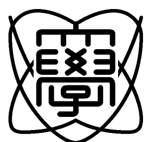

平成22年度 修士論文

Schemeにおける

動的ソフトウェア更新機構の設計と実装

電気通信大学 大学院情報システム学研究科

情報システム基盤学専攻

0953004 荻山 温夫

指導教員 小宮 常康 准教授
多田 好克 教授
古賀 久志 准教授

提出日 平成23年1月27日

目 次

第 1 章　序論 1

第 2 章　背景 5

2.1 動的ソフトウェア更新機構 . . . 5

2.1.1 動的ソフトウェア更新に求められる特性 . . . 6

2.2 動的ソフトウェア更新機構の利用モデル . . . 8

2.2.1 更新プログラムの作成 . . . 9

2.2.2 ソフトウェア更新の適用 . . . 10

第 3 章　関連研究 11

3.1 C 言語における DSU 機構 . . . 11

3.2 Java における DSU 機構 . . . 12

3.3 専用の仮想機械を用いる DSU 機構 . . . 13

第 4 章　設計 15

4.1 ソフトウェア更新の実行 . . . 15

4.2 DSU 機構の設計方針 . . . 16

4.2.1 本研究で対象とするソフトウェア更新 . . . 16

4.2.2 設計方針 . . . 17

4.3 データ置換処理の分割化 . . . 18

4.4 グローバル定義の更新，追加 . . . 21

4.4.1 アクセスバリアの追加 . . . 22

4.5 データの置換 . . . 23

4.5.1 前提とする条件 . . . 23

4.5.2 データ型およびアクセサ，コンストラクタ，型判定述語の命名規則 . . . 26
4.5.3 データの置換処理 . . . 27
4.6 状態の管理 . . . 30
4.7 更新プログラム . . . 30
4.7.1 更新プログラムを構成する要素 . . . 31
4.7.2 更新プログラムの自動作成 . . . 33
4.7.3 適切なプログラム更新を設計する補助 . . . 34

第 5 章　実装 **36**

5.1 言語処理系への機能の追加 . . . 36
5.1.1 TUTScheme のシステム . . . 37
5.1.2 Scheme データの内部表現 . . . 40
5.1.3 組み込みグローバル変数の追加 . . . 42
5.1.4 ソフトウェアの更新処理を進める処理の追加 . . . 44
5.1.5 ごみ集めへの処理追加 . . . 52
5.1.6 プリミティブ関数の追加 . . . 55
5.1.7 その他の処理追加 . . . 56
5.2 更新プログラム作成ツールの実装 . . . 58
5.2.1 更新プログラムの構成 . . . 58
5.2.2 更新プログラムの生成 . . . 61
5.2.3 ソースコードの差分検出 . . . 62
5.3 利用例:更新対象データの置換 . . . 63
5.3.1 クロージャの置換 . . . 63
5.3.2 構造体の置換 . . . 66

第 6 章　評価 **70**

6.1 ソフトウェアへの DSU の適用 71
6.1.1 無限ループ処理を行うソフトウェアの更新 71
6.1.2 超循環評価器 73
6.2 通常実行時のオーバヘッド 78
6.3 データの更新中におけるオーバヘッド及び停止時間 78
6.4 考察 81

第 7 章 結論 **84**

付録 A 超循環評価器の更新に用いた更新プログラム **88**

図 目 次

2.1 適切な動的ソフトウェア更新の状態遷移 6
2.2 DSU 機構の利用モデル 9

4.1 一括でのデータ置換と漸次的なデータ置換 19
4.2 アクセスバリアの挿入 20
4.3 更新前のバージョンの関数が更新後のバージョンの関数を呼び出した際の関数呼び出しエラー 22
4.4 student リスト構造のコンストラクタ 24
4.5 クロージャの環境 28
4.6 分岐構文の追加 34

5.1 TUTScheme の実行スタック 39
5.2 データスロットの形式 41
5.3 クロージャの内部表現 41
5.4 構造体の内部表現 42
5.5 更新処理の進み方 45
5.6 `xcall` に追加された処理 46
5.7 実行スタックへのフレームの追加 47
5.8 集められた更新対象データの置換 50
5.9 コピー GC で行う更新対象データの発見と weak リストへの格納 . . 53
5.10 更新対象の探索と置換を行う処理 55
5.11 生成される `student` 構造体のデータの内部表現 57
5.12 `*dsu-object-checklist*`に格納されるリストを組み立てるプログラム 60

5.13 TUTSchemeの更新対象データの置換を行う関数群 60
5.14 cond式から入れ子になったif式への変形 63
5.15 クロージャを生成するコンストラクタ 64
5.16 クロージャの内部表現 65
5.17 置換されたクロージャの内部表現 67
5.18 クロージャの置換を行う関数 67
5.19 置換処理後のstudent構造体のデータの内部表現 68
5.20 クロージャの置換を行う関数 69

6.1 無限ループ処理 72
6.2 C言語における無限ループ 72
6.3 ループ抽出 73
6.4 両バージョンにおけるフレーム構造の構成 75
6.5 フレームの置換を行う関数 76
6.6 実験に用いたtak関数 77
6.7 実験に用いたコンストラクタ 79
6.8 実験に用いたmainloop関数及び補助関数 81

表 目 次

6.1 評価に用いた計算機の性能諸元 71
6.2 超循環評価器の性能測定実験結果 77
6.3 通常実行時のオーバヘッド測定実験結果 78
6.4 データの更新によるオーバヘッド測定実験結果 80

第 1 章

序論

計算機上で動作するソフトウェアは，不具合の修正や機能の追加のために，常に新しいバージョンに更新することが求められる．しかしながら，ソフトウェアの更新は一般的に，更新の対象となるソフトウェアが実行されていない時にしか行えない．ソフトウェアを更新するためにその実行を一旦終了させ，再始動することは，手間やコストがかかる．特に，ネットワーク上でのサービスを提供するサーバソフトウェアなど，非常に高い可用性を要求されるソフトウェアは，停止させることに多大なコストがかかるか，停止させることができない．

動的ソフトウェア更新 (Dynamic Software Updating, DSU) 機構は，ソフトウェアの手続きやデータを変更し，実行中のソフトウェアの実行状態を，新しいバージョンのソフトウェアの同等の実行状態に移行する．これを用いることで，高い可用性を要求されるソフトウェアの修正および機能追加が可能になる．

DSU 機構によって更新が行われたプログラムは，正常に動作し続けることが求められる．そのためには，更新されたソフトウェアの実行状態が，ソフトウェアが新しいバージョンとして始動し，同じ入力を受けていた時と同じ実行状態に移行できることが十分条件となる．これまでに多くの動的ソフトウェア更新機構が提案され [5, 6, 7, 8]，動的に適用できるソフトウェア更新の領域が広げられてきた．しかし，これらの機構のほとんどは，C や Java のような静的型付けプログラミング言語で記述されたソフトウェアを対象としている．

また，プログラムの実行性能等が，動的更新を行うための機構によって影響を受

ける度合いを最小限に抑えなければならない．すべてのデータを一括で更新するために，特定のクラスのすべてのインスタンスを参照する配列を用いる機構では，メモリ領域を余分に消費する．また，ソフトウェアのソースコードに更新処理を行う関数の呼び出しを追加することを要求する機構では，更新処理を行う関数の呼び出しが頻繁に行われ，ソフトウェアの実行性能が低下する．

また，広大なメモリ領域を実行に用いるソフトウェアでは，ソフトウェアの扱うデータ構造を，一括で新しいバージョンのデータ構造に置き換える際に，長い間ソフトウェアの実行が中断される．DSU は，主に多数のユーザにサービスを提供するソフトウェア，及び高可用性が求められるソフトウェアを主な対象として想定している．しかし，ソフトウェアの実行が中断されている間に多くの問い合わせがあった場合には待ち行列の長さが伸び，ソフトウェアの応答性を損ねる．または，ソフトウェアの実行が僅かな時間中断されることが，致命的な事態につながる場合が考えられる．

そこで本研究では，新しいバージョンのデータ構造への変換を漸次的に行う DSU 機構を設計，実装した．この機構では，データ構造の変換による，ソフトウェアの実行の停止される時間を一定時間内に抑えることができる．この機構では，変換を要する全てのデータ構造を一括して変換する代わりに，更新されるソフトウェアの実行と並行して，一度に一定数ずつデータ構造を変換する方式を取る．この方式では，ソフトウェアの実行は，更新前のバージョンのデータと，更新後のバージョンのデータが混在する状態の中で進められるため，更新後のバージョンのソフトウェアが更新前のバージョンのデータの内部を参照することによる不具合を防がなければならない．本研究で実装した機構では，データの内部への参照，変更には必ず特定の関数が用いられる前提の上で，それらの関数の処理の中で，関数の引数となるデータを更新後のバージョンのデータに置き換えるようにすることで，この問題を解決する．

本研究で実装した DSU 機構は，Scheme プログラミング言語 [1] で記述されたソ

フトウェアを対象とする．Schemeでは，プログラムをリスト構造の編集によって自在に組み立てることができる．この特徴は，プログラムのソースコードの比較を行うツールを作成することを容易にする．また，Schemeの言語機能上の特徴として，動的型付け言語であること，第一級関数をサポートしていることが挙げられる．これらの言語機能は，現在Webアプリケーションの開発に使われることの増えている，Perl, Python, Ruby, JavaScriptなどの動的言語(Dynamic Languages)と共通している．

Webアプリケーションは，セキュリティや機能の追加のための修正が頻繁に行われている一方で，高い可用性が求められる．そのため，WebアプリケーションはDSUが大いに貢献できる分野であると考えられている[2]．また，最近のWebアプリケーションには，継続ベースWebアプリケーション[3]のような，サーバ上で常駐して実行されるアプリケーションも存在する．これらのアプリケーションは，CGIスクリプトによるWebアプリケーションとは異なり，スクリプトを置き換えて再読み込みを行うだけで更新を行う事ができない．これらのアプリケーションには，主にSchemeやRubyなど，継続をサポートする動的言語が用いられる．このようなWebアプリケーションには，動的言語を対象とするDSU機構が大いに貢献する可能性がある．

本研究のDSU機構は，Schemeプログラミング言語の処理系であるTUTScheme[4]のバイトコード命令，およびごみ集め(GC)処理に，DSUを行う処理を追加することで実装した．

本論文の構成を以下に示す．2章では，DSU機構が必要とされる背景および，これまでに開発されたDSU機構の一般的な仕組みを説明する．3章では，これまでに開発されたDSU機構を個々に取り上げ，その特徴と問題点を議論する．4章では，2，3章で議論した問題点をもとに，本研究で実装したDSU機構の設計方針および，その実現のために必要な機能を取り上げる．5章では，本研究でDSU機構を実装したTUTSchemeの紹介を行い，4章で述べた設計を，動作するプログラム

に実装するための方法を説明する．6 章では，5 章で実装した DSU 機構の実行性能，停止時間を測定し，その結果を議論する．最後に 7 章で，結論と今後の指針を述べる．

第 2 章

背景

本章では，DSU 機構の登場した背景を詳説し，従来の DSU 機構の典型的な構成および利用形態を説明する．

2.1 動的ソフトウェア更新機構

ソフトウェアを実行時に更新する個別の機能は，古くから提供されてきた．例えば，ネイティブコードにコンパイルされたソフトウェアでは，実行中のプログラムに対し，その命令列を直接変更する「パッチを当てる」処理が用いられてきた．また，動的言語においては，ソフトウェアが実行中に新しいモジュールを読み込むこともできる．しかし，これらの機能を単一で用いるだけでは，動作中のソフトウェアを安全に新しいバージョンに更新する事ができない場合がある．

また，今日ではソフトウェアの巨大化のため，ソフトウェアの更新の要する変更処理は，非常に多数の処理内容を変更するものになる．さらに，計算機システムを用いた高可用性システムの利用も広がっている．また，今日では計算機上で実行されるソフトウェアは，複数のスレッドを扱うなど，極めて複雑な動作をするようになり，多様なデータを扱っている．そのため，多様なソフトウェアの，多様な更新内容に対応できる，汎用的な動的ソフトウェア更新機構が求められている．

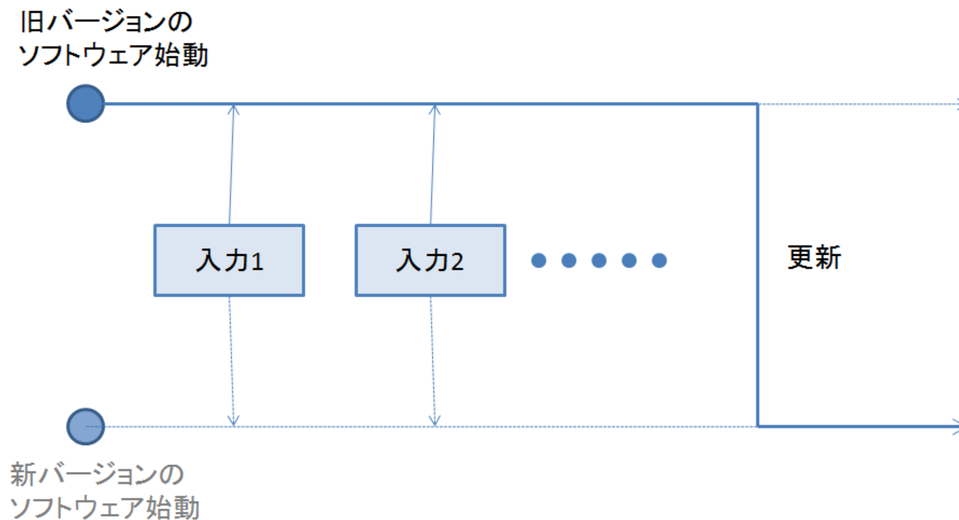

図 2.1: 適切な動的ソフトウェア更新の状態遷移

2.1.1 動的ソフトウェア更新に求められる特性

本節では，DSU 機構に求められる特性として挙げられる 3 つの特徴について述べる．

多様なソフトウェア更新への対応

今日では，分散環境で動作するソフトウェアが増えており，並列ソフトウェアに対応した DSU 機構 [5] も登場している．

近年では，より抽象度の高いプログラミング言語で記述されたソフトウェアが増加している．これまでの DSU 機構では，C，C++や Java のような静的型付け言語で記述されたソフトウェアを主な対象としており，動的型付け言語で記述されたソフトウェアの動的更新を行う機構の例は少ない．本研究では，動的型付け言語である Scheme で記述されたソフトウェアを対象とする DSU 機構を実装した．

適切な更新が行えること

適切な更新とは，更新されたソフトウェアが，最初から更新後のバージョンで動作しているプログラムが同じ入力を受けていたのと同じ状態に到達することとする．図 2.1 では，古いソフトウェアとして始動され，外部からの入力を複数回受けた後に更新された際に，新しいバージョンが始動され，同じ入力を受けたのと同じ状態に移行する模式図を示している．この状態に移行できることは，更新されたソフトウェアが正常に動作し続けることの必要条件である．

動的更新されたソフトウェアが，更新処理によって，ソフトウェアの開発者が想定しない状態に移行した場合，例えば実行されるはずのない処理が実行されることやデータに含まれる値が許容されない値をとることがあれば，ソフトウェアが開発者の意図しない動作を行うこと，または異常終了することが起こり得る．このことはセキュリティ上の危険性や，事故につながる恐れがある．

ソフトウェアの実行への影響の小さいこと

DSU をサポートする機能は，ソフトウェアの実行に影響を及ぼすことがある．ここでの「ソフトウェアの実行への影響」には，ソフトウェアの実行時のメモリ使用量の増大および処理の実行速度の低下と，ソフトウェアの実行の一時停止がある．

関数の差し替えや，データ構造へのフィールドの追加を実現するため，DSU をサポートするソフトウェアは，DSU をサポートしないソフトウェアよりもいくらか余分に複雑な構造を持っていることがある．また，データの置換のために行われる処理がソフトウェアの実行に割り込んで行われることもある．以上の特徴から，DSU をサポートするソフトウェアの実行性能 (処理速度，メモリ使用量) は，DSU をサポートしないソフトウェアに劣ることがある．DSU をサポートするソフトウェアの，更新を行っていない通常実行時におけるソフトウェアの実行性能は，DSU をサポートしないソフトウェアと全く同等であることが望ましい．

また，DSU の対象となるソフトウェアによっては，更新処理によるソフトウェ

アの処理の停止時間を小さく抑える必要がある．例えば，リアルタイム性を保証するソフトウェアの処理が，更新処理によって，許容できる以上に長い間停止されてはならない．スループットを重要視する，膨大な量のトランザクションを処理するソフトウェアでも，数秒でもアプリケーションが停止すれば，その間に問い合わせキューが伸び，その後しばらくの間，長い問い合わせキューのために負荷が高い状態が続くことが想定される．

2.2 動的ソフトウェア更新機構の利用モデル

従来の DSU 機構の，典型的な利用モデルを図 2.2 に示す．前提として，ソフトウェアの開発者は，更新対象のソフトウェアの更新前，更新後の両バージョンのソースコードを保有しており，更新対象のソフトウェアを更新するために必要なことをあらかじめ知っているものとする．また，本研究では，ソフトウェアの不具合への自動での対処することは行わない．更新前後のソフトウェア，および更新プログラムには，ソフトウェアが正常な動作を続けられなくなるような不具合が無いことを前提とする．

DSU 機構を用いてソフトウェアの更新を行う場合，ソフトウェアに更新プログラムを読み込ませることが必要である．更新プログラムは，新しいバージョンの関数などの定義および，データの置き換え処理を行う関数，およびその他にソフトウェアの更新を行うために必要な情報を含んでいるプログラムである．更新プログラムが作成され，更新対象のソフトウェアが更新されるまでの流れを以下に示す．

1. まずソフトウェアの開発者が，更新対象のソフトウェアの更新前と更新後のバージョンのソースコードを更新プログラム作成ツールに読み込ませて，更新プログラムを自動生成する．開発者はこの更新プログラムに，修正を加えることもある．

2. 開発者は作成した更新プログラムを，更新対象のソフトウェアの利用者に配

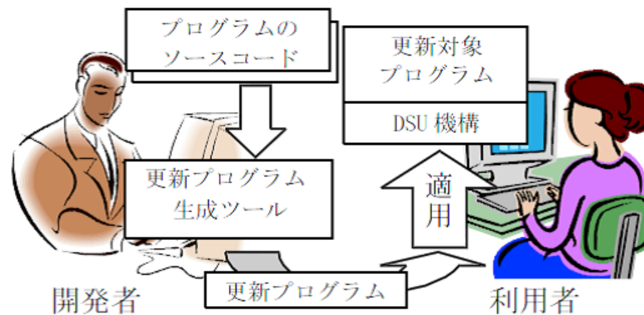

図 2.2: DSU 機構の利用モデル

布する．更新プログラムを入手した利用者は，ソフトウェアに対し，更新を指示する．

3. ソフトウェアは更新の指示を受けると，更新プログラムを読み込み，ソフトウェアの実行状態を，新しいバージョンのソフトウェアの対応する状態に移行させる．

2.2.1 更新プログラムの作成

更新プログラムは，対象となるソフトウェアの動的更新を行う上で必要な処理などの情報を含んでいる．具体的には，変更・追加される関数やデータ構造の定義，および変更される関数名のリスト，データの置換手続きが含まれる．

更新対象のソフトウェアの開発者は，更新プログラム作成ツールを用いて，更新プログラムを作成する．

更新プログラム作成ツールは，更新対象のソフトウェアの更新前のバージョンと更新後のバージョンのソースコードを入力として受け，ソースコード内の各々の定義にバージョン間での差異があるかを調べ，更新プログラムを自動生成する．

更新プログラム作成ツールは，開発者が更新プログラムを作成するための補助を行うためのものであり，必ずしも適切な更新が行える更新プログラムを生成するとは限らない．更新の種類によっては，更新プログラム作成ツールが自動生成した

更新プログラムを，そのまま動作中のプログラムに適用した場合，適切な更新が行えない場合がある．このような場合には，開発者は，自動生成された更新プログラムを改変しなければならない．

