5.7　簡単な最適化

コンパイラは生成されるターゲットプログラムの効率を改善するために**最適化** (*optimization*)[1]と呼ばれるプログラム変換を行う．言語 $\mathcal{C}$ の上で簡単な最適化をやってみよう．

5.7.1　無駄な束縛の除去

無駄な束縛の除去 (*elimination of redundant bindings*) は，その後使われることがなく，除去してもプログラムの意味を変えないとわかっている束縛を除去する変換である．例えば，プログラム let x = 3 in 4 は，x を 3 に束縛して 4 を返すが，束縛された x はその後一切使われないので，この束縛は**無駄** (*redundant*) である．これを束縛を行わない（多くの場合より効率のよい）プログラム 4 に変換するのが，無駄な束縛の除去である．

この変換は（というか，一般に多くの最適化は）何度か繰り返すことでより効率のよいプログラムを得ることが可能となる．例えば，プログラム let x = 3 in let y = x + 2 in 4 において，束縛変数 x は y の束縛先の計算を行う式 x+2 で使われているので無駄ではないが，束縛変数 y はその後使われていないので無駄である．そこで，無駄な y の束縛を除去すると，このプログラムは let x = 3 in 4 になるが，このプログラムにおいては x の束縛が無駄であるから，再度無駄な束縛を除去することにより，4 を得ることができる．

無駄な束縛の除去は図 5.1 に示す変換によって定義することができる．ここで $\mathbf{FV}(e)$ は e 中に現れる自由変数の集合である．この変換のキモは let $x = e_1$ in e_2 の e_2 中で束縛されている変数 x や let rec f = fun $x \to e_1$ in e_2 の e_2 中で束縛されている変数 f が無駄かどうかを判定する部分である．前者については x が e_2 に自由変数として現れておらず，かつ e_1 が関数適用の形をしていなければ，x の束縛を無駄であるとして除去する．前者の自由変数に関する条件は，x がその後使われないための十分条件になっている．後者の条件は e_1 が無限ループする式だった場合にプログラムの意味を変えないための条件である．例えば，関数定義 $d_1 :=$ let rec f x = f x を含むプログラム $(\{d_1\}, \text{let x = f 3 in 4})$ を考えよう．このプログラムは必ず無限ループする．もし後者の条件がなければ，x の束縛が（x が式 4 に自由に現れていないために）無駄だとして除去されてしまい，$(\{d_1\}, 4)$ となって，無限ループしないプログラムになってしまう．最適化はあくまでプログラムの効率を上げるために行う変換なので，プログラムの意味を変えるのはマズい．これを防ぐために，x の束縛先が，無限ループに陥る可能性のある式である関数適用である場合は，束縛を除去しないようにしている．[2][*AI:* e_1 が *if* の場合があるので，このままではちょっとまずいと思います．] ←

Exercise 5.7.1 [⋆⋆]　図 5.1 の無駄な束縛の除去は，無駄な関数定義を除去することはできない．例えば，$d_1 :=$ let rec $fx = gx$ で $d_2 :=$ let rec $gx = fx$ であるとき，プログラム

[1]「最適化」という言葉は情報科学の分野では様々な意味を持つので注意が必要である．実数上の関数を与えられた制約の下で数値的な手法を用いて最小化または最大化する手法を研究する分野を**数値最適化** (*numerical optimization*)，離散値上の関数を与えられた制約の下で最小化または最大化する手法を研究する**組み合わせ最適化** (*combinatorial optimization*) 等があるが，これらをコンパイラで行われる最適化と混同しないこと．(もちろん，組み合わせ最適化や数値最適化を用いてコンパイラでの最適化を行うことはありうる．)

[2] $\mathcal{C}$ では，ある式の評価が無限ループに陥るためには，関数適用を行わなければならない．(多分．)

Definition of $\mathcal{R}(e)$

$$
\begin{array}{rcl}
\mathcal{R}(x) & = & x \\
\mathcal{R}(n) & = & n \\
\mathcal{R}(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{R}(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{R}(x_1\ op\ x_2) & = & x_1\ op\ x_2 \\
\mathcal{R}(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ x\ \mathsf{then}\ \mathcal{R}(e_1)\ \mathsf{else}\ \mathcal{R}(e_2) \\
\mathcal{R}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \begin{cases} \mathsf{let}\ x = \mathcal{R}(e_1)\ \mathsf{in}\ \mathcal{R}(e_2) & (\text{if } x \in \mathbf{FV}(e_2) \text{ or if } e_1 = x_1\ x_2) \\ \mathcal{R}(e_2) & (\text{otherwise}) \end{cases} \\
\mathcal{R}(x_1\ x_2) & = & x_1\ x_2
\end{array}
$$

Definition of $\mathcal{R}(d)$

$$\mathcal{R}(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e) \quad = \quad \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to \mathcal{R}(e)$$

Definition of $\mathcal{R}(P)$

$$\mathcal{R}(\{d_1, \ldots, d_n\}, e) \quad = \quad (\{\mathcal{R}(d_1), \ldots, \mathcal{R}(d_n)\}, \mathcal{R}(e))$$

図 5.1: $\mathcal{C}$ 上で無駄な束縛の除去を行う変換 $\mathcal{R}$.

