コンパイラ実験課題

4 班コンパイラ係 05-191022 平田 賢吾

2020 年 1 月 29 日

次の課題に取り組んだ.

第 8 回コンパイラ係向け課題

グラフ彩色によるレジスタ割り当てを実装せよ.

- どのような塗り分け方法をとったか, 選択の理由とともに説明せよ.
- 実験などで効果を確認せよ.

目次

1　build 方法

コンパイラの実装は Haskell で行なった. 開発環境は Stack を用いた. build の際にはまず, 本気で Haskell したい人向けの Stack チュートリアルなどを参考にして Stack を install していただきたい.

```
stack build
```

で build ができる.

```
stack exec min-caml ~~.ml
```

でコンパイルを行う. アセンブリはソースコードと同じディレクトリに生成される. アセンブリの isa に関しては, https://github.com/CPUEX2019-GROUP4/compiler/wiki を参照していただきたい. シミュレータは https://github.com/CPUEX2019-GROUP4/simulator にある.

compile.sh にコンパイルする shell スクリプトを置いてある. test ディレクトリ内にあるプログラムに関しては, 例えば test/fib.ml をコンパイルしたいときは

```
./compile.sh fib
```

とすると, test/fib.s が生成される. build も同時にするようにしてある.

2　実装の概要

min-caml で元々実装されていたレジスタ割付アルゴリズムを次のように分割し, それぞれ対応するプログラムファイルに実装した.

1. ブロック分割 – src/Back/ToBlock.hs
2. 生存解析 – src/Back/Reg/LivenessGlobal.hs
3. Stack 回避変数の収集 – src/Back/Reg/Stack.hs
4. Save をプログラムに挿入する – src/Back/Reg/InsertSave.hs
5. 小ブロック分割 – SmallBlock.hs
6. Restore をプログラムに挿入する – src/Back/Reg/InsertRestore.hs
7. 生存解析 – src/Back/Reg/Liveness.hs
8. 干渉グラフの生成 – src/Back/Reg/Interference.hs
9. 干渉グラフの彩色 – src/Back/Reg/Coloring.hs
10. 彩色結果で元のプログラムを置換 – src/Back/Reg/BlockRegAlloc.hs

残念ながらバグが取れていないので min-caml コードのコンパイルができない.....

3　アルゴリズム

3.1　目的および概要

min-caml のコンパイラでは, レジスタ割り付けのとき, 割り付けすると同時に変数のスタックへの退避命令 (Save) と変数のスタックからの復帰命令 (Restore) をプログラムに必要次第挿入する. しかし, この方法では

グラフ彩色としてレジスタ割り当てを行う上で処理が煩雑になると考えた．また, 関数呼び出しの前後で生存する変数に対して先に生存解析を行っておくことで, Save, Restore の挿入するべき位置をより効率的にすることができる．そのため, 上のようにアルゴリズムを分割した．ここでは実装できなかったが, レジスタ割り当て時に Spill が起きたとき, Save されることが決まっている変数を優先的にレジスタから追い出すこともできる．

3.2 ブロック分割

まだ実装できていないが, 基本ブロックに分割しておくとデータフロー解析 (定数畳み込み, 共通部分式削除, 不要分削除) などがやりやすくなるため, 基本ブロックに分割した．分割の際, if 文に対しては元のデータ構造 (Back.Asm.T) では

```
let x = if y
  then
     let a = ... in
     k
  else
     let b = ... in
     l
```

という形をしていたが,

```
if y
  then
     let a = ...
     let x = k
  else
     let b = ...
     let x = l
```

となるようにした．

3.3 生存解析

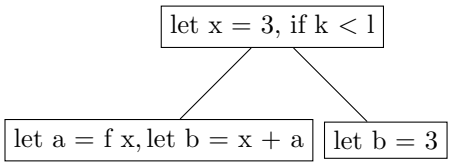
このコンパイラにはループ表現を導入していないので, 基本ブロックのグラフは, acyclic になっている．そのため, 各命令 p に対し, p 以降で使用されうる変数で, p 以降に定義がないものが, p の直前で (スタックまたはレジスタに) 生存している変数である．これを $\mathrm{Live}\,[p]$ と書くことにすると, (p は プログラムの p 番目の命令)

$$\mathrm{Live}\,[p] = \bigcup_{q \in child[p]} \mathrm{Live}\,[q]$$

とかける．acyclic なので, トポロジカルソートしておけば, これは後ろから順番にもとまる．

3.4 Stack の計算および Save の計算

関数呼び出しの前後で生存する変数は必ずスタックにつむことになる．これを計算したい．このとき,



なるように基本ブロックののグラフがあるなら, 左のブロックで関数 f の呼び出し前後で x が生存しているので, x の save 命令をどこかに挿入しないといけない. このとき, 単純な方法として x の定義直後に x を save する命令を入れる方法があるが, それでは右のブロックに if 文で分岐したときに 1 命令の無駄が発生する. そのため, if 文で分岐が起こるとき, 上のブロックで x を save するべき条件は, それよりも下のブロックのいずれかで x が save されていて, さらに下のブロックの両方で x が生存していることとなる. 式で書くと下のようになる.