2.2.2 ソフトウェア更新の適用

ソフトウェアの更新処理は，ユーザがソフトウェアの実行環境に対して更新を指示することで開始される．実行環境は指示を受けると，更新プログラムを読み込み，その内容に従ってソフトウェア更新の各手順を実行中のプログラムに適用する．

第 3 章

関連研究

本章では，これまでに開発された DSU 機構を紹介し，その特徴について議論する．

3.1 C 言語における DSU 機構

Python や Javascript などの動的言語，および Java の一般的な言語処理系では，ソフトウェアのソースコードは，実行されるハードウェアの機械語命令列 (ネイティブコード) とは異なる，処理系または言語に独自の命令列 (バイトコード) にコンパイルされる．バイトコードは仮想機械と呼ばれるソフトウェアに読み込まれて，初めてハードウェア上で実行される．

一方，OS やハードウェアドライバなど，ハードウェアの抽象化を行うソフトウェアや，実行速度が第一に優先される分野のソフトウェアは，依然としてネイティブコードへのコンパイルを行う処理系を用いている．ネイティブコードで実行されるソフトウェアに DSU 機能を付加するため，コンパイルの前の段階で，ソースコードに DSU の機能を呼び出す処理を追加し，各関数呼び出しおよび構造体の参照の処理を DSU に対応する形に変換する．変換されたソースコードは変換前に比べて，関数呼び出しの回数が増加しているため，DSU をサポートしないコードよりソフトウェアの実行速度は遅くなる．

Neamtiu らは，C で記述されたソフトウェアのための DSU 機構 Ginseng [6] を開

発した．Ginseng では，ソフトウェアのソースコードを改変し，関数の呼び出しをすべて関数ポインタを用いて行うようにし，また構造体にバージョンを表すフィールドおよびパディング領域を設ける．これは，将来の更新での関数の変更や，構造体の定義の変更に対応するためのものである．また，構造体の内部へのアクセスは各々の構造体の専用のアクセサを用いるように変更される．これは，新しいバージョンのソフトウェアが古いバージョンの構造体にアクセスした時に対象となる古い構造体を新しいバージョンの構造体に変換するために用いられる．

この機構によってコンパイルされたソフトウェアは，各構造体に設けられたパディング領域により，通常の gcc コンパイラでコンパイルされたものに比べ，最大で 4 割余分なメモリ領域を使用する．また，更新処理を行っていない通常実行時も構造体へのアクセスには更新実行時と同じアクセサを用いているので，通常実行時におけるオーバヘッドが残る．

3.2 Java における DSU 機構

近年では，Java のような仮想機械を用いるプログラミング言語で記述された業務アプリケーションが増えている．また今日までに，Java で記述されたソフトウェアを更新するための，多くの DSU 機構が提案された．

Orso らは，DSU をサポートするための特別な仮想機械を用いずにクラス階層の変換によって DSU をサポートする機構 DUSC[7] を開発した．これは，Java で記述されたソフトウェアのクラスを変換し，変換される前のクラス 1 つに対して，4 つのクラスの定義を生成する．この 4 つのクラスは，変換される前のクラスの実装を表すクラス，クラスのメソッド呼び出しのためのインターフェースを提供するクラス，クラスの更新を行うための，メソッドの呼び出し状況を管理し実装を表すクラスのすべてのインスタンスを参照する配列を保持するクラス，クラスの更新時にインスタンスの状態を保存するためのクラスである．この 4 つのクラスへの変換

によって，クラス間で行われるメソッド呼び出しの意味論を保全しつつ，各クラスの実装を置き換え可能にしている．

しかし，DSU をサポートするためのクラスを新たに作ったために，DSU をサポートしないソフトウェアに比べ，最大で 2 割近く余分なメモリ領域を使用する．また，クラス階層の変換を行うためにリフレクションなど，Java の言語機能のいくつかは用いることができない．その上，定義の変更されたクラスのインスタンスの更新は一括で行われるため，ソフトウェアが非常に多数のオブジェクトと広大なヒープ領域を用いる場合，次節で述べるような，停止時間の問題も起こり得る．

3.3 専用の仮想機械を用いる DSU 機構

仮想機械を用いて実行されるソフトウェアは，仮想機械の各命令を解釈し，各命令に対応する処理を実行する．これらの処理に DSU のための処理を付け加える事によって，ソフトウェアのコードを全く変更せずに DSU を実装することができる．この方式は，ソースコードに全く変更を加えず，その上，ソフトウェアの更新を行っていない時には，DSU のための機能を一切利用しないため，ソフトウェアの実行時間のオーバヘッドをほとんどもたらさない DSU 機構を設計することが可能になる．さらに，更新を行った後に，JIT コンパイラを用いて更新後のソフトウェアのコードを最適化し，ソフトウェアの実行を高速化することもできる．

Subramanian らは，Java ソフトウェア向け DSU 機構 JVOLVE [8] を開発した．これは，Java 仮想機械 Jikes RVM の JIT コンパイラおよびコピーごみ集め処理に変更を加えた専用の仮想機械を用いている．JVOLVE は，クラスの定義を更新した後，コピーごみ集めを行い，コピーごみ集めに追加された処理によって定義の変更されたクラスのインスタンスを置換する．置換処理はコピーごみ集めを行う時に，定義の変更されたクラスのインスタンスに対して，新しいバージョンのクラスのインスタンスをコピーごみ集めの複製先に作成する事で行う．仮想機械の上で動

作するソフトウェアに特別な変形を施さず，ソフトウェアの実行性能とメモリ使用量を悪化させずに実際のサーバソフトウェアの更新を行える DSU 機能を実現している．

しかし，インスタンスの置換は一括で行われるため，その上この置換処理がごみ集めの直後に行われるために，プログラムの実行が長時間中断される．この中断時間は，更新対象のソフトウェアの用いるオブジェクトの数によっては 1 秒以上にもなる．今日の基幹業務システムで用いられる Java アプリケーションでは，ソフトウェアの実行を停止させてヒープ全体に対してごみ集めを行う Stop-the-world ごみ集めによる停止時間が問題となっており，その停止時間がオブジェクトの更新処理によってさらに長くなると，業務の遅滞時間が延び，業務上の損失が大きくなることが考えられる．

第 4 章

設計

本章では，本研究で実装した DSU 機構の設計方針及び，DSU を行う処理の流れ，および DSU を行う各機能の設計について述べる．

本章の構成は以下のとおりである．まず，4.1 節で一般的な DSU 機構による更新処理について述べ，4.2 節で DSU 機構の主な問題点と，それらに対処する方針を述べる．4.3 節では，本研究で提案する漸次的なデータ置換の方式を説明する．4.4 節では，グローバル変数などの定義の置き換えを行う手続きについて述べる．4.5 節では，Scheme 言語のデータの置換を行うための前提条件を議論し，個々のデータの置換を行う方法を示す．4.6 節では，ソフトウェアの更新処理の状態の管理について述べる．最後に，4.7 節で，更新プログラムおよび，更新プログラムを自動生成するツールの設計について述べる．

4.1 ソフトウェア更新の実行

一般的な DSU 機構は，ソフトウェアの状態を新しいバージョンのソフトウェアの状態に移行するため，グローバル関数やグローバル変数，構造体などのグローバルな定義と，ソフトウェアの変数に束縛された構造体などのデータを，すべて新しいバージョンのものに更新する．

DSU 機構は，グローバル変数や構造体などの定義を新しいバージョンのものに置き換えた後に，構造体などのデータを更新する．これらのデータは，ソフトウェ

アのグローバル変数とローカル変数から直接，または間接的に参照されている．新しいバージョンとして実行の再開されたソフトウェアによってこれらのデータが参照されるまでの間に，データの値を新しいバージョンのものに更新する．

一般的な DSU 機構では，以下の流れに沿ってソフトウェアの更新を行う．

1. 開発者は，更新プログラムを生成するツールを用いて，更新対象のソフトウェアの新旧バージョンのソースコードから更新プログラムを作成し，(修正が必要な場合は更新プログラムを修正し，) 更新対象のソフトウェアの利用者に配布する．
2. ソフトウェアの利用者が，言語処理系にプログラムの更新を指示する．
3. 2. の直後に，ソフトウェアは更新プログラムの読み込みを行う．
4. 更新プログラムが読み込まれた後，新しいバージョンで変更されたグローバル定義を更新し，新しいバージョンで追加されるグローバル定義を追加する．
5. グローバル定義の更新後，更新前後のバージョンで，定義が変更されているデータ構造がある場合，古いバージョンのソフトウェアで生成されたデータを，新しいバージョンのソフトウェアが扱うデータに置換する．

本研究で実装する DSU 機構でも，大まかな更新の流れは上記と同じである．

4.2 DSU 機構の設計方針

4.2.1 本研究で対象とするソフトウェア更新

本研究では，ソフトウェアの実装の枠組み，すなわち個々の関数で行うべき役割を保ちながら，関数で行う処理の一部を変更する更新や，いくつかの新しい定義を追加する程度の更新を対象とする．この類の更新には，分岐処理の追加，抽象

データへのフィールドの追加，および関数の中で用いるアルゴリズムの改良が含まれる．

一方で，ソフトウェアの設計段階からの大規模な見直しを行い，ソフトウェアの実行の流れが大幅に変わっているような更新は適用できない．また，ソフトウェアの制御スタックおよび継続 (continuations) の中の制御情報を変更できないため，保存された継続をいつまでも呼び出さないようなソフトウェアの更新や，ソフトウェアが実行されている間，常に実行されている関数を変更する更新も行えない．

4.2.2 設計方針

本研究で実装する DSU 機構の主な設計方針として，以下の 3 つを挙げる．

- **Scheme プログラミング言語における，抽象データの置換のサポート**
  本研究では，Scheme プログラムにおける抽象データを適切に置き換えるための機能を提供する．抽象データは，必ずそのデータ型に固有の関数 (コンストラクタ) によって生成され，特定の関数群 (アクセサ) によって参照が行われるため，コンストラクタに変更があるかを見て，その抽象データの定義に変更があるかを判別できる．Scheme プログラムでは，抽象データを実装するデータ構造として，リストおよびクロージャが主に用いられる．本研究では，これらの抽象データを用いるソフトウェアを適切に動的更新するため，抽象データを実装するリスト及びクロージャの置き換えを行うアプローチを提供する．また，多くの Scheme 言語処理系では，Scheme の標準仕様に最近まで含まれなかった構造体の機能を独自に提供しており，Scheme プログラムで構造体を扱うことへの要求が高いことが窺えるため，本研究では TUTScheme の構造体の置換をサポートする．

- **データの置換処理によるソフトウェアの停止時間の低減**
  本研究では，ソフトウェアの更新時における，ソフトウェアの応答性を向上

させるため，ソフトウェアの更新における，データの置換処理による中断時間に着目した．この時間は，ヒープ領域中に存在するデータの数に比例した時間がかかる．非常に多くのデータを扱うソフトウェアの更新を行うとき，データを置き換える処理によって，ソフトウェアが非常に長い時間停止してしまう．そのため，本研究では，データの置き換え処理を，更新されたソフトウェアの実行と並行して行う方式を提案する．

- **DSU をサポートする機構による実行性能の低減抑制**
  本研究では，データの置換処理を行うために特定の関数に追加される処理を，データの置換処理を行わない時には撤去されるようにし，これらの処理の追加による実行性能の低減を抑制する．

4.3 データ置換処理の分割化

本研究では，ソフトウェアのデータの置換を，ソフトウェアの実行と並行して行う方式を提案する．この方式では，ソフトウェアのグローバル定義などの更新後，データの置換を一括で行う代わりに，ソフトウェアの実行と並行して，データの置換処理を，定期的に，一定数のデータごとに行う．この処理を**漸次的なデータ置換**と呼ぶ．

漸次的なデータ置換を行うためには，まず置換を要するデータを発見する必要がある．この処理は，ごみ集め (GC) を行う間に行うことで，置換を要する全てのデータを発見することができる．発見されたデータは，置換を要するデータのリストから参照され，GC の後，ソフトウェアの実行の間に，一定個数ずつに分けて置換される．この方式では，1 回の GC で全ての置換を要するデータを見つける必要はない．GC を行うごとに一定個数ずつ，置換を要するデータを集めてもよい．

従来のデータの置換を一括で行う場合と，この方式を用いた場合でのソフトウェアの更新処理の進み方を図 4.1 に示す．更新対象のソフトウェアが本来行う処理が

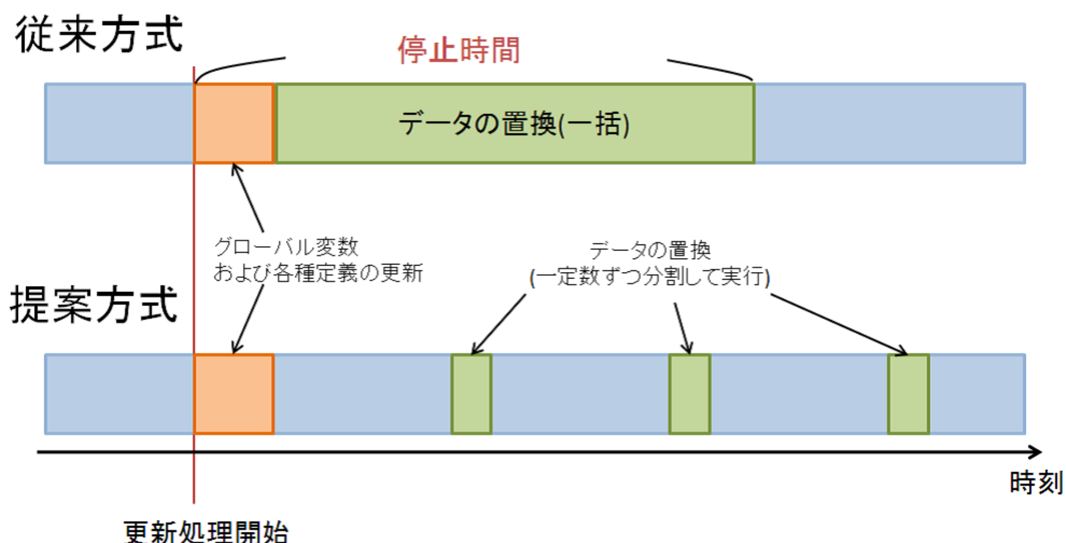

図 4.1: 一括でのデータ置換と漸次的なデータ置換

行われているのは，図中の薄い青色で示した区間である．従来の一括でデータを置換する方法を用いる場合では，更新処理の開始時点からしばらくの間，更新対象のソフトウェアが一切の処理を行えなくなる．一方，提案方式では，データの置換によるソフトウェアの停止時間を，一定時間以内に抑えることができる．

この方式では，データの置換が一括で行われない．そのためソフトウェアの実行は，新しいバージョンのソフトウェアの扱える新しいデータ (新データ) と，古いバージョンのソフトウェアの生成した，古いデータ (旧データ) が混在する中で進められることになる．新しいバージョンのソフトウェアが旧データの内部を参照することによる不具合を防ぐため，置換されるデータが，その内部への参照が専用の関数 (アクセサ) を通して行われる抽象データであることを前提として，それらの関数に，引数が旧データであるかを判定し，引数が旧データであればこれを新データに置換する処理 (アクセスバリア) を追加するようにした．

ソフトウェアの更新処理において，ソフトウェアのグローバル関数やデータ型などの定義が更新されたとき，変更されたデータ型に対応する各アクセサおよび，引数のデータが特定のデータ型であるか判定する関数 (データ型判定述語) にアク

```
(define (account? x)          (define (account? x)
  <exps1>)                      (define (old-account? x)
                                  <exps1>)
                                (define (new-account? x)
                                  ...)
                                (cond ((new-account? x) #t)
                                      ((old-account? x)
                                       (update-account x) #t)
                                      (else #f)))
```

図 4.2: アクセスバリアの挿入

セスバリアが挿入される．アクセスバリアをデータ型判定述語 `account?`に挿入した例を図 4.2 に示す．

アクセスバリアは，まず新しいバージョンのデータ型判定述語を引数のデータに適用し，データが新データであるかを判定する (図中の `(new-account? x)`)．データが新データであれば，アクセスバリアの挿入された関数の本来の処理を行う．引数のデータが新データでなければ，このデータが旧データであるかを判定する (図中の `(old-account? x)`)．この判定には，更新前のデータ型判定述語と同じ処理 (図中の`<exps1>`) が用いられる．データが旧データであれば，これを新データに置換し (図中の `(update-account! x)`)，アクセスバリアの挿入された関数の本来の処理を行う．データが新データでも旧データでもなければ，アクセスバリアの挿入された関数の本来の処理を行う．

2.1.1 節で述べたとおり，通常実行時では，アクセスバリアによるオーバヘッドを取り除くことが必要である．データの置換が完全に終わった際には，アクセスバリア内の，引数が旧データであるかの判定処理は不要になる．しかしアクセスバリアを挿入されたアクセサをそのまま用いる場合では，アクセスバリアの判定処理を行うために追加の処理時間がかかる．

本研究で実装した機構では，こうした状況下での処理時間の節約のため，GC を実行した際に旧データが 1 つも見つからなかった場合，データの更新が完了したとみなす．その際に，次の更新を受け付けられるようにするのと同時に，アクセスバ

リアを含む関数を新データのみに対応した関数に置き換える．これを行うことで，ソフトウェアの通常実行時における，アクセスバリアによるオーバヘッドを回避できる．

4.4 グローバル定義の更新，追加

グローバル定義の更新は，更新対象の関数すべてが実行されていないときに一括して行う．本研究で更新するグローバル定義は，関数を含むグローバル変数の定義，マクロの定義および構造体の宣言である．

更新されていない古いバージョンの関数が，更新された新しいバージョンの関数を呼び出す場合，図 4.3 に示すように，更新された関数の引数の個数が異なるために，この処理が実行時エラーにつながる場合がある．このような事態を防ぐため，関数の更新は全て一括で行う．本研究で実装した DSU 機構は，実行中の命令を参照するプログラムカウンタを，新しいバージョンのソフトウェアの対応する命令を参照するように書き換えることは行わない．実行中の関数を置き換える事ができないので，関数の更新は，その関数が実行されていない時に行う．

実行中の関数が更新された場合，その関数が次に呼び出される時から新しいバージョンの関数が実行される．しかし，実行中の関数は古いバージョンの関数として実行が進められる．この場合，実行中の古いバージョンの関数が，更新された新しいバージョンの関数を呼び出すことが考えられる．この時，関数呼び出しの際に引数の個数が関数の定義に合わないため，実行時エラーが発生することが考えられる．さらに，各関数の更新が行われても，更新が行われる関数がいつまでも実行中である場合，その関数の実行はいつまでも古い関数として続けられる．この関数から，更新された関数を呼び出す際に問題が起こる事が考えられる．更新された新しいバージョンの関数は，更新後の新しい関数から呼び出される事を前提に設計されている．そのため古いバージョンの関数から呼び出される場合，不具合が発

