Vol. 20 No. 6　　情報処理学会論文誌　　Nov. 1979

ショートノート

変数の浅い束縛について†

黒川利明††

いわゆるブロック構造をもっているプログラミング言語における局所変数の束縛方式について検討する．いわゆる「浅い」変数束縛と「深い」変数束縛について，まずそれぞれの定義を与える．
次に Algol 系のプログラミング言語で用いられる「display」方式の変数束縛を取り上げて，これが上の定義に照らして「深い」のか「浅い」のかを考える．この3方式を比較して「浅い」束縛の利点を，処理速度，デバッグ，インタプリタとコンパイラの共存の3点について示す．最後に「浅い」束縛での問題点を述べ，この問題の背景について触れる．

1. まえがき

変数の浅い束縛 (shallow binding) は深い束縛 (deep binding) と対にして用いられる．これらはブロック構造を持つ言語において変数を扱う対照的方式を示している．本論文では，まず浅い束縛の定義を与え，第2節でコンパイラに用いられる display 方式は浅い束縛と良く似ているが，動的自由変数に対しては深い束縛になることを示し，第3節でこれら3方式の比較を行い，どのような場合に浅い束縛が優れているかを示す．一方浅い束縛では余りうまくゆかぬ事項を最後の **5.** で示す．

2. 浅い束縛と深い束縛——その定義——

簡単のために変数値というものは常に変数名にリンクしているものだと考えよう．これはインタプリタ方式の言語プロセッサでは自然な仮定である．ブロック外ですでに現われている変数名をブロック内の局所変数として実現するためには，同一変数名に対して複数の変数値を保持せねばならない．浅い束縛方式は変数名1つに対してスタックを1つ割り当て，最上部の値を現在の有効値とする (**図1**)．深い束縛は変数名—変数値のリンクをスタック内に複数個許し，最上部のリンクを有効とする (**図2**)．

浅い束縛に対する定義を以下に示す．

(1) 現在の変数値の存在場所が変数名に対して一意に決まっている．

(2) 変数値の取り出しが現在のブロック構造に無関係に一定のステップ数で行える．

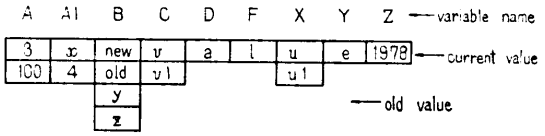

図 1 浅い束縛の一事例
Fig. 1 A model of shallow binding.

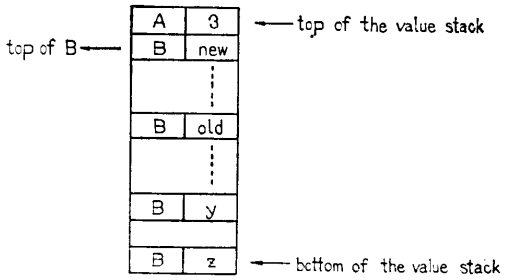

図 2 深い束縛の一事例
Fig. 2 A model of deep binding.

上記2定義は同値である．(1)→(2)は自明．(2)→(1)はブロック構造に無関係なある変換 f：変数名→変数値が得られるので，これをアドレス変換関数と思えば良い．

3. display 方式は浅い束縛か？

display 方式とは Algol 系のコンパイラで良く用いられる方法[1]で，プロセス中の局所変数の領域を動的に確保してゆき，変数値を局所変数領域の相対アドレスを用いて格納するものである (**図3**)．

この方式は1回の間接修飾をしているだけなので前節の定義(2)に合致する*．しかし，変数値の存在場所が一意に決まっているかというとそうでもないから

† On the Shallow Binding of Variables by TOSHIAKI KUROKAWA (Fuchu Works, Toshiba Co.).
†† 東芝(株)府中工場

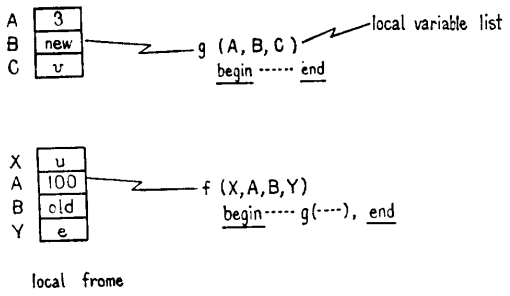

図 3 display 方式の事例
Fig. 3 A model of display binding.

