Vrije Universiteit Brussel

Software Languages Lab

Programming Language Engineering
Master of Computer Science

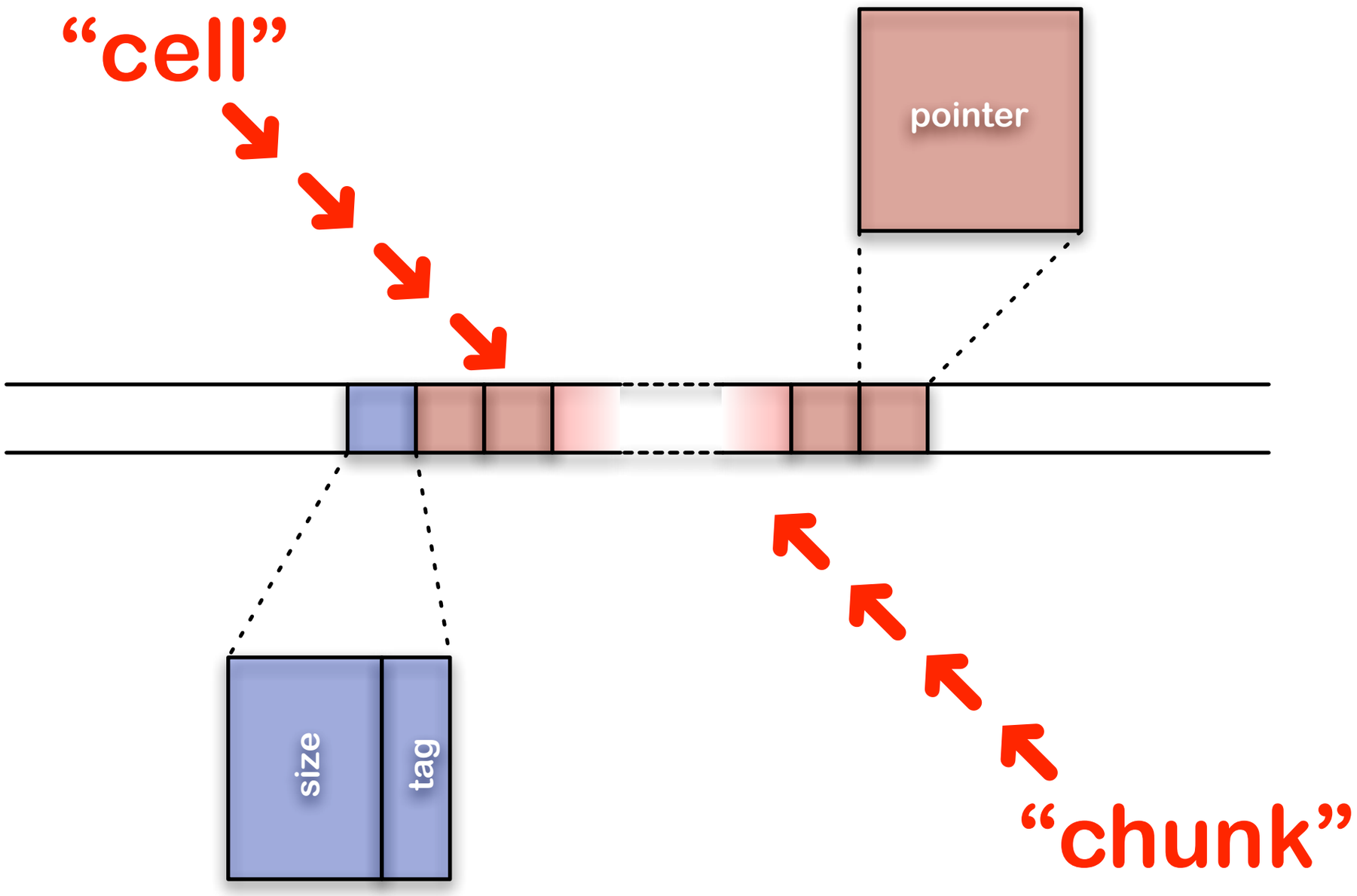
Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