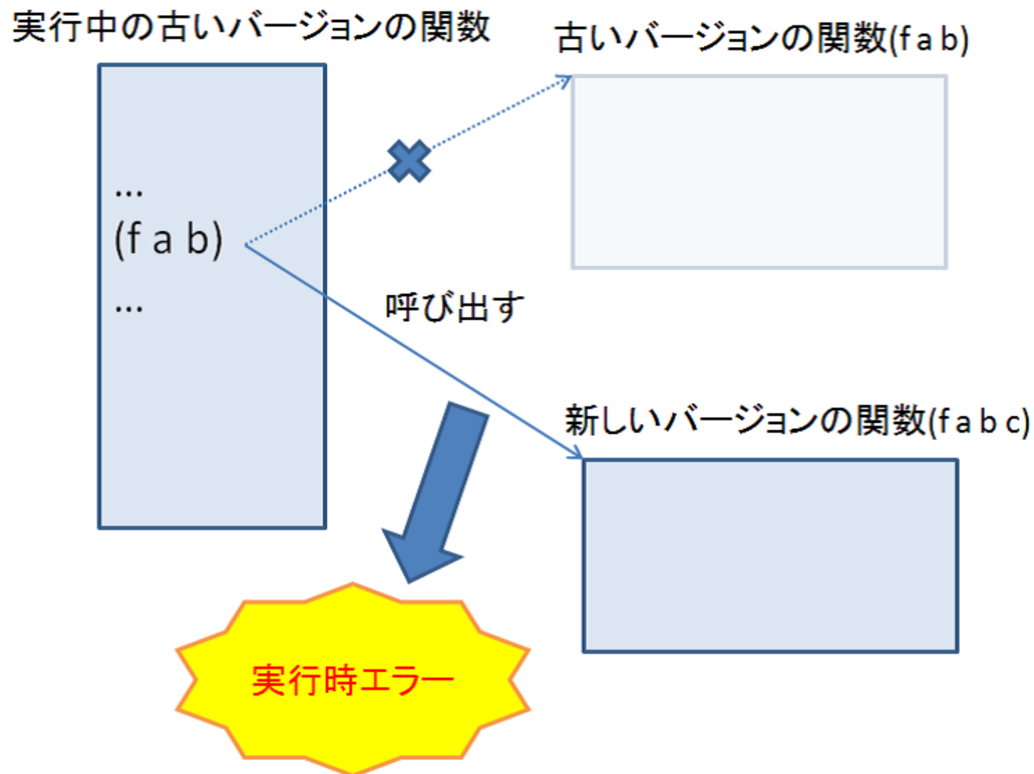

図 4.3: 更新前のバージョンの関数が更新後のバージョンの関数を呼び出した際の関数呼び出しエラー

生することが考えられる．

このため，本研究の DSU 機構では、ソフトウェアの実行スタックから，更新される関数のスタックフレームを探し，更新される関数のスタックフレームが実行スタックに無いことを確認した場合にのみグローバルに定義された変数を更新する．

4.4.1 アクセスバリアの追加

データの置換を行う段階に入る前に，すなわちグローバル変数の更新と同時に，変更されるデータ型に対応するアクセサ及びデータ型判定述語に，アクセスバリアを導入する．

アクセスバリアを挿入する対象となる関数は，更新前後で変更されるコンストラ

クタで生成されるデータ (更新対象データ) の内部を参照するためのアクセサ，および，更新対象データであるかを判定する関数 (データ型判定述語) である．これらの関数は更新対象データの内部を直接操作するため，それらが本来の処理を行う前に，データの置換を行う．また，これらの関数が扱うデータは抽象データであり，上に挙げた関数以外の処理がデータの内部を参照することはない．よってこれらの関数にのみアクセスバリアを加えることで，新しいバージョンのソフトウェアが古いバージョンのデータの内部を直接参照することはなくなる．

アクセスバリアは，渡されたデータが新データであるか，旧データであるかを判別する必要がある．そのため，アクセスバリアの導入されたデータ型判定述語の他に，更新対象のソフトウェアの，更新前と更新後のバージョンのそれぞれの元々のデータ型判定述語を参照できるようにする必要がある．そこで，データ型判定述語`<データ型名>?`に対し，グローバル変数`inner-<データ型名>?`に更新後のデータ型判定述語を，`old-<データ型名>?`に更新前のデータ型判定述語を束縛する．

4.5 データの置換

本節では，本研究で実装した DSU 機構が行うデータの置換処理について，置換処理を行うための前提条件を述べ，その次に置換処理を行う具体的な方法を述べる．

4.5.1 前提とする条件

本研究の DSU 機構でデータの置換を行えるように，更新対象のソフトウェアにコーディング規約を設ける．

まず，本研究では，置換可能なデータ構造は，抽象データを実装するリストとクロージャ，および TUTScheme の構造体の 3 種類とする．抽象データ構造には，データを生成するコンストラクタおよび，データの内部を参照するアクセサ，および引数がそのデータ構造と同じ型であるかを判定するデータ型判定述語が定義さ

```
(define (make-student name gender grade)
  (list 'student-1 name gender grade))
```

(a) 更新前のバージョン

```
(define make-student
  (let ((*id* 0))
   (lambda (name gender grade)
     (set! *id* (+ *id* 1))
     (list 'student-2 id name gender grade))))
```

(b) 更新後のバージョン

図 4.4: student リスト構造のコンストラクタ

れているものとする．

また，抽象データを生成するそれぞれのコンストラクタには，対応するデータサンプル生成関数が用意されているものとする．データサンプル生成関数は，引数を取らない関数で，コンストラクタが生成するデータと同じ型のデータを生成する．これらの関数は，4.7 節で述べる更新プログラム内の，更新対象データのリストを作成するのに用いられる．

抽象データをリストとして実装した場合，その要素は抽象データ型に固有のアクセサを用いなくとも，`car`，`cdr` などの基本関数を用いてアクセスする事ができる．アクセサを用いずにリストで実装された抽象データの内部を参照すると，参照する対象のリストの形式がバージョン間で異なる場合，ソフトウェアが意図しない動作を行うことが考えられる．例として，図 4.4 に示すプログラムを考える．更新前のバージョンで作られた student リストは，シンボル `student-1`，学生の名前，性別，学年が順番に格納される．一方，更新後のバージョンでは，学生の名前の前に ID が追加される．更新後のソフトウェアが学生の学年を取得することを意

図して，更新前に生成されたstudentリスト`s`に対して式`(cadddr (cdr s))`を評価する場合，実行時エラーが発生する．この場合では，アクセサにアクセスバリアを用いても正しい結果を得ることはできない．更新対象のソフトウェアは，抽象データの扱い方を守り，データの内部へのアクセス(参照，変更)に必ず専用のアクセサを用いることを前提とする．

本研究で扱うデータ型について

本研究で実装したDSU機構では，GCを用いて集められたデータを一定数ずつ置換する処理，およびアクセスバリアの2つの方法を用いて，データの置換を行う．

GCを用いて更新対象データを集める際には，GCが参照する各データについて，それが更新対象データであるかを判別する必要がある．Schemeの関数を用いてこの判別処理を行うには，GC時にその関数を実行する必要があり，実装が難しくなる．そこで本研究では，データの各スロットの参照の比較だけで更新対象データを判別する機構をGCに実装した．新しいバージョンのものに置換可能な抽象データは，生成されたコンストラクタのバージョン，およびデータ型を，この機構で判別できる必要がある．

本研究では，抽象データを実装するリストとクロージャ及び構造体を置換可能なデータ構造とする．このうちクロージャは，関数本体への参照を比較することで，どのコンストラクタで生成されたかを判定でき，更新対象データであるかを判別できる．構造体には，構造体の型を表すスロットが用意されているため，このスロットから参照されるシンボルが更新を要するデータのものと一致するかを調べることで構造体が更新対象データであるかを判別できる．しかし，リストで実装される抽象データには，データ型を判定する方法がない．そこで，リスト構造によって実装される抽象データ構造を置換可能なデータ構造として扱うための条件を以下に定める．

1. すべての置換可能なリストは，先頭要素に，データ型を表すシンボルを格納

する．このシンボルはリスト構造のそれぞれの型について固有である必要がある．

2. 抽象データを実装するリスト構造の先頭要素以外は，1. で述べたシンボルと同一のシンボルを格納してはならない．

4.5.2 データ型およびアクセサ，コンストラクタ，型判定述語の命名規則

本研究で実装した DSU 機構では，更新プログラムの自動生成ツール (4.7.2 節で説明) を用いて，更新プログラムを作成する．このツールは，アクセスバリアの追加処理を出力するために，データのコンストラクタおよびアクセサの定義を更新対象のソフトウェアのソースコード中から探す処理を行う．このツールがそれぞれのデータ型のアクセサ及びコンストラクタ，型判定述語を，その名前から容易に判別できるよう，これらの関数の命名規則を設ける．

まず，データ型の名前はハイフン (‘-’) を含まない，TUTScheme のサポートする任意の文字列とする．また，各データ型のそれぞれのフィールドの名前は，TUTScheme のサポートする任意の文字列とする．コンストラクタの名前は，そのコンストラクタが生成するデータ型の名前`<datatype>`を用いて，`make-<datatype>`とする．アクセサの名前は，そのアクセサが参照するデータ型の名前`<datatype>`とフィールドの名前`<field>`を用いて，`set-<datatype>-<field>!`または`get-<datatype>-<field>`とする．データ型`<datatype>`のデータ可動化を判別する型判定述語の名前は，`<datatype>?`とする．また，データサンプル生成関数の名前は，`make-<datatype>-sample` とする．

4.5.3 データの置換処理

更新対象のソフトウェアの扱う，各抽象データを生成するコンストラクタがソフトウェアの更新により変更される場合，そのコンストラクタの生成する抽象データのデータ表現も，バージョン間で異なっている可能性がある．そのため，ソフトウェアの更新を行う際にはコンストラクタの変更された抽象データを新しいバージョンのコンストラクタで生成されたものと同じデータに置換する．

しかし，この置換は，データの同一性 (identity) が保持されるよう行われなければならない．例えば，同一の構造体が変数 a,b から参照されている場合を考える．この場合，新しく作成した新データを変数 a に代入するだけでは不十分である．なぜならば，更新対象のソフトウェアは，変数 a，変数 b が同一の構造体を参照していることを前提に動作しているため，これらが別々の値を参照するようになれば，ソフトウェアの動作に不具合が生じるためである．この場合では，変数 b にも，変数 a に束縛された新データを代入しなければならない．

本研究で実装した機構では，データの置換を破壊的に行う．すなわち，置換を要するデータを上書きすることでデータの置換を行う．データの置換を破壊的に行う場合，置換を行う処理はより複雑になるが，変数への再代入処理を行う必要がなくなる．GC を行う時にデータの置換を行う方式では，同一性を保つためにフォワーディングポインタを用いることもできるが，GC を行うときにデータの置換を行う機能は，実装が難しいと考えられるため，今回は採用しなかった．

この節の以下の部分では，Scheme のクロージャと構造体の置換を行う方法を述べる．

クロージャの置換

クロージャは，関数型言語や多くの動的言語で，関数を第一級関数として扱うための構造である．クロージャは関数の命令列への参照のほかに，関数の引数など，関数についての情報を保持する．また，クロージャは，環境と呼ばれる，変数と値

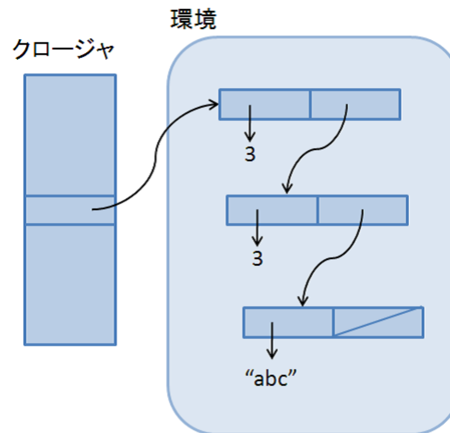

図 4.5: クロージャの環境

の対応 (束縛) の表を含んでいる．

クロージャが呼び出されたとき，クロージャから参照される関数本体が実行される．その際に，関数本体の処理内に現れる自由変数の値は，環境内の束縛を参照することで得る．

クロージャを置換するためには，関数の命令列を新しい関数のものに差し替え，その上で環境の各束縛を変更する．また，新しいクロージャの環境に変数の束縛が追加されている場合，クロージャの環境に新しい束縛を追加する．しかし，クロージャの参照する関数本体は，Scheme の第一級オブジェクトとして扱えない場合が普通である．本研究では，更新対象データのクロージャと，新データのクロージャを引数に取り，更新対象データから参照される関数本体を，新データのクロージャの関数本体に差し替える関数を用意した．また，クロージャの環境の個々の束縛を変更，追加することでクロージャの置換を行う．

クロージャの環境の更新では，クロージャの環境内の，特定の名前の変数に束縛された値を知る必要がある．たとえば，図 4.5 に示すような環境内に複数の変数があるクロージャの置換を行う際に，変数 a の値取り出す場合を考える．ここで，一般的にはクロージャの環境は，図 4.5 に示すように変数の値のみを保持しており，変数名の情報は失われている事が多い．この場合ではどの値が変数 a に束縛されて

いるかを判断できない．本研究では，クロージャの環境の置換を行う際に，環境内の変数の名前を参照できるようにするため，クロージャの環境に変数名の情報を格納するようにした．

構造体の置換

Scheme における構造体 (structure, または record) は，Scheme の最新仕様である R^6RS で，初めて言語の標準機能として正式採用された．また，以前からも Scheme の拡張仕様 SRFI 9 で Record 型の仕様が定義されているほか，多くの Scheme 言語処理系で独自に実装されている．TUTScheme では，Common Lisp の `defstruct` 構文によく似た，構造体を定義する構文がサポートされている．

TUTScheme の構造体は，専用のフォームで宣言され，その際に構造体の各フィールドを参照，変更するためのアクセサ，および構造体の型判定関数が自動的に定義される．そのため，構造体の置換は，4.5.1 節で述べた前提に依らず，簡素に行うことができる．

しかし，TUTScheme の構造体では，構造体の型を再定義する際に，構造体のフィールドの数が増えるような変更を拒絶する．また，構造体の各フィールドは連続したメモリ領域上に配置されるのが普通である．しかし，新しいバージョンの構造体がより多くのフィールドを持つ場合，追加されたフィールドの分のメモリ領域を割り当てるために，もともとの構造体のフィールドのメモリ領域と連続する領域を割り当てることができず，破壊的なデータの置換が困難になる．

本研究では，構造体の型は，バージョンごとにそれぞれ異なるようにする．具体的には，構造体の型を表すスロットに，その構造体が定義されたときのソフトウェアのバージョン番号が追加されるようにした．さらに，TUTScheme の構造体に 1 つ余分なフィールドを追加し，新しいバージョンで追加されたフィールドは，その余分なスロットから参照されるリストに格納するようにした．この方式ではメモリ領域の再割当てを行わずに，同一性を保ちながら構造体のフィールドの数を増

やすことができる．

4.6 状態の管理

本研究で実装したDSU機構では，ソフトウェアの定義やデータをすべて新しいバージョンのものに更新し終えてから次の更新を受け付けることを前提としている．動的ソフトウェア更新を行っている途中で，別の更新プログラムを読み込むと，適切な更新を行う妨げになる．

グローバル定義の更新を行わないうちにデータの置換を行うと，古いバージョンのグローバル関数が新しいバージョンのデータを参照した際に，ソフトウェアが実行時エラーを起こす場合が考えられる．

本研究では，DSUにおける処理の進み具合を管理するため，動的ソフトウェア更新の状態を管理する変数を導入した．この変数は，更新を行っていない状態，ユーザから更新の処理を受けた状態，更新プログラムを読み込んだ状態，グローバル定義の更新が行われ，データの置換を行っている状態の4つの状態を区別する．その上で，更新プログラムを読み込む処理は、更新を行っていない状態の時にのみ行うようにし，データの置換を，グローバル定義の更新が終わった後，データの置換を行なっている状態にのみ行うようにした．

4.7 更新プログラム

本節では，更新プログラムに必要となる要素を取り上げ，その各要素の内容と，動的ソフトウェア更新における役割を説明する．

4.7.1 更新プログラムを構成する要素

更新プログラムを構成する各要素を以下に示す．更新プログラムはSchemeプログラムとして記述されており，ソフトウェアの更新時に，更新対象のソフトウェアによって読み込まれる．更新プログラムの各要素は更新対象のソフトウェアの各グローバル変数に束縛され，言語処理系に追加された機能によって，DSUを行う各段階で必要なときに利用される．

更新される関数のリスト

4.4節で述べたとおり，グローバル変数は，更新される関数すべてが実行されていない間に更新されなければならない．そのため，更新プログラムには，更新される関数についての情報を含んでいる必要がある．ソフトウェアの更新時には，このリストに含まれている関数がすべて実行中でないときに限り，グローバル変数の更新を行う．

更新対象データのリスト

5.1.5節で述べる，置換を要するデータを探す処理を行うために，どのデータ型が置換を要するかの情報が必要となる．この処理はごみ集め(GC)の実行中に行われるため，データ型判定述語など，Schemeの関数を用いてデータ型を判定することができない．そのため，データ型を判定する別の方法が必要となる．

4.5.1節では，本研究で更新をサポートするデータ型の条件，およびデータ型の判定基準を述べた．GCを行う際には，これらの判定基準によって，GCで走査するデータと，更新対象データの情報とを比較し，更新対象データを判別する．更新対象データの情報の形式と，GCの実行中に更新対象データを判別する方法の詳細は，5.1.5節で述べる．

グローバル定義の更新を行う関数

4.4 節で述べたとおり，グローバル定義の更新は一括で行う．よって更新プログラムの中では，グローバル定義の更新を行う処理を，1 つの Scheme 関数として表現する．

まずこの関数では，変更される定義および追加される定義を表すローカル変数を定義する．

その次に，グローバル変数の束縛を変更する．変更される定義については，単に代入を行うだけで更新する事ができる．新しく追加される定義については，グローバル変数が定義されていないため，関数の中からグローバル変数を新しく定義するため，`eval` 関数を用いる．TUTScheme の `eval` 関数は，式の評価を行う際にグローバル環境を用いるため，`eval` 関数に `define` 式を渡すことでグローバル変数の定義を行える．

また，グローバル定義の更新を行うのと同時に，コンストラクタが変更されるデータ構造のアクセサに，アクセスバリアを挿入する．このとき，アクセスバリアの追加されていない，元々の更新前バージョン及び更新後バージョンのアクセサおよびデータ型判定述語が，新しく定義される追加のグローバル変数 `old-<関数名>` および `inner-<関数名>` に束縛される．

データの更新を行う関数

旧データを新データに更新する処理を行う．実際のソフトウェアの更新では，複数の種類の旧データが存在することもあるため，データの更新を行う関数自体は引数のデータ型による分岐処理であり，それぞれのデータ型に応じた更新関数を呼び出す．

アクセスバリアの削除を行う関数

グローバル定義の更新を行う関数で差し替えられたすべてのアクセサ，データ型判定述語を，新バージョンで定義された，アクセスバリアのない関数に差し替える．また，この関数はデータの更新がすべて完了したときに呼び出されるため，データの更新が完了したことを他のソフトウェアやユーザに通知する処理を追加することもできる．

4.7.2 更新プログラムの自動作成

更新プログラムの作成のためには，開発者が更新前と更新後のバージョンのソフトウェアのソースコードを比較しなければならない．しかし，ソースコードの分量が膨大になると，どの関数が変更されたか，どのデータ構造の定義が変更されたかを調べるのは非常に煩雑な作業になる．

そのため，更新前後のバージョンのソフトウェアのソースコードを入力とし，ソースコードの比較を自動で行い，変更のある定義を抽出し，更新プログラムを自動で作成する更新プログラム生成ツールは，更新プログラムの作成に有用である．

一方，ツールによって生成された更新プログラムがそのまま更新対象のソフトウェアに適用できるとは限らない．例えば，ソフトウェアの実行が進められる際にデータを蓄積するグローバル変数が，ツールによって生成された更新プログラムによって更新されると，それまでに蓄積されたデータが失われてしまう．このような場合，開発者が生成された更新プログラムに修正を加えなければならない場合がある．

そのため，更新プログラム生成ツールには，定義が変わった関数をユーザに通知する以外に，定義の変更がどのようなものであるか，どのような条件のもとで更新プログラムに変更を加えなくても適切な更新ができるかを示せる機能があることが望ましい．本研究では，更新前後のソフトウェアのソースコードから `if` 式の追

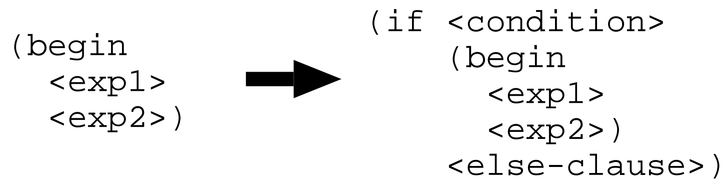

図 4.6: 分岐構文の追加

加を検出する機能を実装し，こうした条件の一部をユーザに通知するようにした．

この自動作成では，関数の定義名によって，その関数がアクセサ，コンストラクタ，型判定述語であるかを判別でき，かつその関数が扱うデータ型を識別できる事を前提とする．これは，アクセスバリアを追加するために自動でコンストラクタ，アクセサ及び型判定述語を識別するための条件である．