(1)の定義に合致しない．さて display 方式は浅い束縛の一種なのであろうか？ 答は否であるが，その理由は(1)の定義に反するからではない．display 方式では局所変数は変数領域が取られるごとに暗に名前がつけ変わっているようなもので，変数値の存在場所を一意に定めているつもりなのである．

問題なのは以下のような局所的な自由変数の処置にある．例として $f \equiv$ **if** $a=0$ **then** 1 **else** 0 **endif** のような関数を考えて，f が $g \equiv$ **local** a; $a:=0$; f; **end** と，$h \equiv$ **local** a; $a:=1$; f; **end** という2関数で呼ばれる場合を考えてみると良い．この場合 a は g と h において局所変数であるにもかかわらず，f においては自由変数であるために，親関数の局所変数領域で a の値を求めねばならない．すなわち深い束縛でスタック中を探したのと同じ探索過程が必要となるのである．

Algol 系の言語では文法上の制約によって上例の f のような関数を与えることが出来ないが，Lisp のような動的性格をもつ言語ではこの f のような関数の扱いが「環境問題」として重要なものとなるのである[2]．

4. 3方式の比較——浅い束縛の利点——

○処理速度：変数の値を取り出す時，浅い束縛は速いから有利である．しかし浅い束縛の場合にはブロックの入口だけでなく出口においても局所変数の値を更新せねばならねという煩らわしさがある．ある程度量的な評価を行うために以下の式を導入しよう．変数は

α_D＝深い束縛と浅い束縛の変数アクセスの差の平均値．α_d は display 方式と浅い束縛の差．

β＝浅い束縛でブロックの出入口における1変数値の保存回復に必要な時間．

γ_D＝深い束縛で関数を呼び出す時のスタックの処理時間．γ_d は display 方式で変数域切り替えの処理時間．

m＝局所変数ののべの個数．

n＝局所変数値の参照，変更の回数．

p＝関数（サブルーチン）の呼び出し回数．

の8個であるとして，処理時間の差を求めると

$$T_D = n\alpha_D - m\beta + p\gamma_D \qquad (1)$$

が深い束縛と浅い束縛の処理時間の差を与える．

$$T_d = n\alpha_d - m\beta + p\gamma_d \qquad (2)$$

は display 方式と浅い束縛との間の時間差を与える．

上の2式だけではまだ充分なイメージがわかないであろうから，具体的な数字をあげてみることにする．TOSBAC-5600 上での LISP 1.9 インタプリタの場合には，$\alpha_D=4\ \mu$sec，$\beta=4\ \mu$sec，$\gamma_D=0$ sec となるので $T_D \fallingdotseq 4(n-m)$ となる．一般的に変数参照の回数が個数よりはるかに大きいので，$T_D>0$ と考えることができる．一方 LISP コンパイラの場合には，display 方式をとっていて $\alpha_d \fallingdotseq 0$, $\gamma_d=4\ \mu$sec となっている．よって $T_d \fallingdotseq 4(p-m)$ となって関数呼び出しの回数と変数参照の回数の差によってどちらが速いかということになる．この場合も一般的には変数の処理の方が関数呼び出しの回数より頻繁に起りそうなので浅い束縛の方が速いだろうと推測される．

○デバッグ：エラー発生時点で現在の変数値が取り出し易いという点は浅い束縛の利点である．深い束縛や display 方式では現在の状況を示すポインタが狂ってしまうと変数値がわからなくなる．ことに display 方式では変数名の失なわれているケースが多い．

○インタプリタとコンパイラの共存：会話型の言語プロセッサにおいてはユーザの便宜を考えるとインタプリタの方が好ましい．しかしながらインタプリタだけではオブジェクト・プログラムの処理速度が遅いので主要なサブルーチンをコンパイルすることによって全体としての処理効率と使い易さを両立させることになる．このような場合，浅い束縛を用いたコンパイラのオブジェクト・プログラムは簡単にインタプリタとのリンケージが取れるだけでなく，コンパイラ作成そのものが容易になる．これは特に display 方式と比較し

