Vrije Universiteit Brussel

Software Languages Lab

Programming Language Engineering
Master of Computer Science

Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 7: Partial Evaluation

Theo D'Hondt
Software Languages Lab

"… a technique for several different types of program optimization by specialization…"

Static vs. dynamic features

version 5

```
static EXP_type evaluate_set(PAI_type Operands)
  { sET_type set_thread;
    CNT_type continuation;
    EXP_type expression;
    PAI_type expressions,
             residue;
    SYM_type variable;
    TAG_type tag;
    tag = Tag_of(Operands);
    switch (tag)
      { case NUL_tag:
          return Main_Error_Text(MSV_error_string,
                                 Main_Set_String);
        case PAI_tag:
          variable = Operands->car;
          if (is_SYM(variable))
            { expressions = Operands->cdr;
              tag = Tag_of(expressions);
              switch (tag)
                { case NUL_tag:
                    return Main_Error_Text(E1X_error_string,
                                           Main_Set_String);
                  case PAI_tag:
                    expression = expressions->car;
                    residue    = expressions->cdr;
                    if (is_NUL(residue))
                      { continuation = Thread_Push(Continue_set,
                                                   sET_size);
                        set_thread = (sET_type)continuation;
                        set_thread->var = variable;
                        return evaluate_expression(expression); }
                    return Main_Error_Text(TMX_error_string,
                                           Main_Set_String); }
              return Main_Error_Text(ITF_error_string,
                                     Main_Set_String); }
            return Main_Error_Text(IVV_error_string,
                                   Main_Set_String); }
      return Main_Error_Text(ITF_error_string,
                             Main_Set_String); }
```

Static vs. dynamic features

version 5

static

```
static EXP_type evaluate_set(PAI_type Operands)
  { sET_type set_thread;
    CNT_type continuation;
    EXP_type expression;
    PAI_type expressions,
             residue;
    SYM_type variable;
    TAG_type tag;
    tag = Tag_of(Operands);
    switch (tag)
      { case NUL_tag:
          return Main_Error_Text(MSV_error_string,
                                 Main_Set_String);
        case PAI_tag:
          variable = Operands->car;
          if (is_SYM(variable))
            { expressions = Operands->cdr;
              tag = Tag_of(expressions);
              switch (tag)
                { case NUL_tag:
                    return Main_Error_Text(E1X_error_string,
                                           Main_Set_String);
                  case PAI_tag:
                    expression = expressions->car;
                    residue    = expressions->cdr;
                    if (is_NUL(residue))
                      { continuation = Thread_Push(Continue_set,
                                                   sET_size);
                        set_thread = (sET_type)continuation;
                        set_thread->var = variable;
                        return evaluate_expression(expression); }
                    return Main_Error_Text(TMX_error_string,
                                           Main_Set_String); }
              return Main_Error_Text(ITF_error_string,
                                     Main_Set_String); }
          return Main_Error_Text(IVV_error_string,
                                 Main_Set_String); }
    return Main_Error_Text(ITF_error_string,
                           Main_Set_String); }
```

Static vs. dynamic features

version 5

```
static EXP_type evaluate_set(PAI_type Operands)
  { sET_type set_thread;
    CNT_type continuation;
    EXP_type expression;
    PAI_type expressions,
             residue;
    SYM_type variable;
    TAG_type tag;
    tag = Tag_of(Operands);
    switch (tag)
      { case NUL_tag:
          return Main_Error_Text(MSV_error_string,
                                 Main_Set_String);
        case PAI_tag:
          variable = Operands->car;
          if (is_SYM(variable))
            { expressions = Operands->cdr;
              tag = Tag_of(expressions);
              switch (tag)
                { case NUL_tag:
                    return Main_Error_Text(E1X_error_string,
                                           Main_Set_String);
                  case PAI_tag:
                    expression = expressions->car;
                    residue    = expressions->cdr;
                    if (is_NUL(residue))
                      { continuation = Thread_Push(Continue_set,
                                                   sET_size);
                        set_thread = (sET_type)continuation;
                        set_thread->var = variable;
                        return evaluate_expression(expression); }
                    return Main_Error_Text(TMX_error_string,
                                           Main_Set_String); }
              return Main_Error_Text(ITF_error_string,
                                     Main_Set_String); }
          return Main_Error_Text(IVV_error_string,
                                 Main_Set_String); }
  return Main_Error_Text(ITF_error_string,
                         Main_Set_String); }
```

static

dynamic

Preprocess static features

version 6

```
static EXP_type compile_set(PAI_type Operands)
  { EXP_type compiled_expression,
             expression;
    PAI_type expressions,
             residue;
    SET_type compiled_set;
    SYM_type variable;
    TAG_type tag;
    tag = Tag_of(Operands);
    switch (tag)
      { case NUL_tag:
          return Main_Error_Text(MSV_error_string,
                                 Main_Set_String);
        case PAI_tag:
          variable = Operands->car;
          if (is_SYM(variable))
            { expressions = Operands->cdr;
              tag = Tag_of(expressions);
              switch (tag)
                { case NUL_tag:
                    return Main_Error_Text(E1X_error_string,
                                           Main_Set_String);
                  case PAI_tag:
                    expression = expressions->car;
                    residue    = expressions->cdr;
                    if (is_NUL(residue))
                      { compiled_expression = compile_expression(expression);
                        compiled_set = make_SET(variable,
                                                compiled_expression);
                        return compiled_set; }

                    return Main_Error_Text(TMX_error_string,
                                           Main_Set_String); }
              return Main_Error_Text(ITF_error_string,
                                     Main_Set_String); }
          return Main_Error_Text(IVV_error_string,
                                 Main_Set_String); }
    return Main_Error_Text(ITF_error_string,
                           Main_Set_String); }
```

Preprocess static features

version 6

```
static EXP_type compile_set(PAI_type Operands)
  { EXP_type compiled_expression,
             expression;
    PAI_type expressions,
             residue;
    SET_type compiled_set;
    SYM_type variable;
    TAG_type tag;
    tag = Tag_of(Operands);
    switch (tag)
      { case NUL_tag:
          return Main_Error_Text(MSV_error_string,
                                 Main_Set_String);
        case PAI_tag:
          variable = Operands->car;
          if (is_SYM(variable))
            { expressions = Operands->cdr;
              tag = Tag_of(expressions);
              switch (tag)
                { case NUL_tag:
                    return Main_Error_Text(E1X_error_string,
                                           Main_Set_String);
                  case PAI_tag:
                    expression = expressions->car;
                    residue    = expressions->cdr;
                    if (is_NUL(residue))
                      { compiled_expression = compile_expression(expression);
                        compiled_set = make_SET(variable,
                                                compiled_expression);
                        return compiled_set; }

                    return Main_Error_Text(TMX_error_string,
                                           Main_Set_String); }
              return Main_Error_Text(ITF_error_string,
                                     Main_Set_String); }
            return Main_Error_Text(IVV_error_string,
                                   Main_Set_String); }
      return Main_Error_Text(ITF_error_string,
                             Main_Set_String); }
```

Preprocess static features (cont'd)

```
static EXP_type evaluate_set(SET_type Set)
  { sET_type set_thread;
    CNT_type continuation;
    EXP_type expression;
    SYM_type variable;
    variable   = Set->var;
    expression = Set->exp;
continuation = Thread_Push(Continue_set,
                           sET_size);
set_thread = (sET_type)continuation;
set_thread->var = variable;
return evaluate_expression(expression); }
```

version 6

Preprocess static features (cont'd)

version 6

```
static EXP_type evaluate_set(SET_type Set)
  { sET_type set_thread;
    CNT_type continuation;
    EXP_type expression;
    SYM_type variable;
    variable   = Set->var;
    expression = Set->exp;
continuation = Thread_Push(Continue_set,
                           sET_size);
set_thread = (sET_type)continuation;
set_thread->var = variable;
return evaluate_expression(expression); }
```

```
typedef struct SET * SET_type;

typedef
  struct SET { CEL_type hdr;
               SYM_type var;
               EXP_type exp; }
SET;
BYT_type   is_SET(EXP_type);
SET_type make_SET(SYM_type,
                  EXP_type);
```

Abstract grammar

```
<expression>    ::=  <pair> | <vector> | <symbol> | <null> | <real> |
                     <number> | <string> | <character>
<pair>          ::=  PAI <expression> <expression>
<vector>        ::=  VEC <expression>*
<symbol>        ::=  SYM [string]
<true>          ::=  TRU
<false>         ::=  FLS
<null>          ::=  NUL
<real>          ::=  REA [float]
<number>        ::=  NBR [integer]
<string>        ::=  STR [string]
<character>     ::=  CHA [character]

<continuation>  ::=  CNT <c function> <continuation> <expression>* |  <null>
<environment>   ::=  ENV <symbol> <value> <environment>  |  <null>
<native>        ::=  NAT [address]
<c function>    ::=  CFN [address]
<unspecified>   ::=  UNS

<procedure>     ::=  PRC <expression> <expression> <environment>
```

version 5

Rich abstract grammar

```
<expression>     ::=  <pair> | <vector> | <symbol> | <null> | <real> |
                      <number> | <string> | <character>
<pair>           ::=  PAI <expression> <expression>
<vector>         ::=  VEC <expression>*
<symbol>         ::=  SYM [string]
<true>           ::=  TRU
<false>          ::=  FLS
<null>           ::=  NUL
<real>           ::=  REA [float]
<number>         ::=  NBR [integer]
<string>         ::=  STR [string]
<character>      ::=  CHA [character]

<continuation>   ::=  CNT <c function> <continuation> <expression>* |  <null>
<environment>    ::=  ENV <symbol> <value> <environment>  |  <null>
```

version 6

```
<native>         ::=  NAT [address]
<c function>     ::=  CFN [address]
<unspecified>    ::=  UNS

<procedure>      ::=  PRC <vector> <vector> <environment> |
                      PRZ <vector> <symbol> <vector> <environment>
<application>    ::=  APL <expression> <vector>
<begin>          ::=  BEG <vector>
<define>         ::=  DFV <symbol> <expression> |
                      DFF <symbol> <vector> <vector> |
                      DFZ <symbol> <vector> <symbol> <vector>
<if>             ::=  IFF <expression> <expression> <expression> |
                      IFZ <expression> <expression>
<lambda>         ::=  LMB <vector> <vector> |
                      LMZ <vector> <symbol> <vector>
<quote>          ::=  QUO <expression>
<set>            ::=  SET <symbol> <expression>
<while>          ::=  WHI <expression> <vector>
```