4.7.3 適切なプログラム更新を設計する補助

既存のプログラムのエラーチェックを強化するために図 4.6 のようにある関数の処理に if 式を追加することを考える．ソフトウェアの更新が行われる際に，図 4.6 のように関数の処理に if 式が追加される場合，新バージョンのソフトウェアは，<condition>が真である場合は旧バージョンと同じ動作をする．ここで，更新対象のソフトウェアが更新されるまでの実行では，<condition>が常に真であり，かつ<condition>の評価を行う際に副作用をもたらす処理をしていなければ，そのソフトウェアは最初から新バージョンで動作していたのと同じである．よって，この場合は単に関数の定義を差し替えるだけで適切な更新が行えると断言できる．このような更新のパターンを発見し，更新プログラムを開発者に更新のパターンと，適切な更新が行えるための条件を示すこと，およびパターンに当てはまらない更新が行われている箇所を示すことで，自動生成された更新プログラムの修正が容易になると考えた．

本研究では，このような分岐構文が追加されたパターンをソースコードの差分

から見つけ出し，追加された分岐構文の条件節と追加された位置，および分岐構文の then 節と else 節のどちらが旧バージョンのコードを含むかを開発者に知らせる機能を実装する．

本研究では，開発者が適切な更新を行う更新プログラムを作成するための補助として，新旧バージョンのソフトウェアのソースコードを比較し，図 4.6 のような，新しいバージョンで追加された分岐構文を探す処理を，更新プログラム作成ツールに組み込んだ．

動的型付け言語である Scheme で記述されているソフトウェアは，第一級関数をサポートするため，静的にプログラムの流れを追うフロー解析は困難になる．本研究の DSU 機構で行う，ソフトウェアのソースコードの比較処理は，個々の関数の内部の処理の比較のみとし，複数の関数にまたがる解析は行わない．また前提として，同一の名前の関数は，バージョンが異なってもソフトウェアの中での役割が変わらないものとする．

第 5 章

実装

この章では，DSU 機構を実装する対象の言語処理系である TUTScheme のシステムを概説し，TUTScheme へ追加した機能とその実装法を述べる．また，更新対象のソフトウェアのソースコードから更新プログラムを自動で作成する，更新プログラム作成ツールの実装法を述べる．

5.1 言語処理系への機能の追加

TUTScheme は，Scheme プログラミング言語で記述されたプログラムをバイトコードに変換するコンパイラと，バイトコードを解釈しプログラムを実行するバイトコードインタプリタを持つ．バイトコードインタプリタを含む TUTScheme の処理系本体は C 言語で，コンパイラや各種ライブラリは Scheme でそれぞれ記述されている．

4.1 節で述べた，動的ソフトウェア更新の一連の流れを実装するため，本研究では，DSU の進み具合を管理する変数を追加した．

本研究で実装した機構では，処理系に DSU の指示を行うために，キーボード割り込みを用いる．ユーザがキーボード割り込みを行うと，割り込みハンドラが起動し，その中で DSU の実行状態を管理する変数の値が変更され，DSU の処理が開始される．

更新対象のソフトウェアのソースコードには，DSU の機能を呼び出すための特

殊な処理を追加していない．本研究では，Scheme プログラムの実行時に頻繁に実行されるバイトコード命令に着目し，その命令の実行時に更新処理を行うことにした．本研究では，グローバル関数を呼び出すバイトコード命令 `xcall` の処理に，更新を行う処理を呼び出すよう変更を加えた．

漸次的なデータの変換は，TUTScheme のコピー GC と，`xcall` 命令に追加された処理によって行われる．コピー GC には，ヒープ領域から旧データを探す処理を追加し，`xcall` 命令には，コピー GC で見つけられた旧データを変換する処理を追加した．

コピー GC で発見された旧データは，発見された時点で変換を行わず，コピー GC の後で呼び出される xcall 命令の処理の中で行われる．そのため，発見された旧データへの参照を記憶するための領域が必要となる．そのため，旧データへの参照を保持するリストを，グローバル定義の更新時に作成するようにした．

本節では，まず本研究で DSU 機構を実装する，TUTScheme のシステムについて説明する．続いて TUTScheme に DSU 機能を実装するため，TUTScheme のバイトコード命令および GC，および他の機能に追加した処理について説明する．

5.1.1 TUTScheme のシステム

TUTScheme は，Scheme 言語で記述されたプログラムをバイトコードに変換するコンパイラと，バイトコードを解釈し，プログラムを実行するバイトコードインタプリタを持つ．バイトコードとは，ハードウェアに固有の機械語命令 (ネイティブコード) とは異なる，仮想の機械語命令であり，関数の呼び出し，関数からの復帰，変数への値の代入，スタックへの値の追加，取り出しなどの命令がある．

バイトコードインタプリタは，プログラムの状態を管理する各種レジスタと，実行スタックを持つ．バイトコードインタプリタは，現在のプログラムカウンタが参照するバイトコード命令に対応する処理を実行し，レジスタおよび実行スタックの値を変更し，新しいデータ構造のためにヒープ領域を確保し，または入出力処

理を行う．

TUTScheme の実行スタック

TUTScheme の実行スタックは，図 5.1 のように，スタックフレームが複数積まれた構造で表される．スタックフレームは，関数本体への参照とその引数，関数の復帰アドレス，静的リンク，動的リンク，および関数の復帰時に実行される処理からなる．

TUTScheme では，ローカル関数のバイトコード命令列は，そのローカル関数が属しているグローバル関数と同じバイトコード列に含まれている．そのため，ローカル関数を呼び出すときのスタックフレームの形式は，グローバル関数の呼び出しのスタックフレームとは異なり、関数のバイトコードオブジェクトへの参照を含まない．

スタックフレームのそれぞれの要素の説明を以下に示す．

- **関数オブジェクトと引数**：呼び出されている関数と，引数の値を参照する．
- **静的リンクと動的リンク**：動的リンクは，呼び出し元の関数の動的リンクを参照する．静的リンクは，1 つ外側のレキシカルレベルの変数を参照するためのものである．ローカル関数のスタックフレームの静的リンクは，呼び出し元の，外側のレキシカルレベルの関数のスタックフレームを参照する．
- **関数の復帰時に実行する C 言語の関数へのポインタ**：関数が復帰する命令が実行するときに呼び出される処理である．通常の関数の復帰処理の代わりに，任意の C 言語の関数を格納し，関数の復帰時に実行できるため，ここに通常の復帰処理の代わりに別の関数を挿入することで，リターンバリア [10] が実現できる．
- **関数の復帰アドレス**：関数が復帰する先のアドレス (バイトコード列の先頭からのオフセット) である．

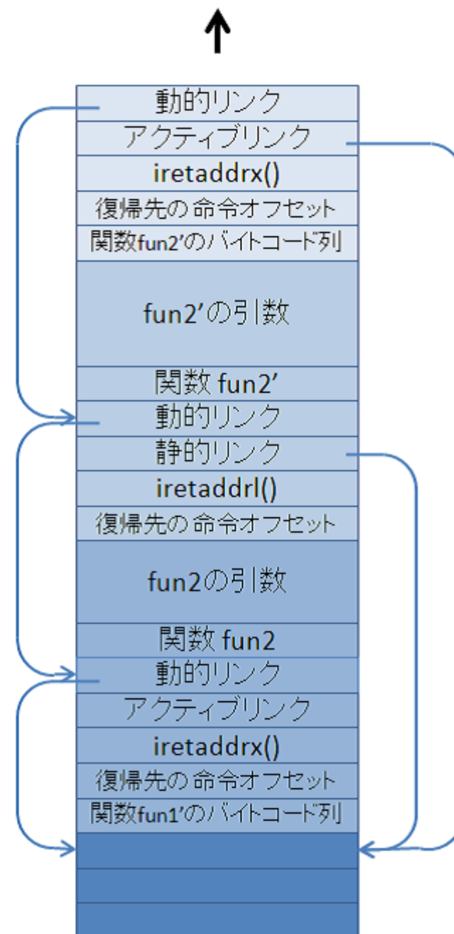

図 5.1: TUTScheme の実行スタック

- **呼び出し元のバイトコード列**：グローバル関数のスタックフレームだけが持つ．関数が復帰する先のバイトコード列を参照する．

TUTScheme **の関数呼び出し/復帰命令**

TUTScheme には，Scheme のグローバル関数の呼び出しと，ローカル関数の呼び出しに別々の命令が用意されている．また，関数の最適化された末尾呼び出しを行う命令が用意されている．最適化された末尾呼び出しとは，呼び出される関数の返り値が現在実行している関数の返り値となる場合，現在のスタックフレームを再利用して関数呼び出しを行うことである．

TUTScheme における関数呼び出し命令を以下に示す.

- **xcall** グローバル関数，クロージャおよび継続の (末尾呼び出しではない) 呼び出しを行う.
- **lcall** ローカル関数の呼び出しを行う.
- **trcall** グローバル関数からグローバル関数の最適化された末尾呼び出しを行う.
- **trlcall** ローカル関数の最適化された末尾呼び出しを行う.
- **ltrxcall** ローカル関数からグローバル関数の最適化された末尾呼び出しを行う.

グローバル関数とローカル関数で，スタックの構成が異なるため，関数の復帰命令も 2 種類存在する.

- **xreturn** グローバル関数からの復帰処理を行う.
- **lreturn** ローカル関数からの復帰処理を行う.

5.1.2 Scheme データの内部表現

TUTScheme における Scheme データは，連続したメモリ領域上に配置された，複数のデータスロットからなるデータの実体を指すポインタで表す．ポインタ及び各データスロットは，図 5.2 に示すような，32 ビットの固定長である．このうち上位の 6 ビットはタグと呼ばれる．タグは，ポインタが指すデータについてのデータ型を保持する.

クロージャは，図 5.3 の左上に示すように，9 個のデータスロットからなる．このうち，先頭のデータスロットはヘッダと呼ばれ，9 個のデータスロットからなることを示している．`const_vec` スロットは定数ベクタ，`body` スロットは関数のバイ

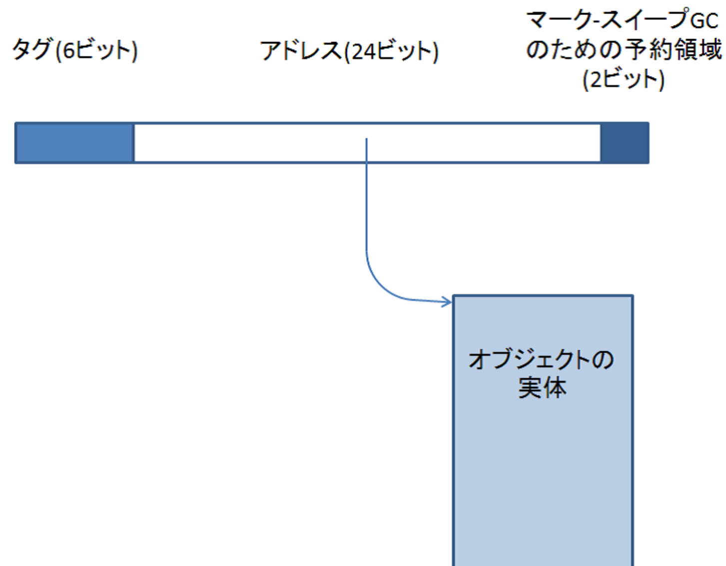

図 5.2: データスロットの形式

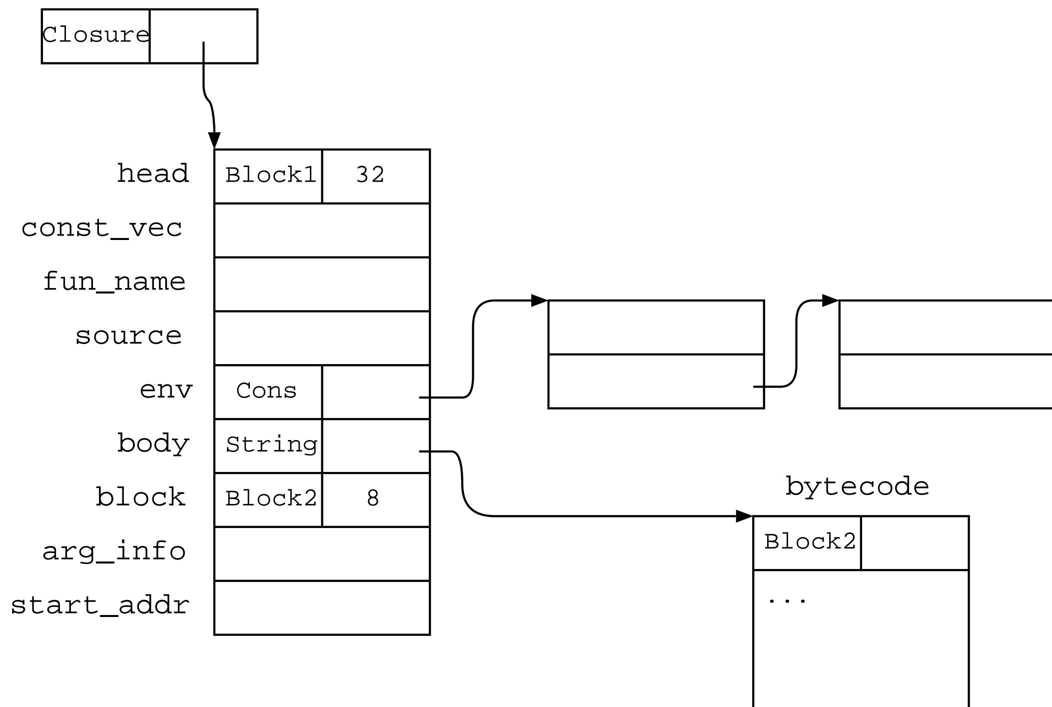

図 5.3: クロージャの内部表現

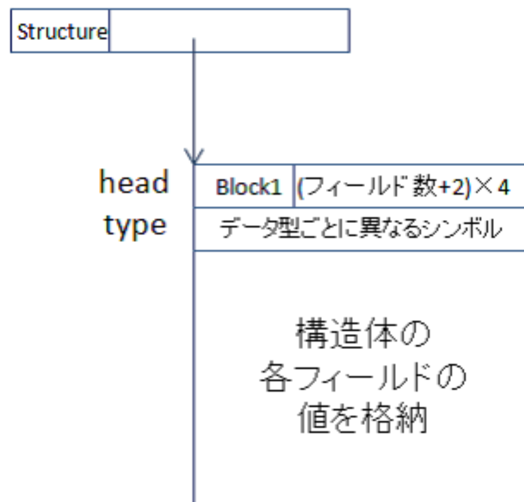

図 5.4: 構造体の内部表現

トコード列 (文字列オブジェクトとして実装) をそれぞれ参照しており，arg_info スロットは関数の引数の個数を，start_addr は関数の開始アドレスのオフセットを格納している．また，env スロットはクロージャが作られたときの環境を参照している．クロージャに格納される環境は値のリストの形で格納され，クロージャが実行されるときは，環境内で束縛された値へのアクセスは，変数名ではなくリスト内での順番によって，どの変数の値を参照するかを指定する．

TUTScheme における Scheme の構造体は，図 5.4 に示すように，構造体の型に応じた個数のデータスロットからなる．このうち，先頭のデータスロットはクロージャのヘッダと同じく，内部表現についての情報を格納している．また，先頭から 2 番目のスロット type は構造体の型を表すシンボルを格納する．残りのデータスロットは，構造体の各フィールドに格納される値を参照する．

5.1.3 組み込みグローバル変数の追加

DSU 機能のために，以下の組み込みの Scheme グローバル変数を追加した．これらの変数は，Scheme プログラムから，自由に値を代入し，参照することができる．

- ***dsu-mode*** DSU が行われているかどうか，および，DSU が行われている場合，現在どの段階まで進んでいるかを示す整数である．
- **dsu:load-updater** 更新対象のソフトウェアが読み込まれた際に，更新プログラムを読み込む関数が格納される．ユーザからの更新の指示が行われた後，次の `xcall` 命令の実行時に呼び出される．
- ***dsu-object-checklist*** 更新プログラムが読み込まれた際，変換を要するデータ型の情報を含んだリストが格納される．GC の実行時に旧データを判別するのに用いられる
- ***dsu-functions*** 更新プログラムが読み込まれた際，更新が行われる関数のリストが格納される．
- **dsu:update-definitions** 更新プログラムが読み込まれた際，グローバル定義の更新を行う関数が格納される．
- **dsu:object-updater** 更新プログラムが読み込まれた際，更新対象データを新データに置換する関数が格納される．
- **dsu:remove-accessbarrier** 更新プログラムが読み込まれた際，データの置換の完了時にアクセスバリアを外す関数が格納される．
- ***dsu-objects*** ごみ集めによる更新対象データの探索を行う際，見つかった更新対象データを格納するためのリストである．グローバル定義の更新が行われる直前に，指定された長さの weak リストが格納される．weak リストとは，weak-cons セルからなるリスト構造であり，weak-cons セルの car 部以外から参照されていないデータは，GC を行う際に回収される．*dsu-objects* に weak リストを用いた理由は，すでに参照されなくなったデータを置換する処理を行わず，無駄な処理を省くためである．

5.1.4 ソフトウェアの更新処理を進める処理の追加

本研究で実装したDSU機構は，4.1節で述べたとおり従来のDSU機構とほぼ同様に，ソフトウェア更新の指示を受け，グローバル関数などの定義を更新し，データの置換を行う．本研究で実装したDSU機構のソフトウェアの更新処理全体のフロー図を図5.5に示す．この処理の実行状態は変数`*dsu-mode*`に整数の値として格納されている．更新の指示を受けてから更新プログラムを読み込むまで，この値は1である．更新プログラムを読み込んでから，グローバル定義の更新を行うまで，この値は2である．グローバル定義の更新が行われた後は，この値は3である．

ソフトウェアの実行に割り込んで更新の処理を行うため，TUTSchemeのバイトコード命令`xcall`の処理に，`*dsu-mode*`の値に応じて，更新の処理を呼び出す処理を追加した．この処理は，更新処理を次の段階に進められるかを調べた後に，更新処理を進められる場合に更新処理を次の段階に進める処理を行う．`xcall`命令に追加された処理のフローを図5.6に示す．

`xcall`に追加された処理によって呼び出される処理を以下に示す．

- 更新プログラムを読み込む処理
- 更新される関数のスタックフレームの数をカウントする処理
- グローバル定義を更新する関数を呼び出す処理
- 更新対象データを格納するリストを作成する処理
- ごみ集めで集められた更新対象データを置換する処理
- アクセスバリアを外す関数を呼び出す処理

これらの処理は，`xcall`の処理が本来行う関数の呼び出し処理の直前に行われる．本節では，これらの各処理の実装法の詳細を述べる．

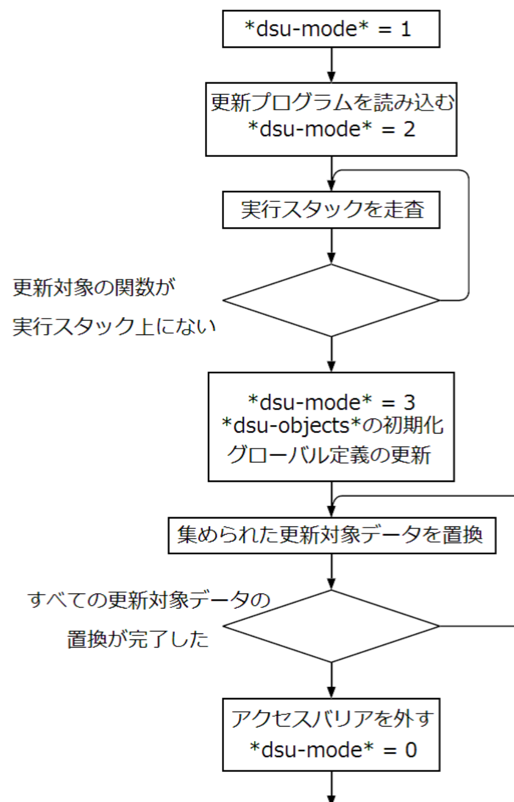

図 5.5: 更新処理の進み方

更新プログラムを読み込む処理

ソフトウェアの更新を行うには，更新の内容を表す各関数および更新に必要な情報を含む，更新プログラムを読み込む必要がある．ユーザがキーボード割り込みを行った時点での処理 (5.1.7 節を参照) では，更新プログラムを読み込む処理は行っていない．本研究では，キーボード割り込みシグナルのシグナルハンドラの中では，変数`*dsu-mode*`の値を 1 にセットする処理のみを行っている．

