

コンパイラ・インフラストラクチャを用いた 静的プログラムスライシングツール

A static program slicing tool using a compiler infrastructure

溝淵裕司[†]

Yuji MIZOBUCHI

中谷 俊晴[†]

Toshiharu NAKAYA

佐々 政孝[†]

Masataka SASSA

[†] 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

Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology

{mizobuc0, Toshiharu.Nakaya, sassa }@is.titech.ac.jp

プログラムをより効率良く理解し、解析する方法の一つにプログラムスライシング技術がある。本研究では静的スライスを求めるツールをコンパイラ・インフラストラクチャを用いて実装し、合わせてコンパイラ・インフラストラクチャのソフトウェア工学的適用性の評価を行なった。

1 はじめに

プログラムスライシング技術はプログラムの意味を解析する技術であり、Mark Weiser[5] によって考案された。最初はプログラムのデバッグを支援するために考えられていたが、現在では、ソフトウェアのテスト、デバッグ、改造、統合、リバースエンジニアリング、プログラム理解、メトリックス、コード最適化など、広範囲に応用されている有用な技術である。

プログラムスライスには、プログラムを静的に解析することにより得られる静的スライスと、動的に解析することにより得られる動的スライスとがある。一口に言えば、静的スライスとは、どんな入力にたいしてプログラムを実行しても、ある変数に関しては、もとのプログラムと同じ値を計算する実行可能なプログラムである。一方、動的スライス p とは、ある入力を与えてプログラムを実行した場合に、その同じ値を計算する実行可能な部分プログラムである。静的スライスは、プログラムの中からある機能を実現している部分だけを取り出すことができるため、ソフトウェアの改造、統合、リバースエンジニアリング、モジュール強度を調べるメトリックスの研究などに応用されている。

このように静的スライスはプログラムを解析する上で有用な方法である。本研究では、スライシング技術を用いた様々な応用に向けて、静的スライスを求める静的スライシングツールを実装した。

スライスを計算するとき必要な情報はコンパイラで必要となる情報と共通する部分が多い。したがって、スライスを求めるためにはコンパイラの解析を利用できると工数を減らすことができる。なかでも、COINS コンパイラ・インフラストラクチャ[6] は、各ユーザーがそれぞれの目的に合う機能部品を加えられる様に、組合せ可能なコンパイラ部品で構成されている。COINS の開発グループよりソフトウェア工学への適用性を試したいという希望もあったため、本

研究ではユーザーという視点から COINS を用いてスライシングツールを実現した。

2 静的スライス

この節では静的スライスの定義を説明する [10]。静的スライスを定義するためには、まず、どの文に関して (あるいは、どの文のどの変数に関して) 静的スライスを求めるのかを決める必要がある。これをスライシング基準という。

プログラム P のスライシング基準 (slicing criterion) $C(s, V)$ とは、次の二組をいう。

1. s はプログラム P 内の文
2. V はプログラム P 内の変数の部分集合

定義 (静的スライス)

プログラム P のスライシング基準 $C(s, V)$ に関する静的スライス (static slice) とは、以下の条件を満たす実行可能なプログラム P' である。

1. P' は、 P から 0 個以上の文を削除することにより得られたもの。
2. 入力 x に対してプログラム P が停止するならば、プログラム P' も停止する。
3. プログラム P に入力 x を与えた時の実行系列を EX 、プログラム P' に入力 x を与えた時の実行系列を EX' とする。すべての $v \in V$ に対して、実行系列 EX' において文 s を実行する直前における変数 v の値が、実行系列 EX において文 s を実行する直前における変数 v の値に等しい。

3 静的スライスを求める方法のあらまし

静的スライスを求める方法にはいくつかあるが、一般的に用いられているのは Ottenstein [2] が提案した

プログラム依存グラフ(Program Dependence Graph, PDG) という有向グラフを用いる方法である。PDG とは、文の間の依存関係をグラフ化したものである。文の間の依存関係には制御依存関係とデータ依存関係がある。本研究でも PDG を用いてスライスを求める。PDG を用いてスライスを求める方法の概要は以下のとおりである。