Slip REP-loop

```
static NIL_type read_eval_print(NIL_type)
  { EXP_type compiled_expression,
             expression,
             value;
    TRA_type status;
    TXT_type input;
    Slip_Print("\n>>>");
    Slip_Read(&input);
    if ((status = setjmp(Trampoline)) == Initiate_loop)
      { expression = Read_Parse(input);
        compiled_expression = Compile_Compile(expression);
        Rollback_environment = Dictionary_Environment;
        Thread_Initialize();
        Thread_Push(Continue_print_function,
                    0);
        if ((status = setjmp(Trampoline)) == Initiate_loop)
          { value = Evaluate_Evaluate(compiled_expression);
            for (;;)
              value = Evaluate_Continue(value); }
        if (status == Abort_loop)
          Dictionary_Environment = Rollback_environment; }}
```

Slip REP-loop

```
static NIL_type read_eval_print(NIL_type)
  { EXP_type compiled_expression,
             expression,
             value;
    TRA_type status;
    TXT_type input;
    Slip_Print("\n>>>");
    Slip_Read(&input);
    if ((status = setjmp(Trampoline)) == Initiate_loop)
      { expression = Read_Parse(input);
        compiled_expression = Compile_Compile(expression);
        Rollback_environment = Dictionary_Environment;
        Thread_Initialize();
        Thread_Push(Continue_print_function,
                    0);
        if ((status = setjmp(Trampoline)) == Initiate_loop)
          { value = Evaluate_Evaluate(compiled_expression);
            for (;;)
              value = Evaluate_Continue(value); }
        if (status == Abort_loop)
          Dictionary_Environment = Rollback_environment; }}
```

Compiling expressions

```
static EXP_type compile_symbol(SYM_type Variable)
  { return Variable; }

static EXP_type compile_value(EXP_type Value)
  { return Value; }
```

```
static
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
          return compile_form(Expressic
        case SYM_tag:
          return compile_symbol(Express
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
          return compile_value(Expression); }
    return Main_Error_Tag(IXT_error_string,
                          tag); }
```

```
static EXP_type compile_form(PAI_type Form)
  { EXP_type operands,
             operator;
    operator = Form->car;
               ...dr;
               ...in_Begin)
               ...le_begin(operands);
               ...in_Define)
               ...e_define(operands);
    if (operator == Main_If)
          return compile_if(operands);
    if (operator == Main_Lambda)
          return compile_lambda(operands);
    if (operator == Main_Quote)
          return compile_quote(operands);
    if (operator == Main_Set)
          return compile_set(operands);
    if (operator == Main_While)
          return compile_while(operands);
    return compile_application(operator,
                               operands); }
```

Compiling expressions

```
static EXP_type compile_form(PAI_type Form)
  { EXP_type operands,
             operator;
    operator = Form->car;
                    dr;
                    in_Begin)
                    le_begin(operands);
                    in Define)
                    fine(operands);
                    f)
                    e_if(operands);
                    ambda)
                    mbda(operands);
                    uote)
                    uote(operands);
                    et)
                    _set(operands);
                    hile)
                    hile(operands);
                    tion(operator,
                         operands); }
```

```
static EXP_type compile_symbol(SYM_type Variable)
  { return Variable; }
```

```
static EXP_type compile_expression(EXP_type Expression)
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
          return compile_form(Expression);
        case SYM_tag:
          return compile_symbol(Expression);
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
          return compile_value(Expression); }
    return Main_Error_Tag(IXT_error_string,
                          tag); }
```

Compiling expressions

```
static EXP_type compile_symbol(SYM_type Variable)
  { return Variable; }

static EXP_type compile_value(EXP_type Value)
  { return Value; }
```

```
static EXP_type compile_form(PAI_type Form)
  { EXP_type operands,
             operator;
    operator = Form->car;
                 dr;
                 in_Begin)
                 le_begin(operands);
                 in_Define)
                 e_define(operands);
                 ain_If)
             return compile_if(operands);
    if (operator == Main_Lambda)
         return compile_lambda(operands);
    if (operator == Main_Quote)
         return compile_quote(operands);
    if (operator == Main_Set)
           return compile_set(operands);
    if (operator == Main_While)
         return compile_while(operands);
    return compile_application(operator,
                               operands); }
```

```
static
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
          return compile_form(Expressio
        case SYM_tag:
          return compile_symbol(Express
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
          return compile_value(Expression); }
    return Main_Error_Tag(IXT_error_string,
                          tag); }
```

Compiling expressions

```
static EXP_type compile_symbol(S
  { return Variable; }

static EXP_type compile_value(EX
  { return Value; }

static
  { TA
    tag = Tag_of(Expression);
    switch (tag)
      { case PAI_tag:
          return compile_form(Expressio
        case SYM_tag:
          return compile_symbol(Express
        case CHA_tag:
        case FLS_tag:
        case NUL_tag:
        case NBR_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case VEC_tag:
          return compile_value(Expression); }
  return Main_Error_Tag(IXT_error_string,
                        tag); }
```

```
static EXP_type compile_form(PAI_type Form)
  { EXP_type operands,
             operator;
    operator = Form->car;
    operands = Form->cdr;
    if (operator == Main_Begin)
          return compile_begin(operands);
    if (operator == Main_Define)
          return compile_define(operands);
    if (operator == Main_If)
          return compile_if(operands);
    if (operator == Main_Lambda)
          return compile_lambda(operands);
    if (operator == Main_Quote)
          return compile_quote(operands);
    if (operator == Main_Set)
          return compile_set(operands);
    if (operator == Main_While)
          return compile_while(operands);
    return compile_application(operator,
                               operands); }
```

Compiling applications

```
static VEC_type compile_sequence(PAI_type Expressions,
                                 UNS_type Size)
  { EXP_type compiled_expression,
```

```
static EXP_type compile_application(EXP_type Operator,
                                    EXP_type Operands)
  { APL_type compiled_application;
    EXP_type compiled_operator;
    TAG_type tag;
    VEC_type compiled_operands;
    compiled_operator = compile_expression(Operator);
    tag = Tag_of(Operands);
    switch (tag)
      { case PAI_tag:
          compiled_operands = compile_sequence(Operands,
                                               1);
          break;
        case NUL_tag:
          compiled_operands = Main_Empty_Vector;
          break;
        default:
          return Main_Error_Text(ITF_error_string,
                                 Main_Application_String); }
    compiled_application = make_APL(compiled_operator,
                                    compiled_operands);
    return compiled_application; }
```

```
xpression);
(expressions,
 Size + 1);


ing,
_String); }
ion;
```

Compiling applications

```
static VEC_type compile_sequence(PAI_type Expressions,
                                 UNS_type Size)
  { EXP_type compiled_expression,
                                                    xpression);
                                                    (expressions,
                                                     Size + 1);
                                                    ing,
                                                    _String); }
                                                    ion;
```

```
static EXP_type compile_application(EXP_type Operator,
                                    EXP_type Operands)
  { APL_type compiled_application;
    EXP_type compiled_operator;
    TAG_type tag;
    VEC_type compiled_operands;
    compiled_operator = compile_expression(Operator);
    tag = Tag_of(Operands);
    switch (tag)
      { case PAI_tag:
          compiled_operands = compile_sequence(Operands,
                                               1);
          break;
        case NUL_tag:
          compiled_operands = Main_Empty_Vector;
          break;
        default:
          return Main_Error_Text(ITF_error_string,
                                 Main_Application_String); }
    compiled_application = make_APL(compiled_operator,
                                    compiled_operands);
    return compiled_application; }
```

Compiling applications

```
static EXP_type comp

  { APL_type compile
    EXP_type compile
    TAG_type tag;
    VEC_type compile
    compiled_operato
    tag = Tag_of(Ope
    switch (tag)
      { case PAI_tag
          compiled_o

        break;
      case NUL_tag
        compiled_o
        break;
      default:
        return Mai

  compiled_applica

  return compiled_application; }
```

```
static VEC_type compile_sequence(PAI_type Expressions,
                                 UNS_type Size)
  { EXP_type compiled_expression,
             expression;
    PAI_type expressions;
    TAG_type tag;
    VEC_type compiled_sequence;
    expression  = Expressions->car;
    expressions = Expressions->cdr;
    compiled_expression = compile_expression(expression);
    tag = Tag_of(expressions);
    switch (tag)
      { case NUL_tag:
          compiled_sequence = make_VEC(Size);
          break;
        case PAI_tag:
          compiled_sequence = compile_sequence(expressions,
                                               Size + 1);
          break;
        default:
          return Main_Error_Text(ITF_error_string,
                                 Main_Sequence_String); }
    compiled_sequence[Size] = compiled_expression;
    return compiled_sequence; }
```

Evaluating expressions

```
static EXP_type evaluate_expression(EXP_type Expression)
  { TAG_type tag;
    tag = Tag_of(Ex
    switch (tag)
      { case APL_ta

        case BEG_ta

        case DFF_ta
                  re
        case DFV_ta
                  re
        case DFZ_ta
          return ev
        case IFF_ta

        case IFZ_ta

        case LMB_ta

        case LMZ_ta

        case QUO_ta

        case SET_ta
```

```
            case SYM_tag:
                                  return evaluate_symbol(Expression);
            case WHI_tag:
                                   return evaluate_while(Expression);
            case CHA_tag:
            case FLS_tag:
            case NAT_tag:
            case NBR_tag:
            case NUL_tag:
            case PAI_tag:
            case PRC_tag:
            case PRZ_tag:
            case REA_tag:
            case STR_tag:
            case TRU_tag:
            case USP_tag:
            case VEC_tag:
                                        return evaluate_value(Expression);
            case CFN_tag:
            case CNT_tag:
            case ENV_tag:
              return Main_Error_Tag(ILT_error_string,
                                    tag); }
          return Main_Error_Tag(IXT_error_string,
                                tag); }
```

Evaluating expressions

```
static EXP_type evaluate_expression(EXP_type Expression)
  { TAG_type tag;
    tag = Tag_of(Expression);
    switch (tag)
      { case APL_tag:
                    return evaluate_application(Expression);
        case BEG_tag:
                         return evaluate_begin(Expression);
        case DFF_tag:
                  return evaluate_define_function(Expression);
        case DFV_tag:
                  return evaluate_define_variable(Expression);
        case DFZ_tag:
          return evaluate_define_function_vararg(Expression);
        case IFF_tag:
                  return evaluate_if_double(Expression);
        case IFZ_tag:
                  return evaluate_if_single(Expression);
        case LMB_tag:
                     return evaluate_lambda(Expression);
        case LMZ_tag:
                return evaluate_lambda_vararg(Expression);
        case QUO_tag:
                      return evaluate_quote(Expression);
        case SET_tag:
                        return evaluate_set(Expression);
```

