コンパイラ係面談

目次

コンパイラの概要

- フロントエンドはmincamlをそのまま利用
  + 最適化のモジュールを追加，必要に応じて仕様を一部変更

- バックエンドはSSA形式のものをスクラッチ
  - llvm mirを参考にSSAの命令列 + コントロールフローグラフ(CFG)による表現を採用
    - 基本ブロックとの対応をアセンブリにコメント
    - デバッグの際にアセンブリの一部分が，元のプログラムのどこへ対応しているかわかりやすい
    - 局所的な変更をSSA命令列，大局的な構造の変更をCFGでそれぞれ独立に行える

  - SSAで使える命令は実際のアーキテクチャにある命令　+ Φ関数 + Save/Reload
    - 2オペランド命令，1オペランド命令，メモリアクセス命令のようなフォーマットを区別する外側のヴァリアントと，それぞれのフォーマット内で利用可能なオペコードによる内側のヴァリアントに分けた
    - 基本的に場合分けは外側のヴァリアントについてのみ行えば十分なので，似たようなコードを何回も書かなくてよくなった

レジスタ割り当て(理論編)

- SSA形式における干渉グラフは弦グラフ (完全グラフの一種)になる
  - (制約がなければ)多項式時間での最適な彩色が可能
  - 具体的にはdominator treeのdfsをしながら貪欲に割り当てすればよい

- 現実的には関数呼び出し規約によって , 一部のノードに対して色の制約がある
  - 各色1ノードまでの制約なら多項式時間で最適解を出せる
  - 上記の条件を満たすためには関数呼び出し時点で生存区間分割 [1]
    → 独立な干渉グラフごとに彩色 + 間の変数の受け渡しをΦ関数で
  - 実装が大変なので , 実際には関数引数の定義に割当してほしいレジスタをヒントとして与えるだけに

- callee reg導入
  - レイトレは末尾再帰 (ループ)が多いので効果的だと予想
  - 元のレジスタ割当から約 10億命令の減少
  - 復帰と退避のオーバーヘッドが増えるので多ければ多いほど良いというものではなかった

[1] Sebastian Hack. Register Allocation for Programs in SSA Form.

レジスタ割り当て(実験編)

- 引数の彩色ヒント
  - while式なしで256 * 256レイトレを実行
  - インライン数0で142.3億→136.4億と命令数約4.1%減
  - インライン数600では73.8億→72億と命令数約2.4%減
  - インライン数が大きいほうが関数呼び出しが減るはずなので減少幅も減る

- callee regの数 (レイトレ256, whileなし, インライン数100)

合計32reg	4	6	8	10	12
命令数[億]	78.0	77.3	74.1	75.7	75.4
合計64reg	6	8	12	16	
命令数[億]	77.3	74.1	75.7	75.7	

この場合は8個が最適のよう. 少ないよりは多いほうがよい .

実装した機能

- フロントエンド
  - 共通部分式削除CSE (KNormal.t)
    - 基本的に即値が共通化されるだけ
  - 組型引数展開 (KNormal.t)
    - インライン展開と干渉するので使われていない
  - +,-演算子のオーバーロード

- バックエンド
  - グローバル値番号付けGVN
  - 分岐で共通の引数設定を分岐前に移動
    - 引数設定が全部共通化されると分岐で直接関数ラベルに飛べる
    - データフロー解析による最適化としてコンパイラ実験の第 9回課題に出したかった. レジスタ割当の実装に手こずり間に合わず .
  - while式導入
    - ループ不変式を外に出す ：逆に命令数が増えてしまったので見直しが必要
  - ループのブロック出力順変更 ：ループのジャンプが1つ減る
  - mvと変数の定義を合体させるpeephole最適化