1. プログラムの各文に対して制御依存解析を行う。
2. プログラムの各文に対してデータ依存解析を行う。
3. PDG の構築
4. スライスの計算

3.1 制御依存関係

文 s から文 t へ制御依存関係 $CD(s, t)$ があるとは、構造的プログラミング言語ではあらず、文 s が分岐文 (の条件の部分) であり、文 t がその分岐文内に直接含まれている場合、あるいは、文 s がループ文 (の条件の部分) であり、文 t がそのループ文内に直接含まれている場合をいう。一般には、制御フローグラフでの逆支配辺境により求められる (4.2 節)。

3.2 データ依存関係

文 s から文 t に対してデータ依存関係 $DD(s, t)$ があるとは、ある変数 w が存在して、文 s において定義された変数 w が、変数 w を使用している文 t に到達する場合をいう。変数 $w \in Use(t)$ に関してデータ依存関係があることを明示する場合には、 $DD_w(s, t)$ と表すこととする。ここでいう到達するとは、フローグラフ上で文 s から t に至る実行系列が存在し、その実行系列上では変数 w が再定義されないことをいう。

3.3 PDG の構築

3.1 と 3.2 で説明した依存関係をグラフ化し、マージさせたものが PDG となる。PDG のノードは文であり、スライス抽出における粒度となる。辺は文の間の依存関係を表しており、制御依存関係を表す制御依存辺とデータ依存関係を表すデータ依存辺からなる。

3.4 スライスの計算

与えられたスライシング基準から制御依存辺およびデータ依存辺を逆方向にたどって推移的に到達可能なノードの集合を求める。そのノードの集合に対応した文がスライスとなる。

PDG の例として図 1 を示す。図では制御依存辺が点線で、データ依存辺が実線で表されている。スライシング基準 (10) のスライスを求めてみると、(1)、(3)、(4)、(5)、(9)、(10) となる。

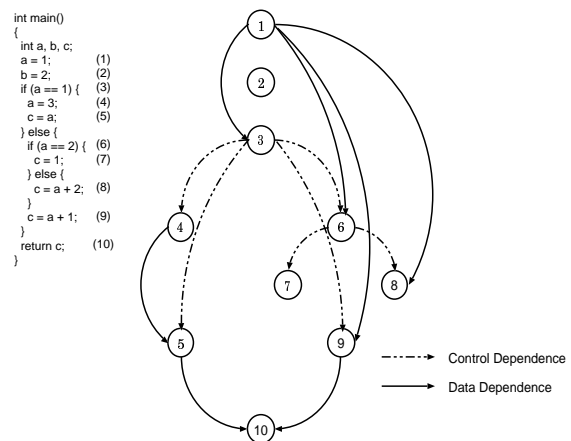


図 1: C のサンプルプログラムに対する PDG

4 実装

入力言語は C 言語とし、実装は COINS インフラストラクチャ [6] による C コンパイラ (以下 COINS) を利用して Java で記述した。ただし、本研究でのスライシングツールは COINS からのフロー情報を基に依存解析をするため、入力プログラムは COINS でフロー解析できるプログラムに限る。

4.1 基本的な流れ

本研究では COINS でソースプログラムを一旦高水準中間表現 (HIR) まで解析する。同時に COINS から HIR 上での制御フロー情報とデータフロー情報も提供してもらう。

本スライシングツールでは COINS から提供してもらうそれらの情報を用いて、HIR 上で制御フロー情報から制御依存グラフ、データフロー情報からデータ依存グラフを作り、それらをマージすることによって PDG を構築する。PDG を構築した後、ユーザーにスライシング基準を入力してもらうとスライスが求まる。本スライシングシステムの解析の流れは図 2 のようになる。

図 3 にソースプログラム (a) とそれから作られる HIR の一部分のダンプ (b) を示す。

ソースプログラムの各文の右側に振った括弧の中の番号と HIR の右側に振った括弧の中の番号は対応関係にある。図の HIR の各行にある番号は HIR のプログラムポイントである。

4.2 制御依存解析

HIR 上で制御依存解析する方法を示す。制御依存解析をするには、逆支配辺境を求めれば良く、支配辺境を求めるアルゴリズムを使うことになる [8]。支配辺境とは、制御フローグラフ上でノード X からグラフを辿っていき、初めて X の支配から外れたノー