Evaluating expressions

```
static EXP_type evaluate_expression(EXP_type Expression)
  { TAG_type tag;
    tag = Tag_of(Ex
    switch (tag)
      { case APL_ta

        case BEG_ta

        case DFF_ta
                  re
        case DFV_ta
                  re
        case DFZ_ta
          return ev
        case IFF_ta

        case IFZ_ta

        case LMB_ta

        case LMZ_ta

        case QUO_ta

        case SET_ta
```

```
        case SYM_tag:
                                  return evaluate_symbol(Expression);
        case WHI_tag:
                                   return evaluate_while(Expression);
        case CHA_tag:
        case FLS_tag:
        case NAT_tag:
        case NBR_tag:
        case NUL_tag:
        case PAI_tag:
        case PRC_tag:
        case PRZ_tag:
        case REA_tag:
        case STR_tag:
        case TRU_tag:
        case USP_tag:
        case VEC_tag:
                                      return evaluate_value(Expression);
        case CFN_tag:
        case CNT_tag:
        case ENV_tag:
          return Main_Error_Tag(ILT_error_string,
                                         tag); }
      return Main_Error_Tag(IXT_error_string,
                                  tag); }
```

Evaluating applications

```
static const TXT_type Application_String = "application";

static CFN_type Continue_application;

typedef struct aPL
typedef struct aPL

static const UNS_t

static EXP_type ev
  { aPL_type appli
    CNT_type conti
    EXP_type expre
    VEC_type opera
    expression = A
    operands   = A
    continuation =

    application_th
    application_th
    return evaluate_expression(expression); }

static NIL_type initialize_application(NIL_type)
  { Continue_application = make_CFN(continue_application); }
```

```
static EXP_type continue_application(EXP_type Procedure,
                                     CNT_type Continuation)
  { aPL_type application_thread;
    TAG_type tag;
    VEC_type operands;
    application_thread = (aPL_type)Continuation;
    operands = application_thread->opd;
    tag = Tag_of(Procedure);
    switch (tag)
      { case NAT_tag:
            return evaluate_native_call(Procedure,
                                        operands);
        case PRC_tag:
            return evaluate_bindings(Procedure,
                                     operands);
        case PRZ_tag:
            return evaluate_vararg_bindings(Procedure,
                                            operands); }
    return Main_Error_Text(PNR_error_string,
                           Application_String); }
```

Evaluating applications

```
static const TXT_type Application_String = "application";

static CFN_type Continue_application;

typedef struct aPL * aPL_type;
typedef struct aPL { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     VEC_type opd; } aPL;

static const UNS_type aPL_size = chunk_size(aPL);


static EXP_type evaluate_application(APL_type Application)
  { aPL_type application_thread;
    CNT_type continuation;
    EXP_type expression;
    VEC_type operands;
    expression = Application->exp;
    operands   = Application->opr;
    continuation = Thread_Push(Continue_application,
                               aPL_size);
    application_thread = (aPL_type)continuation;
    application_thread->opd = operands;
    return evaluate_expression(expression); }

static NIL_type initialize_application(NIL_type)
  { Continue_application = make_CFN(continue_application); }
```

Evaluating applications

```
static const TXT_type Application_String = "application";

static CFN_type Continue_application;

typedef struct aPL
typedef struct aPL

static const UNS_t

static EXP_type ev
  { aPL_type appli
    CNT_type conti
    EXP_type expre
    VEC_type opera
    expression = A
    operands   = A
    continuation =

    application_th
    application_th
    return evaluate_expression(expression); }

static NIL_type initialize_application(NIL_type)
  { Continue_application = make_CFN(continue_application); }
```

```
static EXP_type continue_application(EXP_type Procedure,
                                     CNT_type Continuation)
  { aPL_type application_thread;
    TAG_type tag;
    VEC_type operands;
    application_thread = (aPL_type)Continuation;
    operands = application_thread->opd;
    tag = Tag_of(Procedure);
    switch (tag)
      { case NAT_tag:
            return evaluate_native_call(Procedure,
                                        operands);
        case PRC_tag:
            return evaluate_bindings(Procedure,
                                     operands);
        case PRZ_tag:
            return evaluate_vararg_bindings(Procedure,
                                            operands); }
    return Main_Error_Text(PNR_error_string,
                           Application_String); }
```

Iterative constructs

version 6

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
```

```
static EXP_type sequence(VEC_type Expressions,
                         UNS_type Index)
  { sEQ_type sequence_thread;
    CNT_type continuation;
    EXP_type expression;
    NBR_type position;
    UNS_type size;
    expression = Expressions[Index];
    size = size_VEC(Expressions);
    if (Index <
      { positio
        continu

        sequenc
        sequenc
        sequenc
    return eval
```

```
EQ;

_size(sEQ);
```

```
static EXP_type continue_sequence(EXP_type Value,
                                  CNT_type Continuation)
  { sEQ_type sequence_thread;
    VEC_type expressions;
    NBR_type position;
    UNS_type index;
    sequence_thread = (sEQ_type)Continuation;
    expressions = sequence_thread->exs;
    position    = sequence_thread->pos;
    index = position->lng;
    return sequence(expressions,
                    index); }
```

Iterative constructs

version 6

```
static EXP_type sequen

  { sEQ_type sequence_
    CNT_type continuat
    EXP_type expressio
    NBR_type position;
    UNS_type size;
    expression = Expre
    size = size_VEC(Ex
    if (Index <
      { positio
        continu
          sequenc
          sequenc
          sequenc
    return eval
```

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     VEC_type exs;
                     NBR_type pos; } sEQ;

static const UNS_type sEQ_size = chunk_size(sEQ);
```

```
static EXP_type evaluate_sequence(VEC_type Expressions)
  { return sequence(Expressions,
                    1); }

static NIL_type initialize_sequence(NIL_type)
  { Continue_sequence = make_CFN(continue_sequence); }
```

```
static
    { sE
      VE
      NB
      UN
    sequence_thread = (sEQ_type)Continuation;
    expressions = sequence_thread->exs;
    position    = sequence_thread->pos;
    index = position->lng;
    return sequence(expressions,
                    index); }
```

Iterative constructs

version 6

```
static EXP_type continue_with(EXP_type Value)
  { CNT_type continuation;
    CCC_type function;
    CFN_type c_function;
    continuation = Thread_Pop();
    c_function = continuation->cfn;
    function = c_function->ccc;
    return function(Value,
                    continuation); }
```

```
static EXP_type sequence(VEC_type Expressions,
                         UNS_type Index)
  { sEQ_type sequence_thread;
    CNT_type continuation;
    EXP_type expression;
    NBR_type position;
    UNS_type size;
    expression = Expressions[Index];
    size = size_VEC(Expressions);
    if (Index < size)
      { position = make_NBR(Index + 1);
        continuation = Thread_Push(Continue_sequence,
                                   sEQ_size);
        sequence_thread = (sEQ_type)continuation;
        sequence_thread->exs = Expressions;
        sequence_thread->pos = position; }
    return evaluate_expression(expression); }
```

```
atic CFN_type Continue_sequence;

pedef struct sEQ * sEQ_type;
pedef struct sEQ { CEL_type hdr;
                   CFN_type cfn;
EQ;
_size(sEQ);
alue,
ontinuation)
;
    index = position->lng;
    return sequence(expressions,
                    index); }
```

Iterative constructs

version 6

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
```

```
static EXP_type sequence(VEC_type Expressions,
                         UNS_type Index)
  { sEQ_type sequence_thread;
    CNT_type continuation;
    EXP_type expression;
    NBR_type position;
    UNS_type size;
    expression = Expressions[Index];
    size = size_VEC(Expressions);
    if (Index <
      { positio
        continu

        sequenc
        sequenc
        sequenc
    return eval
```

```
EQ;

_size(sEQ);
```

```
static EXP_type continue_sequence(EXP_type Value,
                                  CNT_type Continuation)
  { sEQ_type sequence_thread;
    VEC_type expressions;
    NBR_type position;
    UNS_type index;
    sequence_thread = (sEQ_type)Continuation;
    expressions = sequence_thread->exs;
    position    = sequence_thread->pos;
    index = position->lng;
    return sequence(expressions,
                    index); }
```

Iterative constructs (cont'd)

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     VEC_type exs;
```

```
static EXP_type continue_sequence(EXP_type Value)
  { sEQ_type sequence_thread;
    EXP_type expression;
    VEC_type expressions;
    NBR_type position;
    UNS_type index,
             size_x;
sequence_thread = (sEQ_type)Thread_Peek();
expressions = sequence_thread->exs;
position    = sequence_thread->pos;
index = position->lng + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

version 7

Iterative constructs (cont'd)

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     VEC_type exs;
                     NBR_type pos; } sEQ;

static const UNS_type sEQ_size = chunk_size(sEQ);
```

```
static EXP_type evaluate_sequence(VEC_type Expressions)
  { sEQ_type sequence_thread;
    EXP_type expression;
    UNS_type size_x;
    size_x = size_VEC(Expressions);
    expression = Expressions[1];
    if (size_x > 1)
      { sequence_thread = (sEQ_type)Thread_Push(Continue_sequence,
                                                sEQ_size);
        sequence_thread->exs = Expressions;
        sequence_thread->pos = Main_One; }
    return evaluate_expression(expression); }

static NIL_type initialize_sequence(NIL_type)
  { Continue_sequence = make_CFN(continue_sequence); }
```

version 7

Iterative constructs (cont'd)

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     VEC_type exs;
```

```
static EXP_type continue_sequence(EXP_type Value)
  { sEQ_type sequence_thread;
    EXP_type expression;
    VEC_type expressions;
    NBR_type position;
    UNS_type index,
             size_x;
sequence_thread = (sEQ_type)Thread_Peek();
expressions = sequence_thread->exs;
position    = sequence_thread->pos;
index = position->lng + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

```
ssions)

Continue_sequence,
sEQ_size);

ce); }
```

version 7

Iterative constructs (cont'd)

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     VEC_type exs;
```

```
static EXP_type continue_with(EXP_type Value)
  { CNT_type continuation;
    CCC_type function;
    CFN_type c_function;
    continuation = Thread_Peek();
    c_function = continuation->cfn;
    function = c_function->ccc;
    return function(Value); }
```