Section 8: Memory Management

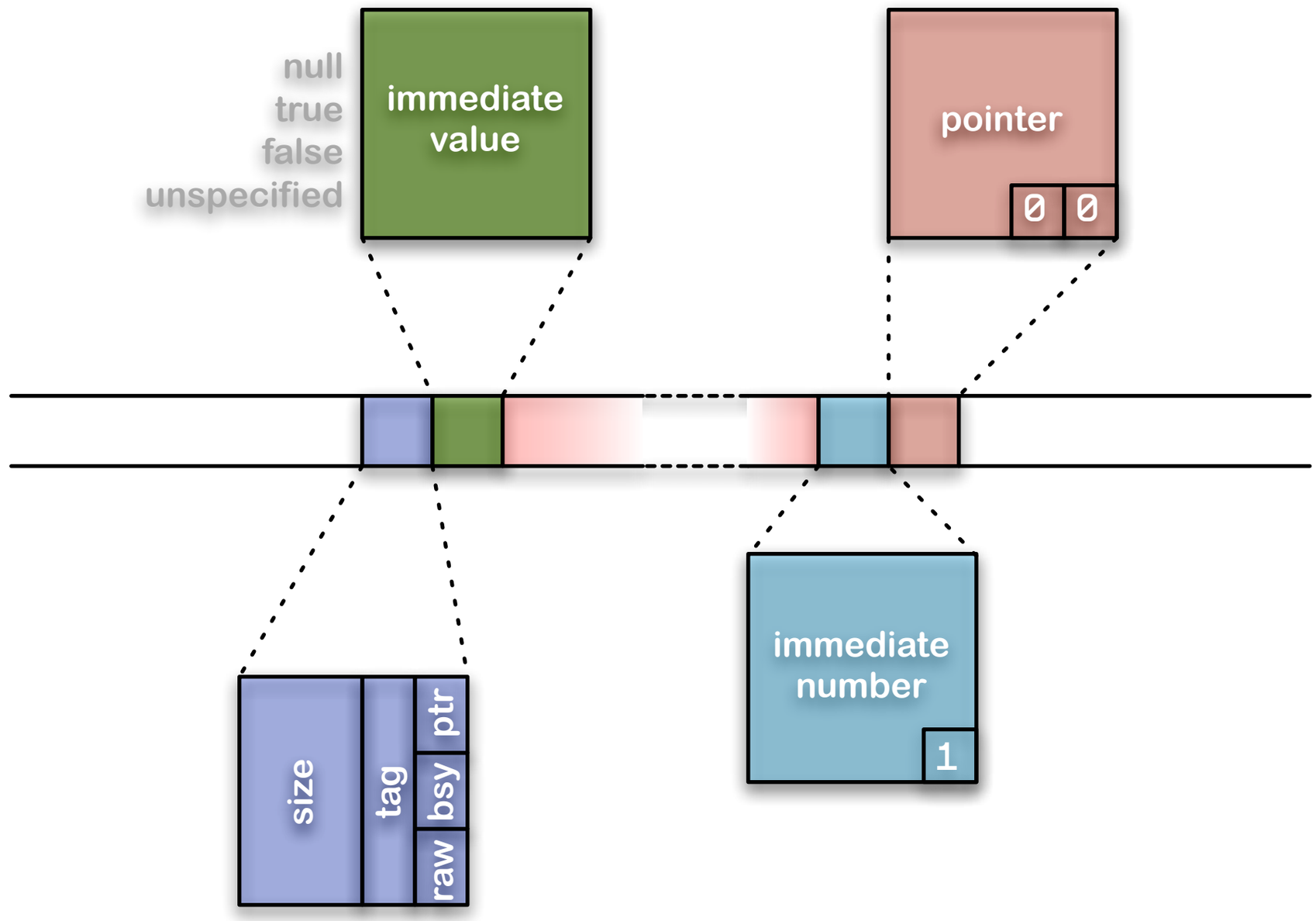
Theo D'Hondt
Software Languages Lab

Simple Memory Architecture (recap)