---

* この段落での表現について，誤解を生じやすいという指摘を受けた．実際，そうなので以下のように補いたい．
まず，display 方式が浅い束縛であるかどうかという点であるが，答は，局所自由変数を使用しない場合では浅い束縛であり，局所自由変数を使用する場合には浅い束縛ではない．
変数束縛の浅い／深いという相違が環境によって変化することは決して奇異なことではない．この相違は，変数値の決定性に根ざしているのであるから，環境によって決定性が変更されることは，ありうることなのである．実際，スタックを用いる通常の浅い束縛方式ですら，変数が1つしか存在しないという条件下では，浅い束縛に他ならない．なんとなれば，スタックの最上部が常にその変数値を保持しているからである．

ての利点である.

5. 浅い束縛での問題点*

前節で浅い束縛の利点を述べたが，以下に述べるような欠点も存在する.

第1の問題は Moses[2] の指摘した文脈の保存である. Baker[3] は文脈の木構造表現を確保しておくことによって浅い束縛でも文脈切替えを行うことが原理的には可能であることを示したが，その場合のオーバヘッドは大き過ぎて実用上は余り役に立たない.

第2の問題は変数名の消去である. 前節のデバッグの項目と矛盾するが，サブルーチンライブラリなどに用いるには局所変数名が消去された方が望ましい. この点では display 方式が優れている.

第3の問題点は global exit である. すなわちブロック構造の何重かの内部から一気にブロック構造の外まで抜け出すような制御において局所変数の値をどう処理するかである.

display 方式や深い束縛では現在位置を示すポインタを何度分か下げれば，それでブロック構造の抜け出しは完了する. これは前述の文脈問題の1事例であるが，浅い束縛の場合にはブロック構造を一つずつ抜け出すような特別な処理を施さねばならない.

6. あとがき

浅い束縛の問題は Lisp 関係者には無意識的にせよ馴染深い問題であろうが，しかしながら浅い束縛を用いた時の問題点や利点についてはまだ充分に知られていないところがある. 特にコンパイラ・インタプリタ共存方式での変数の処理方式については，ここで述べた方式以外にもまだ良い方法があるのではないかと思われる. 実際筆者が本論文を書く動機となったのは LISP 1.9 システムのコンパイラ[4] を display 方式で実現したところが，筆者の予想に反して浅い束縛よりも処理速度で劣っていたことにある. その反省の結果は現時点においてはむしろ浅い束縛の方式コンパイラにも有益であるという 4. の結論となった. 実際米国の Interlisp においても深い束縛を display 方式で実現することを試みた spaghetti-stack[5] が，インプリメントの段階では浅い束縛方式を取り入れたという話[6] もある. 本論文での議論は既存のハードウェアを暗黙の仮定としたが，Lisp マシンのような新ハードウェアにおいてはより強力な新方式が提案されるかもしれない.

* この項目に関しても，この3者を比較することの意味について疑念が寄せられた. その疑念とは「この3者だけで比較が充分なのか？他に良い方法がまだあるのではないか？」ということであった. この疑念は，しごくもっともなものである. 実際筆者としては，この他のより良い方法の呈示を行うつもりで，この論文に着手したのだが，時間的余裕がなく，中間結果をこのような形で発表したものである. 無論，今筆者の頭の中にあるものが最上であるかどうかはわかっていない. ともかく，この3者を比較することは，その出発点となる筈である.

参考文献

1) Randell B. and Russel, L. J.: ALGOL 60 Implementation, Academic Press (1964).
2) Moses, J.: The Function of FUNCTION in LISP, Memo 199, MIT AI Lab. (1970).
3) Baker, H. G. Jr.: Shallow Binding in Lisp 1.5, C. ACM, Vol. 21, No. 7, pp. 565-569 (1978).
4) LISP User's Manual, EPICS-5-ON-4, 電子技術総合研究所—東芝総合研究所 (Mar. 1978).
5) Babrow, D. G. and Wegbreit, B.: A Model and Stack Implementation of Multiple Environments, C. ACM Vol. 16, No. 10, pp. 591-603 (1973).
6) Teitelman, W.: Private Communications (Apr. 1976).

(昭和53年10月17日受付)
(昭和54年7月19日採録)