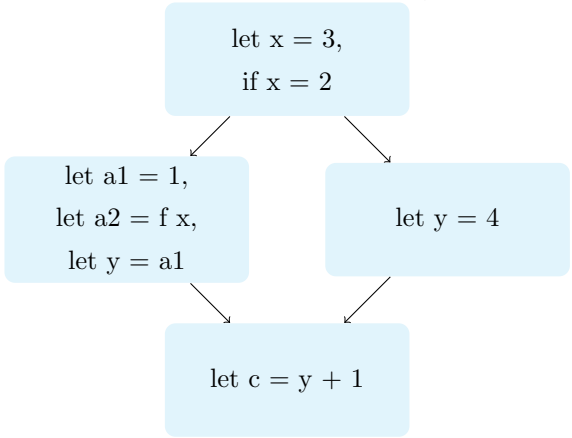
$$\mathrm{Store}\,[u] = \bigcap_{v \in \mathrm{child}\,[v]} \mathrm{Live}\,[v_0] \cap \bigcup_{v \in \mathrm{child}} \mathrm{Store}\,[v]$$

$$\mathrm{Save}\,[u] = \mathrm{Store}\,[u] - \bigcap_{v \in \mathrm{parent}\,[u]} \mathrm{Store}\,[v]$$

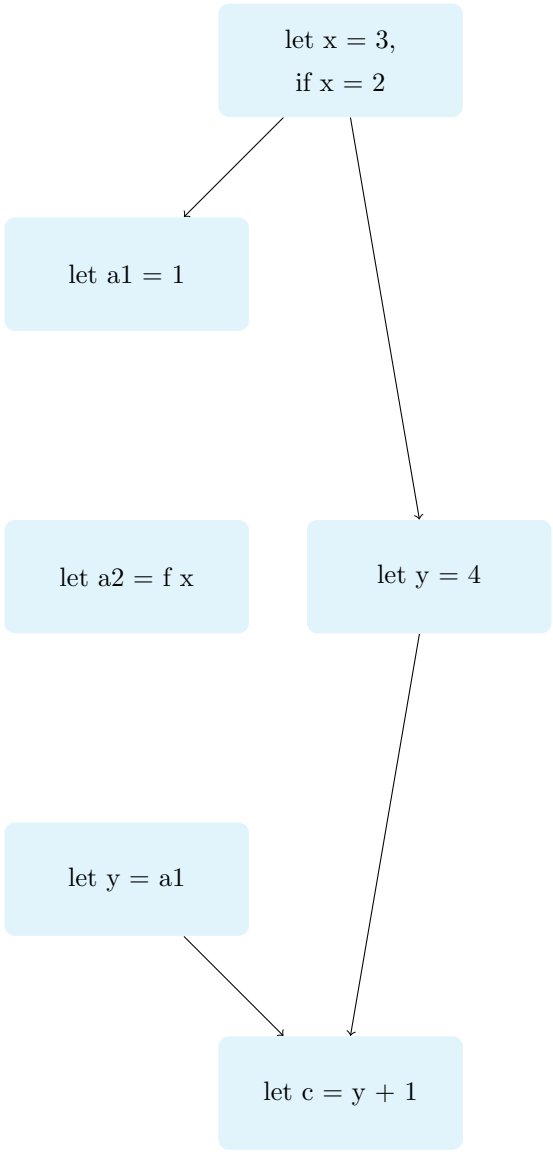
(u, v は基本ブロック, Live[v_0] は, 基本ブロック v_0 の先頭で生きている変数) Save[u] が, 基本ブロック内で save されるべき変数の集合になる. Store は基本ブロックグラフの下から, Save はどこからでも計算できる.

3.5 小ブロック分割および干渉グラフの作成方針

干渉グラフを作成するときは, 同時に生存する変数の集合を集めるが, 関数呼び出し前後では, 同じレジスタに格納されている必要はない. そのため, 基本ブロックで扱っていた処理を関数呼び出しが末尾にしか存在しないようにブロックを切り分けることで, 同じ値が同じレジスタに格納されるべき単位に分ける.



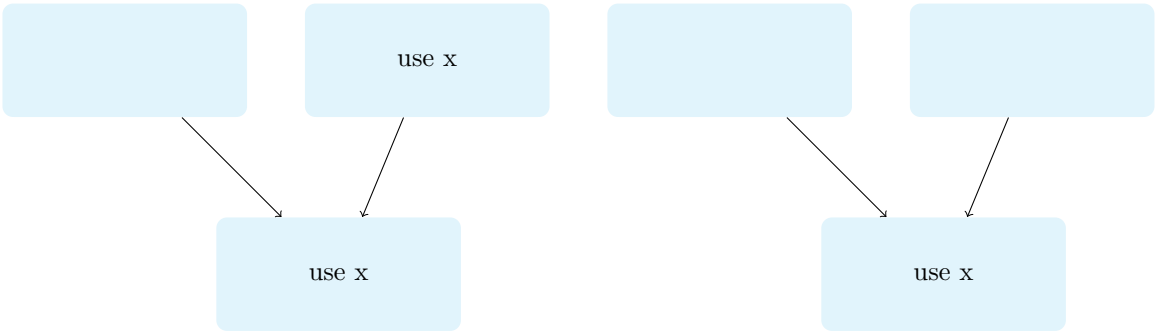
このようなプログラムは, 以下のように切り分ける. (以下小ブロックと呼ぶ.)