version 8



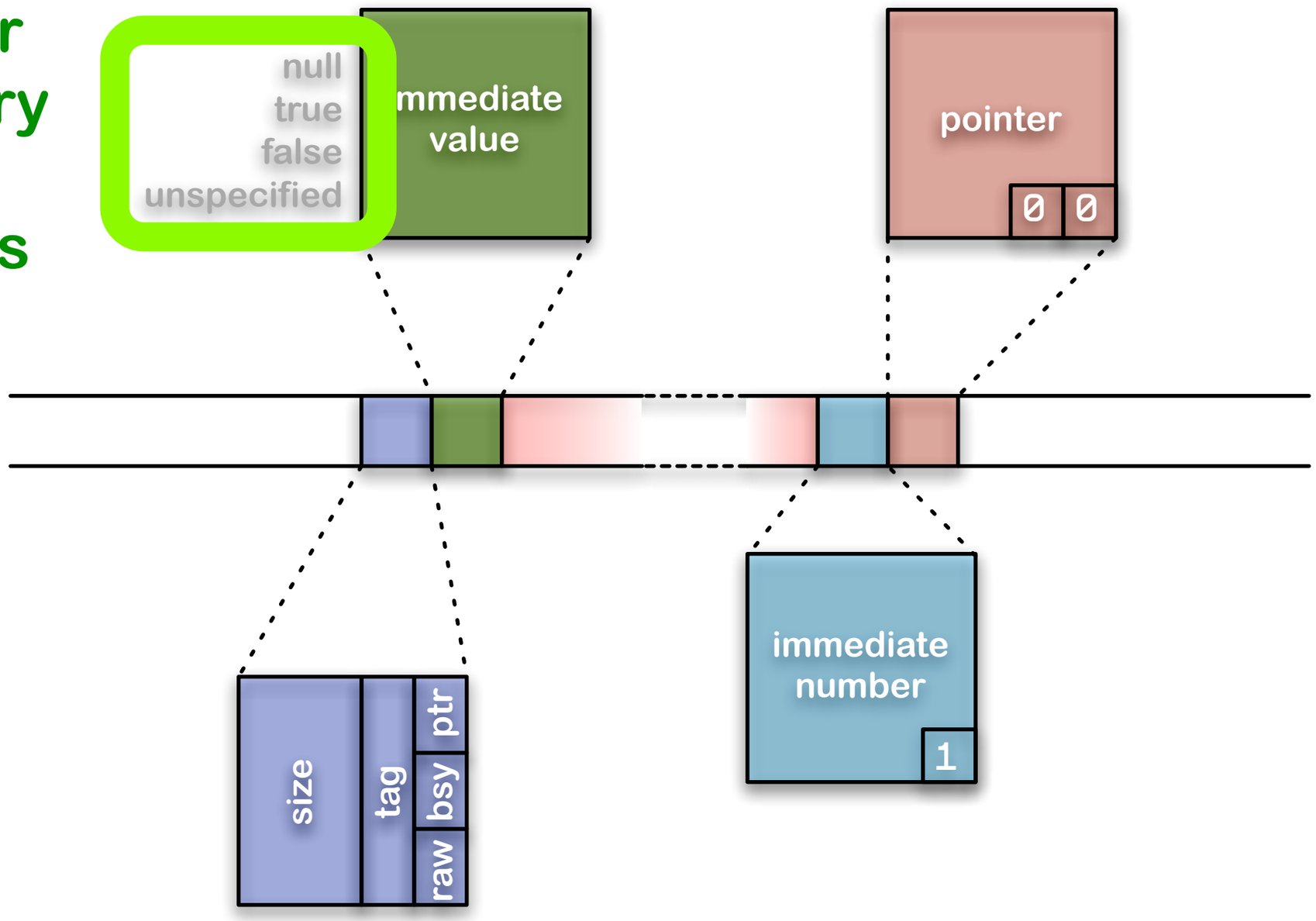
Simple Memory Architecture (recap)