図 5.6 に示されるように，このシグナルハンドラによって`*dsu-mode*`に 1 がセットされた後，次に `xcall` 命令が実行されるときに更新プログラムを読み込む関数 `dsu:load-updater` を呼び出す．この処理が行われた後，変数`*dsu-mode*`の値を 2 に変更する．

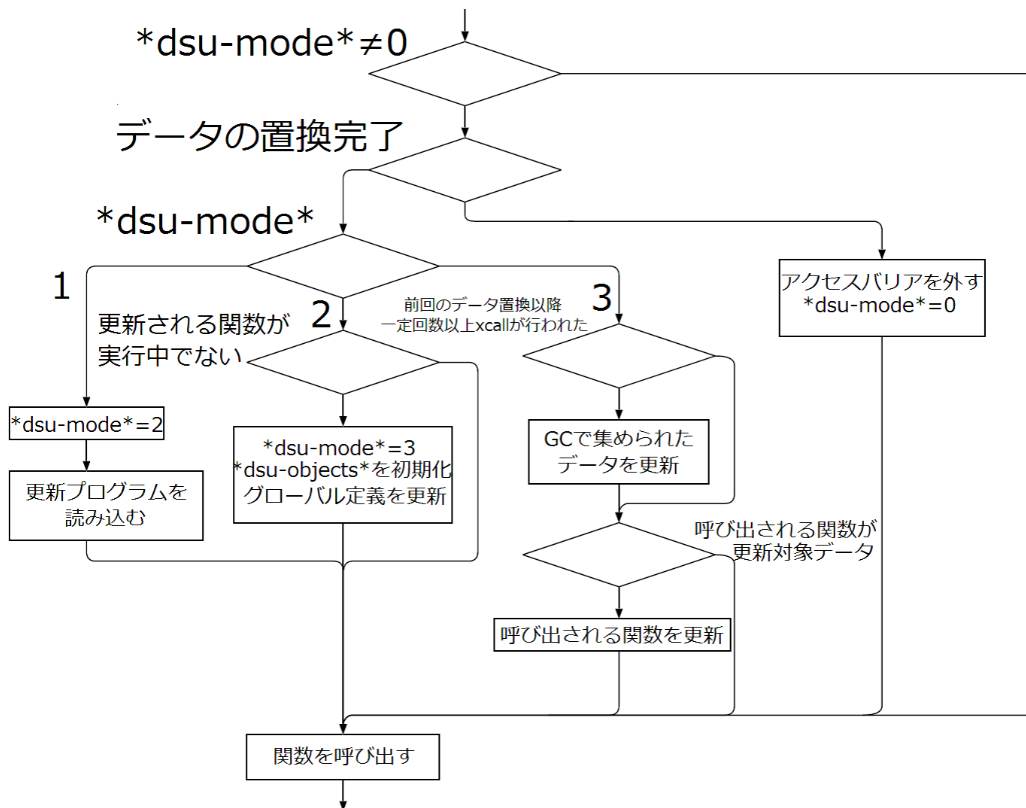

図 5.6: `xcall` に追加された処理

ここで，関数 `dsu:load-updater` の呼び出しは，`xcall` 命令が本来呼び出す関数の呼び出しに割り込んで行われる．関数の呼び出しへの割り込みを行うため，関数 `dsu:load-updater` の呼び出し処理を割り込ませ，この関数が復帰する際に本来呼び出される関数を呼び出すようにした．

`dsu:load-updater` を呼び出すスタックフレームを積んだときの実行スタックの状態を図 5.7 に示す．`xcall` 命令が呼び出される際には，実行スタックの一番上にはすでに，本来呼び出される関数のスタックフレームが用意されている．`xcall` 命令の本来の処理が行われる前に，関数 f のスタックフレームの上に `dsu:load-updater` 関数のスタックフレームを追加し，xcall の関数呼び出しの処理で `dsu:load-updater` 関数が呼び出されるようにした．また，追加されたスタックフレームの関数の復帰時の処理には，C 言語の関数 `xcall_ep` を指定した．これは通常の関数が復帰

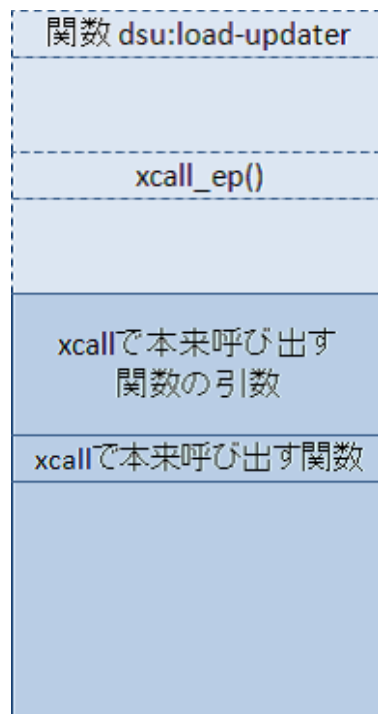

図 5.7: 実行スタックへのフレームの追加

した時の処理を行った後，再び xcall 命令の処理を呼び出す．xcall_ep の処理が dsu:load-updater の復帰する時に実行され，本来呼び出される関数の呼び出しが行われるため，以後更新対象のソフトウェアは本来の動作を続ける．

更新される関数のスタックフレームの数をカウントする処理

グローバル定義の更新を行う処理は，4.4 節で述べたとおり，更新される全ての関数が実行されていない時に行う．更新される関数の実行中にこの処理が行われないよう，この処理を行う前に，更新される関数が実行中かどうかを確かめる．

本研究では，変数*dsu-mode*の値が 2 であるとき，xcall 命令の実行時に，TUTScheme の実行スタック上のスタックフレームを調べて，そのうち更新されるグローバル関数のスタックフレームがいくつ存在するかを数える．更新される関数のリストは，更新プログラムの読み込み時に変数*dsu-functions*に束縛さ

れている．実行中の関数１つ１つについて，このリストに一致するものがあるかを調べることで，更新される関数が実行中かを調べることができる．

図 5.1 から，実行中の関数は全て各スタックフレームのフレームリンクの１つ上のアドレスに格納されていることがわかる．各フレームのアクティブリンクを，スタックの底まで辿りながら，フレームリンクの１つ上のアドレスの関数と，更新される関数を比較することでこれらの関数が実行中かを調べられる．アクティブリンクは，全てのグローバル関数のスタックフレームを参照するため，実行スタック上の実行中の関数のバイトコードをすべて調べることができる．

しかし，TUTScheme では継続の生成時に実行スタックをヒープ領域中に退避するため，ヒープ領域上に退避されたスタックフレームについても，更新される関数のスタックフレームがあるかを調べなくてはならない．退避された実行スタックは，`more` と呼ばれる C 言語の変数から参照できる．退避された実行スタックは，スタックフレームのリストが入れ子になった構造をとっており，スタックフレーム中の呼び出された関数と引数は TUTScheme のベクタオブジェクトに格納され，実行中の関数はそのベクタの先頭に格納されている．本研究では，スタックフレームのリストをたどって，実行中の関数が更新される関数と一致するかを調べることで，スタックフレーム上の全ての関数が更新される関数でないかを確認する．以上の実行スタックと `more` の走査によって，実行中のすべての関数を調べることができる．

実行中の全ての関数を調べる処理には，非常に長い時間がかかる場合がある．この時間の長さは，実行中の関数の数と更新される関数の数の積に比例する．本研究では，実行スタックをすべて調べる処理を一回だけ行い，その後更新される関数のスタックフレームの個数の増減を記録する．この方法では，更新対象の関数のスタックフレームの個数の増減が正しく記録できているならば，更新対象の関数のスタックフレームが実行スタック上から無くなったことを認知できる．しかし，この方式では，`xcall` 命令の実行のたびに，更新される関数の数に比例する，スタッ

クの走査によるオーバヘッドがかかる．

実行スタック全体の更新される関数のスタックフレームの数を，最初の 1 回の xcall の実行時に数えた後，この数を C 言語の変数 running_updated_fun に記録し，この値が 0 になるまで，グローバル関数の呼び出しおよび復帰を行う xcall, xreturn, trcall, ltrxcall 命令の実行時に変数 running_updated_fun のカウント処理を行う．この処理では，更新対象の関数が呼び出される場合に，変数 running_updated_fun をインクリメントし，更新対象の関数が復帰するときにデクリメントする．

グローバル定義を更新する処理

変数 running_updated_fun の値が 0 になったときに xcall 命令が実行されたときに，グローバル定義の更新を行う処理を実行する．このとき，グローバル定義の更新後に GC を用いたデータの置換処理を行うため，更新対象データを格納するためのリストを作成する．これらの処理が終わった後，変数*dsu-mode*の値を 3 に変更し，関数 dsu:update-definitions の処理を実行された後，GC を用いた漸次的なデータ変換と並行して，更新対象のソフトウェアの実行を進める．

グローバル定義の更新を行う処理は，xcall 命令の処理に割り込んで関数 dsu:update-definitions を呼び出すことで行われる．この処理を実行するために，図 5.7 に示す処理と同様に，関数 dsu:update-definitions のスタックフレームが積まれる．

その直後に，GC を行う際に見つかった旧データを格納するリストを作成する．このリストは weak リストであり，このリスト以外から参照されていない値は，GC が行われた際に回収の対象となる．weak リストを用いた理由は，このリストに格納された後に他の場所から参照されなくなった，ごみの旧データを更新する無駄を省くためである．このリストは，Scheme 変数*dsu-objects*に束縛される．

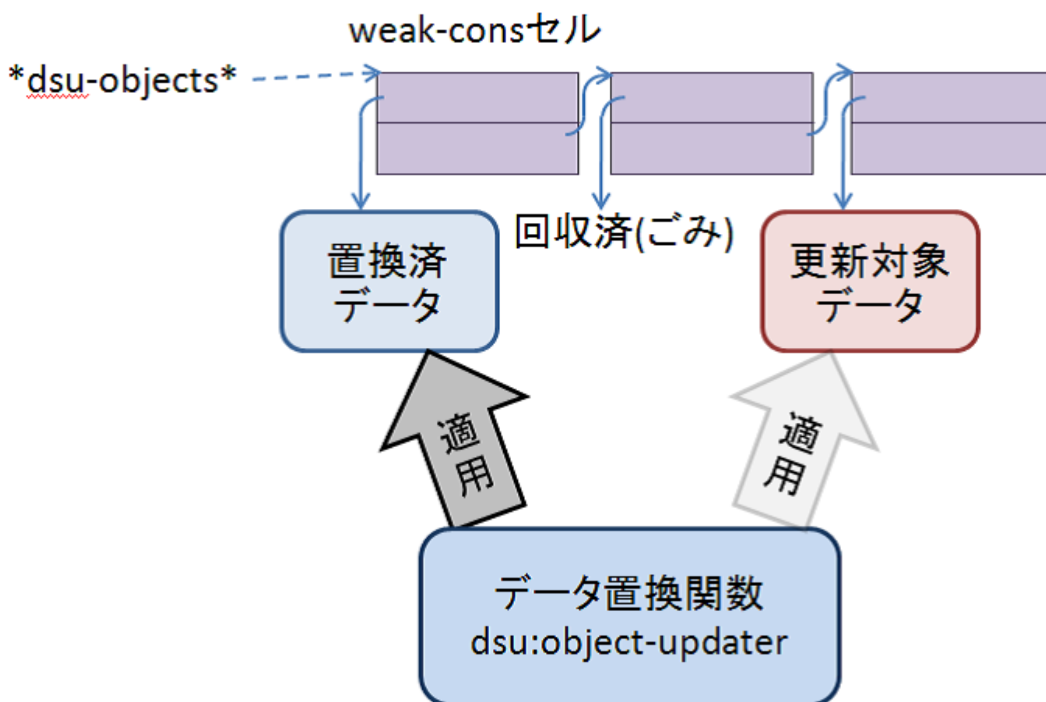

図 5.8: 集められた更新対象データの置換

更新対象データの置換を行う処理

関数などのグローバルな定義が更新された後，xcall 命令が実行されたときに，GC によって集められた更新対象データの置換を行う．また，xcall によって呼び出される関数が更新対象データである場合，その関数を置換してから呼び出されるようにする．

GC によって集められた更新対象データの置換は，変数*dsu-objects*に格納されたリストから参照される個々の更新対象データに対して，関数dsu:object-updater を適用することで行う．*dsu-objects*に格納されたリスト内の更新対象データの置換の様子を図 5.8 に示す．この処理では，*dsu-objects*に格納されたリストの要素である更新対象データを次々と置換済みの新データに置換していく．

dsu:object-updater の呼出しは，図 5.7 と同様の実行スタックの操作を繰り返し行うことで実現できる．本来呼び出される関数のスタックフレームの上に，データ

の置換を行う処理のスタックフレームを追加し，xcall 命令を実行する．このとき追加されるスタックフレームは，更新対象データを引数とした dsu:object-updater の呼び出しである．また，関数の復帰時の処理には，*dsu-objects*の次の要素の更新対象データを置換するためのスタックフレームを作成する処理を指定し，更新対象データを置換する処理が繰り返し呼び出されるようにする．*dsu-objects*から参照される更新対象データを一定数置換した後は，本来呼び出される関数を呼び出し，更新対象のソフトウェアの実行を進める．なお，*dsu-objects*のリストは weak リストであり，このリストには GC で回収された要素も含まれるため，現在参照している weak-cons セルの要素が回収されている場合，その要素を置換せず weak リストのさらに次の weak-cons セルを調べて次の更新対象データに行き着くまで weak リストの cdr をたどる．

xcall 命令の実行するごとに必ず更新対象データを置換するようにすると，GC の行われた直後に更新対象データの置換が行われるため，事実上 GC による停止時間が伸びたのと同じ事になる．本研究では，xcall 命令が一定回数行われるごとに 1 回，更新対象データの置換を行うことで置換処理を行う間隔を設ける．置換処理を行う間隔は TUTScheme のソースコード中で定義されている．

xcall 命令で呼び出される関数自身が更新対象データのクロージャである場合は，その関数を新データに置換してから呼び出す．この処理は，呼び出される関数を引数とした関数 dsu:object-updater の呼び出しの割り込みによって行う．

アクセスバリアを取り除く処理

GC を用いた漸次的な置換によって全ての更新対象データを置換できるので，全ての更新対象データを置換した後では，アクセスバリアは不要になる．アクセスバリアはソフトウェアの実行にオーバヘッドをもたらすので，本研究ではこうした場合にはアクセスバリアを外して，ソフトウェアの実行性能を向上させる．

GC を行う時に更新対象データを探し集めた際に，更新対象データが全く

見つからなかった場合には，C 言語の変数 object_update_finished_flg に 1 がセットされる．この直後の xcall 命令の実行に割り込んで，Scheme 関数 dsu:remove-accessbarrier を呼び出す．さらに，変数*dsu-mode*の値を0にする．

5.1.5 ごみ集めへの処理追加

本研究で提案する漸次的なデータ置換では，GC の処理中に最大一定数まで更新対象データを探し，その後，見つかったデータを一定数ずつ置換する．ここでは，GC の処理に更新対象データを探す処理を追加する方法を示す．

コピー GC では，ルート集合から直接または間接的に参照されているヒープ領域の from 領域内の全てのデータを，ヒープ領域の to 領域内に複製する．この複製処理によって，ルート集合から間接的に参照されていないデータを to 領域に移さないことで「回収」し，またルート集合から参照されている「生きている」データを連続したメモリ領域に配置すること (コンパクション) ができる．TUTScheme のコピー GC では，この複製処理を個々のデータ毎に行う．

本研究では，TUTScheme のコピー GC の，個々のデータの複製を行う C 言語の関数 gcunit を変更し，複製されるデータが更新対象データであるかを判定する処理を追加した．また，gcunit に，複製された更新対象データを*dsu-objects*に格納された weak リストに格納する処理を追加した．この処理の概要を図 5.9 に示す．

更新対象データの判定処理では，複製されるデータが cons セル，構造体，クロージャであるかを判定する．この判定は，コピー GC で複製されるデータを参照する変数のタグが cons セル，構造体，クロージャのものと同じであるどうかを調べることで行う．複製されるデータが cons セル，構造体またはクロージャであれば，複製されるデータの各スロットを変数*dsu-object-checklist*に格納されたリストに含まれる更新対象データの情報と比較して，複製されるデータが更新対象データであるかを判定する．

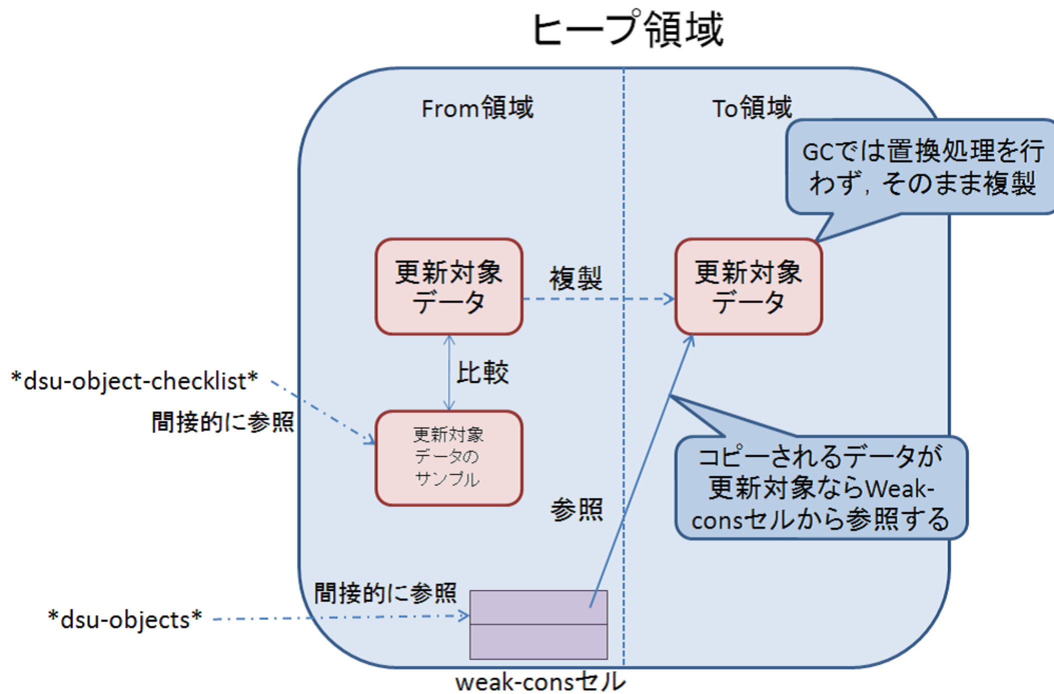

図 5.9: コピー GC で行う更新対象データの発見と weak リストへの格納

変数`*dsu-object-checklist*`には，更新対象のデータ型についての情報を含むリストが束縛されている．このリストの個々の要素は，`(list-car <シンボル>)`，`(struct <シンボル>)`，`(closure <クロージャ>)` のいずれかである．ここで<シンボル>は更新対象データの型を表すシンボルであり，<クロージャ>は，更新対象データと同じ，変更されたコンストラクタ内の `lambda` 式で生成されたクロージャである．このリストの全ての要素と，コピーされるデータを以下のように照らし合わせることで，コピーされるデータが更新対象データであるかを判別する．

- リストの要素が `(list-car <シンボル>)` である場合，コピーされるデータが cons セルであるかを調べる．コピーされるデータが cons セルであり，そのデータの car スロットと<シンボル>が同一のシンボルであるならば，コピーされるデータは更新対象データであるとする．

- リストの要素が `(struct <シンボル>)` である場合，コピーされるデータが構

造体であるかを調べ，コピーされるデータが構造体であれば，そのデータの構造体型と<シンボル>が同一のシンボルであるならば，コピーされるデータは更新対象データであるとする．