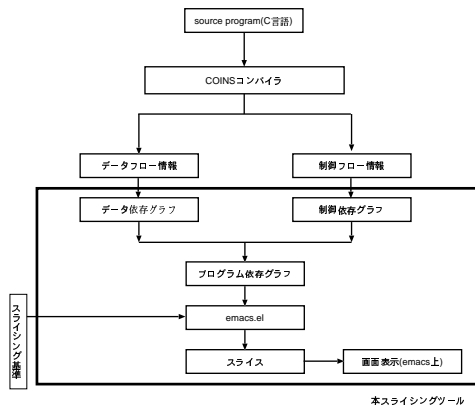


図 2: 本スライシングツールの流れ

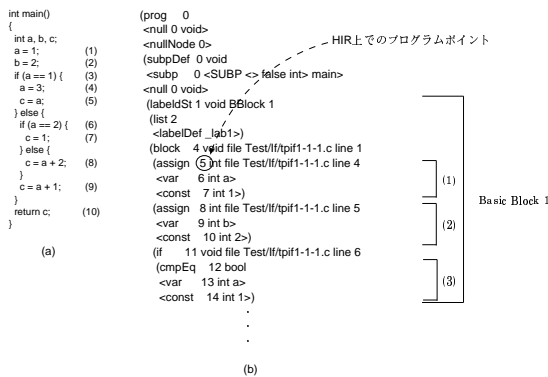


図 3: HIR の例

ドの集合のことである。

COINS の制御フロー情報としては、基本ブロック B に対して、以下のような情報が得られる。

- $Pred(B), Succ(B)$: 基本ブロック B の先行ブロックの番号、後続ブロックの番号
- $Dominator(B), ImmediateDominator(B), DominatedChildren(B)$: 基本ブロック B を支配するブロックの番号、直接支配するブロックの番号、支配木で基本ブロック B の子となるブロックの番号
- $PostDominator(B), PostImmediateDominator(B), PostDominatedChildren(B)$: 逆転フローグラフにおける基本ブロック B を支配するブロックの番号、直接支配するブロック B の番号、逆支配木でその基本ブロック B の子となるブロックの番号

COINS の制御フロー情報を用い、支配境界を求め、アルゴリズムを用いて逆支配境界を求め、制御依存グラフを作る。求められた逆支配境界は基本ブロッ

ク単位に求められるため、制御依存グラフを作る際には、文単位のグラフに変換する。

4.3 データ依存解析

HIR 上でデータ依存関係を求める方法を示す。データ依存解析をするにあたって、COINS から得られるデータフロー情報は、基本ブロック B に対して以下の情報である。

- $USE(B)$: 基本ブロック B で使用される変数に対応する HIR のプログラムポイントの集合
- $DEF(B)$: 基本ブロック B で定義される変数に対応する HIR のプログラムポイントの集合
- $REACH(B)$: 基本ブロック B に到達する定義された変数に対応する HIR のプログラムポイントの集合

これらデータフロー情報より、データ依存グラフを求める。制御依存グラフと同様、これらから求められるデータ依存グラフは基本ブロック単位のものであるため、文単位のグラフに変える。

5 実験と評価

5.1 画面出力

この節では、本スライシングツールを emacs の画面で使用したときの様子を説明する。図4では、return c のスライスを求めた結果が emacs 上のソースプログラムにハイライトされている。

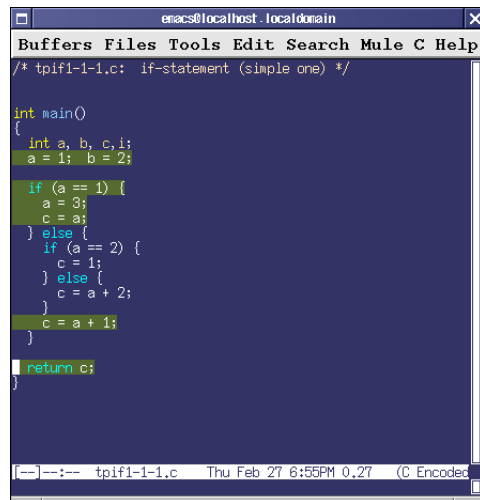


図 4: 画面出力

5.2 評価

5.2.1 本スライシングツールの評価

COINS でフロー解析のできるプログラムについてはスライスを求めることができた。本システムは Java で約 700 行で実装できた。コンパイラ・インフラストラクチャを利用した効果があると言える。