while式

- 再帰関数はインライン展開されると逆に命令数が増える傾向
  - 分岐などで結局ジャンプが多い
  - 同じブロックが複数できるのでむしろ展開しないでまとめておきたい
  - インライン展開されたところまでで終わらないと元の関数に結局飛ぶことに
- しかし, 関数自体は呼び出し規約をなくすべくインライン展開したい
  - そこでwhile式によりループ構造を保ちつつレジスタ割当の自由度を上げる
- ひとまず条件を常にtrueにして, returnとcontinueにより実行の流れを制御
  - 配列のインデックスをループ変数として持っていることが多いので , オフセットを加味したアドレス自体をループ変数にするなど , ループ変数の最適化も今後やりたい
  - continueの無条件分岐を適切な条件分岐にしてジャンプを減らせるようにもしたい
- インライン数を上げたときに命令数削減の効果が顕著
  - しかしまだバグがあって出力結果が安定しない
  - 目視ではそれっぽいレイトレに見える
  - インライン数によってdiffが異なるのでレギュレーション違反

ループのブロック出力順変更

- 左の素朴なコンパイルではループ内で returnブロックへの条件分岐と , エントリへの無条件分岐がある
- 右のアセンブリでは fall-throughを利用してループ内は条件分岐 1つだけに
- 最初にジャンプが一つ増えているが , ループが複数回回れば変換後の方がジャンプが少ない

```
label0:
        (entry)
label1:
        (pred)
        bcond x1, x2, label2
        (continue)
        j label0 # 再帰
label2:
        (return)
```

```
        j label0
label3:
        (continue)
label0:
        (entry)
label1:
        (pred)
        ~bcond x1, x2, label3
label2:
        (return)
```

(name) : アセンブリ列
bcond : beq, bleなど条件分岐命令
~bcond : bcondの比較を反転させたもの

peephole最適化 - mvと変数の定義の合体 -

- 対象はループの continueブロック

- ループ変数の更新を mvでしているが, 更新する値の定義が同じブロック内にあることが多い
  - 可能ならmvしなくていいように最初からループ変数を入れているレジスタに対して値を定義したい

- 一つの命令に変換できる条件を考える
  - 変換できるようにスケジューリングもするので , 依存関係のグラフを構築
  - 変換した命令は , 変換前の二つの命令を縮退させたノードの依存関係を持つ
    - これは中間レジスタの依存関係を余計に含んでいる
  - このときにサイクルができると依存関係が解決できない → 縮退をやめる
  - 変換後のグラフでトポロジカルソートし , 根から葉の順で命令を実行すれば OK

- 例外
  - 関数呼び出しをまたいでスケジューリングはできない
    最後の関数呼び出し以降でやる . ループ変数の更新は最後にやるのでこれで十分なはず .
  - I / O 命令は順番を保つ必要があるので , 依存関係が生じる(未対応)

命令数の変遷 (256 * 256 レイトレ)

日付	最適化	インライン数	命令数[億]
12/08	グローバル配列導入でクロージャ削除		172
	共通部分式削除	10	98
1/2	SSAレジスタ割当,callee reg導入	10	90
1/6	空ジャンプ省略, 関数ラベルへの条件分岐	10	88
1/12	xor,fabs命令追加, レジスタ数32 → 128	50	78
1/23	再帰関数をインライン展開しない	600	72.5
1/23	while式導入(まだ出力があやしい)	88	66
2/4		600	51.5

今後の課題

- 出力結果のずれの原因究明 & デバッグ
  - while式を使うとインライン数ごとに結果が変わってしまう
  - peephole最適化をしたら256 * 256レイトレが51 → 22億命令に
  - 最大でも全てのループについて , ループ引数の数 × ループ回数を足した和しか命令数が減らないはずなので減少量がおかしそう . 一部の関数を計算を省略している可能性あり
  - 見た目からは違いがわからない

- まだ実装できていない最適化を実装
  - 排他な条件分岐を検出 , 不要な比較を減らす
  - スケジューリング

- すでに実装済みの最適化をデバッグ , 新しい仕様に対応させる
  - ループ不変式
  - ループの配置変換
  - 空ジャンプ省略など