version 9

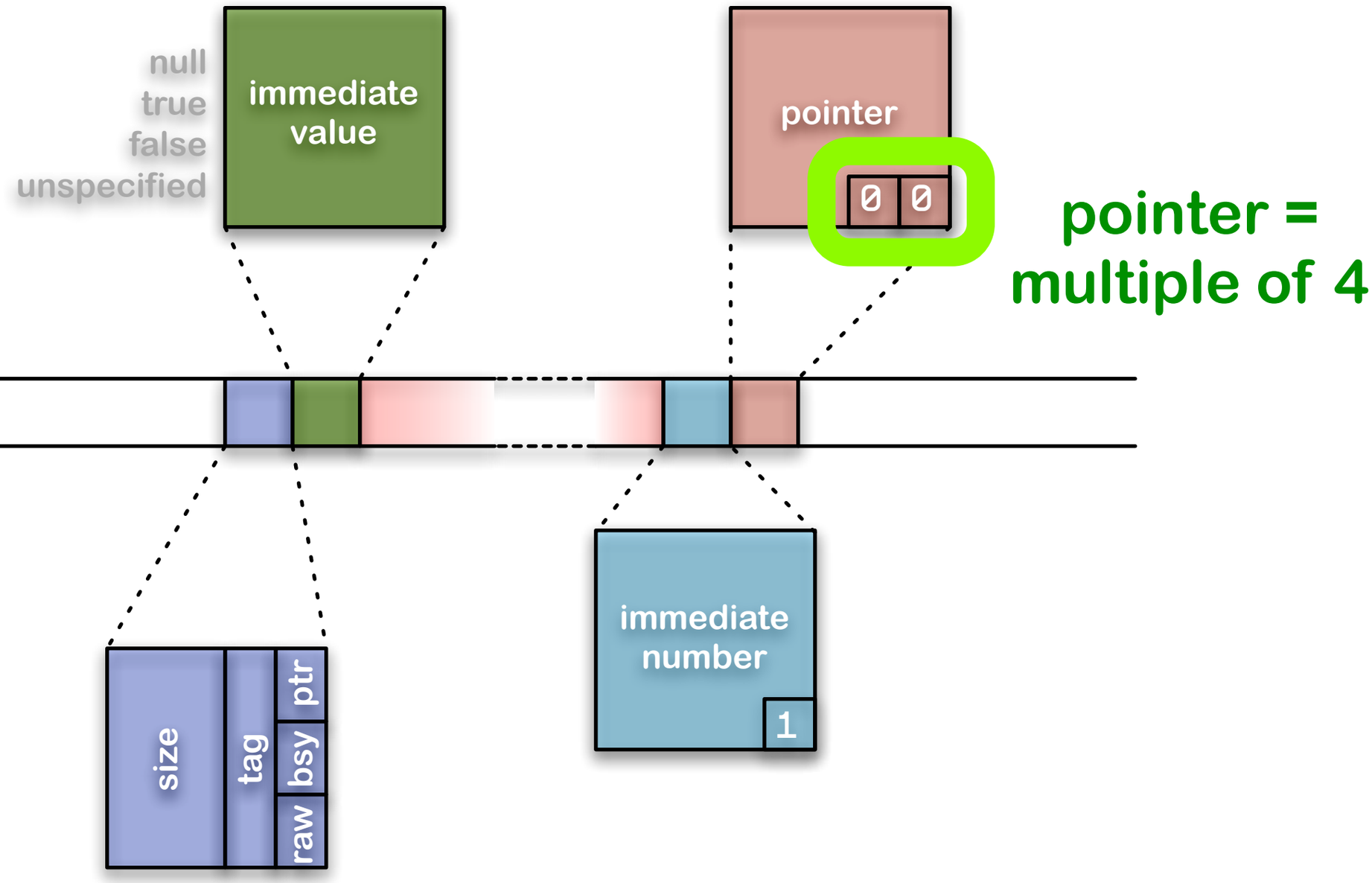
Simple Memory Architecture (recap)

