メモリ管理シミュレータの構想

内山 雄司　脇田 建
東京工業大学大学院 情報理工学研究科 数理・計算科学専攻
{utiyama, wakita}@is.titech.ac.jp

概 要

本稿では, アルゴリズムの性能評価を容易にするためのシミュレータを提案する. シミュレータは, 既存の処理系からメモリ管理機能を分離し, この部分の実装のみを独立に変更可能にしたものであり, この上に様々なアルゴリズムを容易に記述することが可能となる. 本手法は, 性能評価のために既存の実用的なアプリケーションをそのまま利用できる点, シミュレータ上にメモリ管理を実装したものを実用的な処理系として利用できる点に特徴がある.

1　はじめに

現在, 広く利用されている多くの言語処理系には, 実行時システムによる自動メモリ管理機能が実装されている. 自動メモリ管理機能は, プログラマを明示的なメモリ管理という繁雑で困難な作業から解放するという点で重要である.

メモリ管理のための処理はプログラムの実行を一時的に停止させて行われるため, 処理系に実装される自動メモリ管理機能には高い性能が求められる. 効率の良いメモリ管理アルゴリズムを目指す研究は, 現在でも活発に続けられている.

メモリ管理のアルゴリズムの性能を評価する手段としては, 種々のアルゴリズムを既存の処理系に実装し, それぞれの実装の上で同一のプログラムを実行させたときの性能を比較するというものが一般的である. しかしながら, 既存の処理系の多くには特定のアルゴリズムに基づくメモリ管理機能が実装されており, そのような処理系では, 処理系の実装自体がメモリ管理機能の実装に強く依存していることが多い. そのため, アルゴリズムの評価のために処理系の実装を変更するという作業には多大な労力を要する場合が多い.

そこで本研究では, メモリ管理アルゴリズムの性能評価を簡単に行うためのシミュレータを提案する. 提案するシミュレータは, メモリ管理に関わる部分の実装のみを独立に変更できる言語処理系として提供される. 本研究で提案されるシミュレータを利用することで, 一般的なメモリ管理アルゴリズムを容易に記述し, その上で既存の実用的なアプリケーションを用いた性能評価を行うことが可能となる.

以下に本論文の構成を示す. 2 節では, メモリ管理のシミュレータに求められる要件を明らかにした上で, 本研究のアプローチを示す. 3 節では本システムが対象とするモデルを示す. 4 節では, 前節でのモデル化に基づいて, 本システムが提供するインタフェースについて解説する. 5 節では本システムの実装について解説する. 6 節では, システムの効率をあげるための特化方法を提案する. 7 節では, 本システムに関する未解決な問題と将来の課題について述べ, 8 節で本稿をまとめる.

2　アプローチ

メモリ管理のシミュレータに求められる要件として, 以下の 3 点を挙げることができる.

簡潔性 シミュレータを用いた性能評価の必要性は, メモリ管理のアルゴリズムを実際の処理系に実装することの困難さから生じるものである. したがってシミュレータには, アルゴリズムを容易に記述して性能を評価できることが求められる.

予測性 シミュレータ上での実験によって得られる結果は, メモリ管理アルゴリズムを実際の処理系に実装した際の性能を予測できるものでなければならない.

一般性 メモリ管理を実現するアルゴリズムには, 様々なものがある. シミュレータは, これらの

アルゴリズムに対して広く適用できなければならない.

本研究では, これらの要件を満たすシミュレータとして, 既存の言語処理系をシミュレータとして利用するというアプローチをとる. アルゴリズムの評価のために既存の処理系を変更することの困難さは, 処理系とメモリ管理の実装とが互いに強く依存している点にあった. そこで本研究では, 既存の処理系からメモリ管理に関する部分を独立させ, メモリ管理の実装のみを変更可能にする. 処理系の本体とメモリ管理の実装とは, 処理系によって定義されたインタフェースを利用して, 互いに協調して動作する.

本研究で提案するシミュレータは, 上に述べた 3 点の要件を満足する.

- まず, 簡潔性については, 言語処理系からメモリ管理機能を独立させることで, 処理系の実装に立ち入らずにメモリ管理の処理のみを記述することが可能となる.