```
static EXP_type continue_sequence(EXP_type Value)
  { sEQ_type sequence_thread;
    EXP_type expression;
    VEC_type expressions;
    NBR_type position;
    UNS_type index,
             size_x;
    sequence_thread = (sEQ_type)Thread_Peek();
    expressions = sequence_thread->exs;
    position    = sequence_thread->pos;
    index = position->lng + 1;
    expression = expressions[index];
    size_x = size_VEC(expressions);
    if (index < size_x)
      { sequence_thread = (sEQ_type)Thread_Keep();
        position = make_NBR(index);
        sequence_thread->pos = position; }
    else
      Thread_Zap();
    return evaluate_expression(expression); }
```

version 7

Iterative constructs (cont'd)

```
static CFN_type Continue_sequence;

typedef struct sEQ * sEQ_type;
typedef struct sEQ { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     VEC_type exs;
```

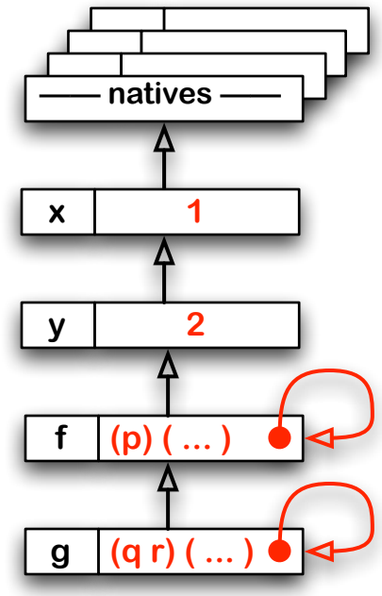
```
static EXP_type continue_with(EXP_type Value)
  { CNT_type continuation;
    CCC_type function;
    CFN_type c_function;
    continuation = Thread_Peek();
    c_function = continuation->cfn;
    function = c_function->ccc;
    return function(Value); }
```

```
static EXP_type continue_sequence(EXP_type Value)
  { sEQ_type sequence_thread;
    EXP_type expression;
    VEC_type expressions;
    NBR_type position;
    UNS_type index,
             size_x;
    sequence_thread = (sEQ_type)Thread_Peek();
    expressions = sequence_thread->exs;
    position    = sequence_thread->pos;
    index = position->lng + 1;
    expression = expressions[index];
    size_x = size_VEC(expressions);
    if (index < size_x)
      { sequence_thread = (sEQ_type)Thread_Keep();
        position = make_NBR(index);
        sequence_thread->pos = position; }
    else
      Thread_Zap();
    return evaluate_expression(expression); }
```

version 7

Environment revisited (cont'd)

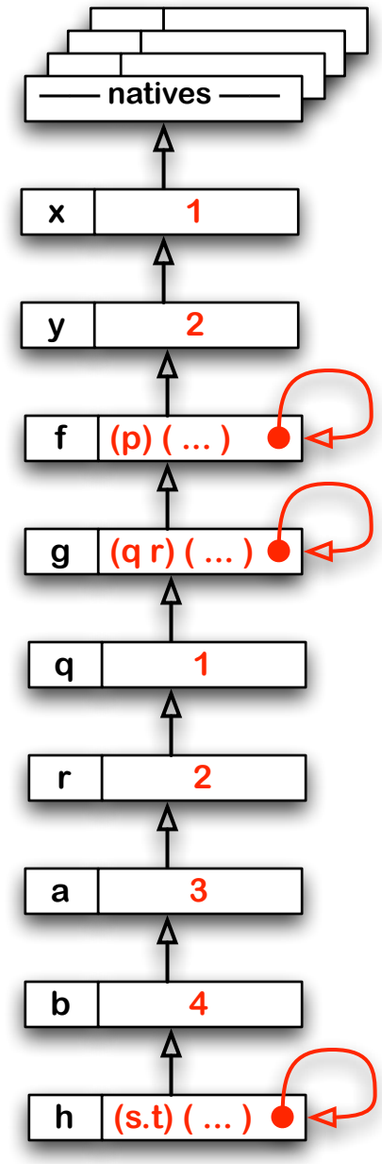
```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



⇒ 8

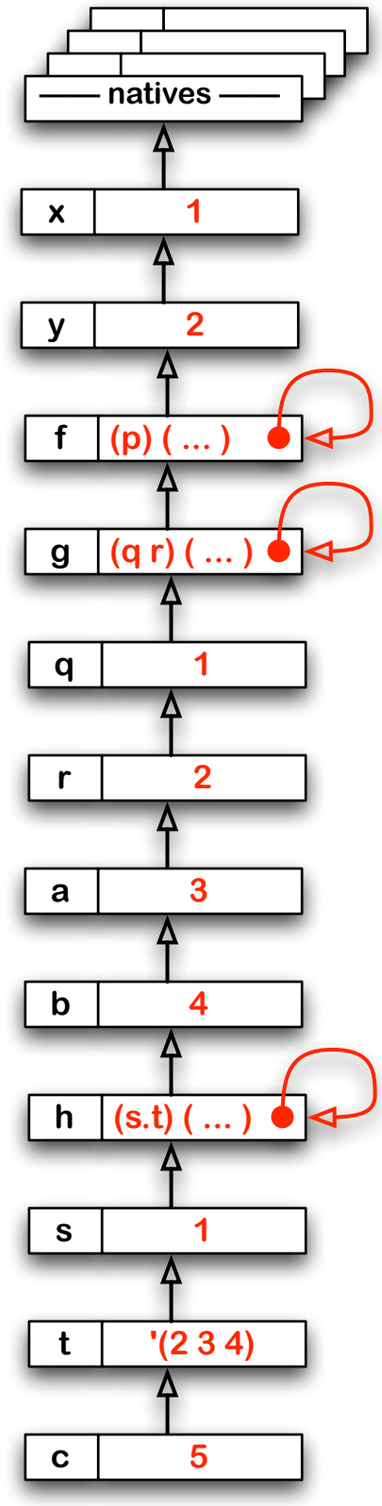
Environment revisited (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



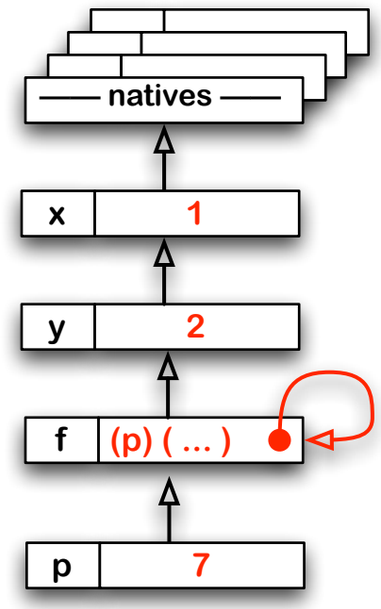
Environment revisited (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



Environment revisited (cont'd)

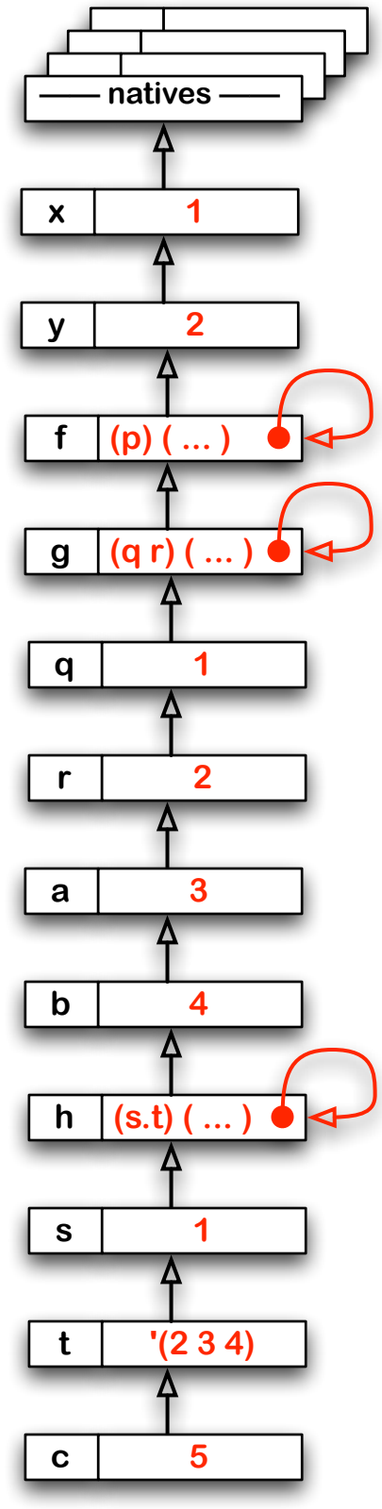
```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



Environment revisited (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```

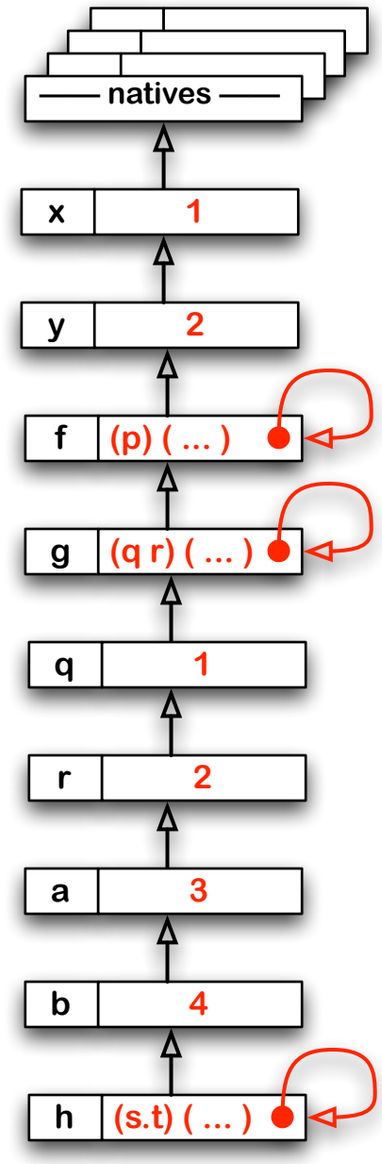
⇒ 8



Environment revisited (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```

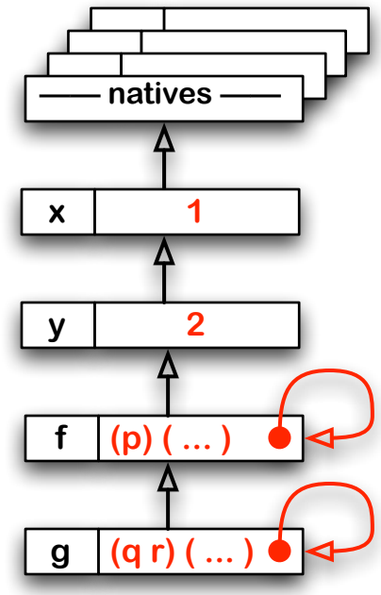
⇒ 8



Environment revisited (cont'd)

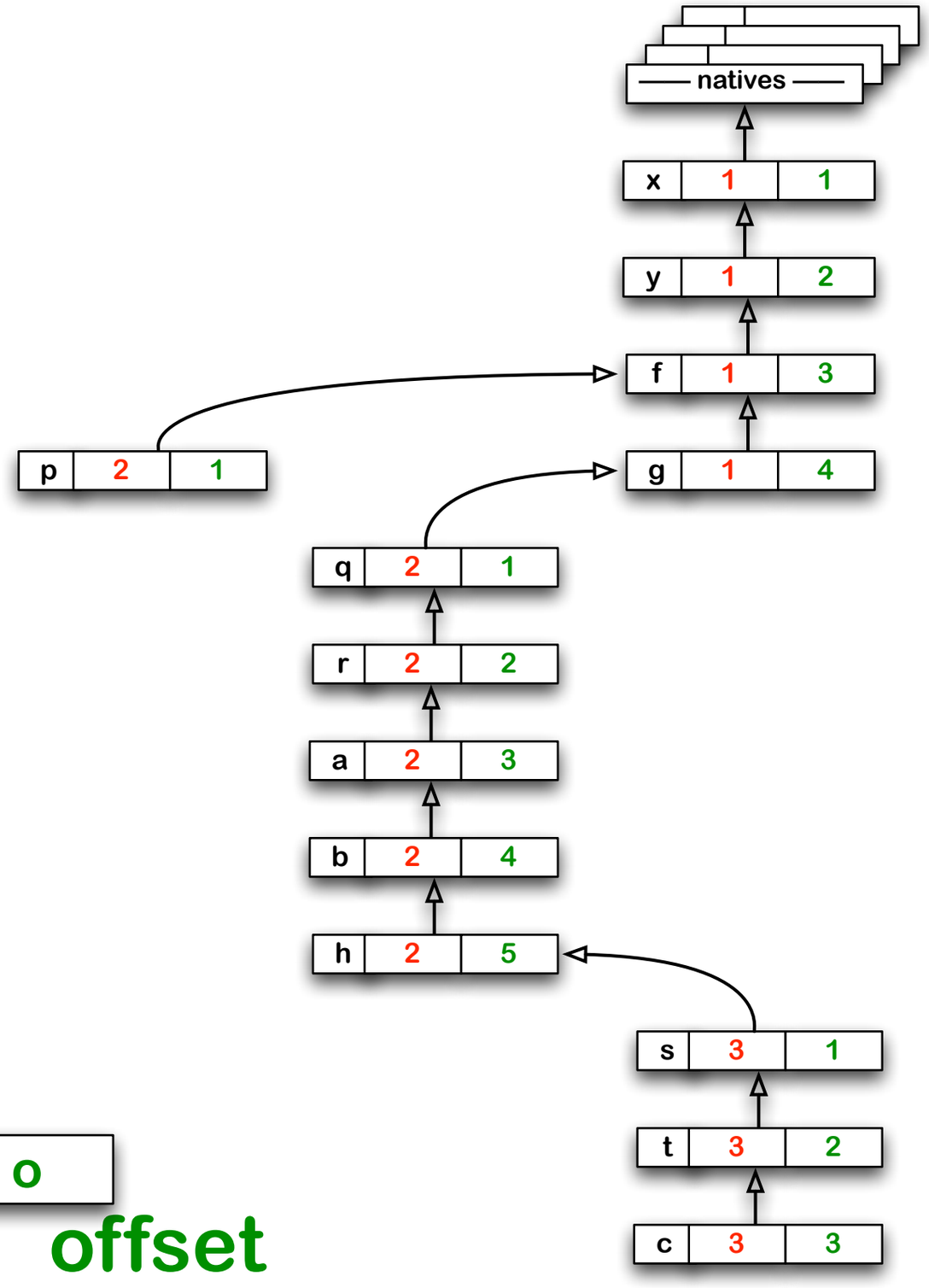
```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```