values
< lower
boundary
for
pointers



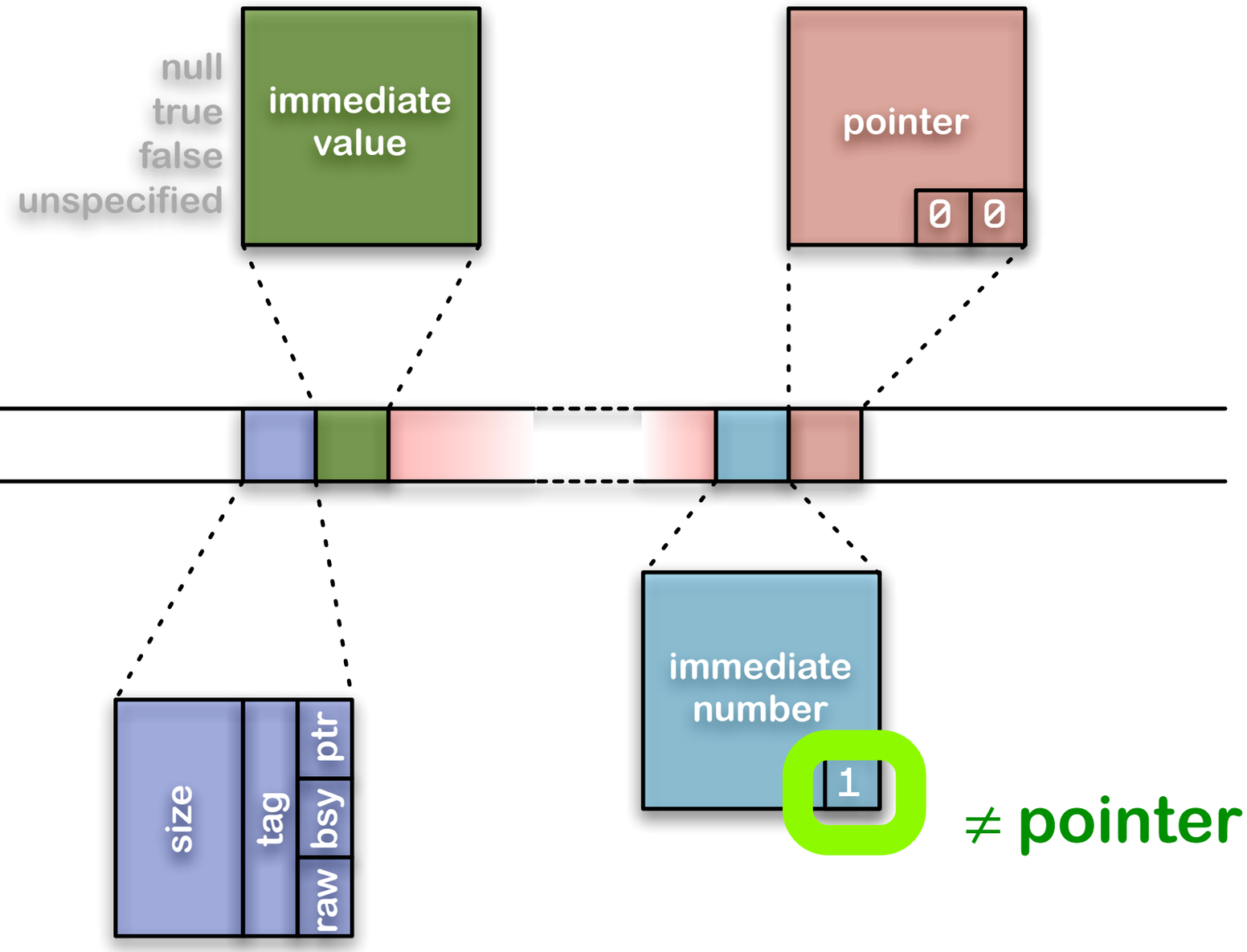
version 9

Simple Memory Architecture (recap)



