧世代に昇格するようなことはしない.

どの方法を採用するのが妥当であるかは，実際にプログラムを実行することにより，比較しなくては決定が難しい．それは，サンクの生成や強制のされ方は実行するプログラ

ムにより大きく変わってくるためであり，指標となるような研究はこれまでなされていない．しかし，GHC は元々コピー式ごみ集めを採用しているため，比較的容易に，オブジェクトの世代を調整し，各方法を比較することができる．

まず，再利用サンクのコピー先の世代を調整する実装については，ごみ集め時のコピー先を固定して指定すればよい．常に新世代に再利用サンクを置くならば，再利用サンクのコピー先を必ず新世代とする．同様にして，常に旧世代に再利用サンクを置くならば，再利用サンクのコピー先を必ず旧世代とする．固定しない場合には，通常のオブジェクトと同様に，新世代にある再利用サンクが一度ごみ集めの対象となれば旧世代へとコピーする．

次に，再利用サンクとその再利用サンクが保持する環境中のオブジェクトの世代を合わせるかについて，必ず合わせる場合には，2.5.7 節で述べたように，通常のオブジェクトと同じに扱えばよいので，元々の GHC の動作に変更を加える必要がない．無理には合わせない場合については，ペイロードが書き換えられる可能性のあるオブジェクト（Mutable Array など）に対するごみ集めを参考に実装できる．具体的には，ごみ集め時に再利用サンクが環境に保持するオブジェクトを Scavenge する際に，旧世代への移動を止めるように，旧世代への移動を制御するフラグ（`eager_promotion`）を False にする．そのフラグに基づいて，Evacuation 時にオブジェクトを無理に旧世代へ動かさないような判定が GHC に用意されているので，世代を参照元のオブジェクトと揃えるかどうかを切り換えるのは容易である．

以上の世代の調整にかかる変更点を数えたところ，C 言語で書かれた 4 つのファイルに対して，合計 16 行の変更がなされただけであった．

実装した各方法について，次節でベンチマークプログラムによる実験結果を示し比較検討する．

5.8.2 各方法の比較

再利用サンクとその環境を置く世代に関して，5.8.1 節で議論した各方法を比較する．比較のため，`nofib` ベンチマーク集を用いる．実験環境と `nofib` ベンチマーク集の詳細については，後述の 6 章を参照されたい．本節のすべての表において表中の値は，再利用なしのオリジナルの GHC の場合を 100% としたときの比である．

表 5.3 に，総実行時間について，全ベンチマークの平均，最大，最小を示す．100% を下回る値であれば，Thunk Recycling により実行時間が減少したことを示す．最大と最小

表 5.3: 各方法における実行時間の比較

	総実行時間		
	平均	最大	最小
方法 1	106.6%	143.5% atom	90.1% paraffins
方法 2	107.0%	158.3% lcss	89.3% paraffins
方法 3	106.1%	135.6% atom	86.0% life
方法 4	106.3%	144.1% atom	85.9% life
方法 5	104.0%	132.8% atom	89.3% paraffins
方法 6	103.5%	127.0% atom	89.4% paraffins

の値については，その値となったベンチマーク名を表中に載せた．世代を調整する以外の項目については，後述する 3 つの改善手法を適用したものとなっている．5.8.1 節で述べたように，新世代に再利用サンクを残す場合（方法 3 と 4）では，その数を一定個数に限定しないと，新世代が再利用サンクで埋まってしまう可能性がある．そのため，この実験では，新世代に残す再利用サンクは 1,000 個までとした．また，multiplier は，特定の世代に固定することで大きな影響を受けるベンチマークであるため，集計から除いた．詳細は，後述する．

各方法を比べると，方法 6（再利用サンクの世代を固定せず，その環境の世代を合せない場合）が最大で 127% となり，他の方法に比べて最悪の場合の速度低下が抑えられている．また，最小値をとるベンチマークは，方法 3 と 4 が他の方法と比べて低い値となった．方法 3 と 4 で最小になる life について，図には載せていないが，方法 5 で 101.2%，方法 6 で 100.7% となっている．全ベンチマークの平均をとった値は，再利用サンクの世代を固定せずに，通常のオブジェクトと同じように扱う方法 5 と 6 が他の方法に比べて低くなった．表 5.3 の値から判断すると，方法 1 〜 4 に比べて，方法 5 と 6 が有望である．

表 5.4: multiplier ベンチマーク

	実行時間		コピー総量	マイナーごみ集め回数
	全体	ごみ集め		
方法 1	115.7%	113.1%	66.9%	55.1%
方法 2	10,066.9%	17,311.5%	5,740.9%	67.9%
方法 3	106.8%	100.6%	67.4%	59.5%
方法 4	7,483.2%	12,834.1%	4,171.4%	67.9%
方法 5	87.5%	74.7%	74.4%	67.9%
方法 6	89.9%	79.4%	74.4%	67.9%

表 5.3 における集計から除いた multiplier の詳細を表 5.4 に示す．方法 2 と 4 において，オリジナルの GHC に対して実行時間がそれぞれ 100 倍と 75 倍にもなってしまう．これは，ごみ集め中のコピー総量が，異常に増え（方法 2 が 57 倍，方法 4 が 42 倍）ごみ集め時間がかかる（方法 2 が 173 倍，方法 4 が 128 倍）ことと関係している．しかし，どちらの方法を採った場合でも，オリジナルの GHC に対して 67.9% となり，マイナーごみ集め回数に増加はみられない．このことから，環境の世代を無理には合わせないことで，再利用サンクと合わせてすぐに昇格させてしまえばよかったオブジェクトがマイナーごみ集めの対象となり，コピーされてしまったことが想定される．元々の GHC はこれを想定して，通常サンクの環境をそのサンク自体の世代に合わせることをしていた．つまり，このベンチマークでは，再利用サンクを通常サンクと同様に扱い，参照元のオブジェクトと同じ世代に環境を配置するのが望ましいと分かる．

以上のことより，世代を固定し，環境を再利用サンクと無理には合わせない場合（方法 2 と 4）の multiplier の実行時間の低下は，許容範囲を明らかに逸脱しており，これらの方法を採用することは妥当ではないと判断できる．

ここまでの結果より，どの方法を採用するかは，表 5.3 の値からすれば，平均を重視すれば方法 5 か 6 がよさそうである．方法 3 は最小の値は他の方法と比較して低くなるが，平均をみると，方法 5 や 6 には及ばず，新世代に固定することは効果のあるベンチマークを選ぶようである．また，方法 3 で新世代に固定する再利用サンクの個数を 1,000 個としたが，これは予備的な実験に従ったものであり，本来はその個数もベンチマークごとに適切に変化させるのが望ましい．ただ，再利用サンクの割当て具合を静的に解析する方法は明らかでないため，現状では方法 5 と 6 に劣っていると言わざるをえない．

表 5.5: atom ベンチマーク

	ごみ集め時間	ごみ集め回数	
		マイナー	メジャー
方法 5	146.3%	92.4%	98.5%
方法 6	137.9%	94.9%	81.1%

次に，方法 5 と 6 を比較して，再利用サンクの世代を固定しない場合に，その環境を同じ世代に合わせるか検討する．表 5.3 に示した平均ではほぼ同じであるが，方法 6 のほうが若干実行時間が改善する．さらに，両方の方法で最大値をとる atom においては，表 5.5 に示すように，メジャーごみ集めの回数に差が見られた．方法 5 では，環境の世代を合わせることによって，旧世代に昇格したあと再利用サンクから参照されなくなったオブジェクトが発生し，メジャーごみ集めの起動回数が増えたものと考えられる．

以上の結果より，再利用サンクと環境の世代に関して，本論文の最終的な結論は，方法 6 に基づく以下のとおりとする．

> 再利用サンクを置く世代は固定せず，その再利用サンクの保持する環境の世代を再利用サンクと無理に合わせることはしない．

以後の実験で世代の調整をする場合は，方法 6 を採用したものの結果を示す．

5.9　実行効率の改善

5.5.4 節と 5.8.2 節の実験結果からも分かるように，Thunk Recycling を適用したプログラムの実行時間については，平均の値で見ると改善の余地があった．そこで，Thunk Recycling の実行効率の改善を目指して，3 つの改善手法を実装した．

5.9.1　再利用が起こる関数の特化

5.4 節で述べたように，letreuse では，再利用サンクがすでに存在し，かつ再利用するのに十分な大きさがあるかを，実行時にチェックする必要がある．しかし，そのようなチェックの一部は，静的な変換によって取り除くことができる．

再利用サンクが割り当て済みとの場合，特化した関数を用意することで，実行時のチェックを除去する．たとえば，enumFromTo について，返り値のコンスセルの tail 部にくる再帰呼び出しを特化した，以下のような定義を考える．

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo n m | m < n     = []
               | otherwise = n : enumFromTo' (n+1) m