- 次に, シミュレータの予測性について述べる. 本手法の特色として, アルゴリズムの性能評価のために, 既存の処理系で動作する実用的なアプリケーションをそのまま利用できるということがある. 言語処理系によるメモリ管理が重要になるのは, プログラマが明示的なメモリ管理を行うことが困難な大規模アプリケーションの場合であるから, 既存のアプリケーションを用いた性能評価を行えることは, シミュレータの要件である予測性のうえで非常に重要である.

- 最後に, シミュレータの一般性については, 3 節で述べるようにアプリケーションプログラムの動作をモデル化し, そのモデルに基づくインタフェースを処理系が正しく提供することで, シミュレータ上に主要なアルゴリズムを記述する表現力を提供する.

メモリ管理機能の実装を変更可能にする研究としては, Customisable Memory Manager (CMM)[1] が知られている. これは, アプリケーションに特化したメモリ管理を行うことで実行性能を向上させることを目的としたものであり, プログラマがメモリ管理の実装を容易に変更できる枠組みを提供するという点で本研究との関連性がある. CMM では, メモリ割り当ての際の処理をプログラマが自由に記述できるが, 処理系の協力を仮定していないために, メモリ割り当て時以外の処理が必要なアルゴリズム (参照カウンタ方式, 世代別コピー方式など) を記述することは難しく, シミュレータとしては一般性の点に問題がある. これに対して本手法では, 処理系がメモリ管理システムのためのインタフェースを提供して協調することで, これらのアルゴリズムをも容易に実装することが可能である.

また, 実行時システムをメモリ管理システムから独立させるものとして, Java の ExactVM (EVM) での研究 [3] がある. これは, Java の仮想機械にメモリ管理機能を実装する際のインタフェースを定めたものであり, これによって, メモリ管理に関する処理を記述することが容易になる.

3 実行モデル

本研究で提案するシミュレータは, 既存の言語処理系からメモリ管理を行う部分を独立させたものである. そのシステム構成を図 1 に示す.

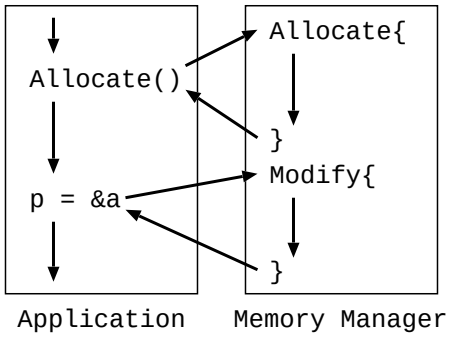

図 1: システム構成

実行時システムは, プログラムの実行によってメモリ空間に対する操作が発生するたびに, ユーザによって記述されたメモリ管理システムの関数を呼び出す.

以下では, 本システムが対象とするモデルについて説明する. まず, 実行時システムが扱うメモリ空間をモデル化する. 次に, そのようなメモリ空間に対してプログラムが行う操作をモデル化する.

3.1 メモリ空間

メモリ空間は, 内部に複数のオブジェクトを含むものとする. メモリ空間内に存在する各オブジェク

トは, 複数のフィールドから構成され, 各フィールドには値または他のオブジェクトへの参照が格納されるものとする. 実行時システムは, 各フィールドに格納されているものが値か参照かを識別可能であり, メモリ管理システムがこの情報を得るための手段を提供するものとする.

また, 本システムが扱うメモリ空間は, オブジェクトを動的に割り当てるためのヒープ領域と, それ以外の領域 (レジスタ, スタック, 大域データ領域など) とに分類される. ヒープ領域以外の領域を root set と呼ぶ.

このようなメモリ空間のモデルに対して, ユーザが記述するメモリ管理システムは, ヒープ領域を管理するものであるとする. root set に存在するオブジェクトが保持しているヒープ領域への参照を得る手段は, 実行時システムが提供するものとする.

図 2 に, メモリ空間のモデルを示す.

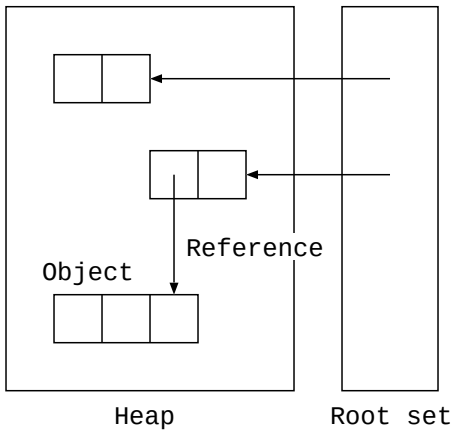

図 2: メモリ空間

3.2 プログラム

メモリ空間内に存在するオブジェクトは, 複数のフィールドを持つ. ユーザが実行するプログラムは, これらのフィールドにアクセスし, フィールドに格納されている値, または参照を変更する存在としてモデル化できる.