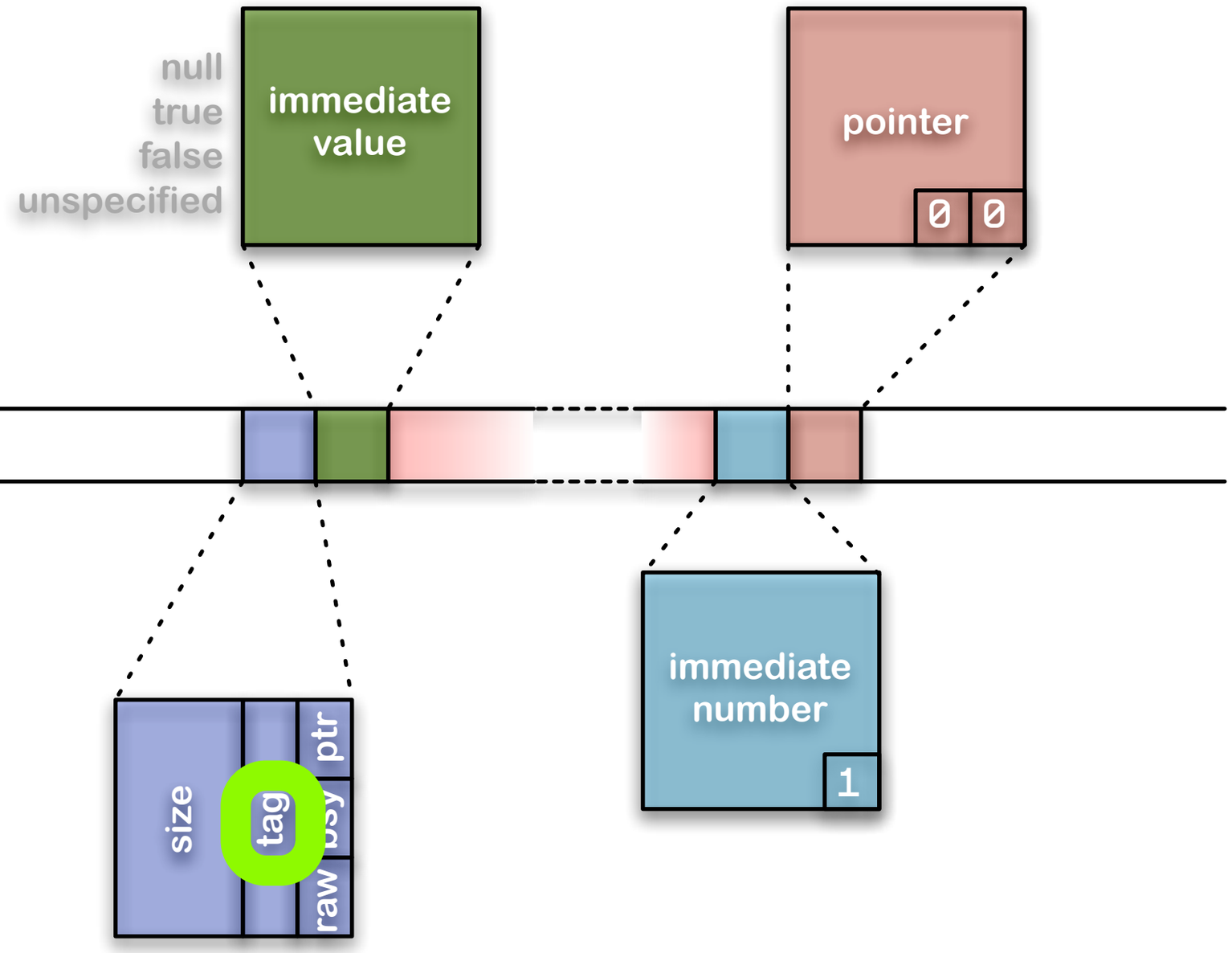
version 9

Simple Memory Architecture (recap)



version 9

Simple Memory Architecture (recap)