⇒ 8



Lexical addressing

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
(g x y))
```

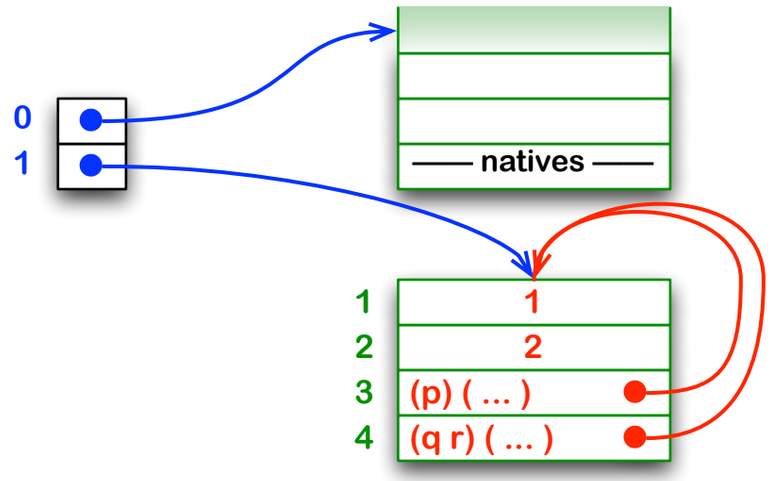


name offset

scope

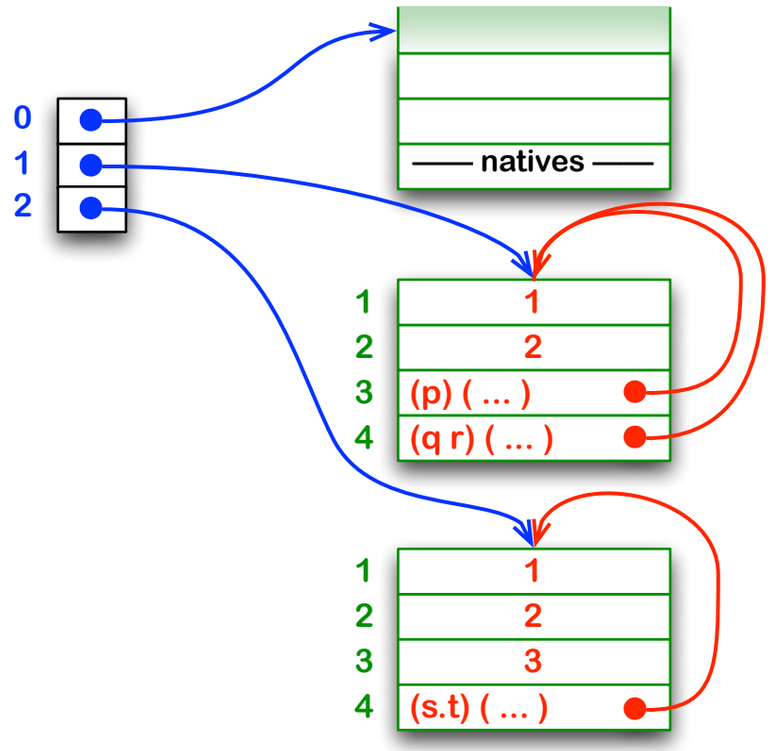
Using lexical addressing

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



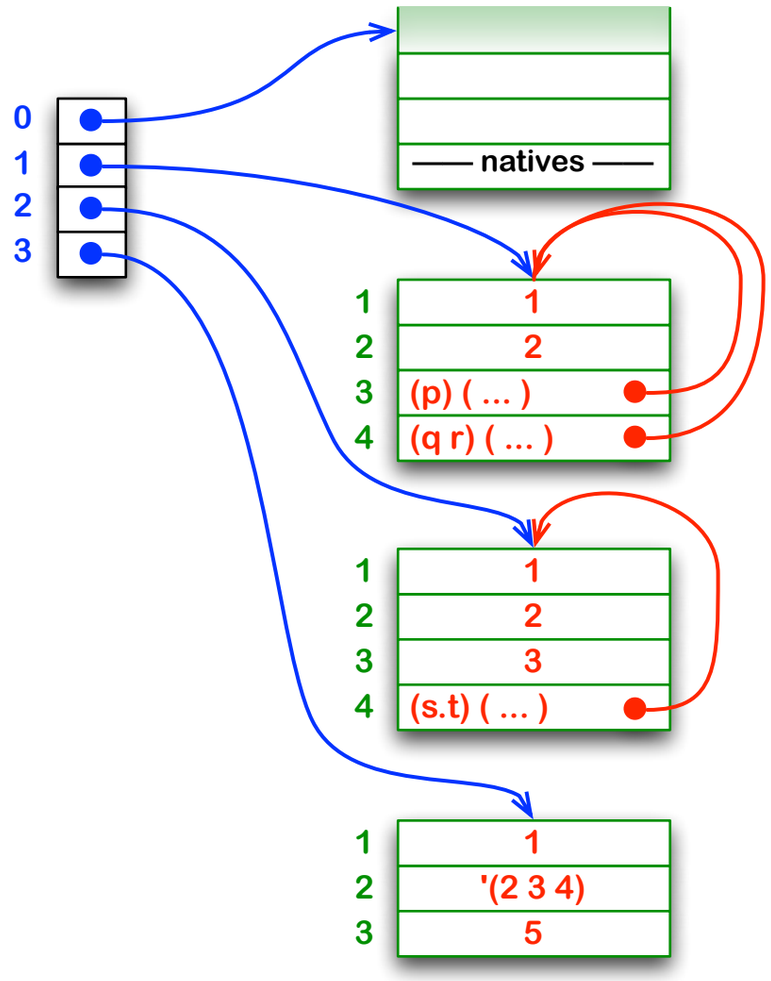
Using lexical addressing (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



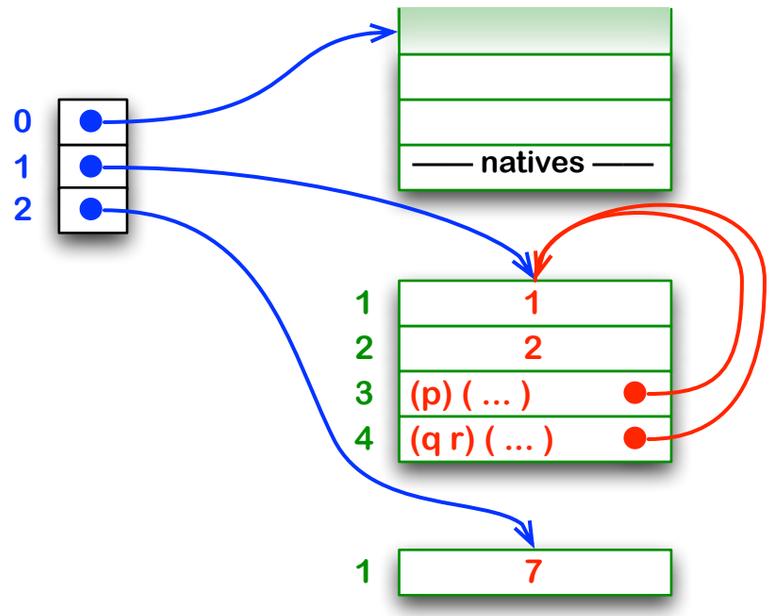
Using lexical addressing (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



Using lexical addressing (cont'd)

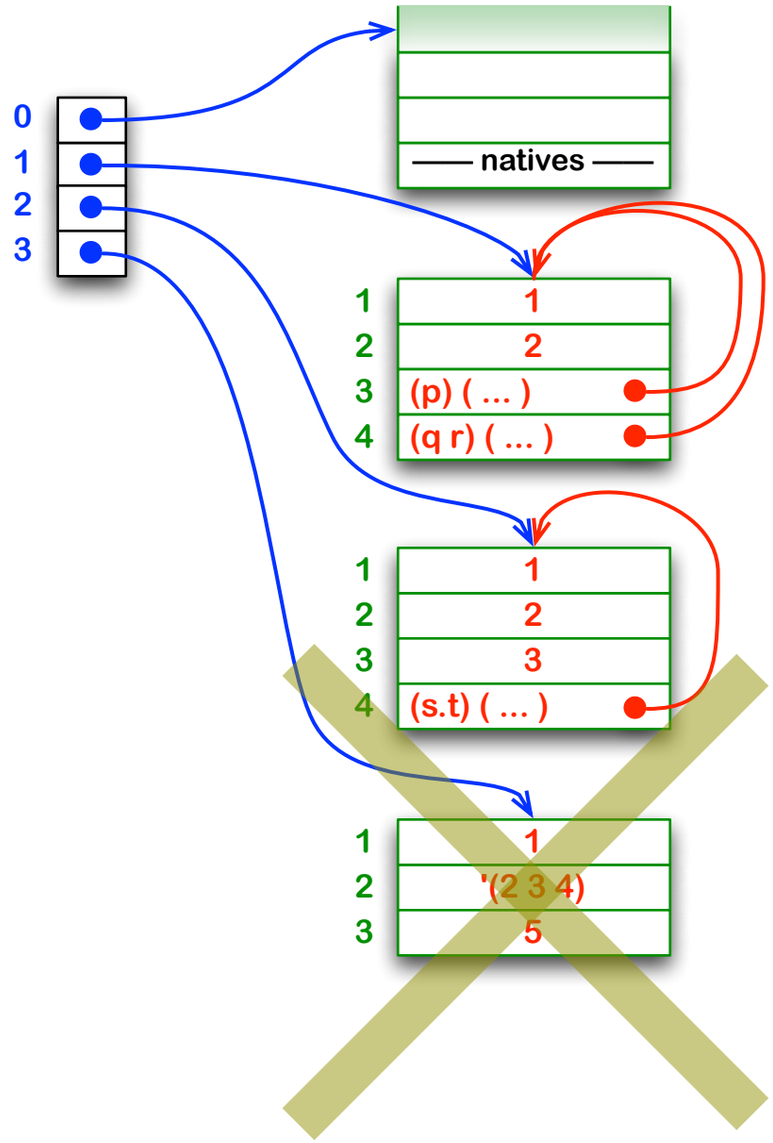
```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