enumFromTo' :: Int -> Int -> [Int]
enumFromTo' n m | m < n     = []
                | otherwise = n : enumFromTo' (n+1) m
```

ここで enumFromTo' は enumFromTo だけから呼ばれ，他の関数から直接呼ばれることはないものとする．関数 enumFromTo が最初に呼ばれたときに，新たに構成するコンスセルの tail 部の式を遅延させるのに使える再利用サンクがすでにあるかどうかは一般には分からないので，再利用サンクが生成済みかどうかのチェックが必要である．しかし，enumFromTo' の呼び出しに変更した二回目以降の tail 部での再帰呼び出しに関しては，enumFromTo' は enumFromTo から呼ばれているので，再利用サンクがすでに生成されていることが保証されている．つまり，enumFromTo' でコンスセルを構成する箇所で，再利用サンクが割り当て済みかどうかチェックせずに済ませることができる．そのようなチェックを省いた関数を用意するために，enumFromTo' のような関数の複製を自動的に用意する変換のことを，以後**関数特化**と呼ぶ．

変換 $\mathcal{P}$ によって letreuse 式が導入された関数 f に対して，関数特化を行い，関数定義を複製する．

$$
\begin{aligned}
f \ldots &= \ldots \,\mathsf{letreuse}\, xs \;=\; f' \ldots \,\mathsf{in}\, \mathsf{Cons}\ \_\ xs \\
f' \ldots &= \ldots \,\mathsf{letreuse}\, xs \;=\; f' \ldots \,\mathsf{in}\, \mathsf{Cons}\ \_\ xs
\end{aligned}
$$

ここの関数特化は，簡単のために上のような tail 部で自己再帰する関数とした．

5.4 節で述べたように変換 $\mathcal{P}$ は STG 言語上で実装するため，関数特化も STG 言語上で行う．STG 言語は部分式の自由変数の情報を保持したり，変数名を一意にしているため，この変換の実装にかかるコストは大きい．GHC の Core 言語はこのような関数の追加を容易に行えるように設計されているが，関数特化は再利用サンクが特定された段階で行う必要があるため，STG 言語の上に実装せざるを得なかった．そのため，変数名の

付け替えなどの処理を GHC 既存の処理を用いて実装することができず，自前で用意する必要があった．

5.9.2　ブラックホーリング

letreuse 式によって再利用サンクが強制された時，5.6 節で説明した方法により，その再利用サンクを再利用する．ここで再利用の対象は，再利用サンク本体の領域であり，再利用サンクがペイロードに保持している環境は，再利用時に破壊的に上書きされる．もしも（上書きされる）環境がこのサンクからしか指されていないならば，いずれ上書きされる不要なオブジェクトへの参照を再利用サンク内に保持していることになり，メモリリークの原因になる．たとえば，図 5.17 のように，再利用フレームとともにスタックに積まれたコンスセルから参照され，これから再利用されるサンク RT_1 が，Obj_1 と Obj_2 を環境中に保持するものとする．その後再利用が起こって RT_1 が Obj_3 や Obj_4 を参照するようになるとこれらは不要になる．そのため，もし Obj_1（Obj_2）を指すポインタが RT_1 だけにあるとすれば，メモリリークになる．

このようなメモリリークは，Thunk Recycling のない環境でも起こりうる問題として知られている．通常サンクの場合，あるサンクが繰り返し強制されるのを防ぐために，強制が終わるまで強制中のサンクへの参照を保持するためである．

この問題に対して，通常サンクでは，サンクの**ブラックホーリング** [20] という手法が有効であることが分かっており，GHC に実装されている．ブラックホーリングとは，強制中のサンクに印を付けておくことで，ごみ集めはサンクがペイロードに保持している参照（サンクに局所的な環境）をたどらずに済ませるというものである．よって，このサンクだけからしか参照されないオブジェクトはこれによりごみとして回収される．

通常サンクの場合と同様にして，再利用サンクに関するメモリリークもブラックホーリングにより解消できる．図 5.17 であれば，RT_1 をスタックに積む際に，RT_1 に印を付けてごみ集め時に Obj_1 と Obj_2 をたどらないようにする．

図 5.18 に再利用サンクのブラックホーリングの様子を示す．スタックに積む再利用サンクに対して，ブラックホーリング済みであること示す Info Table に書き換え，ブラックホーリングを実現する．その Info Table に指定するオブジェクトの種類は，ブラックホーリングを示すもの（`BH_THUNK_REUSE_`n，ここで n はペイロードの大きさ) であり，コードは空の命令列とする．コードが空でいいのは，再利用サンクのコードが再び実行されることはないためである．サイズは，ごみ集めに必要であるため，ブラックホーリン

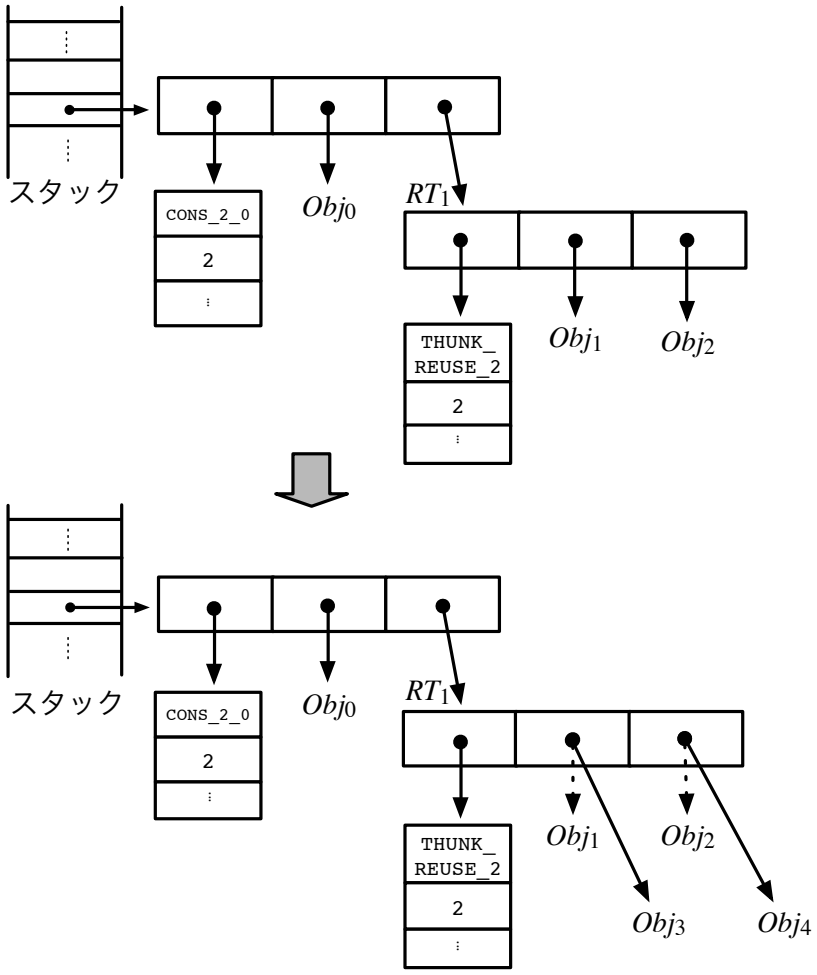

図 5.17: 再利用サンクのメモリリーク

グする再利用サンクと同様にする．ある再利用サンクに対して，どのブラックホーリング済み再利用サンク Info Table を用意すべきかは，コンパイル時に決定できるため，共通の Info Table をランタイムシステムに用意して利用する．なお，サイズに関しては，通常サンクの場合と同様に，オブジェクトの種類に情報を含めて，BH_THUNK_REUSE_1 などとすることで，効率化を図る．Info Table はコードが空であるため決まった大きさ（64 bit の Linux 環境で 88）となり，それぞれのサイズに対応した Info Table は，メモリ上に等間隔に連続的に配置できる．等間隔に連続で配置されていることにより，あるサイズの再利用サンクをブラックホーリングする際に指定すべき Info Table は，サイズ 1 の場合の（BH_THUNK_REUSE_1 というオブジェクトの種類を持つ）Info Table からのオフセットで指定できる．なお，現在はサイズが 10 である BH_THUNK_REUSE_10 まで用意し，それ以上のサイズのサンクに対してはブラックホーリングをしないようにしている．ベン

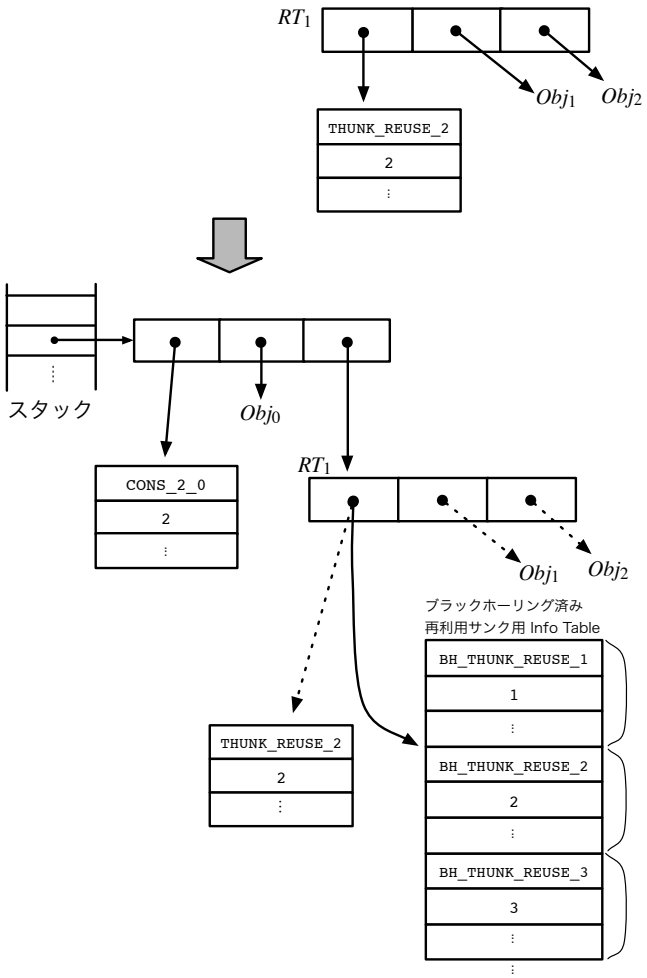

図 5.18: 再利用サンクのブラックホーリング

チマークプログラムの実行によると，大きい再利用サンクは必要なく，実用上はこれで問題ないと考えている．

ごみ集め時には，ブラックホーリング済みの再利用サンク本体を Evacuation の対象とするが，ペイロードに保持する環境への参照は Scavange の対象としないことにより，メモリリークを防ぐ．

5.9.3　間接参照に伴うメモリリークの除去

3.3 節で述べたように，Thunk Recycling は，再利用サンクへの参照をコンスセルへの間接参照で代用することで単一参照性を実現する．この方法は，複雑な解析をすることなく単一参照性が実現できるという利点があるものの，メモリリークを引き起こす可能

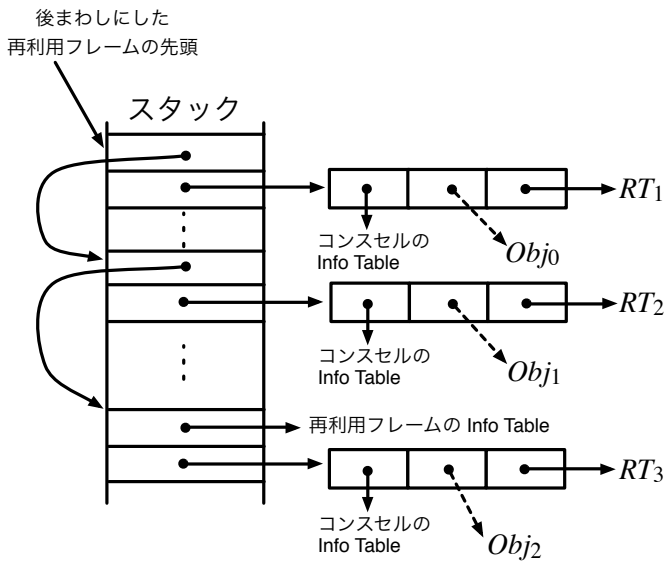

図 5.19: 後まわしにしたスタック領域の記録

性を生んでしまう．つまり，間接参照のみに用いられるコンスセルがある場合，そのコンスセルの head 部は本来は必要な参照ではない．5.6 節の図 5.10 であれば，Obj_0 への参照は，メモリリークを引き起こす可能性がある．これは，5.5 節で述べたポインタタギングを用いたとしても存在する問題であり，これを解決するためにはごみ集め時に間接参照のためのコンスセルを適切に扱わなくてはならない．

まず，tail 部に強制中の再利用サンクを持つコンスセルについては，再利用フレームをスタックに積むという処理によりスタックに必ずそのようなコンスセルへの参照があることに注意すれば，ごみ集め時のスタック走査においてこれを特別に扱うことで，上の問題を解決できる．つまり，スタックに積まれた参照は必ず生きているオブジェクトへの参照であるため優先して走査するのが通常の動作であるが，再利用フレームに積まれたコンスセルへの参照は，その tail 部だけ Evacuation し，後の操作のため，図 5.19 に示すように再利用フレームを数珠つなぎにしておく．再利用フレームは図 5.10 で示したように，再利用後の処理のコードの Info Table と，再利用サンクへの参照を持つコンスセルへの参照からなる．再利用後の処理コードの Info Table はすべて共通であるため，数珠つなぎのため一時的に利用する．

それ以外のすべてのオブジェクトに対する Scavenge が終わったら，後まわしにしたコンスセルが置かれているスタックの位置すべてに対して，以下の手順によって処理する．

1. スタックのその位置にある参照が，コピー式ごみ集めのフォワーディングポインタ

（コピー先の領域を指すポインタ）に変更されていれば，フォワーディングポインタをたどった先のアドレスを書き込む．この場合には，その位置が参照していたコンスセルは他からも参照されていたため，head 部に関するメモリリークは起きないものであったことになる．

2. 上の条件に当てはまらなければ，スタックからしか参照されないコンスセルであると判断できるので，コンスセルのみを Evacuation し，head 部は Scavenge しない．tail 部の再利用サンクは，先に Evacuation し，Scavenge してあるので，ここでは何もしない．

3. 対象となった再利用フレームの Info Table を書き戻し，数珠つなぎされた次の位置に同様のことを繰り返す．

以上のようにして，メモリリークを取り除いた．

なお，5.5 節で述べたポインタタギングを導入すれば，間接参照に用いるコンスセルへのポインタには，それを識別するタグが付けられることとなる．そのことで，上のスタックに積まれたコンスセルと同様にして，間接参照のタグが付いたコンスセルに関するごみ集め処理を後まわしにすれば，head 部のメモリリークの問題を解決することができる．ただし，ヒープ内にあるコンスセルへの参照の場合には，後回しにする領域の集合を効率良く実現するための実装コストは大きい．そのため，このようなコンスセルの head 部のメモリリークを防ぐ手法は，現状では実装していない．

5.9.4　実験

本節で示した 3 つの実行効率改善手法の効果を確認するため，`nofib` ベンチマーク集を用いた実験を行った．実験環境と `nofib` ベンチマークの詳細については，6 章を参照されたい．

それぞれの手法を次のとおり表記する．

改善手法なし 5.5 節の方法 1（ポインタタギングを用いた間接参照）を採用し，5.8.2 節の結果に基づいて 5.8.1 節の方法 6（再利用サンクを置く世代は固定せず，そのサンクの保持する環境の世代を再利用サンクと無理には合わせない）を採用した．

関数特化 5.9.1 節の関数特化による手法のみを適用したもの．

ブラックホーリング 5.9.2 節で述べたブラックホーリングを行ったもの．

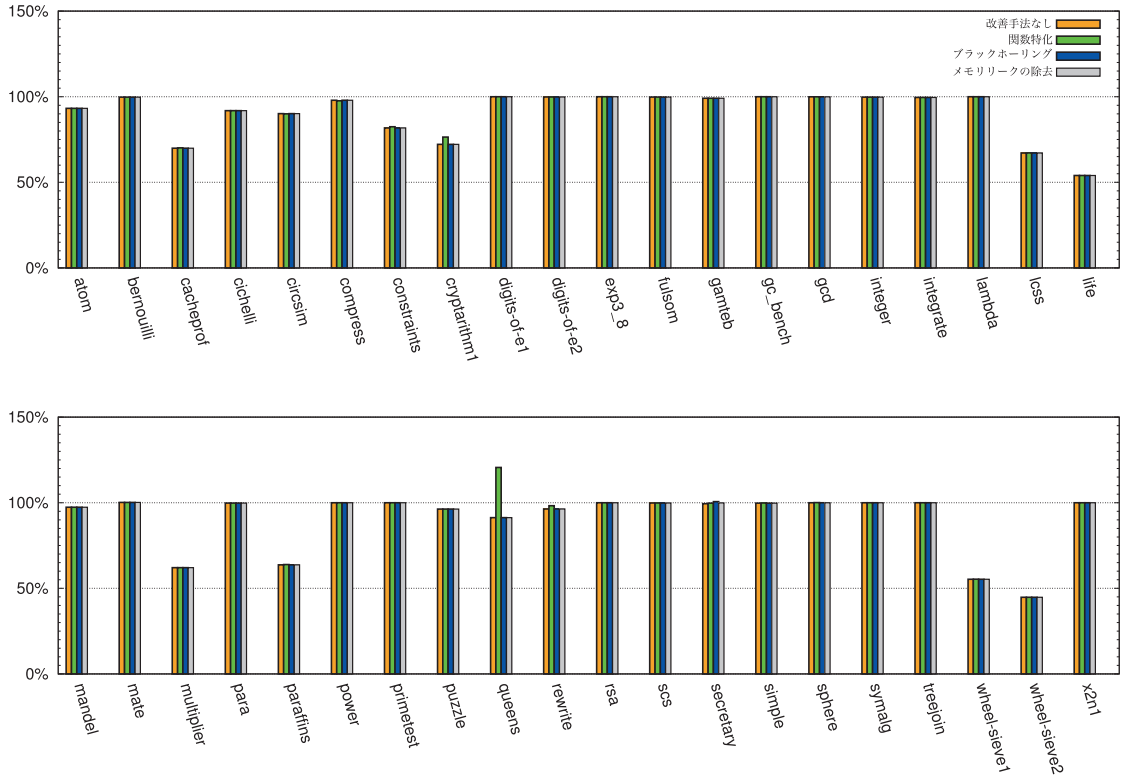

図 5.20: 実行効率改善手法を適用した場合の総メモリ割当量

メモリリークの除去 5.9.3 節で述べた間接参照のみに用いるコンスセルのメモリリークを除去したもの.

本節では，これら 3 つの改善手法をそれぞれ適用し，総メモリ割当量と実行時間について，オリジナルの GHC と比較する．3 つの改善手法を同時に適用した場合の実験結果は，6 章で示す.

総メモリ割当量の実験結果を図 5.20 に示す．縦軸に Thunk Recycling なしのオリジナルの GHC の計測結果を 100% としたときの比をとった．多くのベンチマークにおいて，改善手法なしの場合と 3 つの改善手法をそれぞれ適用した場合の総メモリ割当量と同じである．ただし，queens や cryptairithm1 などのベンチマークでは関数特化によりメモリ割当量が増加した．これは現在の関数特化においては，5.9.1 節で述べたパターンに該当する関数を，内部関数まで複製しているため，関数の複製による増加が起きてしまったと考えられる．そのため，関数特化のアルゴリズムには，改善の余地がある．全ベンチマークの平均をとると，次のようになる.

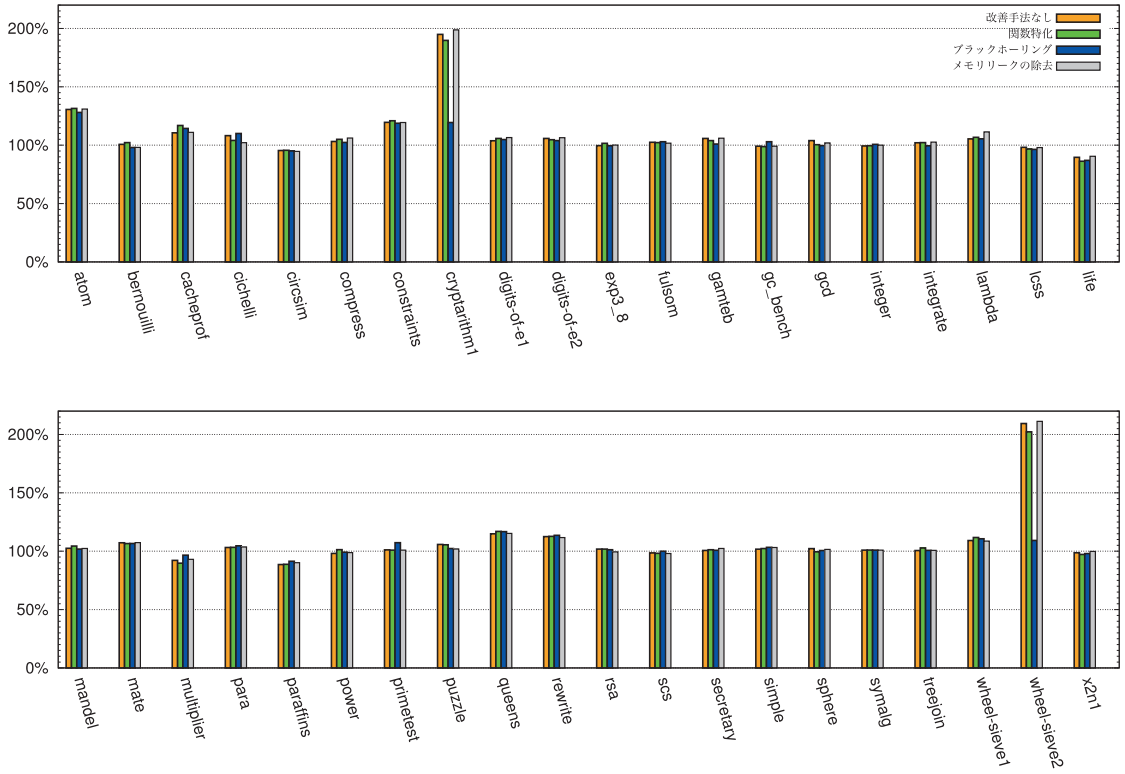

図 5.21: 実行効率改善手法を適用した場合の実行時間

- 改善手法なし： 86.8%
- 関数特化： 87.6%
- ブラックホーリング： 87.1%
- メモリリークの除去： 86.8%

3 つの改善手法は実行時間の改善を目指したものであり，メモリ割当量に関する影響は少ないことが分かる．