このような, プログラムのふるまいは, 以下のように分類できる. このモデルに基づいて, 参照カウンタ方式 [2], マークスイープ方式 [9], コピー方式 [4] に代表される, 主要なメモリ管理アルゴリズムを記述することが可能である.

Creation メモリ管理システムにメモリ割り当てを要求し, その結果としてヒープ領域に割り当てられた新たなオブジェクトへの参照を生成する操作.

Duplication ヒープ領域内に存在するオブジェクトへの参照を他のオブジェクトのフィールドに代入することによって, 新たな参照を生成する操作.

Deletion オブジェクトのフィールドへの代入, またはオブジェクトの消滅が発生した結果として, それまでに存在した参照を消去する操作.

Use オブジェクトへの参照を介して, そのオブジェクトを利用する操作.

上述のモデルによる, メモリ空間の参照に対する操作の様子を, 図 3 に示す.

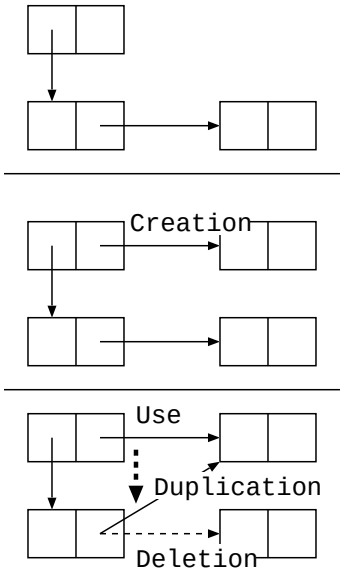

図 3: プログラムによる参照の操作

4 インタフェース

シミュレータは, ユーザが処理系の実装から独立にメモリ管理の処理を記述できるように, 必要なインタフェースを提供する. シミュレータが提供するインタフェースは, 3 節でのモデルに基づく.

以下, シミュレータが提供するインタフェースについて説明する.

4.1 ヒープ領域の管理

実行時システムは, プログラムの開始時と終了時に, メモリ管理システムがヒープ領域の初期化, または後処理を行うための関数を呼び出す. このために定義されるインタフェースは, 以下の通りである.

Initialize() プログラムの開始時に一度だけ実行される. ユーザは, これに対する処理として, ヒープ領域の初期化作業を記述できる.

Exit() プログラムを終了する直前に実行される. ユーザは, これに対する処理として, 必要な後処理を記述できる.

4.2 プログラムによるメモリ操作

実行時システムは, プログラムがメモリ空間に対する操作を行うたびに, そのことをメモリ管理システムに伝え, 必要な処理を要求する. このためのインタフェースは以下の通りである.

Allocate(size) 実行時システムがメモリ割り当てを要求する際に, このインタフェースを通してメモリ管理システムの関数が呼ばれる. メモリ管理システムは, この要求に対する処理としてヒープ領域内に適切な大きさの領域を確保し, その領域への参照を実行時システムに返さなければならない. このインタフェースの実装は, メモリ管理のアルゴリズムによらず必須である.

Use(field) 実行時システムが, ヒープ領域内に存在するオブジェクトのフィールドにアクセスする際に, そのことがメモリ管理システムに伝えられる.

Modify(field) オブジェクトのフィールドへの代入によって, 新たな参照の生成が発生した直後に, そのことがメモリ管理システムに伝えられる.

Delete(field) オブジェクトのフィールドへの代入, または root set に存在するオブジェクトの消滅によって, それまでに存在した参照が失われる直前に, そのことがメモリ管理システムに伝えられる.

4.3 オブジェクトの情報

実行時システムは, オブジェクトに関する情報をメモリ管理システムに提供するためのインタフェースを定義する. 利用できるインタフェースは, 以下の通りである.

Root_scan(function) Root set に存在するオブジェクトのフィールドを走査し, それがヒープ領域への参照ならば, 参照先のオブジェクトに対して function を実行する.