5.2.2 COINS の評価

戸板らのスライサー [9] も COINS のフロー解析を用いた依存解析を行っている。戸板らのスライサーは配列にも対応しており、C 言語で約 2700 行で実装できている。

他のスライサーとして文献 [4] を基にして作られた Unravel がある。Unravel はプログラムの解析器とリンクとスライサーからなる。Unravel のスライサー部分は約 4500 行で実装されていて、本システムや戸板らのスライサーの方が少ない労力で作られたことが分かる。

現状の COINS の実装では以下のような点が課題である。

- 宣言文の行番号が狂う
変数の宣言と代入を同じ命令文で記述したとき、その命令文の行番号を特定することができない。
- カラム位置がとれない
同じ行に文を複数書いたとき、それぞれの文のソースコードにおける位置の特定ができない。
- switch 文を解析するために出来る無駄なノード

5.3 課題

本研究ではスライシングツールを実装したが、スライスの今後の応用に向けて課題となる点を論ずる。

1. より多くの C 言語の機能に対応する

配列、ポインタ、大域変数、関数呼出し、副作用、ライブラリ関数の扱いなど対応しきれていない点が多々ある。これらの中には、別名解析や手続き間解析など、COINS で保守的に解析されるために十分なフロー情報を得られないものがある。

2. スライスの精度やそれに伴う負荷について

静的スライスは、起こりうる全ての入力を考慮して依存関係解析をするため、不必要な情報を含むことがあり、スライスの精度が低いという問題点がある。

一方、動的スライスは、特定の入力データに関する実行系列 (実際に実行された文の並び) を扱うため、抽出されるスライスのサイズを減らすことができる。しかし、全ての実行系列を保

持し、そこから依存関係を抽出するために、多大な時間や空間コストを必要とする。動的スライスをコンパイラ・インフラストラクチャを利用して求められるかどうかは検討する必要がある。

6 関連研究

• COINS を用いたスライシングツール

COINS を用いたスライシングツールに前述の戸板らの研究 [9] がある。COINS の中間表現 HIR を用いて XML に変換することでスライシングをプログラミング言語に依存しないグラフ上での抽象的な操作とすることができている。

• スライスの応用

スライシング技術を応用したデバッガに VALSOFT [3] がある。VALSOFT はスライスの精度向上のために Robschink や Snelting が提案した condition path [7] というものを加えている。その他、Code Surfer [1] などのスライシングシステムがある。

参考文献

- [1] Tim Teitelbaum et al. Code surfer user guide and reference. Technical report, Gramma Tech Product Documentation, 2001.
- [2] Ottenstein, K. and Ottenstein, L. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 177–184. SIGPLAN, 1984.
- [3] Jens Krinke and Gregor Snelting. Validation of measurement software as an application of slicing and constraint solving. *Information and Software Technology*, Vol. 40, pp. 661–675, 12 1998.
- [4] J. Lyle and D. Wallace. Using the unravel program slicing tool to evaluate high integrity software, 1997.
- [5] Weiser M. Program slices when debugging. *Communications of the ACM*, Vol.25, pp. 446–452, 1982.
- [6] COINS project. COINS - project homepage. <http://www.coins-project.org/>.
- [7] Torsten Robschink and Gregor Snelting. Efficient path conditions in dependence graphs. In *Proceedings of the 24th International Conference on Software Engineering*, pp. 478–488. ACM Press, 2002.
- [8] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.
- [9] 戸板晃一, 山本晋一郎, 阿草清滋. プログラムスライシングツールのための共通表現. 電子情報通信学会 (SS 研究会), 3 2003.
- [10] 下村隆夫. プログラムスライシング技術と応用. 共立出版株式会社, 1995.