Definition of $\mathcal{P}_\delta(e)$ （δ は識別子から定数への部分関数である.）

$$
\begin{array}{rcl}
\mathcal{P}_\delta(x) & = & \begin{cases} \delta(x) & (\text{if } x \in \mathbf{dom}(\delta)) \\ x & (\text{otherwise}) \end{cases} \\
\mathcal{P}_\delta(n) & = & n \\
\mathcal{P}_\delta(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{P}_\delta(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{P}_\delta(x_1\ op\ x_2) & = & \mathcal{P}_\delta(x_1)\ op\ \mathcal{P}_\delta(x_2) \\
\mathcal{P}_\delta(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ \mathcal{P}_\delta(x)\ \mathsf{then}\ \mathcal{P}_\delta(e_1)\ \mathsf{else}\ \mathcal{P}_\delta(e_2) \\
\mathcal{P}_\delta(\mathsf{let}\ x = y\ \mathsf{in}\ e) & = & \mathcal{P}_{\delta[x \mapsto y]}(e) \\
\mathcal{P}_\delta(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \mathsf{let}\ x = \mathcal{P}_\delta(e_1)\ \mathsf{in}\ \mathcal{P}_\delta(e_2) \quad (\text{where } e_1 \text{ is not a variable}) \\
\mathcal{P}_\delta(x_1\ x_2) & = & \mathcal{P}_\delta(x_1)\ \mathcal{P}_\delta(x_2)
\end{array}
$$

Definition of $\mathcal{P}(d)$

$$
\mathcal{P}(\mathsf{let\ rec}\ f\ = \mathsf{fun}\ x \to e) \quad = \quad \mathsf{let\ rec}\ f\ = \mathsf{fun}\ x \to \mathcal{P}_\emptyset(e)
$$

Definition of $\mathcal{P}(P)$

$$
\mathcal{P}((\{d_1, \ldots, d_n\}, e)) \quad = \quad (\{\mathcal{P}(d_1), \ldots, \mathcal{P}(d_n)\}, \mathcal{P}_\emptyset(e))
$$

図 5.2: $\mathcal{C}$ 上でコピー伝播を行う変換 $\mathcal{P}$.

$(\{d_1, d_2\}, 3)$ においてはどちらの関数定義も無駄であるから，$(\emptyset, 3)$ にしてしまえば良いはずである．無駄な関数定義を除去できるように $\mathcal{R}$ の定義を書き換えよ．$\mathcal{R}(\{d_1, \ldots, d_n\}, e)$ の定義を書き換えればよいが，無駄な関数定義をできるだけたくさん除去できるように（しかし無駄でない関数定義は除去しないように）定義すること.

5.7.2 コピー伝播

コピー伝播 (*copy propagation*) は，let x=y in e を $[y/x]e$ に置き換える変換である．ただし，$[y/x]e$ は e 中の x を y に置き換えた式を表す．これによって x の束縛が無くなるので，効率が良くなることが期待される.

図 5.2 がコピー伝播を行う変換 $\mathcal{P}$ である．変換の途中で式 $\mathsf{let}\ x = y\ \mathsf{in}\ e$ を見つけると，$\mathcal{P}$ は写像 δ に x の束縛式が y であることを記録して e を変換する．変数 x を変換する際には，δ の定義域に x が含まれるかどうかを確認して，含まれているならば $\delta(x)$ に，含まれていないならば x に変換する．これにより，$\mathsf{let}\ x = y\ \mathsf{in}\ e$ を $[y/x]e$ に変換することができる.

Exercise 5.7.2 [$\star$] **定数畳み込み** (*constant folding*) を行う変換を定義せよ．定数畳み込み

とは，実行時の評価結果が分かっている値を計算してしまう変換である．例えば，let $x = 3$ in let $y = 4$ in $x + y$ は 7 に畳み込むことができる．

Exercise 5.7.3 [$\star\star$] インライン化 (*inlining*) を行う変換を定義せよ．インライン化とは関数呼び出しを呼び出されている関数の本体で置き換える変換である．例えば，$d_1 :=$ let rec $f =$ fun $x \to x + 2$ であるときに，プログラム $(\{d_1\}, f\ 5)$ をインライン化すると $(\{d_1\},$ let $t_1 = 5 + 2$ in$t_1)$ のようになる．プログラムが再帰呼び出しを含む場合には，無制限にインライン化を行うとプログラム変換が止まらなくなるので，そのようなことが無いように何らかの制限を加える必要がある．(例えばインライン化する深さを制限する，再帰を含む場合はインライン化を行わない等.)

5.8 扱っていないトピック

ここまで解説したコンパイラは，あくまでソース言語からアセンブリ言語に中間言語をはさみながら変換が可能であることを見せるためのものであり，実際に「使える」コンパイラにするにはかなりのギャップがある．今年度は時間の都合でこの「使えない」コンパイラを説明するだけで終わってしまうのだが，o 概ね以下のトピックを実装するともう少し使えるコンパイラになる.[3]興味のある人は自習してほしい．また，このうちの幾つかのトピックは，実験 4 で「コンパイラ」を選択すると体験することができる予定である．

（ざっと解説を書こうとしたが，中途半端な解説を挙げるよりはキーワードを挙げておく方が役に立ちそうなので，とりあえずキーワードだけ挙げておく.）

5.8.1 仮想マシンコードにおける最適化

フロー解析

定数畳み込み

無駄な定義の除去

生存変数解析

レジスタ割り当て

5.8.2 クロージャ変換

5.8.3 命令選択

5.8.4 その他のトピック

[3]来年度はもう少しこの講義にこれらの内容を取り込みたい．