Is_pointer(field) メモリ空間内に存在するオブジェクトのフィールドに対し, そのフィールドに参照が格納されているかどうかの情報を返す.

5 実装

本研究で提案するシミュレータは, 既存の言語処理系である O'Caml の仮想機械からメモリ管理機能を分離し, ユーザがメモリ管理のアルゴリズムを記述するためのインタフェースを提供することによって実装される. このインタフェースを利用することにより, ユーザは仮想機械の実装を意識せずにメモリ管理の処理のみを実装することが可能になる.

シミュレータの実装に用いる O'Caml は関数型言語 ML の実装である. O'Caml は言語機能として高階関数, 多相型, モジュール機能などを提供しており, 様々な用途に広く利用されている.

以下では, はじめに本システムの実装の概要について述べる. 次に, 実装に用いる O'Caml の仮想機械について簡単に説明する. その後, O'Caml 仮想機械が前節のモデルによる 3 種類のメモリ操作を引き起こす箇所を明らかにし, それぞれの箇所に対して実際に挿入されるコードを示す.

5.1 実装の概要

提案する処理系の実装方法をおおまかに述べる. まず, O'Caml の仮想機械が前節で述べた 3 種類のメモリ操作を行う箇所を特定する. 次に, 特定されたそれぞれの箇所に対して, それぞれの操作に対応する処理を行う関数を呼び出すコードを挿入する. メモリ管理機能は, これらの呼び出しに対する処理を記述することで実装される.

アルゴリズムによっては, すべてのメモリ操作に対処する必要はないものもある. このような不必要な呼び出しに伴うオーバーヘッドはシミュレータの予測性の面で問題となるが, これは言語のマクロ機能を利用して除去することが可能である.

5.2 O'Caml 仮想機械

O'Caml の仮想機械は, 図 4 に示す構造からなる. 仮想機械は, root set として, レジスタ, スタック, 大域データ領域を持つ. これらの領域には 1 ワードの値, もしくは参照が格納される. 2 ワード以上の大きさを持つオブジェクトはヒープ領域に割り当てられ, 参照を通して扱われる.

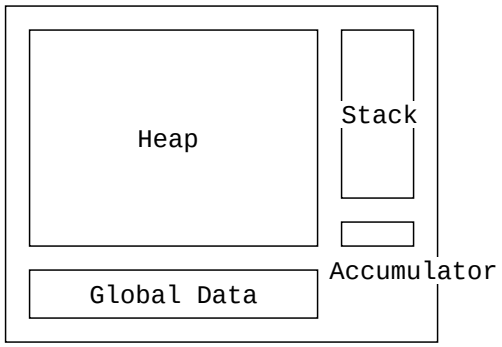

図 4: O'Caml 仮想機械

5.3 Creation

O'Caml 仮想機械では, ヒープ領域にオブジェクトを割り当てた結果として, 新たに割り当てられたオブジェクトへの参照が accu に格納される. このとき, 以前に accu に格納されていた値は上書きによって消去されるので, deletion が発生する可能性がある.

以上のことから, O'Caml 仮想機械がオブジェクトの割り当てを要求する箇所に対して, 以下のような実装がなされる.

```
Instruct(Allocate):
    if (Is_pointer(accu))
      Delete(&accu);
    Allocate(size);
    Read_next_instruction;
```

自動メモリ管理機能の直接的な目的は, ここに記述した Allocate を効率的に実装することである. 特に, マークスイープ方式, コピー方式に基づく基本的なアルゴリズムは, Allocate を処理するコードを実装することで簡単に実現できる.

5.4 Duplication

O'Caml 仮想機械による参照の複製は, 代入が行われるときに発生する. O'Caml 仮想機械は型を持たないため, 代入されるものが値か参照かをバイトコード命令によって識別することはできない. したがって, 代入が起こるすべての箇所に duplication に対応する呼び出しのコードを挿入する必要がある. また, creation の場合と同様に, 代入によって古い値が消去された結果として deletion が発生する可能性もある.

ここでは, スタックへの push を例に示す.

```
Instruct(Push):
    *++sp = accu;
    if (Is_pointer(*sp))
      Modify(sp);
    Read_Next_Instruction;
```

5.5 Deletion

O'Caml 仮想機械で deletion が発生するのは, 以下の場合である.

- Pop 命令の実行によって, スタックが縮小される場合.
- Creation, duplication による上書きの結果として, 古い参照が消去される場合.