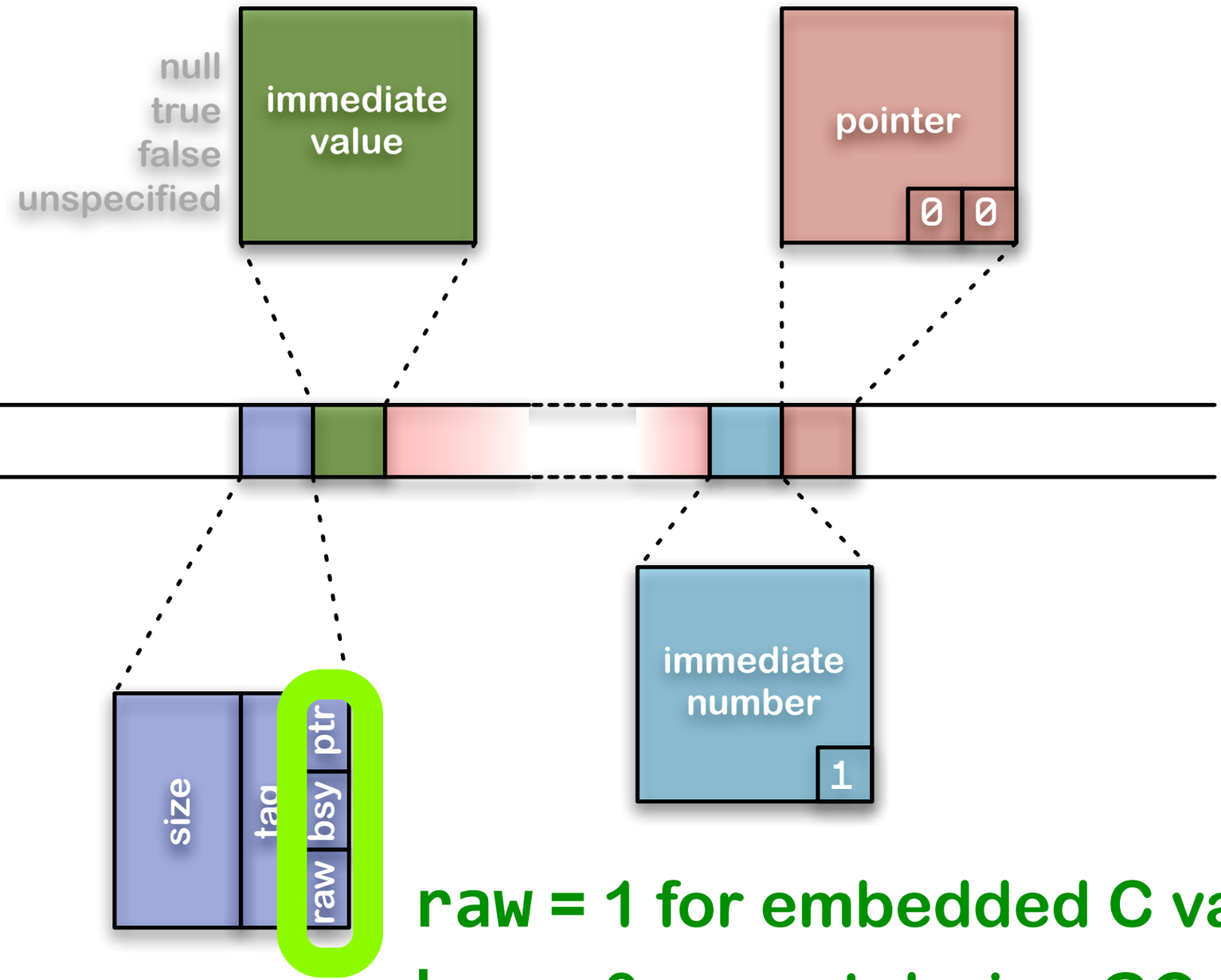


version 9

all chunks are tagged

Simple Memory Architecture (recap)

version 9



`raw` = 1 for embedded C values,

`bsy` = 0 except during GC,

`ptr` = 0 always

Memory Manager interface

```
UNS_type      Memory_Available(NIL_type);
BLN_type         Memory_Claim(UNS_type);
PTR_type       Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type      Memory_Get_Size(PTR_type);
BYT_type       Memory_Get_Tag(PTR_type);
NIL_type    Memory_Initialize(ADR_type,
                              UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type     Memory_Make_Chunk(BYT_type,
                              UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

```
typedef             void * ADR_type;
typedef  unsigned long   CEL_type;
typedef unsigned short   BYT_type;
typedef             char   CHR_type;
typedef           double
typedef        long int
typedef             void
typedef    signed long
typedef             char
typedef  unsigned long   UNS_type;

typedef enum { Memory_False = 0,
               Memory_True  = 1 } BLN_type;

typedef union PTR { CEL_type     cel;
                    union PTR * ptr; } * PTR_type;
```