- リストの要素が (closure <クロージャ>) である場合，コピーされるデータがクロージャであるかを調べ，コピーされるデータがクロージャであれば，そのデータのバイトコードへの参照と，関数の開始アドレスがともに<クロージャ>のバイトコードへの参照，および関数の開始アドレスと同じならば，コピーされるデータは更新対象データであるとする．

複製されるデータが更新対象データであれば，そのデータは，変数`*dsu-objects*`に束縛された weak リストに格納され，後で `xcall` 命令の実行時に，`*dsu-objects*`から参照されたデータを置換する処理が行われるときに，置換される．

また，コピー GC が旧データを探したときに，コピー GC の開始時から終了時までに，1 つも旧データが見つけられなかった時には，変数 `object_update_finished_flg` に 1 を代入する．この処理を行ったとき，次に `xcall` 命令が実行される際に，5.1.4 項で述べた処理が実行され，ソフトウェアの更新が完了する．

この処理は，各定義の更新が行われた後，全ての更新対象データの置換が終わるまで行われる．更新対象の探索と置換を行う処理の流れを図 5.10 に示す．この図の中の定数 N_{GC} と N_{call} は TUTScheme のソースコード内に定義されている．$N_{GC} = N_{call}$ のときは，GC を行う際に集められた全てのデータを一回で置換する．一般的に N_{GC} は更新対象のデータの総数より小さいため，これは全ての更新対象データを一括で更新することではない．$N_{GC} > N_{call}$ のときは，GC を行う際に集められたデータの置換処理を複数回に分けて行う．

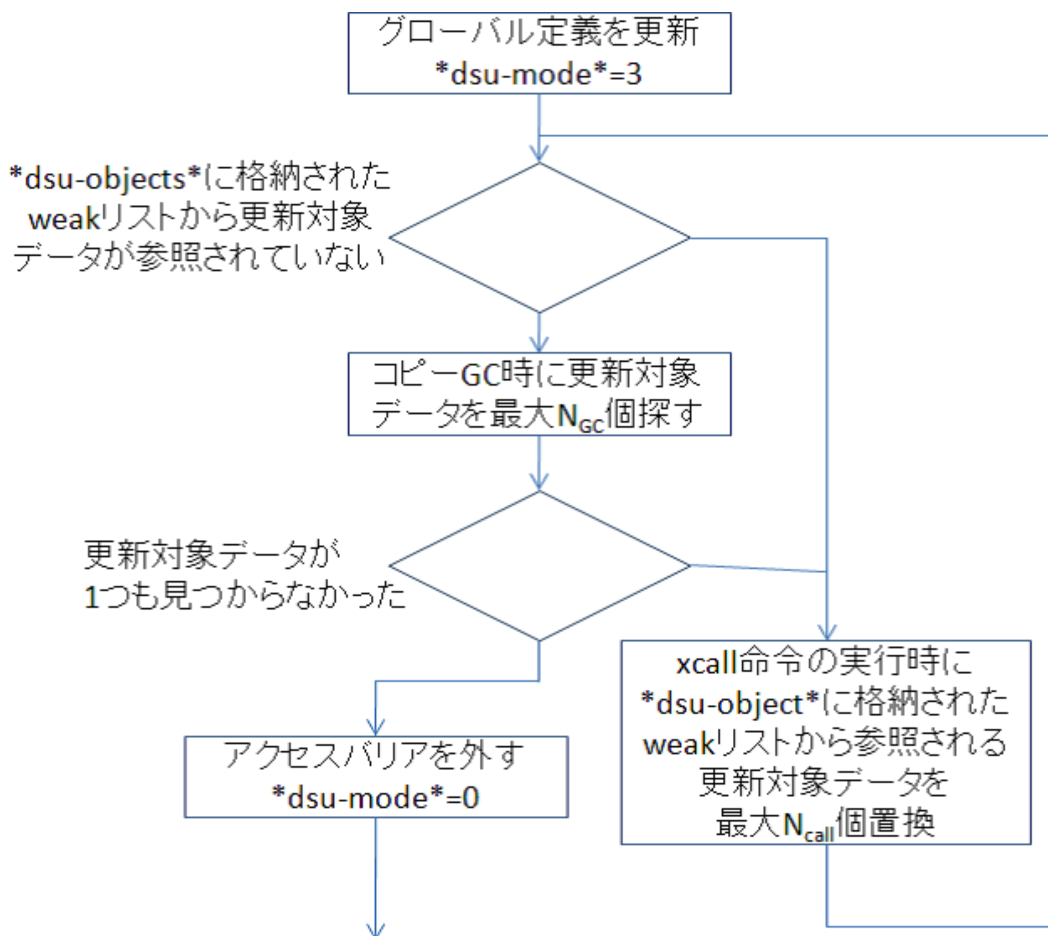

図 5.10: 更新対象の探索と置換を行う処理

5.1.6 プリミティブ関数の追加

更新対象データとなるクロージャおよび構造体の置換を行うため，以下の関数を TUTScheme のプリミティブ関数として実装した．

- (bcbody-eq? a b) 2つのクロージャが同一の lambda 式によって生成されたものならば，#t を返す．それ以外の場合は#f を返す．
- (bcbody-equal? a b) 2つのクロージャの関数本体が同一のバイトコードに含まれる場合，#t を返す．それ以外の場合は#f を返す．(bcbody-eq? a b) との違いは，同じグローバル関数に含まれる別個の lambda 式によって生成さ

れたクロージャを比較した場合にも#t を返す点である．例えばグローバル関数 x と，その中の lambda 式で生成されたクロージャy について，(bcbody-eq? x y) は#f を返すが，(bcbody-equal? x y) は#t を返す．

- (structure-type-set! s t) 構造体の置換を行う際に，構造体のデータの型を新しい構造体のものに変更するために用いられる．構造体 s の type スロットに t の値を代入する．

5.1.7 その他の処理追加

更新の指示を受ける処理

ユーザからのソフトウェアの更新の指示をいつでも受け付けられるようにするため，TUTScheme がユーザによるキーボード割り込みシグナル (SIGINT) を受けた時に呼び出される，割り込みハンドラ関数に，更新の指示を受け付ける処理を追加した．

この割り込みハンドラはもともと，実行中の Scheme プログラムの処理を中断するのに用いられてきた．この関数が呼び出された際に，ユーザの入力を受け付ける機能を追加し，ユーザが処理の中断を行うか，ソフトウェアの更新を開始するかを選択するプロンプトを表示する．このプロンプトに対して u と入力すると，変数 *dsu-mode* の値に 1 がセットされる．

ここで更新プログラムの読み込み処理を行わないのは，キーボード割り込みシグナルに対応するシグナルハンドラの関数内で TUTScheme の実行スタックなどの値を変更すると，TUTScheme が予想外の動作をする恐れがあるためである．*dsu-mode*の値が 1 になれば，後は xcall 命令に追加された処理が更新プログラムの読み込みなど，ソフトウェアの更新の処理をユーザの介入を必要とせずに進める事ができる．

head	Block1	24
type	student-1	
name		
age		
grade		
rest		

図 5.11: 生成される student 構造体のデータの内部表現

クロージャの環境に名前情報を格納する処理

4.5.3 節で述べたとおり，クロージャを置換を行うためには，クロージャの環境内の，特定の名前の変数に束縛された値を知る必要がある．クロージャの置換を可能にするため，本研究では，TUTScheme のコンパイラに変更を加え，クロージャの環境に変数名の情報を格納するようにした．

TUTScheme のコンパイラは，Scheme で記述された 2 パスコンパイラである．このコンパイラの 2 パスめの処理に変更を加え，クロージャの環境に閉じ込められる変数の名前が保存されるようにした．

構造体の型をバージョン固有のものにする処理

4.5.3 節では，構造体を更新可能にするため，構造体の型は，バージョンごとにそれぞれ異なるようにし，全ての構造体に 1 つ余分なフィールドを追加することを提案した．以下では更新可能な構造体を実装する方法について述べる．

構造体の型をバージョンごとにそれぞれ異なるようにするため，更新可能なソフトウェアのバージョンを表すグローバル変数*version*を導入する．この変数は，ソフトウェアの開発者が定義する必要がある．また，各バージョンごとに異なる値を設定する必要がある．その上で，*version*の値が n であるときに宣言された構造体の型は，すべてユーザが指定した型<type>に対して<type>-n となるようにする．

更新可能な構造体の実装のため，TUTScheme の標準ライブラリで定義されている，構造体を定義するためのマクロ defstruct に変更を施した．変更された defstruct マクロは，*version*によって，異なる型の構造体の型を定義する．例えば，フォーム (defstruct student name age grade) を，*version*に 1 が束縛されているときに評価すると，student-1 型の構造体が定義される．また，構造体の末尾に rest と呼ばれるフィールドが追加されるようにした．これは，ソフトウェアの更新によって追加されたフィールドを格納するためのリストを格納する．変更された defstruct によって定義される student 型の構造体の内部表現を図 5.11 に示す.

5.2 更新プログラム作成ツールの実装

更新プログラムを作成する労力を減らすため，新バージョン及び旧バージョンのソフトウェアのソースコードを読み込み，更新プログラムを自動で作成するツールを開発した．ツールは 750 行程度の Scheme プログラムで実装できた．本節では，ツールの作成する更新プログラムの構成および，更新プログラムを出力するまでに必要な解析処理を示す.

5.2.1 更新プログラムの構成

4.7 節で述べた通り，更新プログラムは以下の要素からなる．

- 更新される関数のリスト
- 旧データの情報を表すリスト
- グローバル定義の更新を行う関数
- データの置換を行う関数

- アクセスバリアを外す関数

更新される関数のリストは，関数名のリストで表される．このリストは，更新対象のソフトウェアに読み込まれた時に更新される関数を要素とするリストになる．

旧データの情報を表すリストを作成する際には，各データ構造データサンプル生成関数を用いて得られた更新対象データのサンプルを，リスト，クロージャおよび構造体に分類することで，GC が更新対象データを探すためのリスト *dsu-object-checklist*を作成する．

*dsu-object-checklist*の構造は，更新プログラムの読み込み時に作られる．これは，更新対象データの各サンプルが，それぞれリスト，クロージャまたは構造体のどれになるかをあらかじめ静的解析によって求めるよりも実装が容易であるためである．本研究では，更新プログラムが読み込まれた際に更新対象データのサンプルを生成し，そのサンプルがリスト，クロージャ，構造体であるかに応じて，リスト*dsu-object-checklist*の各要素を生成する．この処理を行う Scheme プログラムを図 5.12 に示す．このプログラムでは，生成された各種の更新対象データのサンプルが，map の第 1 引数の関数に渡される．この関数は引数 x がクロージャであれば (closure x) を，x が構造体ならば (structure <x の型を表すシンボル>) を，x がリストならば (list-car <x の先頭要素に格納されたシンボル>) を返す．map 関数が各更新対象データのサンプルにこの関数を適用することで，5.1.5 節で述べた，*dsu-object-checklist*の要素からなるリストが生成される．

グローバル定義の更新を行う関数 dsu:update-definitions は，新しい定義の宣言と，グローバル変数などの定義を行う eval 式，および新しいマクロの定義からなる．この関数では，まず新しい定義の式をリストまたはローカル関数として列挙し，バージョン間でマクロの定義に変更があれば新しいバージョンのマクロの定義を実行し，新しい定義グローバル変数などの定義を行う．

データの置換を行う関数は，任意の種類の更新対象データの置換を行う関数 dsu:object-updater と，特定のデータ型の更新対象データのみを置換する関数

```
(define *dsu-object-checklist*
  (map
    (lambda (x)
      (cond ((function? x) `(closure ,x))
            ((structure? x) `(struct ,(structure-type x)))
            ((list? x) `(list-car ,(car x)))))
    (list
      (make-foo-sample)
      (make-bar-sample)
      ...)))
```

図 5.12: *dsu-object-checklist*に格納されるリストを組み立てるプログラム

```
(define (update-foo old-foo) ...)
(define (update-bar old-bar) ...)

(define (dsu:object-updater x)
  (cond ((old-foo? x) (update-foo x))
        ((old-bar? x) (update-bar x))
        (else x)))
```

図 5.13: TUTScheme の更新対象データの置換を行う関数群

update-<データ型名>からなる．データの置換を行う関数群の例を図 5.13 に示す．

アクセスバリアを外す関数 dsu:remove-accessbarrier は，アクセスバリアが追加されたアクセサ及びデータ型判定述語を，アクセスバリアのない関数として再定義を行うことで，これらの関数に追加されたアクセスバリアを取り外す．

5.2.2 更新プログラムの生成

更新プログラム作成ツールは，更新前のバージョンのソフトウェアのソースコードと，更新後のバージョンのソフトウェアのソースコードを読み込み，それぞれのグローバル変数の，更新前後のバージョンにおける差分を求める．この差分から，どのグローバル変数の定義に変更を加える必要があるか，どの関数にアクセスバリアを挿入する必要があるか，どのデータ構造を更新対象に指定すべきかを推定する．

まず，ソフトウェアのソースコードの内から，`define`, `macro`, および `defstruct` で始まる式を探す．各グローバル変数の定義が変更されているか，またはマクロや，構造体の定義が変更されているかを調べる．

次に，変更のある `define` 式から，更新される関数のリスト `*dsu-functions*` を作成し，出力する．この処理は，単に変更のある `define` 式から定義される変数名を抜き出し，列挙するだけである．作られたリストには，関数でないものが含まれることもあるが，更新プログラムが読み込まれる段階で，関数でないものが束縛された変数を除外することもできる．

その次に，変更のある関数の定義のリストからコンストラクタの定義を探し，変更のあるコンストラクタで作られたデータのアクセサおよび型判定述語の定義のリストを作る．また，変更された defstruct 式から，構造体の各フィールド名を得る．これらの処理によって得られた情報から，アクセスバリアを追加すべき関数のリストを作り，これを元に，アクセスバリアの追加された関数の定義を組み立てる．

変更のあるコンストラクタのリストから，コンストラクタに対応するデータ型のサンプルを生成する関数を用いて，GC 時に用いる旧データのリスト `*dsu-object-checklist*`の定義を組み立て，出力する．また，各データ型のデータを変換するための関数を出力し，任意のデータ型のデータを変換する関数 `dsu:object-updater` の定義を出力する．更新プログラムの開発者は，これらの関数群の処理を，更新プログラムの自動生成後に書き換えることができる．

更新された変数およびマクロの新しい定義と，アクセスバリアの追加されたアクセサの定義から，グローバルな定義を変更する関数 `dsu:update-definitions` の定義を組み立て，出力する．マクロの再定義は，変数の新しい定義の式を評価する前に行うようにする．

5.2.3 ソースコードの差分検出

更新前後のバージョンのソースコードを比較して，更新時に関数の処理に追加される `if` 式と，その `if` 式の条件節の真偽値が，更新される前のソフトウェアの実行においてどちらの値をとるならばその `if` 式が更新前と同じ処理を行うかを表示する機能を実装した．ソースコードの比較は，ソースファイル中の各々の式に対して行う．また，コードの比較を行う前に，ソースコード内の `case` 構文および `cond` 構文を全て `if` 式に変形する．`cond` 構文を入れ子になった `if` 式に変形される様子を図 5.14 に示す．

ソースコードの変形が終わった段階で，ソースコードの比較を行う．ソースコードの比較は，更新前のバージョンのソースコードを式の木に見立てて，先行順で，ソースコードのそれぞれの式と同じ式を更新後のバージョンのソースコードから探す事で行う．同じ式が見つからなかった場合，前の式に対応する式の次の式が `if` 式かを調べる．これが `if` 式であれば，その中から，古いバージョンのソースコードと同じ式を探す．同じ式が見つかれば，その式が `if` 式の `then` 節にあるか，`else` 節にあるかを調べる．この式が `then` 節にあれば，「ソフトウェアが更新されるまでの実行において，この `if` 式の条件節が常に真であれば，更新後も同じ処理を行う」旨を表示する．この式が `else` 節にあれば，「ソフトウェアが更新されるまでの実行において，`if` 式の条件節が常に偽であれば，更新後も同じ処理を行う」旨を表示する．

ソースコードの比較による，ソフトウェアのバージョン間の差分を求める静的解析は，DSU により大きな可能性をもたらすと考えられる．今回は `if` 式の追加を

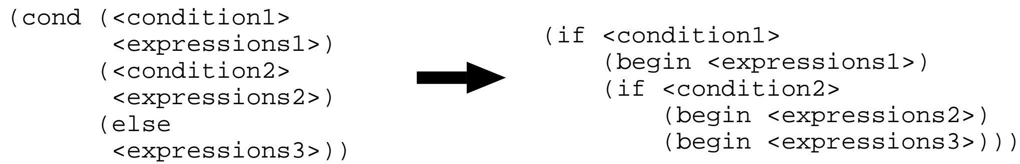

図 5.14: `cond` 式から入れ子になった `if` 式への変形

検出するだけのアルゴリズムを用いたが，一般的なソフトウェアの更新には，この手法では見つけられないバージョン間の差分が含まれる．将来的には，木の編集距離を求めるアルゴリズムを更新前後のソースコードに適用して得られる差分情報をより広い目的に用いることを考えたい．

5.3 利用例:更新対象データの置換

5.2 節で実装した更新プログラム作成ツールで生成される更新プログラムでは，更新対象データの置換を行う処理は，データの種類ごとに作成される関数 `update-<datatype>`の中で定義されている．これらの関数は，置換の対象となるデータを引数として受け取り，このデータを新しいバージョンのデータ構造のデータに上書きする．

本節では，本研究で実装した機構によってソフトウェアの更新を行う際に，更新対象データのクロージャおよび構造体の置換を行うための，更新対象データの置換処理の設計方法を，例を交えて説明する．

5.3.1 クロージャの置換

ここでは，図 5.15(a) に示すコンストラクタが生成するクロージャを，図 5.15(b) に示すコンストラクタが生成するのと同じ型のクロージャに置き換える方法を示す．まず，図 5.15(a) のコンストラクタを用いて `(make-account 100)` を評価して得られるクロージャの内部表現を，図 5.16 に示す．

```
(define (make-account balance)
  (define (withdraw amount)
    (set! balance (- balance amount))
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          (else (error "Unknown message : MAKE-ACCOUNT"))))
  dispatch)
```

(a)