Using lexical addressing (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```

⇒ 8

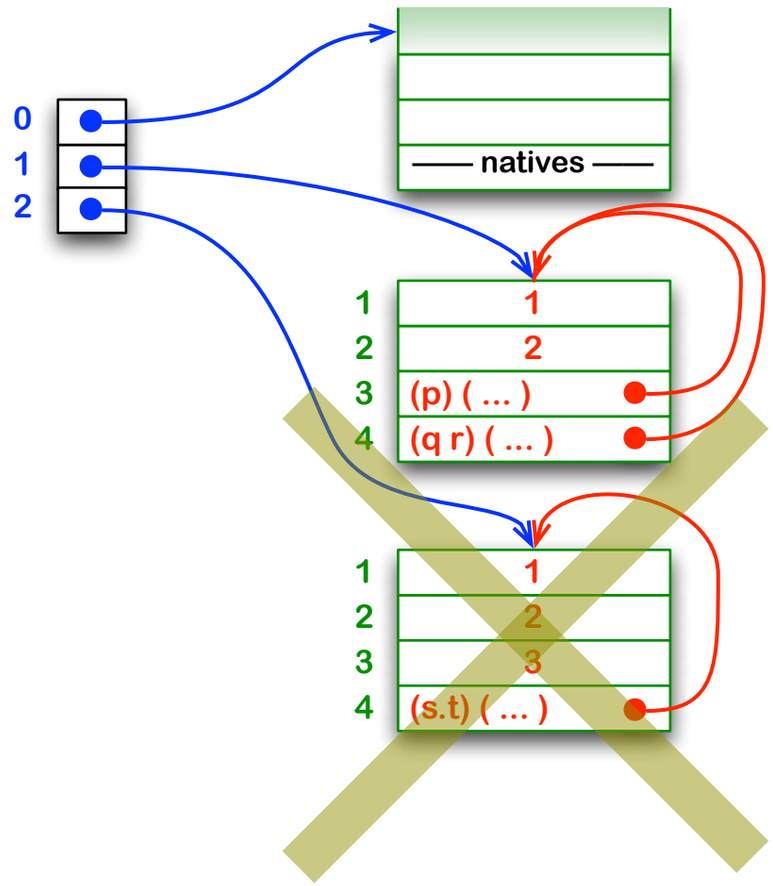


tail call

Using lexical addressing (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```

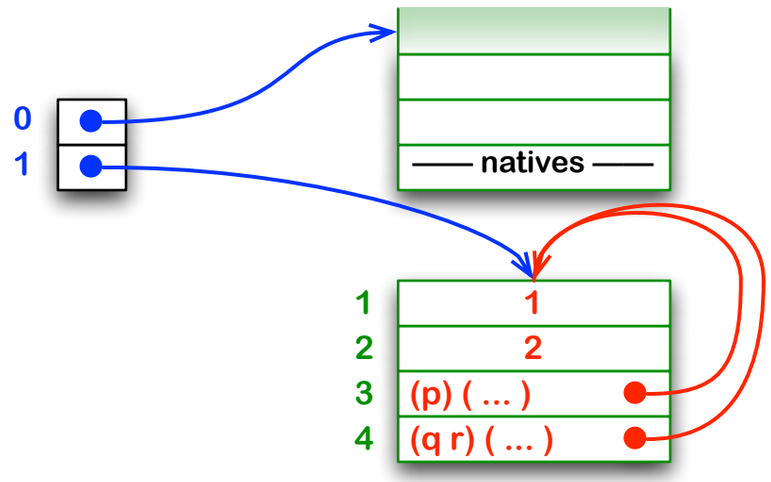
⇒ 8



tail call

Using lexical addressing (cont'd)

```
(begin
  (define x 1)
  (define y 2)
  (define (f p)
    (+ p x))
  (define (g q r)
    (define a 3)
    (define b 4)
    (define (h s . t)
      (define c 5)
      (f (+ c (car t))))
    (h q r a b))
  (g x y))
```



⇒ 8

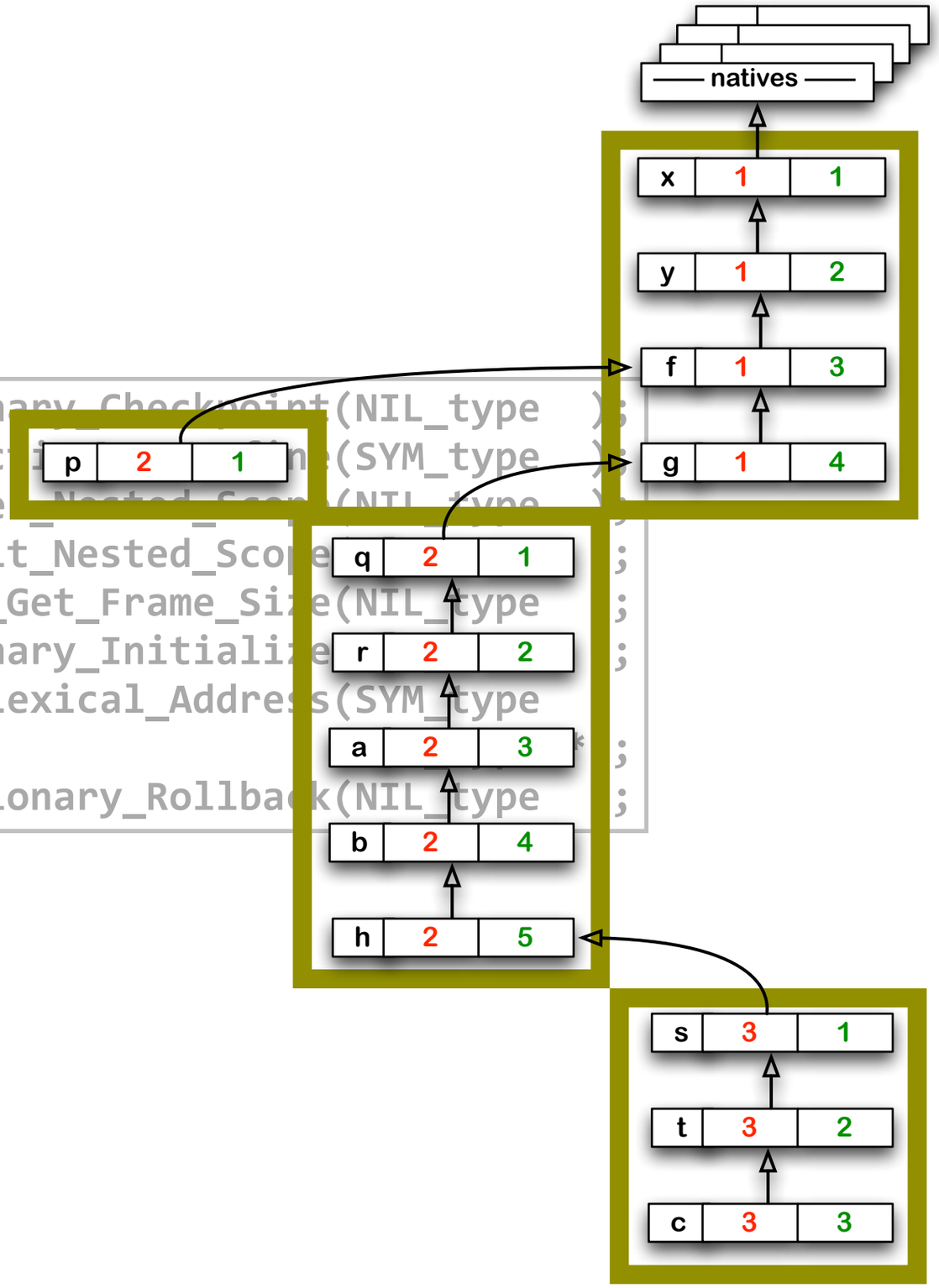
Implementing lexical addressing

```
NIL_type         Dictionary_Checkpoint(NIL_type  );
UNS_type             Dictionary_Define(SYM_type  );
NIL_type Dictionary_Enter_Nested_Scope(NIL_type  );
UNS_type  Dictionary_Exit_Nested_Scope(NIL_type  );
UNS_type     Dictionary_Get_Frame_Size(NIL_type  );
NIL_type         Dictionary_Initialize(NIL_type  );
UNS_type    Dictionary_Lexical_Address(SYM_type  ,
                                       UNS_type *);
NIL_type           Dictionary_Rollback(NIL_type  );
```