次に，実行時間の測定結果を図 5.21 に示す．メモリ割当量の場合と同様に，縦軸に Thunk Recycling なしのオリジナルの GHC の場合を 100% としたときの比をとった．そのため，縦軸について，100% を下回れば，実行時間に関して再利用による効果がみられたことを表し，100% を上回れば再利用によるオーバヘッドによってオリジナルの GHC に比べて速度が低下したことを表す．

まず，ブラックホーリングによる実行時間の削減が顕著である．たとえば，cryptarithm1については，改善手法なしの場合に 195.0% だったものに対して，ブラックホーリングを適用することで，119.4% と，まだオーバヘッドはあるものの実行時間を大幅に削減できた．細かく見ると，このベンチマークは，再利用サンクのメモリリークが起こっており，オリジナルの GHC と比べてごみ集め中のコピーの総量が，オリジナルの GHC に対して 5,757.6 % と大幅に増加していた．このメモリリークをブラックホーリングによって解消したことで，ごみ集め中のコピー総量をオリジナルの GHC に対して 95.8% と大幅に減少させることができた．なお，ブラックホーリングを導入するだけで，全ベンチマークにおけるごみ集め中のコピー総量の平均は，オリジナルの GHC に対して 100.4% となる．

関数特化とメモリリークの除去については，効果のあるベンチマークもあるが，悪影響があるベンチマークもある結果となった．全ベンチマークの平均は，次のようになる．

- 改善手法なし： 106.5%
- 関数特化： 106.5%
- ブラックホーリング： 103.6%
- メモリリークの除去： 106.7%

なお，論文 [58] における実験では，論文 [57] で採用した設計方針を基準として議論を行っていた．そのため，5.5 節の検討における方法 2 と 5.8 節における方法 3 を採用していたため，本論文の傾向とは異なる結果となっている．つまり，本節で示した 3 つの改善手法による効果は，再利用の設計方針にも大きく影響される．さらに，プログラムのふるまいごとに適切な組み合わせが異なると予想されるため，その組み合わせの決定方法は自明ではなく，今後の課題である．

5.10 変更規模

Thunk Recycling の導入に必要であった GHC への変更点をまとめると，表 5.6 のようになる．変更点をファイル数・行数として示す目的は，GHC に対してどの程度の変更が必要であるか示すことにある．

ランタイムシステムは，C・C−−・ヘッダファイルを変更し，コンパイラは Haskell コードに記述を加えた．なお，並列化 GHC に関する部分と新コード生成器[2] に関する変更

[2]以降のバージョンでのコード生成の刷新のための準備として，GHC 7.0 で試験的に導入されたコード生成器である．

表 5.6: Thunk Recycling の導入に必要となる GHC の変更点

	C・C−−・ヘッダファイル		Haskell	
	ファイル数	行数	ファイル数	行数
プログラム変換 $\mathcal{P}$（5.4 節）	0	0	2	171
ごみ集め（5.8 節）	4	175	0	0
コード生成（5.7 節）	3	86	23	457
再利用機構 (5.6 節)	11	650	1	21
関数特化（5.9.1 節）	0	0	4	454
ブラックホーリング（5.9.2 節）	16	387	2	27
メモリリークの除去（5.9.3 節）	8	215	0	0
その他	15	132	8	169
合計	のべ 62	1,729	のべ 61	1,735

点は，再利用機構に必要な変更を加えたが，表の数値からは除いて集計した．

各項目に関して見ると，以下のとおりである．

プログラム変換 $\mathcal{P}$ 4.2.2 節で述べた再利用サンクの特定に関するプログラム変換に必要であった変更点である．変換を行う Haskell プログラムと，その変換を STG-to-STG のフェーズに組み込む部分のみで，変更は 2 ファイルのみである．

ごみ集め ごみ集めで，Evacuation や Scavenge の際にオブジェクトの種類によって行う処理に再利用サンクを追加したことによる変更である．オブジェクトの種類による振り分けなど，定型的な処理の追加が主たる部分であり，本論文では説明しなかった．なお，世代の調整，ブラックホーリング，間接参照に伴うメモリリークの除去に関しては，別項目として集計を行っている．

コード生成 プログラム変換の結果を受けて，再利用を行うコードを生成するための変更点である．主に，STG 言語から C−− 言語に変換する箇所の Haskell プログラムに変更を加えた．GHC の実行モデルに従ったコードを生成する必要があるため，Code Generation フェーズの既存のコードの多くの部分を参照する必要があり，変更にコストがかかる項目であった．作業としてはコストがかかったものの，設計上の選択が必要となることはなかったため，生成されるコードの詳細については本論文では説明を行っていない．

再利用機構 5.6 節の実装に必要であった，再利用フレームなど，ランタイムシステムに用意した組み込み処理に関する変更点である．その他に，本論文では詳しく述べなかったが，再利用サンクのサイズなどによって特殊化された組み込み処理を用意したため，比較的変更量が大きくなった．それらの組み込み処理は，主に C−− で記述した．

ごみ集めにおける世代の考慮 5.8.1 節で考察した，再利用サンクを置く世代に関する変更点である．GHC の既存のごみ集めについて，世代の実現法など多くの調査が必要であったが，書き換え可能配列の実現のために GHC が元々用意していた機構を利用できるため，変更点は非常に少なくて済んだ．

関数特化 5.9.1 節で述べた，再利用時のチェックを省くための手法に関する変更点である．STG 言語上で，ユーザが定義した関数を複製する処理が必要なため，STG-to-STG の変換においてコード量が大きくなった．

ブラックホーリング 5.9.2 節で述べた強制中のサンクに対するブラックホーリングに関する変更点である．ランタイムシステムにおいては，ごみ集めでのブラックホーリングされた再利用サンクを扱うための処理，コンパイラには再利用サンクの強制開始時にブラックホーリングするコードの生成などに，変更が必要であった．

メモリリークの除去 5.9.3 節で述べた間接参照のみに用いるコンスセルのメモリリークを除去するためのごみ集めに対する変更点である．スタックをたどるタイミングなどの知識が必要であり，GHC のごみ集め全体の流れを理解する必要があった．

その他 既存のデバッグ・プロファイル機構に対する変更や，コンパイル時に指定するコマンドラインのフラグなどに関する変更である．

すべての項目に共通で言えるのは，変更を加える箇所や変更方法を決めるためには，GHC のコードの理解が必要であり，ソースツリーの多くの部分に目を通す必要があったことである．今回の変更でいえば，主にコンパイラのコードがある `compiler` 以下とランタイムシステムの実装 `rts` 以下の理解が不可欠であった．なお，`compiler` 以下だけでも 400 個程度の Haskell ソースコードファイルがあり，計 20 万行以上もある．そのため，変更が必要なファイル数・行数の絶対的な量が少なくても，Thunk Recycling の実現に必要となる全体のコストが低いわけではない．

5.11　本章のまとめ

本章では，GHC 上の Thunk Recycling の実装について，詳細に述べた．特に，再利用サンクのオブジェクトの表現方法（5.3 節），再利用対象となるサンクの選定方法（5.4 節），単一参照の実現方法（5.5 節），再利用サンクのごみ集め時のふるまい（5.8 節）においては，複数の実装方針を比較し，実装コスト，総メモリ割当て量，実行時間などの面から比較し，採るべき方針を取捨選択した．再利用サンクのオブジェクトの表現については，通常サンクの表現と同一とし，再利用サンクとその再利用サンクを tail 部で参照するコンスセルの対応関係をスタックを用いるのがよいという結論に至った．再利用するサンクの選定方法は，式の返り値となるコンスセルについて，tail 部の遅延に用いるサンクを再利用対象とするのがよいという結論に至った．単一参照の実現には，ポインタの表現を変更し，tail 部を間接参照することを示すフラグをポインタに持つようにするのがよいという結論に至った．ごみ集めについては，GHC が世代別コピー式ごみ集めを採用しているため，再利用サンクを置く世代は固定せず，その再利用サンクの保持する環境の世代を再利用サンクと無理に合わせることはしない．

Thunk Recycling を適用したプログラムの実行時間には改善の余地があったため，3 つの改善手法を実装し，その効果を確認した．3 つの改善手法とは，再利用が起こる関数の特化（5.9.1 節），ブラックホーリング（5.9.2 節），間接参照に伴うメモリリークの除去（5.9.3 節）である．これらの改善手法のうち，ブラックホーリングによる実行時間の改善が顕著であった．

第6章

GHC 上の Thunk Recycling の実装の評価

本章では，GHC 上の Thunk Recycling の実装を評価するため行ったベンチマークプログラムを用いた実験について述べる．まず，再利用機構の効果をみるため，これまで例で用いてきた関数 `enumFromTo` と関数 `sum` を組み合わせた呼び出しによるマイクロベンチマークの結果を述べる．次に，Haskell 処理系の評価に広く用いられている `nofib` ベンチマークの結果を述べる．

6.1 実験環境

実験は，Intel Core i7-3770 CPU（3.4 GHz, キャッシュ 8MB），メインメモリ 8 GB の PC 上で動作している Linux カーネル 3.11 上で行い，コンパイル時には，GHC の `-O2` オプションを付けることにより，GHC の正格性解析を行ったものとなっている．Thunk Recycling による再利用がある場合には，5.9 節で述べた，3 つの実行効率改善手法をすべて適用したものとなっている．

各結果の計測には，以下のものを用いた．

GHC の実行時オプション `-S`： GHC において，コンパイルしたプログラムの実行完了時の情報を得るためのオプションである．今回は，以下の値を計測するのに用いた．

- 総メモリ割当て量
- ごみ集め中のコピー総量

- 生存しているオブジェクトが占めるバイト数（以下，生存オブジェクト量）
- OS から確保したメモリ量（以下，フットプリント）
- メジャーごみ集めの回数
- マイナーごみ集めの回数
- 総実行時間
- ごみ集めにかかった時間（以下，ごみ集め時間）
- 計算にかかった時間（以下，計算時間）

生存オブジェクト量は，メジャーごみ集め後に全ヒープ中のオブジェクトから計算される．ごみ集め時間については，メジャーごみ集めとマイナーごみ集めの両者にかかった時間を合計する．計算時間とは，プログラムの初期化処理にかかった時間とごみ集め時間を除く実行時間を言う．

GHC の Tikcy-ticky プロファイラ： GHC に組み込まれているプロファイラである．中間言語 STG の段階においてカウンタを挿入し，関数ごとに呼び出し回数やメモリ割当て量などを計測できる．今回は，強制されたサンクの総数を測定した．さらに，Thunk Recycling を適用した場合には，強制されたサンクの数を通常サンクと再利用サンクに分けて計測し，再利用サンクについては，割当てられた総数と再利用が起きた回数も計測した．

Linux のパフォーマンスカウンタ（`perf` コマンド）： `perf` コマンドは，引数として実行したプログラムのハードウェアとソフトウェアの両面での情報を取得できる Linux カーネル組み込みのコマンドである．今回は，以下の値を計測するのに用いた．

- プログラム終了までの総 CPU サイクル数
- キャッシュミスの回数・割合
- 分岐予測ミスの回数・割合

上の各項目について，GHC のランタイムシステムもしくは，ベンチマークのコンパイルコードのどの処理中に起きたかを詳細に取得できる．

表 6.1: マイクロベンチマーク: `sum (enumFromTo 0 1000000)`

	オリジナルの GHC	再利用ありの GHC	再利用ありの GHC / オリジナルの GHC
総メモリ割当量（キロバイト）	170,061	137,689	81.0 %
生存オブジェクト量（キロバイト）	38,109	38,082	100.0 %
フットプリント（メガバイト）	66	66	100.0 %
総実行時間（秒）	1.538	1.233	80.2 %
ごみ集め時間（秒）	0.880	0.707	80.3 %
計算時間（秒）	0.657	0.525	80.0 %
マイナーごみ集めの回数（回）	254	192	75.6 %
メジャーごみ集めの回数（回）	7	7	100.0 %
強制された通常サンク（個）	1,000,007	6	-
強制された再利用サンク（個）	-	1,000,001	-
再利用の回数（回）	-	1,000,000	-
総 CPU サイクル数	6,056,321,519	4,830,730,396	79.8 %
キャッシュミスの割合（%）	57.60	54.33	94.3 %
分岐予測ミスの割合（%）	0.01	0.01	100.0 %

6.2 マイクロベンチマーク

2.1 節で用いた関数 `enumFromTo` と関数 `sum` を組み合わせた以下の呼び出しを用いて，Thunk Recycling の効果を確認した．