ここでは, O'Caml 仮想機械が pop 命令を実行する場合を例として示す.

```
Instruct(Pop):
    if (Is_Pointer(*sp))
      Delete(sp);
    sp--;
    Read_Next_Instruction;
```

なお, メモリ管理システムが不要なオブジェクトを回収した結果として, そのオブジェクトが保持していた参照が失われ, 参照の消滅が連鎖的に発生することがある. しかし, これはメモリ管理システムの内部で発生する事象であるため, プログラムがメモリ領域に対して行う操作による deletion とは区別される.

5.6 Use

O'Caml 仮想機械では, ヒープ領域に存在するオブジェクトのフィールドへアクセスする際に, use が発生する. 以下に, そのような命令を実行する場合の例を示す.

```
Instruct(Getfield):
    if (Is_Pointer(accu))
      Delete(&accu);
    accu = Use(field);
    Read_Next_Instruction;
```

6 実行時システムの特化

メモリ管理のアルゴリズムには, メモリ空間内に複数のヒープを構成し, 割り当てられるオブジェクトの性質によってこれを使い分けるものがある. 例えば, O'Caml のメモリ管理システムは, オブジェクトの大きさによって割り当てるヒープを決定する.

このようなアルゴリズムを既存の処理系に実装する場合には, 静的に得られる情報を利用した効率的な実装が可能である. しかし, 本システムが提供する処理系では, このような, 特定のメモリ管理アルゴリズムに特化した実装による効率化はできない. このために生じるオーバーヘッドは, シミュレータの予測性の点で問題となる.

この問題に対しては, ユーザの記述したメモリ管理アルゴリズムに対応して, 実行時システムを特化する方法が考えられる. ユーザは, オブジェクトの性質によって異なる処理を行いたい場合には, 以下のような 2 段階の記述を用意する.

- 実行時システムからの呼び出しを受ける関数を実装し, ここにオブジェクトの性質による場合分けを記述する.
- 個々の場合に対応して適切な処理を行う関数を実装する.

以下に, 記述例を示す.

```
alloc(size) {
  if (size <= Max_small)
    alloc_small(size);
  else
    alloc_large(size);
}
alloc_small(size) { ... }
alloc_large(size) { ... }
```

実行時システムは, ユーザの記述したメモリ管理アルゴリズムを解析し, オブジェクトの性質が静的に判明する場合には, 対応する関数を直接呼び出すように, 自身の実装を特化する.

7 議論

本節では, 本研究で提案する処理系で記述できるアルゴリズムの範囲を明らかにする. また, 本研究において解決していない問題点と将来の課題についても述べる.

7.1 処理系の適用限界

本処理系では, 3 節で述べたモデルの範囲内で記述できるアルゴリズムを対象としている. 基本的なメモリ管理アルゴリズム (参照カウンタ方式, マークスイープ方式, コピー方式) に関しては, このモデルの上に記述することが可能である.

例えば, 参照カウンタ方式を実装するには, creation の際に参照カウンタの初期化を行い, duplication, deletion に対応する処理として参照カウンタを正しく変更すればよい. deletion に対する処理の結果として参照カウンタが 0 になった場合に, そのオブジェクトを回収することができる.

マークスイープ方式, コピー方式の場合の実装はさらに簡単であり, creation に対応する処理を記述するだけでアルゴリズムを実装できる. 世代別コピー方式 [8] の実装では, メモリ空間を young space と old space に分割し, old space から young space への参照を保存しておく必要があるが, これも duplication に対する処理として記述することができる.

一方, このモデルでは記述できないアルゴリズムとしては, コンパイル時の情報を積極的に利用するものがある. このようなアルゴリズムとして, tag-free garbage collection[5], region inference[11] などが挙げられる. これらのアルゴリズムでは, コンパイル時の型解析や生存区間解析を利用することで効率的なメモリ管理を実現するが, 本研究では実行時の情報のみを利用してメモリ管理を行うために, このようなコンパイル時の情報を利用するアルゴリズムを記述することはできない.

7.2 コンパイル時の情報の活用

5 節で述べたように, メモリ管理システムに処理される必要のない無駄な呼び出しはマクロ機能を利用して除去することが可能である. しかし, マクロ機能による除去では対応できない場合もあり, これが本システムの性能面で大きな問題となる.