version 8

Implementing lexical addressing

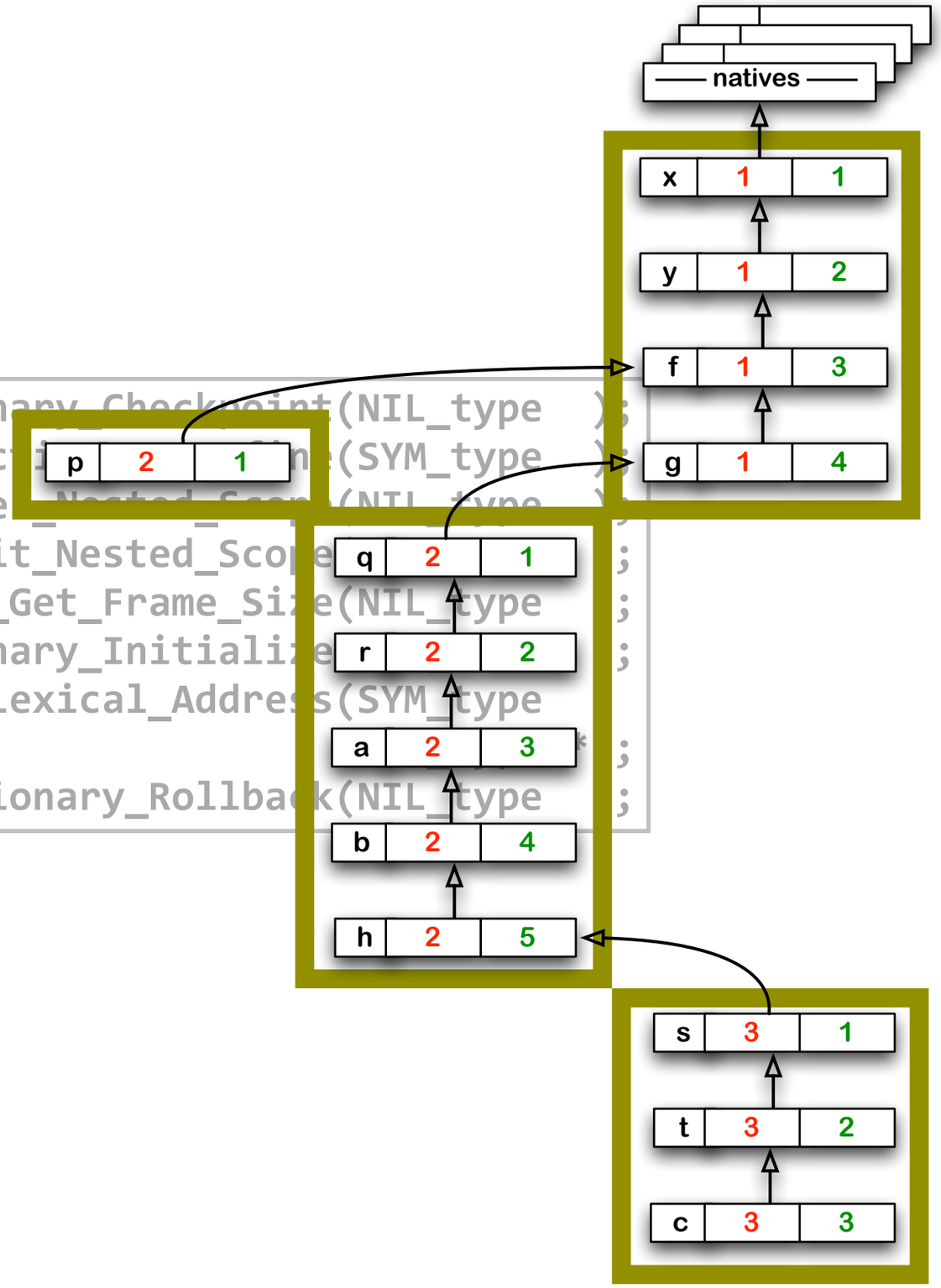
```
NIL_type          Dictionary_Checkpoint(NIL_type  );
UNS_type              Dictionary_Create(SYM_type  );
NIL_type Dictionary_Enter_Nested_Scope(NIL_type  );
UNS_type  Dictionary_Exit_Nested_Scope(           );
UNS_type     Dictionary_Get_Frame_Size(NIL_type  );
NIL_type         Dictionary_Initialize(           );
UNS_type    Dictionary_Lexical_Address(SYM_type
                                                  );
NIL_type             Dictionary_Rollback(NIL_type  );
```



Implementing lexical addressing

```
typedef
  struct FRM { CEL_type hdr;
               SYM_type var;
               FRM_type frm; } FRM;
BYT_type   is_FRM(EXP_type);
FRM_type make_FRM(SYM_type,
                  FRM_type);
```

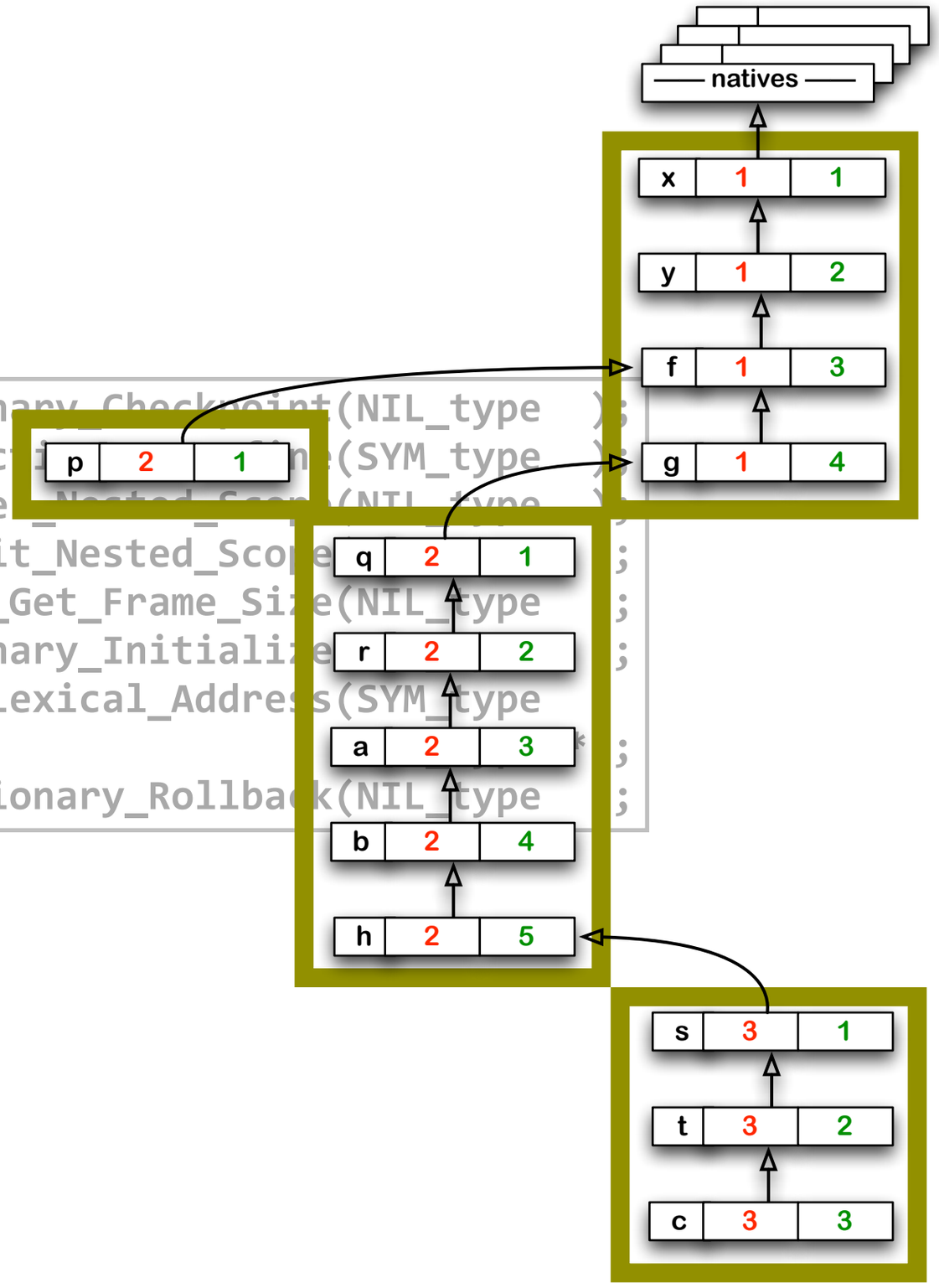
```
NIL_type          Dictionary_Checkpoint(NIL_type  );
UNS_type          Dictionary_Define(SYM_type  );
NIL_type Dictionary_Enter_Nested_Scope(NIL_type  );
UNS_type  Dictionary_Exit_Nested_Scope(NIL_type  );
UNS_type     Dictionary_Get_Frame_Size(NIL_type  );
NIL_type          Dictionary_Initialize(NIL_type  );
UNS_type    Dictionary_Lexical_Address(SYM_type
                                       *);
NIL_type          Dictionary_Rollback(NIL_type  );
```