```
sum (enumFromTo 0 1000000)
```

この呼び出しは，`0` から `1000000` までを要素とするリストを `enumFromTo` で生成しながら，そのリストの全要素の和を `sum` で求める（結果は `500000500000`）．再利用のある場合には，関数 `enumFromTo` で生成するリストの後続の遅延には再利用サンクを用い，関数 `sum` のパターンマッチにおいて tail 部への間接参照が必要となる．

表 6.1 に結果を示す．まず，総メモリ割当量について，Thunk Reycling を適用したことで，81.0% で実行でき，再利用の効果が見られた．ただし，このベンチマークプログラムでは，リストの最後尾にサンクが生成されるのみであるので，生存するサンクの数が変わるわけではない．そのことは，生存オブジェクト量とフットプリントが，オリジナルの GHC の値と Thunk Recycling を適用した場合の値には違いがなかったことから確認できる．

総実行時間は，Thunk Recycling を適用したことで 80.2 % となった．計算時間における減少について，perf コマンドを用いて，関数呼び出しごとに総 CPU サイクル数を分析したところ，threadPaused という関数における変化が大きかった．この threadPaused には，オリジナルの GHC で 825,887,643 サイクル，Thunk Recycling を適用すると 602,405,520 サイクルかかっている．Thunk Recycling を適用して実行した場合に，オリジナルの GHC に対して全体として 72.9 % の CPU サイクルで実行できたことが分かった．threadPaused は，GHC ランタイムで定義されている関数で，ごみ集めの準備として，スタックを走査する．今回のベンチマークにおいて，sum はスタックを消費するように定義されているため，ごみ集めが始まる際に長いスタックを走査することは，オーバヘッドとなる．Thunk Recycling を適用した場合，マイナーごみ集めの回数が減らせるため，ごみ集めの準備としてスタックを走査する処理の総数が減らせ，オリジナルの GHC に比べて実行時間が短くなったと考えられる．なお，threadPaused は，Evacuation と Scavenge というごみ集めの主流の処理ではないという判断から，GHC ではごみ集め時間に含まずに計算時間に含めている．また，ごみ集め時間の減少については，生存しているオブジェクトは変わらないものの，起動回数が減ったことによる効果であったと考えられる．

このベンチマークにおいては，キャッシュミスと分岐予測ミスとも，オリジナルの GHC の値と Thunk Recycling を適用した場合とで大きな違いはなかった．

6.3　nofib ベンチマーク集

nofib ベンチマーク集 [39] は，広い分野における典型的な Haskell プログラムを集めたベンチマーク集である．様々な作者によるプログラムで構成されており，一般のプログラムに想定される動作を網羅しているため，Haskell 処理系の性能測定で広く用いられている．プログラムの規模・内容によって，real，spectral，imaginary という 3 種類に以下のように分類される．

real 現実に用いられているプログラムを元にしており，規模が大きいプログラムが多い．たとえば，Haskell プログラムの構文解析を行う parser やテキストを圧縮する compress などがある．

spectral 規模としては中程度で，現実のプログラムである real の核となる要素を抽出したものを集めている．たとえば，マンデルブロー集合を求める mandel やライフゲームを実装した life などがある．

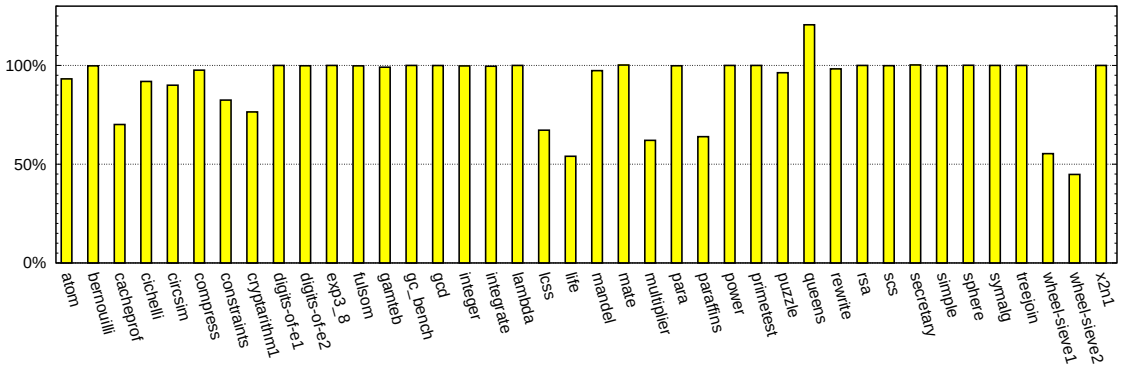

図 6.1: 総メモリ割当て量

imaginary 非常に単純な関数で構成されたプログラムであり，主に処理系のテストやマイクロベンチマークとして用いられる．たとえば，素数判定をする wheel-sieve や N クイーン問題を解く queens などがある．

今回は，nofib ベンチマーク集の 76 個のプログラムのうち，実行時間が 0.05 秒以上となる 40 個のベンチマークについて実験結果を示す．それぞれのベンチマークに対して，総メモリ割当て量，ごみ集め回数，実行時間の結果を示し，それらの平均の値を示す．

6.3.1 総メモリ割当て

GHC の実行時オプション -S によって計測した総メモリ割当て量の実験結果を図 6.1 に示す．queens においては，5.9.4 節で示した実験のように，関数特化を採用したことによる悪影響で，オリジナルの GHC に対しても 100% を超えてしまった．しかし，queens を除けば，オリジナルの GHC に対して総メモリ割当て量を削減でき，最大で wheel-sieve2 の 44.8% と半分以下の割当て量で実行できた．

全ベンチマークの平均をとると，87.6% であった．洗練された実装を持つ GHC において，正格性解析を行った上で，これだけの削減が可能である意義は大きい．

6.3.2 ごみ集め

Thunk Recycling の総メモリ割当て量の削減の効果は，ごみ集め回数の削減という形で現われる．図 6.2 にマイナーごみ集め回数，図 6.3 にメジャーごみ集め回数の結果を

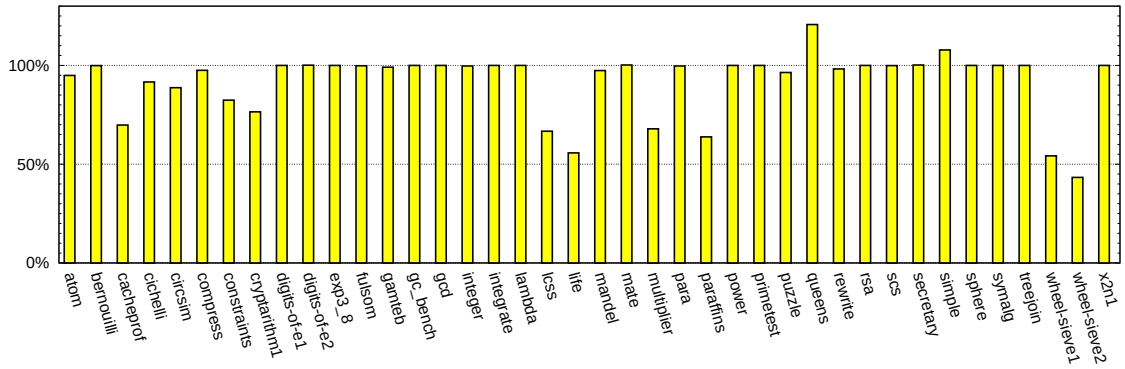

図 6.2: マイナーごみ集め回数

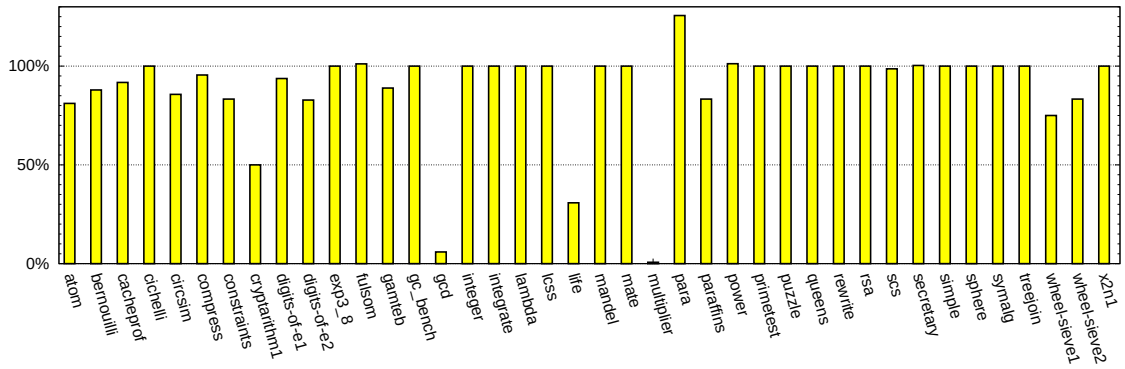

図 6.3: メジャーごみ集め回数

示す.

マイナーごみ集めは，ヒープへのオブジェクト割当て回数を元に起動されるため，図 6.1 に示した総メモリ割当て量と同じ傾向となった.

一方で，メジャーごみ集めは生きているオブジェクトの量に応じて起動されるため，Thunk Recycling によりサンクの割当てを減らせたからといって，その回数が同じように減るわけではない．特に multiplier や gcd で回数の大きな減少が見られたが，これは，5.8.1 で検討した方法 6 において，サンクとそのサンクの環境を揃えないようにしたことが大きいと考えられる．サンクの環境を無駄に昇格させないことで，生きていると判定されるオブジェクトを減らしたことが，メジャーごみ集め回数の減少につながったと考えられる.

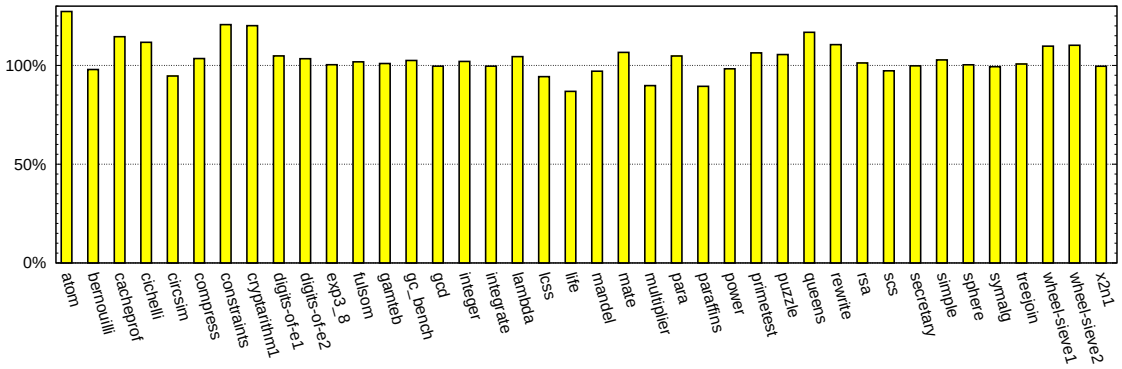

図 6.4: 実行時間

6.3.3 実行時間

GHC の実行時オプション -S によって計測した実行時間を図 6.4 に示す．全ベンチマークの平均をとると，103.1% であった．

ベンチマーク毎にみると，life や multiplier などのベンチマークで 10 % 以上の実行時間の削減がみられた．その一方で，atom や constraints では 20 % 以上，実行時間が増加してしまった．これらの結果より，実行時間に関しては，Thunk Recycling は適用するプログラムを選ぶ必要があると分かった．Thunk Recycling の効果のあったベンチマークと，悪化したベンチマークについて，6.3.4 節で詳細に分析する．

ごみ集めの回数の減少がそのまま実行時間の減少に直結するとは限らない．たとえば，wheel-sieve1 と wheel-sieve2 は，マイナーごみ集めとメジャーごみ集めのいずれの回数も減っているにもかかわらず，実行時間が増加してしまう結果となった．

Thunk Recycling によって，総メモリ割当て量に削減がみられるにもかかわらず，実行時間はオリジナルの GHC に対して平均で 3 % 程度のオーバヘッドがあることは，GHC のサンクの割当て処理とごみ集めが元々高速であるためである．また，総メモリ割当てを削減することによって，OS から確保するメモリ総量の削減につながっていないことも，速度の改善に至っていない理由である．なお，OS から確保するメモリ総量の削減に関して，全ベンチマークの平均をとると，97.4% となり，メモリ確保にかかる実行時間に Thunk Recycling の影響は少ないと考えられる．

実行時間におけるオーバヘッドの改善は今後の課題である．

表 6.2: nofib ベンチマークの分析：ごみ集め

		ごみ集め時間	コピーされた総バイト数	ごみ集め回数		ごみ集め時間 / 総実行時間
				マイナー	メジャー	
改善	life	51.5 %	54.2 %	55.7 %	30.1 %	47.8 %
	multiplier	79.2 %	74.4 %	67.9 %	0.7 %	57.9 %
	paraffins	84.9 %	84.2 %	63.8 %	83.3 %	87.4 %
悪化	atom	138.1 %	118.6 %	94.9 %	81.1 %	54.5 %
	constraints	126.5 %	118.4 %	82.4 %	83.3 %	65.5 %
	cryptarithm1	91.6 %	103.2 %	76.5 %	50.0 %	3.9%

6.3.4　ベンチマークプログラムの分析

本節は，実行時間が最も改善した 3 つのベンチマーク life，multiplier，paraffins と，実行時間が最も悪化した 3 つのベンチマーク atom，constraints，cryptarithm1 について，詳細に分析する．

ごみ集めに関する計測結果について，オリジナルの GHC を 100 % としたときの値を表 6.2 に示す．一番右の列の “ごみ集め時間/総実行時間” とは，オリジナルの GHC で，そのベンチマークにおけるごみ集めにかかる時間の割合を示す．まず，Thunk Recycling の効果があったものは，ごみ集め時間の減少が大きい．それは，マイナーごみ集め回数が減った事により，ごみ集め中にコピーされた総バイト数の削減につながったためであると考えられる．効果のあった 3 つのプログラムでは，コピーされた総バイト数がそれぞれ 54.2%，74.7%，84.2% となっており，ごみ集め中の総コピーバイト数が抑えられた．マイクロベンチマークでも述べたように，生存する tail 部のサンクは Thunk Recycling の有無で変化がないため，総バイト数を抑えられたことは，マイナーごみ集め回数の減少によるものであると考えられる．なお，multiplier のメジャーごみ集めは，オリジナルの GHC で 142 回のメジャーごみ集めを行なっていたのに対し，Thunk Recycling を適用した場合には 1 回のメジャーごみ集めのみで実行が完了しており，このことが実行時間の削減に貢献していた．また，Thunk Recycling の効果が見られたプログラムは，総実行時間に占めるごみ集めの割合が大きいベンチマークであった．これらは，実行時間の約半分以上がごみ集めにかかるようなベンチマークである．しかし，実行時間が悪化した atom と constraints を見ると，ごみ集めが実行時間に占める割合が大きいプログラムであっても必ずしも Thunk Recycling の効果がみられるわけではない．

一方で，悪化した 3 つのプログラムについて見ると，atom と constraints ではごみ集

表 6.3: nofib ベンチマークの分析：再利用

		再利用サンクの個数（個）	総再利用回数（回）	1 個あたりの再利用回数（回）	フットプリント（MB）（再利用ありの GHC / 再利用なしの GHC）
改善	life	67,381	68,587	1.1	2 （100 %）
	multiplier	38,829	5,965,025	153.6	2 （100 %）
	paraffins	34,166	4,012,592	117.4	266 （75.4%）
悪化	atom	5,990,386	2,994,016	0.5	3 （100 %）
	constraints	735,204	1,828,772	2.5	160 （125 %）
	cryptarithm1	7,280,383	14,310,644	2.0	2 （100 %）

め中の総コピーバイト数の増加が，実行時間の増加につながっていた．表には載せなかったが，どちらのプログラムにおいても生存オブジェクト量の増加がみられた（atom では 115.8 %，132.7 %）．考えられる原因は，tail 部での間接参照にともなって，生存オブジェクト量が増加したことである．5.9.3 節で除去した head 部のメモリリークは，現在強制中の再利用サンクを tail 部に持つコンスセルのものだけが除去の対象であった．そのため，tail 部に再利用サンクを持つようなすべてのコンスセルにおいてメモリリークを除去する必要がある．なお，cryptarithm1 は，ほとんどごみ集めが起こらないベンチマークであり，Thunk Recycling による効果がみられなかった．

次に，再利用の観点から，表 6.3 に再利用サンクの個数，総再利用回数，再利用ありの場合のフットプリントの値を示す．まず，multiplier と paraffins について，割当てられた再利用サンクを多くの回数再利用して用いていると分かった．特に paraffins については，フットプリントの削減にもつながっている．実行時間が悪化したベンチマークでは再利用サンク 1 個あたりの再利用回数が少なく，ごみ集めの時間の結果と合わせると，ごみ集め時のコピーの増加に伴う Thunk Recycling のオーバヘッドを上回るだけの再利用回数がなかったと考えられる．

定性的にみると，multiplier は乗算器の動作をシミュレートするプログラムであり，数値をリストとして表現している．また，paraffins は，パラフィンの構造解析をするプログラムであり，基（radical）のデータ構造のリストを生成し，解析している．いずれの場合も，リストを使用するプログラムということは判別できるが，どれだけ長いリストを生成するかは，プログラムを精読しないと判別できない．

結果をまとめると，以下の条件のプログラムでは，Thunk Recycling の効果が現れる傾向にある．

- 再利用が多く起きるプログラム
- プログラム全体に占めるごみ集めの割合が大きいプログラム
- 間接参照にともなう head 部でのメモリリークが発生しないプログラム

再利用サンクの効果を大きくするためには，これらの傾向があるプログラムに対してのみ Thunk Recycling を適用するのが望ましい．しかし，遅延評価において，これらの傾向があるプログラムを静的に解析することは難しい問題である．Thunk Recycling をプログラムの性質に合わせて選択的に適用することは，今後の課題である．

6.4　本章のまとめ

本章では，GHC 上の Thunk Recycling の実装をベンチマークプログラムにより評価した．

まず，2.1 節で用いた関数 `enumFromTo` 関数と関数 `sum` を組み合わせたマイクロベンチマーク用いて，総メモリ割当量や総実行時間などの種々の値を計測した．このベンチマークでは，Thunk Recycling を適用したことにより，総メモリ割当量と総実行時間の両者ともオリジナルの GHC に対して約 80% で済むことが確認できた．Thunk Recycling を適用しても生存するオブジェクトの量は変わらないため，このベンチマークにおける実行時間における削減は，ごみ集めの準備時にスタックを走査する処理の削減によるものと分かった．