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "Insufficient funds"))
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          (else (error "Unknown message : MAKE-ACCOUNT"))))
  dispatch)
```

(b)

図 5.15: クロージャを生成するコンストラクタ

図 5.15(a) で作られたクロージャを図 5.15(b) で作られるのと同じものに置き換えるには，まずクロージャの内部表現の body スロットを，新しいバージョンの make-account を指すよう変更する必要がある．また，関数の引数の個数などの情報も，新しいバージョンの物に変更する．ここで，新しいバージョンのクロージャがどのような構成になっているか，つまり，新しいバージョンのクロージャの body

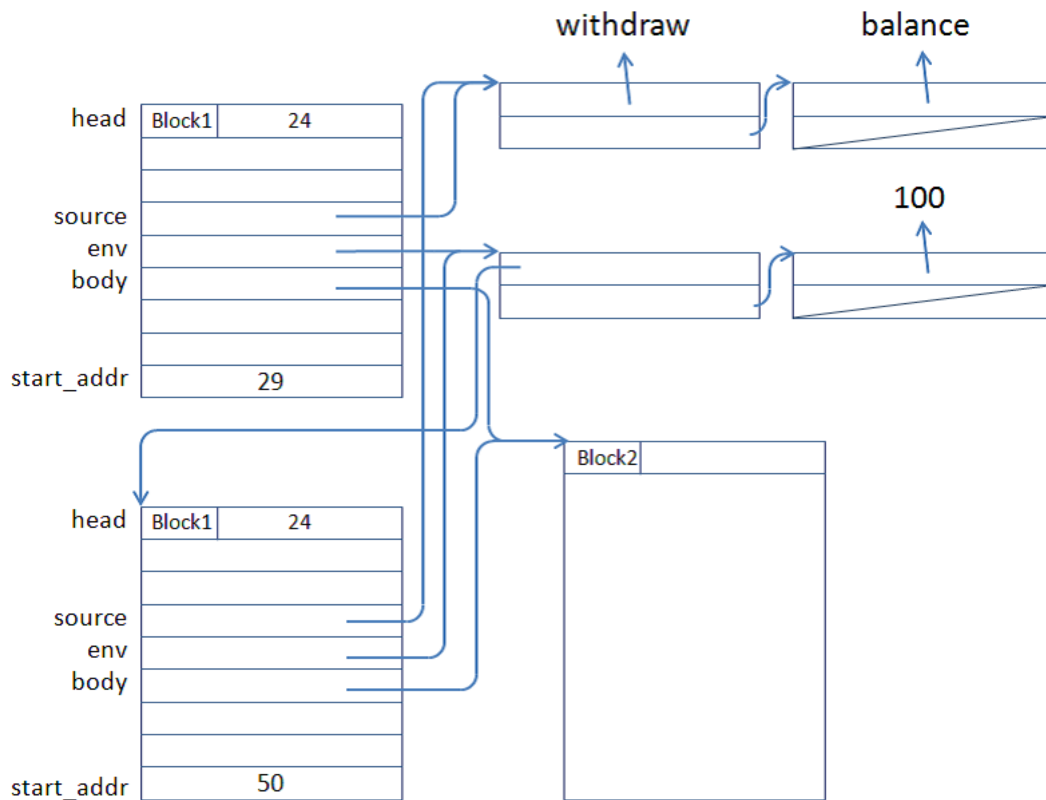

図 5.16: クロージャの内部表現

スロットなど各スロットにどの値が入っているかの情報が必要になる．

本研究で行うクロージャの置換では，データサンプル生成関数によって生成した新データのサンプルを用いて，新しいバージョンのクロージャの構成情報を得る．TUTScheme では，クロージャの const_vec，fun_name，body，arg_info，start_addr の各スロットを別のクロージャのものに置き換える change-bci 関数が定義されている．この関数を用いて，クロージャの body スロットおよび，関数についての各情報を新しい関数のものに更新する．

DSU をサポートする TUTScheme のクロージャの環境は，クロージャの source スロットに格納された変数名のリスト，および env スロットに格納された値のリストからなる．図 5.16 に示したクロージャの source スロットにはリスト (withdraw balance) が，env スロットには図 5.15(a) 中のローカル関数 withdraw のクロージャ

と，整数 100 からなるリストが格納されている．

クロージャの body スロットなどの更新は上に示した方法で全て行えるが，環境内の束縛の更新を行う方法は様々である．図 5.15(a) の make-account で作られたクロージャは，環境に変数 withdraw, balance の束縛を含んでいる．balance は明らかにクロージャの内部状態を表す変数であるので，そのままの値に保つべきであるが，withdraw の値は make-account 関数の中のローカル関数を実行するクロージャである．withdraw の値のクロージャは，make-account で作られたクロージャが更新された後では，新しい make-account のバイトコードによって呼び出しが行われる．そのため，withdraw の値のクロージャも新しい関数のバイトコードに置き換えなければならない．withdraw の値のクロージャは図 5.16 からわかる通り，withdraw の値のクロージャは，make-account で作られたクロージャと，source スロットおよび env スロットの値を共有している．よって，withdraw の値のクロージャの置換は，change-bci 関数を用いて，各スロットの値を新しいバージョンの make-account で作られたクロージャのサンプルの環境の withdraw の値であるクロージャの各スロットの値に置き換えるだけで行える．置換が完了した時のクロージャの内部表現を図 5.17 に示す．図中の赤色で示した部分が更新した箇所である．また，置換処理を行う関数のコードを図 5.18 に示す．なお，図 5.18 中の変数 new-account-sample は更新後のバージョンの make-account 関数で生成されるデータのサンプルである．

5.3.2 構造体の置換

構造体の置換は，構造体の内部表現 (図 5.11) の type スロットの更新と，rest フィールの更新によって行われる．

例として，図 5.11 の student 構造体のデータの置換を行う処理を考える．更新前のバージョンでは student 構造体は変数*version*が 1 であるときに式 (defstruct student name age grade) で定義され，更新後のバージョンでは変数*version*

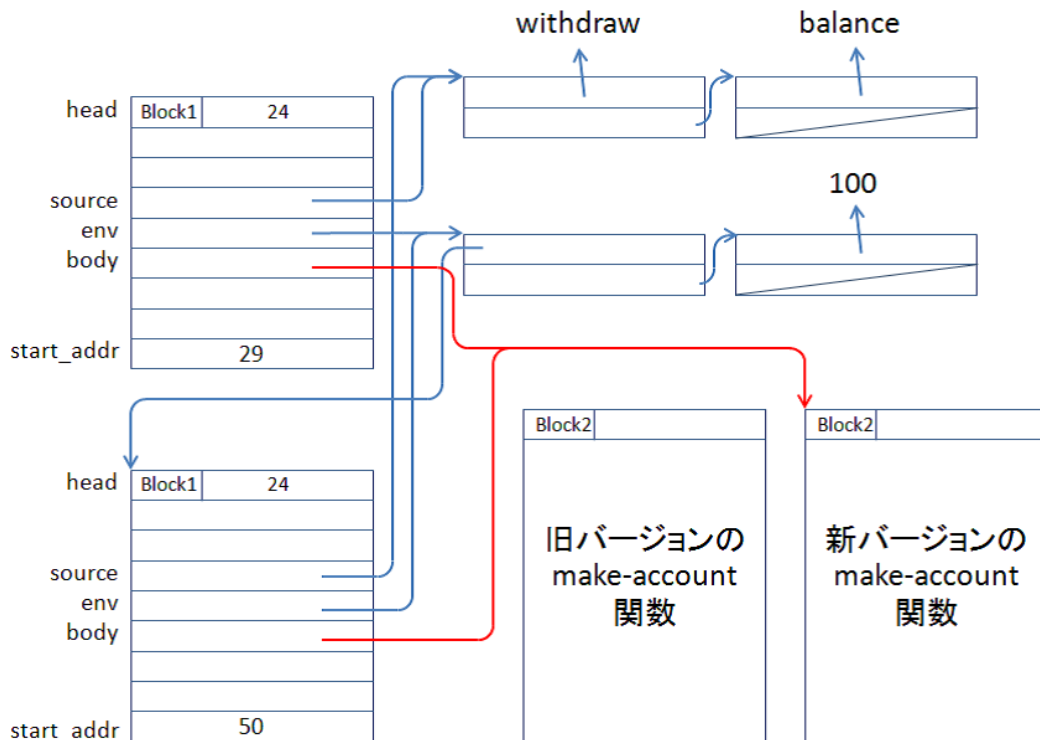

図 5.17: 置換されたクロージャの内部表現

```
(define (update-account x)
  (define (update-account-env n v)
    (cond ((eq? n 'withdraw)
           (car (get-closure-env new-account-sample)))
          (else v)))
  (let ((ns (get-source x)) (vs (get-closure-env x)))
    (change-bci x new-account-sample)
    (put-source x (get-source new-account-sample))
    (put-closure-env x (map update-account-env ns vs))))
```

図 5.18: クロージャの置換を行う関数

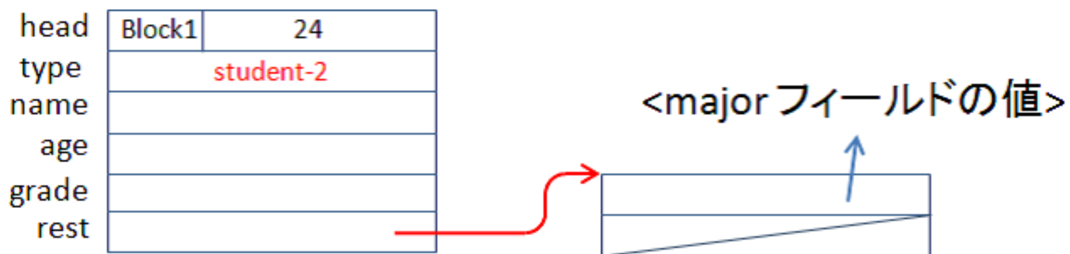

図 5.19: 置換処理後の student 構造体のデータの内部表現

が2であるときに式(defstruct student name age grade major)で定義されるとする．新しいバージョンでは新しいフィールド major が追加される．

まず，構造体のデータの type スロットを更新する．更新前の student 構造体のデータの type スロットにはシンボル student-1 が格納されているが，更新前の student 構造体のデータの type スロットには student-2 が格納される．よって，structure-type-set!関数を用いて構造体のデータの type スロットにシンボル student-2 を格納する．

次に構造体の rest フィールドに，追加されたフィールドを格納する．新しいバージョンで student 構造体に追加されるフィールドは major フィールドであるため，構造体のデータの rest フィールドに，major フィールドの値からなるリストを格納する．置換処理が完了した後の student 構造体のデータを図 5.19 に示す．図中の赤色で示した部分が更新した箇所である．また，置換処理を行う関数のコードを図 5.20 に示す．なお，図 5.20 中の変数 new-student-sample は更新後のバージョンの make-student マクロで生成されるデータのサンプルである．また置換を行った構造体の major フィールドは#f とするようにした．

```
(define (update-student x)
  (structure-type-set! x 'student-2)
  (setf (student-rest x) (list #f)))
```

図 5.20: クロージャの置換を行う関数

第 6 章

評価

本章では，5 章で実装した DSU 機構で可能になる DSU の範囲を明らかにし，また，更新が適用された際の，DSU の機能による，ソフトウェアの実行への影響を測定するために，実験を行った．本章の構成は以下のとおりである．6.1 節では，本研究で実装した機構で更新が行えない例を示し，また実用的な Scheme プログラムの更新を行う例を示し，実際のソフトウェアの更新を行う際の，更新プログラムの作成方法の例を示す．6.2 節では，更新を行わない時の，本研究で実装した DSU をサポートする機構によるオーバヘッドを測定する実験の方法及び結果を示す．6.3 節では，多数のデータを扱うソフトウェアの更新を行う実験を行い，本研究で提案した漸次的なデータの置換の有効性を検証する．6.4 節では，以上の実験の結果について議論する．

実験はすべて表6.1 に示す構成のデスクトップ PC 上で行った．また，TUTScheme の実行形式ファイル raw_tus に渡すオプションを変更し，ヒープ領域の大きさを 48,000[KB] に設定した．また，`xcall` 命令が実行される際に，GC で集められた更新対象データの置換を行う頻度 C_{int} を 100,000 に設定した．この設定は，`xcall` 命令が 100,000 回実行されるごとに 1 回，更新対象データの置換処理を行うことを意味する．

表 6.1: 評価に用いた計算機の性能諸元

プロセッサ	Intel Core Duo (2.00GHz)
主記憶	2GB DDR2 SDRAM
OS	Mac OS X 10.6.6
コンパイラ	GCC 4.0.1 (Apple Inc. build 5465)
コンパイラオプション	`-m32 -g -O2 -lm -lcurses -lreadline -ltermcap`

6.1 ソフトウェアへの DSU の適用

本節では，本研究で実装した DSU 機構で行える更新の範囲を明らかにするため，無限ループ処理を含む関数を実行時に更新する例を試す．

また，実用的なソフトウェアへの DSU の適用の例として，Scheme で記述された超循環評価器 [9] を実行時に更新する．

6.1.1 無限ループ処理を行うソフトウェアの更新

本研究で実装した DSU 機構で，図 6.1 に示す，無限ループ処理を行う関数 `main` を，実行中に更新することを試した．結果，更新を指示しても，関数 `main` の更新処理が始まることはなかった．これは，関数 `main` のスタックフレームが常時実行スタック上に乗っているためである．

このような無限ループ処理を更新する方法として，Ginseng[6] のループ抽出 (loop extraction) が挙げられる．これは，無限ループを行う処理そのものは更新できないが，その内部で呼び出される関数は実行中でない時もあり，更新できることを利用した方法である．ループ抽出ではソースコードを変形し，無限ループ処理の内部で行う処理を別の関数の中に書くようにする．Ginseng における，C 言語で記述さ

```
(define (main)
  (define (loop-body)
    (foo)
    (loop-body))
  (loop-body))

(define (foo) #t)
```

図 6.1: 無限ループ処理

```
int foo() {
...
  L1:while (1) {
    x = x+1;
    if (x == 8) break;
    else continue;
    if (x == 9) return 42;
  }
  return 1;
}
```

図 6.2: C 言語における無限ループ

れたソースコードに対するループ抽出の例を図 6.2 および図 6.3 に示す．図 6.2 は変換前の無限ループ処理のコードであり，図 6.3 はループ抽出の変換が適用されたソースコードである．ここで，関数 foo 中に無限ループを実行し，その中では，関数 L1_loop を呼び出す処理のみを行う．関数 L1_loop は，無限ループ中で行う全ての処理を行う．この場合，関数 foo の処理は更新できないが，L1_loop の処理を更新することが可能な期間は関数 foo の実行中にも存在する．よって，無限ループ中の処理を更新することができる．

```
int L1_loop (int *ret, struct L1_ls *ls) {
  *(ls->x) = *(ls->x) + 1;
  if (*(ls->x) == 8) return (0); // break
  else return (1); // continue
  if (*(ls -> x) == 9) {
    *ret = 42;
    return (2); // return
  }
  return (1);   // implicit continue
}
int foo() {
...
  while (1) {
    code = L1_loop(&retbal, &ls);
    if (code == 0) break;
    else if (code == 1) continue;
    else return (retval);
  }
  return (1);
}
```

図 6.3: ループ抽出

6.1.2 超循環評価器

「計算機プログラムの構造と解釈」(SICP)[9] の 4.1 節では，Scheme プログラムを評価する Scheme プログラム (超循環評価器) の実装法を示している．

SICP の 4.1 節で取り上げられている超循環評価器の実装には，2 つのバージョンが存在する．1 つは SICP の 4.1.4 項までで説明されている，プログラムを実行するごとに構文解析を行っているものであり，もう 1 つは，SICP の 4.1.7 項で取り上げられている，プログラムを読み込む際にのみ構文解析を行い，プログラムの実

行には構文解析済みの「実行手続き」を用いる，高速化版である．

SICP の 4.1.4 項の超循環評価器を，実行時に高速化版に更新する方法を考える．この 2 つのバージョン間には，式を評価するための処理が大幅に変更されており，超循環評価器の扱うデータの変換処理は簡単ではない．そこで，超循環評価器の更新前後のそれぞれのバージョンの実装法を調べる必要がある．

超循環評価器の実行は，ユーザが入力した式の読み込みと，式の評価と，評価の結果の出力を行うループ処理からなる．式の評価では，変数の束縛の集まりである環境から，変数に束縛した値を探す．環境は，1 個以上のフレームと呼ばれるデータ構造を含んでいる．各変数の名前と，束縛された値は，フレームの内部から参照される．

更新プログラムは，5.2.2 節で述べた更新プログラム生成ツールによって生成する．また，生成プログラムをこの章の以下の議論の内容に従い，修正する．

更新プログラムの`*dsu-functions*`の定義を見ると，2 つのバージョンの間でプログラムの評価を行う `my-eval` 関数の定義が異なることがわかる．したがって，`my-eval` の実行中には更新処理は始まらない．よって，変更を行う対象のデータは，`my-eval` が実行されていないときに保持されるデータに限られる．このようなデータは，プログラムの評価を行うための環境を構成するフレーム構造のみである．

フレームの構造は，更新前後のバージョンで同じく，変数名を格納する要素と，変数の値を格納する要素からなるリストである．しかし，このリスト構造は 4.5.1 項に示したような，リストのデータ型を表すタグを備えていない．また，GC を用いて更新対象のフレームのデータを探すために，更新前後のバージョンで，このリスト構造に付けられるタグを変える必要がある．そこで，フレームのコンストラクタである `make-frame` 関数に，フレームを表すリスト構造の先頭に，シンボル `frame-228` が追加されるようにした．また，新しいバージョンの `make-frame` 関数には，同様に，フレームの先頭にシンボル `frame-236` が追加されるようにし

```
#0#=((frame-228 ((f b a ...) (procedure (x) ((+ x x)) #0#) "abc" 3 ...))
```

(a) 更新前のバージョンにおける超循環評価器の環境の構成

```
#0#=((frame-236 ((f b a ...) (procedure (x) #<function> #0#) "abc" 3 ...))
```

(b) 更新後のバージョンにおける超循環評価器の環境の構成

図 6.4: 両バージョンにおけるフレーム構造の構成

た．次に，2 つのバージョンにおけるフレームの構造の違いを比較した．2 つのバージョンのそれぞれの超循環評価器において，Scheme の式 `(define a 3)`, `(define b "abc")`, `(define f (lambda (x) (+ x x)))` を順に評価した際の，フレームの構造を図 6.4 に示す．フレームの変数名を格納する要素には両バージョンともに変数名を表すシンボルのリストが格納されているが，変数の値を格納する要素では手続きを表すリストに違いが見られた．よって，フレームの変数の値を格納する要素のうち，手続きを表すリストを変換する必要があることがわかる．手続きを表すリストのコンストラクタ `make-procedure` の呼び出し元を調べると，古いバージョンでは，`lambda` 式の本体をそのまま手続きを表すリストに入れているのに対し，新しいバージョンでは，`lambda` 式の本体を `analyze-sequence` 関数に適用してから手続きを表すリストに入れている．よって，フレームの置き換え処理は，図 6.1.2 に示すように，リストの先頭要素をシンボル `frame-238` に付け替え，フレーム構造の中の手続きの値を表すリストの中の `lambda` 式の本体を表すリストに `analyze-lambda` を適用し，その結果得られる関数に差し替えればよい．

また，GC を行う際に更新対象のフレームを集めるため，変換を要するデータのリストを表す `*dsu-object-checklist*` に，更新前のフレームの型の識別子を表す `(list-car 'frame-228)` を追加した．データの変換を行う `dsu:object-updater` 関数は，引数がリストであり，先頭要素が `frame-228` である場合に引数を図 6.1.2 の update-frame 関数に適用するようにした．

```
(define update-frame
  (lambda (x)
    (define (analyze-proc val)
      (if (and (list? val) (eq? (car val) 'procedure))
          (set-caddr! val (analyze-sequence (caddr val)))))
    (set-car! x 'frame-236)
    (if (list? (cdadr x))
        (for-each analyze-proc (cdadr x)))))
```

図 6.5: フレームの置換を行う関数

フレームのデータを置き換えるため，フレームの内部を参照する関数にアクセスバリアを設ける．更新後のバージョンで定義された関数で，フレームの内部を参照する関数は `frame-variables`, `frame-values` および `add-binding-to-frame` の 3 つである．グローバルな定義が更新される時に，これらの関数がアクセスバリア付きで定義されれば，新しいバージョンのソフトウェアが古いバージョンのフレームの内部を参照することはない．更新プログラムの，グローバルな定義を更新する関数 `dsu:update-definitions` に，フレームの内部を参照する関数 `frame-variables`, `frame-values` および `add-binding-to-frame` にアクセスバリアを追加する処理を追加した．

追加されたアクセスバリアによるオーバヘッドを小さく抑えるため，更新プログラムのアクセスバリアを外す関数 `dsu:remove-accessbarrier` に，フレームの内部を参照する関数からアクセスバリアを外す処理を追加した．

修正を加えた更新プログラムを付録 A に示す．この更新プログラムを適用する前と，適用した後で，図 6.6 に示す `tak` 関数を超循環評価器上で実行した．本研究で DSU 機構を実装した TUTScheme 上で，超循環評価器の駆動ループを実行させる間に更新を適用した場合の，超循環評価器上で，更新前と更新後に式 `(tak 18 12 6)` を評価した際の実行時間を表 6.2 に示す．実行時間の測定は更新前と更新後