Implementing lexical addressing

```
typedef
  struct FRM { CEL_type hdr;
               SYM_type var;
               FRM_type frm; } FRM;
BYT_type   is_FRM(EXP_type);
FRM_type make_FRM(SYM_type,
                  FRM_type);
```

```
NIL_type          Dictionary_Checkpoint(NIL_type  );
UNS_type             Dictionary_Define(SYM_type  );
NIL_type Dictionary_Enter_Nested_Scope(NIL_type  );
UNS_type  Dictionary_Exit_Nested_Scope(NIL_type  );
UNS_type     Dictionary_Get_Frame_Size(NIL_type  );
NIL_type          Dictionary_Initialize(NIL_type  );
UNS_type    Dictionary_Lexical_Address(SYM_type
                                       *);
NIL_type            Dictionary_Rollback(NIL_type  );
```




```
typedef
  struct ENV { CEL_type hdr;
               FRM_type frm;
               NBR_type siz;
               ENV_type env; } ENV;
BYT_type   is_ENV(EXP_type);
ENV_type make_ENV(FRM_type,
                  NBR_type,
                  ENV_type);
```

Compiling a set!

```
static EXP_type compile_set(PAI_type Operands)
  {

                    { compiled_expression = compile_expression(expression);
                      offset = Dictionary_Lexical_Address(variable,
                                                          &scope);
                      if (offset == 0)
                        return Main_Error_Symbol(VNF_error_string,
                                                 variable);
                    c_offset = make_NBR(offset);
                    if (scope == 0)
                      compiled_set = make_STL(c_offset,
                                              compiled_expression);
                    else
                      { c_scope = make_NBR(scope);
                        compiled_set = make_STG(c_scope,
                                                c_offset,
                                                compiled_expression); }
                    return compiled_set; }


                                    }
```

Compiling a set!

```
static EXP_type compile_set(PAI_type Operands)
  {

                    { compiled_expression = compile_expression(expression);
                      offset = Dictionary_Lexical_Address(variable,
                                                          &scope);
                      if (offset == 0)
                        return Main_Error_Symbol(VNF_error_string,
                                                 variable);
                    c_offset = make_NBR(offset);
                    if (scope == 0)
                      compiled_set = make_STL(c_offset,
                                              compiled_expression);
                    else
                      { c_scope = make_NBR(scope);
                        compiled_set = make_STG(c_scope,
                                                c_offset,
                                                compiled_expression); }
                    return compiled_set; }



                                    }
```

Implementing lexical addressing (cont'd)

```
VEC_type          Environment_Get_Environment(NIL_type);
VEC_type               Environment_Get_Frame(NIL_type);
EXP_type             Environment_Global_Get(UNS_type,
                                            UNS_type);
BLN_type          Environment_Global_Overflow(UNS_type);
NIL_type             Environment_Global_Set(UNS_type,
                                            UNS_type,
                                            EXP_type);
VEC_type        Environment_Grow_Environment(NIL_type);
NIL_type              Environment_Initialize(NIL_type);
EXP_type               Environment_Local_Get(UNS_type);
NIL_type               Environment_Local_Set(UNS_type,
                                            EXP_type);
NIL_type                Environment_Rollback(NIL_type);
NIL_type Environment_Set_Environment_And_Frame(VEC_type,
                                            VEC_type);
```

Implementing lexical addressing (cont'd)

```
const static UNS_type Initial_global_size = 64;



static VEC_type Current_environment;
static VEC_type Current_frame;
static VEC_type Global_frame;


NIL_type Environment_Initialize(NIL_type)
  { Current_environment = Main_Empty_Vector;
    Global_frame = Current_frame = make_VEC(Initial_global_size); }
```

```
VEC_type Environment_Grow_Environment(NIL_type)
  { UNS_type index,
             size;
    VEC_type environment,
             frame;
    size = size_VEC(Current_environment);
    environment = make_VEC(++size);
    for (index = 1;
         index < size;
         index++)
      { frame = Current_environment[index];
        environment[index] = frame; }
    environment[size] = Current_frame;
    return environment; }
```

Implementing lexical addressing (cont'd)

```
const static UNS_type Initial_global_size = 64;



static VEC_type Current_environment;
static VEC_type Current_frame;
static VEC_type Global_frame;


NIL_type Environment_Initialize(NIL_type)
  { Current_environment = Main_Empty_Vector;
    Global_frame = Current_frame = make_VEC(Initial_global_size); }
```

```
VEC_type Environment_Grow_Environment(NIL_type)
  { UNS_type index,
             size;
    VEC_type environment,
             frame;
    size = size_VEC(Current_environment);
    environment = make_VEC(++size);
    for (index = 1;
         index < size;
         index++)
      { frame = Current_environment[index];
        environment[index] = frame; }
    environment[size] = Current_frame;
    return environment; }
```

Implementing lexical addressing (cont'd)

```
const static UNS_type Initial_global_size = 64;



static VEC_type Current_environment;
static VEC_type Current_frame;
static VEC_type Global_frame;


NIL_type Environment_Initialize(NIL_type)
  { Current_environment = Main_Empty_Vector;
    Global_frame = Current_frame = make_VEC(Initial_global_size); }
```

```
VEC_type Environment_Grow_Environment(NIL_type)
  { UNS_type index,
             size;
    VEC_type environment,
             frame;
    size = size_VEC(Current_environment);
    environment = make_VEC(++size);
    for (index = 1;
         index < size;
         index++)
      { frame = Current_environment[index];
        environment[index] = frame; }
    environment[size] = Current_frame;
    return environment; }
```

Implementing lexical addressing (cont'd)

```
const static UNS_type Initial_global_size = 64;



static VEC_type Current_environment;
static VEC_type Current_frame;
static VEC_type Global_frame;


NIL_type Environment_Initialize(NIL_type)
  { Current_environment = Main_Empty_Vector;
    Global_frame = Current_frame = make_VEC(Initial_global_size); }
```

```
VEC_type Environment_Grow_Environment(NIL_type)
  { UNS_type index,
             size;
    VEC_type environment,
             frame;
    size = size_VEC(Current_environment);
    environment = make_VEC(++size);
    for (index = 1;
         index < size;
         index++)
      { frame = Current_environment[index];
        environment[index] = frame; }
    environment[size] = Current_frame;
    return environment; }
```

Implementing lexical addressing (cont'd)

```
const static UNS_type Initial_global_size = 64;



static VEC_type Current_environment;
static VEC_type Current_frame;
static VEC_type Global_frame;


NIL_type Environment_Initialize(NIL_type)
  { Current_environment = Main_Empty_Vector;
    Global_frame = Current_frame = make_VEC(Initial_global_size); }
```

```
VEC_type Environment_Grow_Environment(NIL_type)
  { UNS_type index,
             size;
    VEC_type environment,
             frame;
    size = size_VEC(Current_environment);
    environment = make_VEC(++size);
    for (index = 1;
         index < size;
         index++)
      { frame = Current_environment[index];
        environment[index] = frame; }
    environment[size] = Current_frame;
    return environment; }
```

Evaluating a local set!

```
static CFN_type Continue_set_local;

typedef struct sTL * sTL_type;
typedef struct sTL { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     NBR_type ofs; } sTL;


const static UNS_type sTL_size = chunk_size(sTL);

static EXP_type evaluate_set_local(STL_type Set)
  { sTL_type set_thread;
    EXP_type expression;
    NBR_type c_offset;


    c_offset   = Set->ofs;
    expression = Set->exp;
    set_thread = (sTL_type)Thread_Push(Continue_set_local,
                                       sTL_size);

    set_thread->ofs = c_offset;
    return evaluate_expression(expression); }

static NIL_type initialize_evaluate_set_local(NIL_type)
  { Continue_set_local = make_CFN(continue_set_local); }
```

Evaluating a global set!

```
static CFN_type Continue_set_global;

typedef struct sTG * sTG_type;
typedef struct sTG { CEL_type hdr;
                     CFN_type cfn;
                     CNT_type cnt;
                     NBR_type scp;
                     NBR_type ofs; } sTG;

const static UNS_type sTG_size = chunk_size(sTG);

static EXP_type evaluate_set_global(STG_type Set)
  { sTG_type set_thread;
    EXP_type expression;
    NBR_type c_offset,
             c_scope;
    c_scope    = Set->scp;
    c_offset   = Set->ofs;
    expression = Set->exp;
    set_thread = (sTG_type)Thread_Push(Continue_set_global,
                                       sTG_size);
    set_thread->scp = c_scope;
    set_thread->ofs = c_offset;
    return evaluate_expression(expression); }

static NIL_type initialize_evaluate_set_global(NIL_type)
  { Continue_set_global = make_CFN(continue_set_global); }
```

Evaluating a local set! (cont'd)

```
static EXP_type continue_set_local(EXP_type Value)
  { sTL_type set_thread;
    NBR_type c_offset;

    UNS_type offset;

    set_thread = (sTL_type)Thread_Pop();

    c_offset = set_thread->ofs;

    offset   = c_offset->lng;
    Environment_Local_Set(offset,
                          Value);

    return Value; }
```

Evaluating a local set! (cont'd)

```
static EXP_type continue_set_local(EXP_type Value)
  { sTL_type set_thread;
    NBR_type c_offset;

    UNS_type offset;

    set_thread = (sTL_type)Thread_Pop();

    c_offset = set_thread->ofs;

    offset   = c_offset->lng;
    Environment_Local_Set(offset,
                          Value);

    return Value; }
```

```
NIL_type Environment_Local_Set(UNS_type Offset,
                               EXP_type Value)
  { Current_frame[Offset] = Value; }
```

Evaluating a global set! (cont'd)

```
static EXP_type continue_set_global(EXP_type Value)
  { sTG_type set_thread;
    NBR_type c_offset,
             c_scope;
    UNS_type offset,
             scope;
    set_thread = (sTG_type)Thread_Pop();
    c_scope  = set_thread->scp;
    c_offset = set_thread->ofs;
    scope = c_scope->lng;
    offset = c_offset->lng;
    Environment_Global_Set(scope,
                           offset,
                           Value);
    return Value; }
```

Evaluating a global set! (cont'd)

```
static EXP_type continue_set_global(EXP_type Value)
  { sTG_type set_thread;
    NBR_type c_offset,
             c_scope;
    UNS_type offset,
             scope;
    set_thread = (sTG_type)Thread_Pop();
    c_scope  = set_thread->scp;
    c_offset = set_thread->ofs;
    scope = c_scope->lng;
    offset = c_offset->lng;
    Environment_Global_Set(scope,
                           offset,
                           Value);
    return Value; }
```

```
NIL_type Environment_Global_Set(UNS_type Scope,
                                UNS_type Offset,
                                EXP_type Value)
  { VEC_type frame;
    frame = Current_environment[Scope];
    frame[Offset] = Value; }
```