このようにしてできたグラフの弱連結成分内では, 同じレジスタに同じ値が格納されていると良い. よって, この弱各連結成分ごとに干渉グラフを生成する.

3.6 Restore する位置の戦略

次の 2 つの図 (右と左) のように小ブロックのグラフがあるとする. use x と書かれている小ブロックでは x が計算に用いられている. これらのすべての小ブロックの中に, x の定義は現れず, restore 文を挿入することで x をレジスタに復帰させないといけない時を考える.



このとき, 右側の図では, 明らかに一番下のブロックで restore をするのが良い. その一方で, 左側では右上および下の小ブロックで restore しても良いのだが, load 命令は我が班の cpu では 3 クロックかかり, 遅くなっているため, 出来れば 2 回以上 load 命令が一つの path に現れることは避けたい. そのため, 右上および左上の小ブロックに restore 命令を挿入したい.

グラフの問題に直すと, 次のような問題になる.

problem

有向グラフ D と, 頂点の部分集合 T(= 変数 x が使用される小ブロック) に対して次のような集合 S(= restore する頂点) を求めたい.

任意の T の頂点 $t \in T$ と入次数 0 の頂点 u に対し,
u から t への path 上には S の頂点が一つだけある.
S はそのような集合の中で, S から到達可能な頂点の集合が最小.

これは次のアルゴリズムで求められる.

最初に $S = T$ とする. S の頂点 $u, v(u \neq v)$ で u から v へ向かう path が存在するならば, S から v を除き, S に v の親をすべて追加する.

各変数に対してこのアルゴリズムで restore を挿入すべき頂点集合を求めることができる.

3.7 生存解析および干渉グラフの生成

上の save と restore のアルゴリズムによって, 関数呼び出し前後で必要な store と restore はすべて挿入されているので, もういちど上と同じ生存解析を小ブロックのグラフの弱連結成分に対して行うと, 得られる結果は, 各プログラムポイントにおいて (stack ではなく) レジスタに生存している変数の集合になる. これによって各弱連結成分に対し, 干渉グラフを作ればいい. この時点で関数引数の情報, 返り値の情報, 関数呼び出しの際に与える変数の情報, レジスタ移動命令で同じになる集めている. 実装できていないが, これらを使って移動命令を減らすように彩色の順番を変えることでできる.

3.8 レジスタ彩色

%r0 と, 返り値になる %r1 %f0 のみ元々彩色されていると考え, (ここに格納されないといけない変数はこの処理より前にレジスタと同じ名前の変数名に変換されている) そのほかの変数に彩色を施す. レジスタの個数を最初に与え, その数より次数が少し小さい頂点をリストにつむことを繰り返す. もしすべての頂点の次数がレジスタの個数近くまであるのであれば, その中の一つを選んでリストにつむ. 最終的にできたリストの最

後に追加した頂点から順に色を greedy に塗っていく. 彩色方法として, 頂点の次数の大きいものから順に, 既に色が塗られた頂点を除きながら彩色していくと, それなりに良いアルゴリズムになっている. (参考 Graph Theory (Reinhard Diestel)) 計算量的に軽くしたかったので, このアルゴリズムは採用しなかったが, 今のアルゴリズムでも spill の数は抑えられているはずである.

実装できていないが, 完成形としては, 関数引数や返り値などになる変数に対して, 先にその変数がもし塗られると良い色に塗れるなら塗っておき, その後, レジスタの個数より 2 か 3 程度次数が少ない頂点を除き, それ以上の次数の頂点しかなくなれば次数の小さい順に除くことを繰り返すアルゴリズムにしたい. この方法にしたいのは, この方法がある程度色の数を小さくできるアルゴリズムであることと, さらにプログラムの都合として移動命令が少なくできることの両方を叶えられると考えたからである.

Spill した場合の処理は, 生存期間/(干渉グラフの次数)^2 の大きいものから生存区間を分割するようにしたかったが, それ以前に動いていないので実装していない. 残念ながらバグが取れていないため, 効果を確認できない.

3.9 彩色結果による変数の置換

小ブロックに分けられたプログラムを, 元のブロックに対応するように concat する. また, 関数引数などはレジスタ移動命令を挟むことで正しい位置に移動させる.

4 まとめ

もっと簡単なアルゴリズムでとにかく実装を終わらせることを目標にやるべきだった. 1 月のテスト期間の少し前から取り組んでいたが, もっと早く取り組み始めればよかった. いま現在で動く最新版のコンパイラを提出してある. ここではレジスタ割り付け後に基本ブロック分割をしたプログラムをアセンブリに変換している. 面談までに動くようにバグを頑張って取りたい.