例えば, 参照カウンタ方式では, 参照の代入が発生する際にオブジェクトの参照カウンタを変更する必要があり, これが無視できないオーバーヘッドとなることが知られている. したがって, 参照カウンタ方式を実装する際には, コンパイル時の情報から, オブジェクトに代入されるものが参照ではないと判断できる場合には, 参照カウンタを変更するコードを除去することによって性能劣化を小さくとどめることが一般的である.

しかし, 本処理系ではコンパイル時の情報を有効に利用することができないために, このような最適化を実現することは困難である.

このような, 適切な実装によって解消できるオーバーヘッドの存在は, シミュレータとしての予測性の面で問題となる. この問題を解決することは, 今後の重要な課題である.

7.3 ネイティヴコードに対する実装

本研究では, 実装として O'Caml 仮想機械を変更する方法をとった. この場合には, 仮想機械がメモリ操作を行う箇所に適切なコードを挿入することで, 3 節のモデルを正しく実装できる.

一方, ユーザプログラムをネイティヴコードで実行する場合には, コンパイラのコード生成フェーズを変更して, メモリ管理の処理を呼び出すコードをアセンブリコード中に挿入すればよい.

ただし, この場合には, メモリ管理システムが処理する必要のない無駄な呼び出しを除去することが難しくなる. ネイティヴコードで実行されるプログラムに関して, 無駄な呼び出しの除去を実現する方法としては, コンパイラのコード生成フェーズにおいて, メモリ管理システムの記述を利用したコード生成を行うことが考えられる.

7.4 分散システムへの拡張

本研究では, 単一プロセッサ上のシステムにおけるメモリ管理を対象として, アルゴリズムの評価を容易に行うためのシミュレータを提案した. 現在, メモリ管理のアルゴリズムについてより活発な研究が続けられている分野として, 分散システム上でのメモリ管理がある [10, 6, 7]. 分散システム上でのメモリ管理の実装は, プロセス間の同期などによる複雑さのために, より困難な作業となる. 分散システム上での効率的なメモリ管理は, 分散システムを構成する各プロセッサでのメモリ管理と協調するによって実現される. このことから, 本研究の主眼である単一プロセッサでのメモリ管理に対するシミュレータを拡張し, 分散システムに対するメモリ管理アルゴリズムの性能評価を容易に実現できるようにすることは, 非常に有用であると考えられる.

8 まとめ

本研究では, メモリ管理アルゴリズムの性能評価を容易にするという目的のために, メモリ管理に関わる実装のみを独立に変更可能な言語処理系を提案した. 提案された処理系は, プログラムが実行時に行うメモリ操作のモデルに基づいて実装され, この処理系の上に一般的なアルゴリズムを実装することが可能である. 本研究の今後の課題として重要なものは, 実行時の情報のみでは解消できないオーバーヘッドへの対応である.

参考文献

[1] Giuseppe Attardi, Tito Flagella and Pietro Iglio. A Customisable Memory Management Framework for C++. *Software-Practice and Experience*, 28(11): 1143-1183, 1998.

[2] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 2(12): 655-657, December 1960.

[3] Derek White and Alex Garthwaite. The GC Interface in the EVM. *Sun Microsystems Labolatories Technical Report Series*, TR-98-67, December 1998.

[4] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Computer Systems. *Communications of the ACM*, 12(11): 611-612, November 1969.

[5] Benjamin Goldberg. Tag-Free Garbage Collection for Strongly Typed Programming Languages. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 165-176, Toronto, Ontario, Canada, June 26-28, 1991.

[6] John Hughes. A Distributed Garbage Collection Algorithm. Jean-Pierre Jouannaud, editor, *ACM Conference on Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 256-272, Nancy, France, Software Practice and Experience 1985.

[7] Niels Christian Juul and Eric Jil. Comprehensive and Robust Garbage Collection in a Distributed System. Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 113-115, St.Malo, France, September 1992. Springer-Verlag.

[8] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6): 419-429, June 1983.

[9] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3(4): 184-195, April 1960.

[10] David Plainfossé and Marc Shapiro. A Survey of Distributed Garbage Collection Techniques. *Memory Management, International Workshop IWMM 95*, pages 211-249, Kinross, UK, September 27-29, 1995,

[11] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-Value lambda-Calculus using a Stack of Regions. *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188-201, Portland, Oregon, January 17-21, 1994.