マイクロベンチマークに加えて，一般のプログラムを想定した `nofib` ベンチマーク集を用いた実験を行った．ほとんどすべてのベンチマークプログラムで総メモリ割当量を削減することができ，オリジナルの GHC に対して，全ベンチマークプログラムの平均で約 87 % で済むことが確認できた．実験は正格性解析と併用したものであり，洗練された実装を持つ GHC でこれだけの削減が可能である意義は大きい．また，実行時間については，Thunk Recycling は適用するプログラムを選ぶ必要がある結果となった．Thunk Recycling の効果がみられたベンチマークプログラムにおいては，オリジナルの GHC に対して 90%以下の実行時間で済むが，効果がみられないベンチマークプログラムでは，約 150 % の実行時間がかかるものもあった．これらの結果について個別に分析を行った結果，実行時間が改善したベンチマークプログラムでは，ごみ集め中のコピーが抑制できていた．一方で，実行時間が悪化したベンチマークプログラムでは，割り当てた再利用

サンク一個あたりの再利用回数の少なさや，間接参照を導入したことによるメモリリークなどの複合的な要因が実行時間の悪化につながったと考えられる．

全ベンチマークプログラムの平均では，約 103% の増加となっており，プログラムに合わせて選択的に Thunk Recycling を適用することが今後の課題である．

第7章

議論

本章では，Thunk Recycling の GHC への実装を通して得られた知見について述べる．まず，6 章の実験結果を受けて，Thunk Recycling の適用により効果があるプログラムについて考察する．次に，GHC が Haskell で記述された大規模なプログラムであるという観点から，遅延評価に基づくプログラミング言語処理系に関して得られた知見について述べる．最後に，GHC の設計に合わせて Thunk Recycling を実装したことに対する議論を行う．

7.1 Thunk Recycling の適用効果

Thunk Recycling は，リストの後続の遅延に必要となるサンクを再利用するため，リストを用いるプログラムに対して適用される．ただし，6 章の結果から，単にリストを用いていればよいわけではなく，Thunk Recycling を適用するプログラムを，リストの使い方に基づいて選択しなければ，プログラムの実行時間を減らすことはできない．Thunk Recycling を適用することで，有利に働き得るのは，以下の 2 つである．

- オブジェクトの割当て時間
- ごみ集めの回数

GHC では，オブジェクト割当て時間が全体の実行時間に比べて非常に少ないため，オブジェクトの割当て時間を減らすことによって総実行時間を大きく改善することはあまり望めない．そのため，ごみ集めの回数を減らし，ごみ集め全体の処理に必要な時間を削減することによって，プログラムの実行時間を減らす必要がある．その一方で，生存する

オブジェクトの数は Thunk Recycling によって基本的に減らず，コンスセルの head 部のメモリリークが起こるプログラムでは生存するオブジェクトが増えることもある．そのため，GHC で採用されている世代別コピー式ごみ集めの場合には，ごみ集め本体の処理である Evacuation と Scavenge 操作において，1 度のごみ集めでの実行時間差は生まれない．ごみ集めを何度もする中で，再利用によってオブジェクトのコピー量を減らすことができれば，実行時間が改善する可能性がある．

結論として，Thunk Recycling によりごみ集めの回数の減少を目指すことになるが，そのためには，再利用の回数を増やす必要がある．遅延リストをできるだけ長く読み進めると，Thunk Recycling によって実行時間が改善することが期待できる．しかし，リストがどれだけ読み進められるかやどれだけサンクが生存するかという動的なふるまいをプログラムから自動的に判別するのは，難しい問題である．リストを読み進める量があらかじめ分かっていれば，もともと遅延評価する必要がないからである．

以上のように Thunk Recycling で効果があるプログラムを自動的に判別するような手法は明らかでなく，Thunk Recycling の適用により効果がある問題領域を限定するのは難しい．試してみないと効果は分からないが，ヒューリスティックを導入して，選択的に Thunk Recycling を適用することは可能である．たとえば，Haskell においては，`n` から始まる無限リストを `[n..]` と記述するが，このような記法で与えられたリストは長く読み進められるだろうと予想して，Thunk Recycling を適用することは考えられる．このような Thunk Recycling を選択的に適用する手法と，選択アルゴリズムの開発は今後の課題である．

7.2　Haskell によるコンパイラ記述

これまで説明してきたように，プログラム変換とコード生成のフェーズにおいて，GHC の Haskell で実装されている部分のプログラムに変更を加えた．本節では，プログラム変換とコード生成それぞれのフェーズにおいて，GHC が Haskell のプログラムとして書かれていることから得られた知見について述べる．また，GHC を大規模な Haskell プログラムとして見たときの特徴についても述べる．

7.2.1 Haskell によるプログラム変換の記述

まず，5.4 節と 5.5 節で述べたように，中間言語に相当する Core 言語あるいは STG 言語を入出力とするプログラムによって，Thunk Recycling のプログラム変換を実現した．Core 言語，STG 言語は代数的データ型を用いて設計されていることにより，拡張性に関して以下のような利点がある．

- 中間言語に要素を追加する際に，型による検査が可能であること．
- 宣言的な記述が可能なため，形式的な記述と相性がよいこと．

1 つ目について，再利用サンクに対応するデータ構成子を STG 言語に追加し，再利用サンクのコンパイル機構を実現する際に，変更を加える箇所が十分であるかを型検査により確認することができたことは，拡張のコストを下げている．このことは，Core-to-Core の変換や STG-to-STG の変換というよりも，STG 言語からのコード生成で得られる恩恵が大きかった．つまり，コード生成は，既に GHC で実現されている変換処理に対する広範囲な変更が必要となるが，このとき変更の誤りをコンパイル時に型検査によって発見できたことがあった．さらに，抽象データ型を用いることで，パターンマッチにおいて，必要な変更を網羅した記述が可能であった．

2 つ目について，4.2 節に示したようなプログラム変換の形式的な宣言を，ほぼそのままの形で Haskell のプログラムとして実装できるという利点があった．

以上のような利点はあるものの，抽象構文木を破壊的に書き換えながら変換を進めるという手続き型言語のプログラミングスタイルに慣れている場合には，変換のたびに新たなデータ構造を割り当てるという Haskell のプログラミングスタイルに，効率の面で抵抗があるかもしれない．その点は，コンパイル時間の最適化を考えれば，より重要になってくると考えられる．現状は実装の簡潔さを優先したため，コンパイル時間は重視しなかった．

7.2.2 Haskell によるコード生成の記述

コード生成に関する変更箇所については，あるコード片でのスタック位置や必要ヒープ量の計算などのために，どうしても手続き的に扱う必要が出てくる．たとえば，STG 言語を手続き的な C−− 言語に変換するのに際し，計算を順序付けし，ある部分式で必要となるヒープ量を求めた上で必要であれば合算し，ヒープをまとめて確保するコードを生成する必要がある．このような処理を，GHC では Haskell のモナドという仕組みを

用いて，実現している．モナドは計算を部品化して組み合わせることを可能とし，コンパイラの簡潔な記述に寄与しているが，その部分のコードの解読には慣れが必要であった．ただし，モナドは圏論などの理論的背景から一般的には理解するのが難しいとされているが，GHC に変更を加える上で理論的背景まで要求されることはなかった．

7.2.3　大規模な Haskell プログラムとしての GHC

Thunk Recycling で加えた変更点から離れて，大規模な Haskell プログラムとして GHC を見たとき，以下の二点について特筆すべきことがあった．

- 型情報を明示するプログラミングスタイル
- プログラムにおける遅延性の利用

1 つ目について，Haskell では型推論により処理系が型を与えることもできるが，型情報をプログラマが明示するプログラミングスタイルが推奨されている．このことは，大規模なプログラムの理解において，関数の型情報をもとに既存のコードの調査すべき箇所を絞り込むことの助けとなった．さらに，コメントが書かれていなかったとしても，プログラムの仕様を読み取ることができるという面で非常に重要であった．2 つ目について，データ構造の遅延性を利用しているものを除けば，Haskell が遅延評価であることを意識することはほとんどなかった．

7.3　Thunk Recycling の GHC への実装

7.3.1　GHC の変更点

プログラム変換の導入は，変更するファイルが一部に集中しており，変更に必要な GHC に関する知識が少なく済んだ．これは，GHC がプログラム変換の導入を意識して設計されているためであり，コードが巨大化してきた GHC においても，多くの研究の基盤として用いられ続けている一つの要因となっている．ただし，導入する手法によっては既存の変換との適用順などが問題となる可能性もあるため，拡張を意識しているとはいえ十分な考慮が必要である．

GHC に新たな手法をプログラム変換として導入する際には，基本的には Core 言語上で実装するのがよい．本論文の Thunk Recycling は，サンクの存在が明示されている必要

があったため，STG 言語上でのプログラム変換も必要であったが，STG 上のそれ以外の変換は，もともとはプロファイルに関するものしか実装されていない．Thunk Recycling の実験結果が，メモリ削減効果はあったものの実行時間に関する効果があまりみられなかったことから分かるように，STG 以降のフェーズへの変更は，動作モデルに影響を与えるため，新たな手法によって効果を得るのが難しい場合が多いと考えられる．

プログラム変換が一部のファイルに集中していたのに対して，コンパイラのコード生成部は，変更点が複数ファイルに散在している．そのため，複数のファイルに渡って関数定義を調べるなどコストのかかる作業が必要であった．しかし，既存の GHC のコードは読み易く書かれていたため，関数の個別の実装に難解な部分は少なかった．たとえば，今回の Thunk Recycling に関していえば，通常サンクに関する既存の GHC のコードを参考に実装すればよいことを読み解くのは容易であった．

7.3.2 Thunk Recycling 導入時のデバッグ作業

デバッグについては，GHC の中間言語を表示する機構を利用することが有効であった．具体的には，`-ddump-prep` や `-ddump-stg` などのコンパイルオプションを与えることで，コンパイラの各フェーズ後の状態が得られる．さらに，同様の表示を全フェーズに渡って得ることも `-v4` といったコマンド（“4” は詳細さのレベル）を与えることで可能である．これらのコンパイルオプションを用いることで，Core-to-Core の変換の適用結果を細かく表示でき，どの変換がどれだけ適用されたかも把握することができた．ただし，中間言語の内部構造に対して，情報がすべて表示されるまでは実装されていないことに注意が必要であった．たとえば， STG 言語では，ある式が含む自由変数の情報までは表示されないため，変換を実装する際には，lint のようなチェックプログラムが必要であった．

そのようなデバッグ機構を用いた場合でも，C−− で書かれたコード生成結果のデバッグは，コストのかかる作業であった．それは，C−− フェーズ以後で，コード量も一度に大きくなり，元の言語との対応付けを取るのに手間がかかったためである．対処法としては，出力コードに C 言語を指定し，バグが疑われる箇所に C プログラムのデバッグコードを挿入する方法が有効であった．

7.4 本章のまとめ

本章では，Thunk Recycling の GHC 上の実装を通じて得られた知見について述べた．

Thunk Recycling で実行時間の面での効果を得るためには，選択的に再利用を行う必要があるなど Thunk Recycling には残された課題がある．Thunk Recycling によりごみ集めの回数を減少させるには，再利用の回数を増やす必要があり，リストをできるだけ長く読み進めるプログラムが望ましい．しかし，リストを読み進める量があらかじめ分かっていれば，遅延する必要がないため，プログラムから静的に自動的に判別するのは難しい問題である．

洗練された遅延評価に基づくプログラミング言語処理系である GHC への Thunk Recycling の実装を通じて多くの知見が得られた．特に，GHC が Haskell 自身で書かれていることは，大規模なコンパイラの一部として Thunk Recycling を実装する上で非常に有効に働き，関数型プログラミング言語の有用性を確認できた．

第8章

関連研究

本章では，本研究を関連研究と比較することで，Thunk Recycling の新規性とその位置付けについて述べる．多くの既存研究が，サンクの抑制やデータ構造の再利用について扱っているが，Thunk Recycling とは方針が異なっている．さらに，GHC 上に Thunk Recycling を実装した観点から，GHC 上の既存の実行効率改善手法についても説明する．

8.1 既存のサンク抑制機構

サンクを抑制する既存の仕組みとして，2.3 節で述べた正格性解析の他にも，Cheap eagerness analysis [8, 9] や Optimistic evaluation [7] などの解析手法がある．Cheap eagerness analysis は，遅延させるまでもない軽い計算をプログラム全体で解析し，プログラムの意味を変えない範囲で，値が必要になるかどうかにかかわりなく式を遅延させずに，すなわちサンクを生成せずに前もって計算してしまう．軽い計算とは，たとえば，組み込みの算術演算などの計算コストが小さいとあらかじめ分かっている処理である．式を遅延せずに計算を進めてしまうため，Cheap eagerness analysis は，プログラムを投機的に実行する手法である．論文 [8] では，静的に解析するのみの手法であったが，後続の論文 [9] では，実行時にもそれぞれの式に対して遅延する必要があるか否かの判断を行う手法へと拡張された．前もって計算することで，遅延にともなうコストを削減できるため，プログラムの実行時間を削減できることが，実験により確認されている．遅延データ構造に注目すると，一定の長さだけ正格に評価を進めることで，遅延評価の利点を残しつつ，サンクの生成などのオーバヘッドの削減を可能とした．一方で，Cheap eagerness analysis は，プログラム全体に対する解析が必要であり，実装が複雑になるという欠点が

ある．すなわち，プログラム全体を必要とするので，分割コンパイルには向かず，ライブラリ関数なども含めた解析が必要となってしまう．その欠点を補う形で，Cheap eagerness analysis 同様に投機的に計算を進める Optimistic evaluation [7] という手法が開発された．Cheap eagerness analysis のように前もって遅延データ構造の評価をある長さ分だけ投機的に進めることでサンクを削除するが，Optimistic evaluation は，読み進める量を実行時に取得するプロファイル情報に基づいて決定する．プロファイル情報を利用することで，複雑な解析をせずに済ませる．