```
(define (tak x y z)
  (if (not (< y x))
      z
      (tak (tak (1- x) y z)
           (tak (1- y) z x)
           (tak (1- z) x y))))
```

図 6.6: 実験に用いた tak 関数

表 6.2: 超循環評価器の性能測定実験結果

	(tak 18 12 6) の計算時間
更新前	14.58[s]
更新後	9.77[s]
最初から更新後バージョンで実行	9.72[s]

にそれぞれ 10 回ずつ行い，10 回測定した平均値を結果とした．また，更新を行わず，最初から更新後バージョンで動作している超循環評価器で同じ式を評価した場合の結果を併記した．実行時間を評価した結果，更新後のソフトウェアの実行速度は，最初から更新後のバージョンで動作しているソフトウェアとほとんど変わらないことがわかった．なお，更新後しばらくの間はアクセスバリアの挿入された関数を用いてソフトウェアが実行される．アクセスバリアの挿入された状態では，式 `(tak 18 12 6)` の評価には，19.5[s] 程の時間がかかった．

表 6.3: 通常実行時のオーバヘッド測定実験結果

	(tak 30 20 10) の計算時間
DSU 無し	85.17[s]
DSU 有り	83.76[s]

6.2 通常実行時のオーバヘッド

DSU 機構の実装では，主に `xcall` 命令および，`xreturn`, `lreturn` 命令に DSU の機能を呼び出す処理を追加している．そのため，関数呼び出しを多く行う処理では，DSU 機構を実装していない TUTScheme に比べて，実行時間が余分にかかると考えられる．

そこで，本研究で DSU 機構を実装した TUTScheme と，DSU 機構を実装していない TUTScheme で，図 6.6 に示す `tak` 関数の実行時間を比較した．`tak` 関数の引数には，x=30, y=20, z=10 を用いた．実行時間の計測には，TUTScheme の組み込みマクロ `time` を用いた．

実験では各々の場合について計測を 10 回行い，その結果の平均値を結果とした．結果を表 6.3 に示す．結果から，DSU 機構を実装した TUTScheme ではオーバヘッドが見られないことがわかった．

6.3 データの更新中におけるオーバヘッド及び停止時間

本研究で提案した，漸次的なデータの更新における，ソフトウェアの停止時間を測定するため，またデータの更新中におけるソフトウェアの実行性能へのオーバヘッドを測定するため，以下の実験を行った．

まず，更新の対象となるクロージャを，図 6.7 のコンストラクタを用いて，10^5

```
(define (make-account balance)
 (define version 1)
 (define (withdraw amount)
   (begin (set! balance (- balance amount))
          balance))
 (define (dispatch m)
   (cond ((eq? m 'version) version)
         ((eq? m 'balance) balance)
         ((eq? m 'set-balance) (lambda (x) (set! balance x)))
         ((eq? m 'withdraw) withdraw)
         (else (error "Unknown request -- MAKE-ACCOUNT" m))))
 dispatch)
```

図 6.7: 実験に用いたコンストラクタ

個作成した．これらのクロージャは，変数 acc から参照されるリストに参照されている．

次に，TUTScheme の対話形式プロンプトから，

```
(set! *dsu-mode* 1)
```

と入力し，更新を指示する．更新を支持した直後に，関数の定義の置き換えが行われる．

次に，プロンプトに

```
(mainloop)
```

と入力する．mainloop 関数の定義を図6.8 に示す．mainloop 関数は，make-account で作成されたクロージャのアクセサを呼び出す処理を繰り返す．この繰り返し処理は，関数 inquiries の実行中に，ファイルの読み込みが終わったときに終了する．読み込み対象のファイルは，繰り返し処理の個々の処理内容を記したリストを 10^6

表 6.4: データの更新によるオーバヘッド測定実験結果

	合計実行時間	停止時間
$D_{max} = 10$	96.10[s](108.4%)	0.18[ms]
$D_{max} = 1,000$	93.66[s](105.7%)	12.06[ms]
$D_{max} = 100,000$	92.88[s](104.8%)	839.7[ms]

個含んでいるため，繰り返し処理は 10^6 回行われる．この処理の実行中に，GC が呼び出され，更新対象データがリストに格納され，置換される．

この処理を，一度に更新するデータの数 D_{max} を変更させて，合計実行時間と停止時間を測定した．停止時間の測定には，プロセッサのタイムスタンプカウンタを，合計実行時間には，TUTScheme の組み込み関数 `ptime` を用いた．結果を表 6.4 に示す．なお，合計実行時間の欄の括弧内の値は，データの更新を行わなかった場合と比べた割合である．

また，$D_{max} = 1,000$ のときに処理を行った際の，GC の実行時間を測定した結果は，更新対象データを集める処理を行った場合は平均 252.7[ms] であり，更新対象データを集める処理を行っていないときは平均 151.3[ms] であった．

また，DSU を実装していない TUTScheme での GC の実行時間の平均は 114.1[ms] であった．DSU を実装した TUTScheme の通常実行時での GC に余分な時間がかかるのは，更新対象データを集める処理を行うかの判定を行う分岐処理に原因があると考えられる．

```
(define (mainloop)
  (if (null? prev-time) (set! prev-time (ptime)))
  (if (let ((result (proceed (inquiries))))
(take-stats) result)
      (mainloop)
      '()))
(define (proceed l)
  (if (null? l)
      #f
      (let ((account (car l)) (action (cadr l)) (amount (caddr l)))
(case action
  ((withdraw) (set-account-withdraw (get-accarray-account account) amount))
  ((balance)  (get-account-balance  (get-accarray-account account)))
  ((make)     (add-account (make-account amount))))
#t)))
(define inquiries
  (let ((source (open-input-file *inquiry-file-name*)))
    (lambda ()
      (let ((r (read source)))
        (if (eof-object? r)
            '()
            r)))))
```

図 6.8: 実験に用いた mainloop 関数及び補助関数

6.4 考察

6.1.2 節では，Scheme の超循環評価器の更新を実演した．しかし，SICP に示されていた超循環評価器のプログラムでは，環境フレームなどの，リストによって実装された合成データの一部にデータ型を表すタグがついていなかったので，今回は実装の一部を修正し，環境フレームを表すリストにタグを付加した．また，この

タグをバージョンごとに異なるものにするため，決め打ちでタグにバージョンを付加した．バージョンの情報は，ソフトウェアの開発者が手動で付けるのではなく自動で付けられることが望ましい．今後は自動でバージョン番号を付加する `list` 関数を用いるよう超循環評価器を書きなおして更新を試したい．

6.2 節では，更新を行っていないときの，DSU 機構を実装した TUTScheme の実行性能のオーバヘッドがわずかであることが示された．しかし，さらなる性能改善 `xcall` 関数および `xreturn` 関数の DSU 機能の実装された TUTScheme のパフォーマンス改善のため，復帰命令に追加される処理を減らす事を考える．

`xcall` 命令に追加された処理は，DSU の処理のエントリポイントであるため，外すことはできないが，`xreturn` 命令に追加された処理は，DSU を行う際に特定の関数が復帰する時にのみ行えばよい．`xreturn` 命令に処理を追加し，実行スタック上の更新される関数のフレームの数を示す変数を書き換える代わりに，実行中の更新対象の関数が全て復帰された時に，リターンバリアによってグローバル変数の更新を起動する方法が考えられる．

グローバル変数の更新は，スタック上の更新される関数がすべて取り除かれた時点で，可能になる．更新される関数のスタックフレームのうち，最もスタックの底に近いフレームの関数が復帰した時には，更新対象のどの関数も実行中でなくなる．このスタックフレームの復帰時の処理を変更し，このフレームの関数が復帰する時にグローバル変数の定義を置き換える関数を呼び出すようにすれば，オーバヘッドを小さくしながら安全にグローバル変数の更新が行える．継続の呼び出しによってスタックが入れ替わったときには，再度実行スタックを走査し，最もスタックの底に近い，更新される関数のスタックフレームの関数の復帰時の処理を変更する．継続が呼び出されるのは，`xcall` 命令の特定の箇所に限定されるため，この機能の実装は困難ではないと考える．

6.3 節では，GC の実行時間が DSU をサポートしない TUTScheme に比べて大幅に長くなることが明らかになった．GC に挿入された処理は，ソフトウェアの更新

を行っていない時には無駄な処理になる．コピー GC を行う際に個々のデータをコピーする C 言語の関数 `gcunit` を，更新を行うときの，更新対象データを探す処理付きのものと，更新を行わないときの，オーバヘッドをもたらさないものの 2 つを用意し，関数ポインタを用いてそれぞれの関数の呼び出しを切り替えるようにすれば，GC の実行時間のオーバヘッドは小さくできると考えられる．

6.3 節の結果から，D_{max} の値を小さく設定することによって，データの更新によるソフトウェアの停止時間を，非常に短く区切ることができることがわかる．しかし，D_{max} の値を小さくすると，合計実行時間が増加する．これは，D_{max} の値が小さい時には，その分ソフトウェアの実行を長く進めないと，全ての旧データを更新し終わらないためである．$D_{max} = 10$，$C_{int} = 100,000$ に設定した場合では，100,000 個の旧データを更新し終わるまでには，10^9 回 `xcall` 命令が実行される．この更新が終わるまでの間は，旧データが更新し終わったことを確かめないので，ソフトウェアの実行は，アクセスバリアの挿入されたアクセサを用いて行われる．このとき，新データの内部への参照にも，アクセスバリアの挿入されたアクセサを用いるために，無駄な処理が行われると考えられる．

また，D_{max} を小さい値に設定した結果から，データの更新による停止時間は，一定未満にする事ができないことがわかる．D_{max} を 10 に設定したときの停止時間は 0.18[ms] であり，この結果から単純に計算すると，旧データ 1 つを更新するのに 18[μs] の時間がかかる．実際には，`xcall` 命令からの，データの更新処理を呼び出す処理が停止時間に含まれるため，旧データを 1 つずつ更新する場合の停止時間はこれより長くなると考えられる．

第 7 章

結論

本研究では，Scheme プログラミング言語処理系 TUTScheme に変更を施し，Scheme で記述されたソフトウェアを対象とした DSU 機構を実装した．評価実験では，この DSU 機構を用いて超循環評価器のソフトウェアの更新が行えることを示し，また，ソフトウェアの更新におけるデータの変換による停止時間を小さくできたことを実証した．

今後の課題としては，4.7.3 節に述べた，ソースコードの静的解析をさらに拡充させ，ソフトウェアの変更についてより多くの情報を，更新プログラムを作成する開発者に提供するようにしたい．また，ソースコードの静的解析をコンパイラと統合し，新旧のバージョン間での，バイトコードのラベルを対応付けることで，実行スタック上の，実行中の関数のスタックフレームを，新しいバージョンの関数のフレームに書き換えられるようにしたい．また，この技法を継続の更新に用いることで，継続を多用するソフトウェアの更新，および常に実行されている関数の更新を可能にしたい．また，DSU 機能を実装したことによるオーバヘッドを小さくするために，より良い実装法を模索しなければならない．

最後に，本研究において最も重要視した，ソフトウェアの実行の停止時間の低減を本当の意味で達成するには，本研究で実装した，データの更新処理の分割では不十分である．本研究で提案した方法では，ソフトウェアの実行を一時停止し，ヒープ領域を全探索する Stop-the-World 方式のコピー GC を前提としている．このため，GC による停止時間の問題が依然残っている．今後は，Incremental GC な

ど，停止時間を小さくできるGCを用いて，更新対象となる旧データを発見することと，全ての旧データが更新されたことを確認することができるかを明らかにしたい.

謝辞

大学院で研究を進めるにあたり，本論文の提出に至るまで，多くの方々にお世話になりました．指導教員の小宮常康准教授には，日頃よりご指導ご鞭撻を頂いた上，研究の遂行および論文執筆にあたり多くの相談に応じていただき，またご多忙中にもかかわらず論文の草稿を添削していただき，大変多くのご助言をいただきました．ここに深く御礼申し上げます．多田好克教授，佐藤喬助教には，研究の遂行にあたり多くのご助言，ご意見をいただきました．ここに御礼申し上げます．最後に，基盤ソフトウェア学講座の学生諸氏とは，日頃の研究室生活で多くの討論を行い，ともに研究を進めました．とくに石橋崇氏には研究を進めるにあたり，多くのご助言をいただきました．ここに深く感謝致します．

参考文献

[1] The Revised6 Report on the Algorithmic Language Scheme, http://www.r6rs.org/.

[2] P.Bhattacharya and I.Neamtiu: "Dynamic Updates for Web and Cloud Applications," *Proc. APLWACA '10*, pp.21–25, June 2010.

[3] Kahua Project: http://www.kahua.org/.

[4] TUTScheme, http://www.yuasa.kuis.kyoto-u.ac.jp/~komiya/tus_intro.html.

[5] I.Neamtiu and M.Hicks: "Safe and Timely Dynamic Updates for Multi-threaded Programs," *Proc. 2009 ACM SIGPLAN Conf. PLDI*, pp.13–24, June 2009.

[6] I.Neamtiu, M.Hicks, G.Stoyle, and M.Oriol: "Practical Dynamic Software Updating for C," *Proc. ACM SIGPLAN Conf. PLDI*, pp.72–83, June 2006.

[7] A.Orso, A.Rao, and M.J.Harrold: "A Technique for Dynamic Updating of Java Software," *Proc. 18th IEEE ICSM*, pp.649–658, October 2002.

[8] S.Subramanian, M.Hicks, and K.S.McKinley: "Dynamic Software Updates: A VM-centric Approach," *ACM SIGPLAN Notices*, Vol.44, pp.1–12, June 2009.

[9] ジェラルド・ジェイ・サスマン，ハロルド・エイブルソン，ジュリー・サスマン: 「計算機プログラムの構造と解釈　第二版」，ピアソンエデュケーション，ISBN 4-89471-163-X, 2000年2月

[10] 湯淺 太一，中川 雄一郎，小宮 常康，八杉 昌宏: 「リターン・バリア」，情報処理学会論文誌 第四一巻，No.SIG 9, pp.87–99, 2000.

付録A

超循環評価器の更新に用いた更新プログラム

```
(define *dsu-functions*
  (list my-eval tagged-list? make-frame))

(define (dsu:update-definitions)
  (define new-my-eval
    (lambda (exp env) ((analyze exp) env)))
  (define new-analyze
    (lambda (exp)
      (cond ((self-evaluating? exp) (analyze-self-evaluating exp))
            ((variable? exp) (analyze-variable exp))
            ((quoted? exp) (analyze-quoted exp))
            ((assignment? exp) (analyze-assignment exp))
            ((definition? exp) (analyze-definition exp))
            ((if? exp) (analyze-if exp))
            ((lambda? exp) (analyze-lambda exp))
            ((begin? exp) (analyze-sequence (begin-actions exp)))
            ((cond? exp) (analyze (cond->if exp)))
            ((application? exp) (analyze-application exp))
            (else (my-error "Unknown expression type -- ANALYZE" exp)))))
  (define new-analyze-self-evaluating
    (lambda (exp) (lambda (env) exp)))
```

```
(define new-analyze-quoted
  (lambda (exp)
    (let ((qval (text-of-quotation exp)))
      (lambda (env) qval))))
(define new-analyze-variable
  (lambda (exp)
    (lambda (env)
      (lookup-variable-value exp env))))
(define new-analyze-if
  (lambda (exp)
    (let ((pproc (analyze (if-predicate exp)))
          (cproc (analyze (if-consequent exp)))
          (aproc (analyze (if-alternative exp))))
      (lambda (env)
        (if (true? (pproc env)) (cproc env) (aproc env))))))

(define new-analyze-sequence
  (lambda (exps)
    (define (sequentially proc1 proc2)
      (lambda (env) (proc1 env) (proc2 env)))
    (define (loop first-proc rest-procs)
      (if (null? rest-procs)
          first-proc
          (loop (sequentially first-proc (car rest-procs)) (cdr rest-procs))))
    (let ((procs (map analyze exps)))
      (if (null? procs)
          (my-error "Empty sequence -- ANALYZE"))
      (loop (car procs) (cdr procs)))))

(define new-analyze-lambda
  (lambda (exp)
```

```
    (let ((vars (lambda-parameters exp))
          (bproc (analyze-sequence (lambda-body exp))))
      (lambda (env) (make-procedure vars bproc env)))))

(define new-analyze-assignment
  (lambda (exp)
    (let ((var (assignment-variable exp))
          (vproc (analyze (assignment-value exp))))
      (lambda (env) (set-variable-value! var (vproc env) env) 'ok))))
(define new-analyze-definition
  (lambda (exp)
    (let ((var (definition-variable exp))
          (vproc (analyze (definition-value exp))))
      (lambda (env) (define-variable! var (vproc env) env) 'ok))))
(define new-analyze-application
  (lambda (exp)
    (let ((pproc (analyze (operator exp)))
          (aprocs (map analyze (operands exp))))
      (lambda (env)
        (execute-application
          (pproc env)
          (map (lambda (aproc) (aproc env)) aprocs))))))
(define new-execute-application
  (lambda (proc args)
    (cond ((primitive-procedure? proc)
           (apply-primitive-procedure proc args))
          ((compound-procedure? proc)
           ((procedure-body proc)
            (extend-environment
              (procedure-parameters proc)
              args
```

```
              (procedure-environment proc))))
          (else (my-error "Unknown procedure type -- EXECUTE-APPLICATION" proc)))))
(define new-tagged-list?
  (lambda (exp tag) (if (pair? exp) (eq? (car exp) tag) false)))
(define new-make-frame
  (lambda (variables values) (list 'frame-236 (cons variables values))))
(set! my-eval new-my-eval)
(eval '(define analyze))
(set! analyze new-analyze)
(eval '(define analyze-self-evaluating))
(set! analyze-self-evaluating new-analyze-self-evaluating)
(eval '(define analyze-quoted))
(set! analyze-quoted new-analyze-quoted)
(eval '(define analyze-variable))
(set! analyze-variable new-analyze-variable)
(eval '(define analyze-if))
(set! analyze-if new-analyze-if)
(eval '(define analyze-sequence))
(set! analyze-sequence new-analyze-sequence)
(eval '(define analyze-lambda))
(set! analyze-lambda new-analyze-lambda)
(eval '(define analyze-assignment))
(set! analyze-assignment new-analyze-assignment)
(eval '(define analyze-definition))
(set! analyze-definition new-analyze-definition)
(eval '(define analyze-application))
(set! analyze-application new-analyze-application)
(eval '(define execute-application))
(set! execute-application new-execute-application)
(set! tagged-list? new-tagged-list?)
(set! make-frame new-make-frame)
```

```
  ;; added : access-barrier
  (set! frame-variables
    (lambda (frame)
      (cond ((eq? (car frame) 'frame-236)
              (caadr frame))
             ((eq? (car frame) 'frame-228)
              (update-frame frame) (caadr frame)))))

  (set! frame-values
    (lambda (frame)
      (cond ((eq? (car frame) 'frame-236)
              (cdadr frame))
             ((eq? (car frame) 'frame-228)
              (update-frame frame) (cdadr frame)))))

  (set! add-binding-to-frame!
    (lambda (var val frame)
      (cond ((eq? (car frame) 'frame-236)
              (set-car! (cadr frame) (cons var (frame-variables frame)))
              (set-cdr! (cadr frame) (cons val (frame-values frame))))
             ((eq? (car frame) 'frame-228)
              (update-frame frame)
              (set-car! (cadr frame) (cons var (frame-variables frame)))
              (set-cdr! (cadr frame) (cons val (frame-values frame)))))))

  (eval '(define new-frame-sample (make-frame-sample))))

(define old-frame-sample (make-frame-sample))

(define (mk-checklist-elem e)
```

```
  (cond ((list? e) `(list-car ,(car e)))
        ((function? e) `(closure ,e))
        ((structure? e) `(struct ,(structure-type e)))))

(define *dsu-object-checklist*
  (map mk-checklist-elem
       (list old-frame-sample)))

;; modified : frame abstruction updating
(define update-frame
  (lambda (x)
    (define (analyze-proc val)
      (if (and (list? val) (eq? (car val) 'procedure))
          (set-caddr! val (analyze-sequence (caddr val)))))
    (set-car! x 'frame-236)
    (if (list? (cdadr x))
        (for-each analyze-proc (cdadr x)))))

(define dsu:object-updater
  (lambda (x)
    (define (old-frame? x)
      (and (list? x) (eq? (car x) 'frame-228)))
    (cond
      ((old-frame? x) (update-frame x))
      (else x))))

(define dsu:remove-accessbarrier
  (lambda ()
    ;; added : access-barrier removal
    (set! frame-variables
      (lambda (frame)
```

```
    (caadr frame)))

(set! frame-values
  (lambda (frame)
    (cdadr frame)))

(set! add-binding-to-frame!
  (lambda (var val frame)
    (set-car! (cadr frame) (cons var (frame-variables frame)))
    (set-cdr! (cadr frame) (cons val (frame-values frame)))))))
```