Cheap eagerness analysis と Optimistic evaluation のいずれの場合も，投機的にリストを進めるので，投機に失敗した場合には，無駄なリストを生成する可能性がある．それに対して，Thunk Recycling は遅延リストの後続で，再利用サンクを破壊的に上書きして用いるので，投機的にリストを読み進めるようなことはない．さらに，Thunk Recycling は，プログラム全体に対する複雑な解析が必要ないという違いもある．ただし，遅延データ構造の後続のサンク以外は，Thunk Recycling の対象外であるため，正格性解析の場合と同様に，Thunk Recycling を Cheap eagerness analysis や Optimistic evaluation と併用することによって，サンク削減の効果を高めることができると考えられる．

8.2　遅延データ構造に関する関連研究

Thunk Recycling の着想は，Improving Sequence（IS）[19] と呼ばれるデータ構造を扱う処理系 [55,56] で用いられている手法から得た．IS は，ある最終値へと近づいていく近似値の列を表現するデータ構造であり，要求駆動的に近似値を求めていくことで，探索プログラムの枝刈り記述に用いることができる．意味上は近似値の列を表現するデータ構造であっても，枝刈りに用いるのは最終値に近い近似値だけであるため，IS の処理系は，一番精度の良い近似値だけを更新して保持するよう実装されている．Thunk Recycling は，IS の近似値を更新して用いる手法を，一般の遅延データ構造で必要となるサンクに対して適用したものととらえることもできる．ただし，IS の場合には，更新前の近似値を参照しても，枝刈りの効果が薄まるだけで間違った結果にはならないが，再利用サンクの場合には，更新して用いる再利用サンクが単一参照されていなければ，誤った結果を求めることとなってしまう．そのため，3.3 節で述べたように，再利用可能となるサンクの条件を細かく検討し，単一参照性を保証するための変換を行う必要があった．

Spineless Tagless G-machie（STG）[22] は，関数プログラミングをモデル化するために設計された抽象機械であり，初期の GHC はこのモデルによりプログラムを実行して

いた．STG では，update in place という手法が提案されている．これは，強制中のサンクを強制結果のために書き換えて使用することで，メモリ割り当て量の削減を試みるものである．たとえば，5 つのペイロードを持つようなサンクを強制し，2 つのペイロードからなるコンスセルが結果として得られる場合には，5 つのうち 2 つのペイロードを使って，コンスセルとする．それに対して，Thunk Recycling は，強制中のサンクを強制により作られるコンスセルの後続のサンクとして再利用する．Thunk Recycling の目的は，update in place と同様にヒープ使用量の削減を狙ったものであるが，割り当て済みのサンクを，強制結果ではなく別のサンクとして再利用するという点が update in place と異なっている．3 章で見たように，リストの後続のサンクは強制中のサンクと似た形をとる傾向があるので，Thunk Recycling の方が，効率よく再利用できる可能性がある．なお，update in place は，GHC の開発初期（バージョン 4.0 以前，2000 年頃）には実装されていたものの，現状では取り除かれてしまった．その理由として，次のことが挙げられている [1]．

- ランタイムシステムの実装が複雑となること．
- 並列化と相性が悪いこと．

1 つ目の項目に関して，Thunk Recycling は，通常サンクの仕組みを流用することで，実装が複雑となることを防いだ．実際，5 章で説明したように，再利用機構の多くは GHC の既存のサンクの強制の仕組みを流用したものである．2 つ目の項目に関して，並列化と Thunk Recycling の関係は，現状ではあまり明らかになっていない．これを明らかにしていくことは，今後の課題である．

サンクの単一参照性に注目した遅延評価処理系の効率化に関する研究には，update avoidance [32, 34] がある．これは，単一参照されているサンクを強制結果で上書きする操作は，それ以降そのサンクが参照されなければ無駄であることに着目し，上書きの必要のないサンクを静的に解析する．本研究は，上書きの操作によるオーバヘッドでなく，サンクに必要となる無駄なメモリ領域を削減することを目的としているため，目的，方針ともに大きく異なる．

Minamide による研究 [38] は，先行評価を行う関数型プログラミング言語である ML [37] において，“穴”（初期化されていないフィールド）を持つデータ構造を提案し，hole 抽象と呼ぶ機構を提案し，関数的な記述による効率的なプログラムを書くことを可能とした．hole

[1] メーリングリスト（http://www.haskell.org/pipermail/glasgow-haskell-users/2007-June/012719.html，http://www.haskell.org/pipermail/haskell-cafe/2004-July/006463.html）で議論されている．

抽象はラムダ抽象と似ているが，hole 抽象が評価されるときに，その式を評価し，データ構造を構成する点で異なる．たとえば，tail 部に穴のあるコンスセルは，$\hat{\lambda}x.\texttt{cons}(1,x)$ というように hole 抽象 $\hat{\lambda}$ を使って表現され，x が穴である．この例では，hole 抽象が評価された時点で，コンスセルが構成される．もし，$(\hat{\lambda}x.\texttt{cons}(1,x))\ \texttt{nil}$ のように x に実際の値 `nil` が与えられれば，hole 抽象を評価した際に構成済みのコンスセルの穴に `nil` が入る．穴に対する破壊的な書き換えという副作用のある操作が必要となるが，関数型プログラミング言語における副作用の表現手法 [13,52] を用いて，副作用がユーザのプログラムに影響を与えないようにしている．このことは，Thunk Recycling において，再利用サンクへの参照を単一なものに限定していることに似ている．Minamide による研究と Thunk Recycling の違いは，採用する評価戦略による．Minamide による研究は，先行評価を前提としたプログラム中に，上のコンスセルの例のように，ユーザが遅延するデータ構造を明示しなければならない．その一方で，Thunk Recycling は，遅延評価を前提とするため，サンクがプログラム中に明示されず，再利用は処理系の最適化手法として自動的に適用される．

8.3　型システムに関する研究

線形論理（linear logic） [12] は仮定を一度しか利用できないと限定することで，資源の消費を表現する論理体系である．線形論理をプログラミング言語の型システムに応用したものが，線形型（linear type）[50, 52] であり，線形型を用いれば，すべての変数はちょうど一度しか使用されない．一度しか使用されないと型推論によって判定できれば，破壊的に書き換えて用いることができるため，効率的な実装が可能となる．関数型プログラミング言語における線形型の研究 [15, 50, 54] により，線形型を利用したデータ構造の書き換えによって，オブジェクトの無駄な複製を省くことができることが確認されている．

Thunk Recycling でも単一参照されるサンクに注目し，サンクを再利用するという点で，線形型を利用したデータ構造の書き換え手法と一致する点は多いが，Thunk Recycling では，型に単一参照の情報を持たせるように設計しなかった．それは，型情報を用いずに静的なプログラム変換とすることで，解析を単純にすることができると考えたためである．また，型システムのように言語全体への影響が大きい構成要素に対する変更を避けることで，現実のプログラミング言語処理系への実装が容易になるという違いもある．

Concurrent Clean [40] の一意型（uniqueness typing） [3] も，線形型と同様に型情報

に，値の使われかたを保持し，推論する手法である．ユーザは，破壊的な書き換えのための型注釈を与え，Clean のコンパイラが正しく書き換えることができるかどうかをチェックする．ユーザの記述に依っているという点で，Thunk Recycling がプログラム変換によって再利用サンクの単一参照性を保っていることと異なる．

型情報を用いたデータ構造の書き換えに関する別の研究として，Hage らの研究 [14] がある．この研究では，ユーザがデータ書き換えの情報を与えることができるよう対象言語を拡張し，その情報を型システムで解析した上で書き換え可能であるか確認するような言語機構を提案している．その上で，その言語を用いて，Launchbury の natural semantics [31] に基づいた形式的な定義を与えている．本論文で定義した Thunk Recycling の操作的意味論と同様に，環境とヒープを区別することで，ヒープへの書き換えを明示的に扱っている．一方で，我々の研究との大きな違いは，サンクは処理系で内部的に作られるオブジェクトであるため，これらの研究で対象としているデータ構造の破壊的な書き換えの対象ではないという点がある．また，Hage らの形式化が natural semantics に基づいているため big-step の操作的意味論であるのに対し，我々の形式化は small-step の意味論を採用している点も異なっている．

線形型以前になされた，関数型プログラミング言語のデータ構造の書き換えに関して，Hudak による研究 [16] は，単一参照されている配列やベクトルなどのデータ構造の複製を抑えることを目指し，静的な解析と実行時の参照カウントを合わせた手法を提案した．これは，実行時の参照カウントにより，単一参照であると静的に判断できなかったデータ構造にも手法が適用できる．これに対して Thunk Recycling では，参照カウントの管理の実装は複雑であると考え，参照カウントによってサンクの単一参照性を判定する方法は採用しなかった．

8.4 オブジェクトのメモリ管理に関する研究

リージョンによるメモリ管理機構（Region-based memory management）[1, 46, 47] は，ヒープをリージョンに分割し，そのリージョンに対してオブジェクトを割り当てる．Tofte らの研究 [47] は，関数型プログラミング言語 ML [37] に対してリージョン推定と呼ばれる解析によって各オブジェクトの生存期間を推定した上で，割り当てるリージョンを静的に決定した．生存期間が同じオブジェクトを同じリージョンにまとめることができれば，複数のリージョンをスタックと同じ要領で管理して，生存期間が終わったリージョンから順に解放することで，あるリージョンに属するオブジェクトをまとめて解放でき

る．Aiken らの研究 [1] は，Tofte らの研究を拡張し，リージョンをスタックとして管理せずに，リージョンの生存期間を制約問題に帰着させることで，生存期間推定の精度を上げた．Thunk Recycling は，単一参照されているサンクは強制されればごみとなることに基づいているので，オブジェクトの生存期間を推定している手法であるという点で，リージョンによるメモリ管理機構と関連している．つまり，ある再利用サンクが強制された時点で，その再利用サンクを後続の計算の遅延に用いるというのは，ごみになる瞬間をとらえて，瞬時にそれを回収し，再利用しているという見方もできる．単一参照されているサンクに限定することで，複雑な解析をすることなく，再利用サンクの生存期間が分かるという利点が，Thunk Recycling にはある．

8.5　プログラミング言語処理系に関する研究

GHC に限らず，多くの関数型プログラミング言語処理系では，末尾呼び出し最適化（tail call optimization）[6, 29] と呼ばれる手法が導入されている．末尾呼び出し最適化とは，関数の末尾にある関数呼び出しに対して，スタックを消費するような関数呼び出しではなく，呼び出し先の関数のコードへのジャンプとする実装手法である．後続の計算に必要となるスタックの消費を抑えることができるため，再帰的なプログラムに対してこの最適化を行えば，ループと同様の処理を期待できる．Thunk Recycling は，リストの tail 部を対象とする最適化であるため，後続の計算にかかる資源の削減を目指しているという点では，末尾呼び出し最適化と似ている面もある．しかし，削減する資源が Thunk Recycling ではヒープ領域であり，末尾呼び出し最適化ではスタック領域であるという点で異なる．さらに，その最適化の具体的な手法には，大きな違いがある．末尾呼び出しの最適化では，関数の末尾呼び出しのスタックを削減できることが静的に決められるが，Thunk Recycling では，静的に単一参照性に関する保証が必要であり，さらに実行時の文脈によって再利用できるかどうかが決まる．つまり，Thunk Recycling はサンクの生成と強制という動的なふるまいまで考慮した手法である．

次に，GHC を研究の基盤として用いる観点から，実行効率の改善を目指した研究のうち代表的ものを挙げる．まず，プログラム変換を GHC に導入した代表的な研究として，Let-floating [26]，Allwood らによる研究 [2]，Call-pattern specialization [24] がある．Let-floating は，let 式の位置を調整するプログラム変換により，サンクの生成を効率化するものである．Allwood らは，プログラムにデバッグ情報を挿入し，スタックトレースの表示を可能とする手法を示した．Call-pattern specialization は，再帰関数などにおい

て，不要なデータの生成を抑制するものである．これらの研究は，Core 言語上での変換として完結しており，GHC の Core-to-Core フェーズを変更することにより実装されている．7 節で述べたように，Core-to-Core の変換は GHC が拡張性を考慮して設計されているため，手法の実装は単純である．なお，これらの研究の成果は，GHC の本流に採り入れられている．

一方で，ポインタタギングを用いたパターンマッチの実現法 [36] は 5.5.2 節で詳細を説明したように，処理系全体への変更が必要である．具体的には，パターンマッチやデータ構成に関する C−− コード生成に手を加え，タグ情報を含んだコードを生成するようにする．それと同時に，ごみ集めなどランタイムシステムにおいてもタグを取った後に参照をたどるなどの変更が必要となる．

Marlow らによる研究 [35] は，関数呼び出しに関する評価モデルについて，二つの方法を比較した．一つは，スタックに実引数を積んだ上で関数本体にジャンプし，関数本体において実引数の数を調べ，必要であれば高階関数の処理を関数本体において行う手法（push/enter と呼ぶ），もう一つは，先に関数を評価し必要とする実引数の数を調べ，関数本体に入る前に必要であれば高階関数を扱う手法（eval/apply と呼ぶ）である．これら二つは，関数の実引数の充足性を調べる処理の場所に違いがあり，従来，関数型プログラミング言語においては push/enter が採用されてきた．それに対して，Marlow らは，push/enter よりも，eval/apply のほうが望ましいということを GHC 上に実装した上で，ベンチマークプログラムによる実験とともに示した．評価モデルが大きく異なるため，GHC のコード生成に大きく変更が必要であったと考えられる．実際，eval/apply は GHC バージョン 6.0 で導入されており，一つ前のバージョン GHC 5.04 との差分を取ると，コンパイラで 275 ファイル中 209 ファイル，ランタイムシステムで 90 ファイル中 55 ファイルが変更されていた．この変更点が eval/apply によるものだけでないものの，大きな変更が必要だったことが分かる．

以上のように，多くの研究がなされてきたが，論文の発表時期を見ると GHC の実行改善を目指した研究は，最近では少なくなってしまっている．一方で，記述性や信頼性に関する研究 [28, 45, 51] が盛んに行われれるようになってきた．これらについて GHC を基盤として用いる場合は，本論文では触れなかった Parse から Core 言語に至るフェーズでの変更が必要となるであろう．

8.6　本章のまとめ

本章では，Thunk Recycling の関連研究について述べた．Thunk Recycling は，遅延データ構造の後続に注目するという点で，新規性があった．また，サンクの単一参照性に注目した研究やデータ構造の破壊的な書き換えに注目すると，型情報を利用する研究が多くなされてきたが，Thunk Recycling は，静的解析を単純にし，実装を容易にするという点を重視して，型情報を用いない設計方針を選択した．

プログラミング言語処理系における遅延評価の実装法の確立に向けて，GHC を基盤として用いた研究が果してきた役割は大きい．研究分野で得られた成果が，現実に用いられるプログラムの実行効率の改善に役立てられている．Thunk Recycling もその研究の一つとして，現実の実用的な環境における，遅延評価の実装法の確立を目指したものである．

第9章

結論

9.1 本論文のまとめ

本論文は，遅延評価における再帰的データ構造の特徴をとらえ，再帰的データ構造の後続計算を遅延させるサンクを再利用する実装法 Thunk Recycling について扱った．Thunk Recycling は，既に割当てられているサンクを破壊的に更新することで再利用し，新たなサンクの生成を抑える．たとえば，代表的な線形に定義される再帰的データ構造であるリストにおいては，tail 部に必要となるサンクを再利用できる．Thunk Recycling の再利用機構は，破壊的な更新により矛盾が起こらないようにするコンパイル時の変換と，実行時に再利用を行う機構から構成される．

本論文は，Thunk Recycling に関する一連の研究について，手法の提案，形式的な定義と手法の正しさの証明，GHC 上への実装，その実装を用いた実験などについて述べたものである．研究の結果，リストの tail 部に必要となるサンクを破壊的に書き換えることにより，プログラム実行時の総メモリ割当量を削減できるという普遍的な価値を見いだすことができた．一方で，プログラムの総実行時間に関する再利用の効果は，元の言語処理系の設計方針や適用するプログラムなど多くの要因に左右されることが分かった．総実行時間における再利用の効果を得るためには，Thunk Recycling を選択的に適用する枠組みを今後の開発していく必要がある．

9.2 本研究の貢献点

本論文の貢献点をまとめると，以下のとおりである．

- Thunk Recycling を提案した．Thunk Recycling は，再帰的データ構造において，同じサンクが繰り返し現れることを利用し，そのサンクが保持している値を破壊的に更新し用いる．このようなサンクの再利用機構を，リストに対して適用した．破壊的な更新により矛盾した状態にならないように，コンパイル時には，再利用サンクに対する単一参照性を保つプログラム変換をコンパイル時に行い，プログラムの実行時にはコンスセルの tail 部を再利用サンクを強制した結果の値に破壊的に書き換えることでリストを接続する．

- Thunk Recycling の形式的な定義を与え，その正しさを証明した．簡単な言語上で Thunk Recycling のプログラム変換を記述し，small-step の操作的意味論による形式的な定義を与えた．その操作的意味論を用いて，Thunk Recycling の正しさを証明した．Thunk Recycling の有無で，プログラムのふるまいが変わらないことについて，簡約規則の対応関係と双模倣の考え方を用いた．

- GHC 上に Thunk Recycling を設計し，実装した．GHC を研究基盤として用いる事例として，Thunk Recycling で採り得る設計方針について比較し，方針ごとの利害得失に関して議論した．その方針に基づいた実装について詳細に説明した．

- Thunk Recycling の効果をベンチマークプログラムにより確認した．GHC 上の実装を用いて，プログラムで必要となる総メモリ割当量と，プログラムの実行時間を計測した．サンクの生成を抑えることで，総メモリ割当量の削減に効果があることが分かった．ただし，プログラムの実行時間に関しては，再利用にともなうオーバヘッドが大きくなるベンチマークプログラムもあり，Thunk Recycling は適用すべきプログラムを選ぶという結果であった．

- 遅延評価を行う関数型言語処理系の構成法に関して議論した．GHC を関数型プログラミング言語で作成された高度なソフトウェアであるととらえ，GHC を研究の基盤に用いた経験により得られた遅延型関数型言語処理系に関する知見について述べた．

9.3　よりよい遅延評価機構の実現に向けて

本論文の最後に，Thunk Recycling の実現を通じて得られた，遅延評価機構に関する所感を述べる．

まず，遅延評価がプログラムの記述性に関して大きな効果があることは，明確である．プログラムを宣言的に記述することにより，プログラムの記述が簡潔になり，かつ強力にモジュール化の進んだ記述が可能となる．さらに重要なのは，宣言的なプログラム記述は，形式的な定義と非常に相性がいい点である．実際，本論文の 4 章と 5 章で扱った内容は密接に関係しており，Thunk Recycling の形式的な定義を現実の処理系である GHC に実現するということが可能であった．これは，Thunk Recycling のような最適化手法だけに留まらず，一般のプログラムにおいてもプログラムの性質を形式的に定義した上で，実現することが可能となる．このことは，今後，より重要になってくると考えている．すなわち，複雑で大規模なプログラムにおいては，形式的な手法によりプログラムの正しさを確認しなければ，動作の安全性を保証できなくなってくるであろう．

このような状況において，遅延評価機構のより効率的な実現が必要となると考えられる．本論文で示した Thunk Recycling は，正格性解析などこれまでの静的解析手法が対象としていなかった，動的なサンクのふるまいに注目したものであった．しかし，サンクを再利用し，サンクの生成を抑制することによって，空間的なオーバヘッドを削減することができたものの，時間的なオーバヘッドにおいては，適用するプログラムを選択する必要があった．つまり，Thunk Recycling のようなサンクの動的なふるまいに注目した手法においても，静的な解析により時間的オーバヘッドを十分に削減しなければならない．しかし，それには実行時のサンクの生成・強制という動的なふるまいを実行時間の面から解析することになるため，それほど単純なことではない．実際，4 節で定義した操作的意味論では，ヒープ上への新たなアドレスの確保や，環境への変数の追加などメモリ面でのふるまいを考慮したものであったが，実行時間の面からは扱えていない．そのため，GHC のように複雑なメモリ管理やごみ集めを行う処理系においては，サンクの生成を抑えることが時間的なオーバヘッドの削減に単純には結びつかない．このことは，GHC に実装してみて初めて分かる事実であった．

以上のことを踏まえて，効率的な遅延評価機構の実現には，GHC など実際の処理系と密接に連携したコストモデルを構築する必要があると考えている．そのためには，まず第一に，宣言的に書かれた関数型言語のプログラムを，手続き的な計算機上での動作と結び付けて扱うような形式化が必要となるであろう．さらに厳密にコストを見積るには，サンクの生存期間などを考慮し，ごみ集めにかかるコストも含めて考えなければならない．ただし，GHC は構造が非常に複雑化しており，実際の処理系に則したコストモデルを構築するには，解決すべき問題が多い．

謝辞

本研究を進めるにあたり，多くの方々のご指導とご協力を賜わりました．すべての方々に心より感謝いたします．

まず，主任指導教員として指導していただいた岩崎英哉教授に深く感謝いたします．私が卒業研究で岩崎研究室に配属されてから 12 年もの長い間にわたり，多くの貴重な時間を割いて，手厚いご指導いただきました．岩崎研究室に配属したことで，プログラミング言語研究の面白さを知ることができ，ここまで研究を続けてくることができました．

次に，益田隆司先生に深く感謝いたします．単位取得退学という形で一度はあきらめた博士号を，再び目指すことができたのは，益田先生の後押しがあったためです．

鵜川始陽准教授と中野圭介准教授には，研究において多くの助言を頂きました．深く感謝いたします．

本論文の審査員として，改善のためのご助言を頂きました沼尾雅之教授，小花貞夫教授，寺田実准教授，中山泰一准教授，大山恵弘准教授に深く感謝いたします．

岩崎研究室，中野研究室の皆様には，色々な場面で大変お世話になりました．特に，佐藤重幸君には，研究の面で多くの助言を頂きました．深く感謝いたします．

牧大介君には，株式会社コマ・システムズでの仕事を続けつつ，博士の研究に取り組む時間を与えていただきました．深く感謝いたします．

最後に，あたたかく見守ってくれた両親に深く感謝いたします．

参考文献

[1] A. Aiken, M. Fähndrich, and R. Levien. Better Static Memory Management: Improving Region-based Analysis of Higher-order Languages. *Proc. the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (PLDI 1995), pp. 174–185. ACM, 1995.

[2] T. Allwood, S. Peyton Jones, and S. Eisenbach. Finding the Needle: Stack Traces for GHC. *Proc. the 2nd ACM SIGPLAN Symposium on Haskell* (Haskell 2009), pp. 129–140. ACM, 2009.

[3] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. *Proc. the 13th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science* (FSTTCS 1993), pp. 41–51, 1993.

[4] C. Beshers, D. Fox, and J. Shaw. Experience Report: Using Functional Programming to Manage a Linux Distribution. *Proc. the 12th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2007), pp. 213–218. ACM, 2007.

[5] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2nd edition, 1998.

[6] W.D. Clinger. Proper Tail Recursion and Space Efficiency. *Proc. the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (PLDI 1998), pp. 174–185. ACM, 1998.

[7] R. Ennals and S. Peyton Jones. Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs. *Proc. the 8th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2003), pp. 287–298. ACM, 2003.

[8] K.F. Faxén. Cheap Eagerness: Specuative Evaluation in a Lazy Functional Language. *Proc. the 5th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2000), pp. 150–161. ACM, 2000.

[9] K.F. Faxén. Dynamic Cheap Eagerness. *Proc. the 13th International Workshop on Implementation of Functional Languages* (IFL 2002), pp. 105–120. Springer-Verlag, 2002.

[10] R.R. Fenichel and J.C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, Vol. 12, No. 11, pp. 611–612, 1969.

[11] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial Uses: Going Functional on Exotic Trades. *Journal of Functional Programming*, Vol. 19, No. 1, pp. 27–45, 2009.

[12] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, Vol. 50, pp. 1–102, 1987.

[13] J.C. Guzmán and P. Hudak. Single-threaded Polymorphic Lambda Calculus. *Proc. the 5th Annual IEEE Symposium on Logic in Computer Science* (LICS 1990), pp. 333–343. IEEE, 1990.

[14] J. Hage and S. Holdermans. Heap Recycling for Lazy Languages. *Proc. the 11th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (PEPM 2008), pp. 189–197. ACM, 2008.

[15] M. Hofmann. A Type System for Bounded Space and Functional In-place Update. *Nordic Journal of Computing*, Vol. 7, No. 4, pp. 258–289, 2000.

[16] P. Hudak and A. Bloss. The Aggregate Update Problem in Functional Programming Systems. *Proc. the 12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 1985), pp. 300–314. ACM, 1985.

[17] P. Hudak, J. Huges, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy with Class. *Proc. the 3rd Conference on History of Programming Languages* (HOPL-III), pp. 1–55. ACM, 2007.

[18] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, Vol. 32, No. 2, pp. 98–107, 1989.

[19] H. Iwasaki, T. Morimoto, and Y. Takano. Pruning with Improving Sequences in Lazy Functional Programs. *Higher-Order and Symbolic Computation*, Vol. 24, No. 4, pp. 281–309, 2011.

[20] R. Jones. Tail Recursion Without Space Leaks. *Journal of Functional Programming*, Vol. 2, No. 1, pp. 73–79, 1992.

[21] S. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[22] S. Peyton Jones. Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine - Version 2.5. *Journal of Functional Programming*, Vol. 2, pp. 127–202, 1992.

[23] S. Peyton Jones. Haskell 98: Introduction. *Journal of Functional Programming*, Vol. 13, pp. i–6, 2003.

[24] S. Peyton Jones. Call-pattern Specialization for Haskell Programs. *Proc. the 12th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2007), pp. 327–337. ACM, 2007.

[25] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview. *Proc. Joint Framework for Information Technology* (JFIT) *Technical Conference*, pp. 249–257. ACM, 1993.

[26] S. Peyton Jones, W. Partain, and A. Santos. Let-floating: Moving Bindings to Give Faster Programs. *Proc. the 1st ACM SIGPLAN International Conference on Functional Programming* (ICFP 1996), pp. 1–12. ACM, 1996.

[27] S. Peyton Jones, N. Ramsey, and F. Reig. C−−: a Portable Assembly Language that Supports Garbage Collection. *Proc. the 2nd International Conference on Principles and Practice of Declarative Programming* (PPDP 1999), pp. 1–28. ACM, 1999.

[28] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple Unification-based Type Inference for GADTs. *Proc. the 11th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2006), pp. 50–61. ACM, 2006.

[29] G.L. Steele Jr. Rabbit: a Compiler for Scheme. Technical Report 474, MIT Artificial Intelligence Laboratory, May 1978.

[30] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proc. the 2004 IEEE/ACM International Symposium on Code Generation and Optimization* (CGO 2004), pp. 75–86. ACM, 2004.

[31] J. Launchbury. A Natural Semantics for Lazy Evaluation. *Proc. the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 1993), pp. 144–154. ACM, 1993.

[32] J. Launchbury, A. Gill, J. Hughes, S. Marlow, S. Peyton Jones, and P. Wadler. Avoiding Unnecessary Updates. *Proc. the 1992 Glasgow Workshop on Functional Programming*, pp. 144–153. ACM, 1992.

[33] H. Lieberman and C. Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, Vol. 26, No. 6, pp. 419–429, 1983.

[34] S. Marlow. Update Avoidance Analysis by Abstract Interpretation. *Proc. the 1993 Glasgow Workshop on Functional Programming*, pp. 170–184. ACM, 1993.

[35] S. Marlow and S. Peyton Jones. Making a Fast Curry: push/enter vs. eval/apply for Higher-order Languages. *Proc. the 9th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2004), pp. 415–449. ACM, 2004.

[36] S. Marlow, A. Yakushev, and S. Peyton Jones. Faster Laziness Using Dynamic Pointer Tagging. *Proc. the 12th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2007), pp. 277–288. ACM, 2007.

[37] R. Milnet, M. Tafte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[38] Y. Minamide. A Functional Representation of Data Structures with a Hole. *Proc. the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 1998), pp. 75–84. ACM, 1998.

[39] W. Partain. The `nofib` Benchmark Suite of Haskell Programs. *Proc. the 1992 Glasgow Workshop on Functional Programming*, pp. 195–202. ACM, 1992.

[40] R. Plasmeijer and M. van Eekelen. Concurrent Clean Language Report –version 1.3. Technical Report CSI-R9816, University of Nijmegen, 1998.

[41] I. Pop. Experience Report: Haskell as a Reagent: Results and Observations on the Use of Haskell in a Python Project. *Proc. the 15th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2010), pp. 369–374. ACM, 2010.

[42] D. Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

[43] P. Sansom and S. Peyton Jones. Generational Garbage Collection for Haskell. *Proc. the 6th ACM Conference on Functional Programming Languages and Computer Architecture* (FPCA 1993), pp. 106–116. ACM, 1993.

[44] P. Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, Vol. 7, No. 3, pp. 231–264, 1997.

[45] M. Sulzmann, M.M.T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with Type Equality Coercions. *Proc. the 3rd ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (TLDI 2007), pp. 53–66. ACM, 2007.

[46] M. Tofte and J.P Talpin. Implementation of the Typed Call-by-value λ-calculus Using a Stack of Regions. *Proc. the 21th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 1994), pp. 188–201. ACM, 1994.

[47] M. Tofte and J.P Talpin. Region-based Memory Management. *Information and Computation*, Vol. 132, No. 2, pp. 109–176, 1997.

[48] A. Tolmach. An External Representation for the GHC Core Language, 2008. `http://www.haskell.org/ghc/docs/papers/core-6.10.ps.gz`.

[49] D.A. Turner. Miranda: a Non-strict Functional Language with Polymorphic Types. *Proc. the 2nd ACM Conference on Functional Programming Languages and Computer Architecture* (FPCA 1985), pp. 1–16. ACM, 1985.

[50] D.N. Turner, P. Wadler, and C. Mossin. Once Upon a Type. *Proc. the 8th ACM Conference on Functional Programming Languages and Computer Architecture* (FPCA 1993), pp. 1–11. ACM, 1995.

[51] D. Vytiniotis, S. Peyton Jones, and J.P. Magalhães. Equality Proofs and Deferred Type Errors: a Compiler Pearl. *Proc. the 17th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2012), pp. 341–352. ACM, 2012.

[52] P. Wadler. Linear Types Can Change the World! *Proc. IFIP Working Conference on Programming Concepts and Methods*, pp. 347–359. North Holland, 1990.

[53] P. Wadler and R.J.M. Hughes. Projections for Strictness Analysis. *Proc. the 3rd ACM Conference on Functional Programming Languages and Computer Architecture* (FPCA 1987), pp. 385–407. ACM, 1987.

[54] K. Wansbrough and S. Peyton Jones. Once Upon a Polymorphic Type. *Proc. the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL 1999), pp. 15–28. ACM, 1999.

[55] 田村知博, 高野保真, 岩崎英哉. 純関数型言語の処理系における効率的な枝刈り機構の実装. 情報処理学会論文誌 プログラミング, Vol. 1, No. 2, pp. 28–41, 2008.

[56] 高野保真, 岩崎英哉. Improving Sequence を第一級の対象とする Scheme コンパイラ. 第 8 回プログラミングおよびプログラミング言語ワークショップ, pp. 153–162. 日本ソフトウェア科学会, 2006.

[57] 高野保真, 岩崎英哉, 鵜川始陽. Glasgow Haskell Compiler における再帰的データ構造のための遅延オブジェクトの再利用. 情報処理学会論文誌 プログラミング, Vol. 5, No. 2, pp. 67–78, 2012.

[58] 高野保真, 岩崎英哉, 佐藤重幸. Glasgow Haskell Compiler 上の遅延オブジェクトの再利用手法の設計と実装. コンピュータソフトウェア, Vol. 32, No. 1, pp. 253–287, 2015.

関連論文の印刷公表の方法及び時期

1. 高野保真，岩崎英哉，鵜川始陽.
   Glasgow Haskell Compiler における再帰的データ構造のための遅延オブジェクトの再利用.
   情報処理学会論文誌 プログラミング，Vol. 5, No. 2, pp.67–78.
   2012 年 4 月.
   主に第 3 章と 5 章に関連.

2. Yasunao Takano, Hideya Iwasaki.
   Thunk Recycling for Lazy Functional Languages: Operational Semantics and Correctness.
   Proc. 30th ACM/SIGAPP Symposium on Applied Computing (SAC 2015), pp.2079-2086 , Salamanca, Spain.
   2015 年 4 月.
   主に第 4 章に関連.

3. 高野保真，岩崎英哉，佐藤重幸.
   Glasgow Haskell Compiler 上の遅延オブジェクト再利用手法の設計と実装.
   コンピュータソフトウェア，Vol. 32, No. 1, pp.253–287.
   2015 年 1 月.
   主に第 5 章と 6 章に関連.