```
enum { Memory_Cell_Bias           = 0x00000004,
       Memory_Immediate_Maximum = 0x3FFFFFFF,
       Memory_Void_Value        = 0x00000000 };

static const UNS_type Memory_Cell_Size = sizeof(CEL_type);
```

Memory Manager interface

```
UNS_type      Memory_Available(NIL_type);
BLN_type         Memory_Claim(UNS_type);
PTR_type       Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type      Memory_Get_Size(PTR_type);
BYT_type       Memory_Get_Tag(PTR_type);
NIL_type    Memory_Initialize(ADR_type,
                              UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type    Memory_Make_Chunk(BYT_type,
                              UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

```
typedef           void * ADR_type;
typedef  unsigned long   CEL_type;
typedef unsigned short   BYT_type;
typedef           char   CHR_type;
typedef         double   FLT_type;
typedef      long int   INT_type;
typedef           void   NIL_type;
typedef    signed long   SNG_type;
typedef           char   STR_type;
typedef  unsigned long   UNS_type;

typedef enum { Memory_False = 0,
               Memory_True  = 1 } BLN_type;

typedef union PTR { CEL_type    cel;
                    union PTR * ptr; } * PTR_type;
```

```
enum { Memory_Cell_Bias          = 0x00000004,
       Memory_Immediate_Maximum = 0x3FFFFFFF,
       Memory_Void_Value         = 0x00000000 };

static const UNS_type Memory_Cell_Size = sizeof(CEL_type);
```

Memory Manager interface

```
UNS_type      Memory_Available(NIL_type);
BLN_type         Memory_Claim(UNS_type);
PTR_type       Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type      Memory_Get_Size(PTR_type);
BYT_type       Memory_Get_Tag(PTR_type);
NIL_type    Memory_Initialize(ADR_type,
                              UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type    Memory_Make_Chunk(BYT_type,
                              UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

```
typedef           void * ADR_type;
typedef  unsigned long   CEL_type;
typedef unsigned short   BYT_type;
typedef           char   CHR_type;
typedef         double   DBL_type;
typedef      long int    INT_type;
typedef           void   NIL_type;
typedef    signed long   SNG_type;
typedef           char   STR_type;
typedef  unsigned long   UNS_type;

typedef enum { Memory_False = 0,
               Memory_True  = 1 } BLN_type;

typedef union PTR { CEL_type    cel;
                    union PTR * ptr; } * PTR_type;
```

```
enum { Memory_Cell_Bias          = 0x00000004,
       Memory_Immediate_Maximum = 0x3FFFFFFF,
       Memory_Void_Value         = 0x00000000 };

static const UNS_type Memory_Cell_Size = sizeof(CEL_type);
```

Memory Manager interface

```
UNS_type      Memory_Available(NIL_type);
BLN_type        Memory_Claim(UNS_type);
PTR_type      Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type     Memory_Get_Size(PTR_type);
BYT_type     Memory_Get_Tag(PTR_type);
NIL_type    Memory_Initialize(ADR_type,
                              UNS_type);
BLN_type  Memory_Is_Immediate(PTR_type);
PTR_type    Memory_Make_Chunk(BYT_type,
                              UNS_type);
                Make_Immediate(LNG_type);
```

```
typedef             void * ADR_type;
typedef  unsigned long   CEL_type;
typedef unsigned short   BYT_type;
typedef             char   CHR_type;
typedef           double   FLO_type;
typedef        long int   LNG_type;
typedef             void   NIL_type;
typedef    signed long   SGN_type;
typedef             char * TXT_type;
typedef  unsigned long   UNS_type;

typedef enum { Memory_False = 0,
               Memory_True  = 1 } BLN_type;

typedef union PTR { CEL_type     cel;
                    union PTR * ptr; } * PTR_type;
```

```
    = 0x00000004,
num = 0x3FFFFFFF,
    = 0x00000000 };

_Cell_Size = sizeof(CEL_type);
```

Memory Manager interface (cont'd)

```
UNS_type       Memory_Available(NIL_type);
BLN_type           Memory_Claim(UNS_type);
PTR_type         Memory_Collect(PTR_type);
LNG_type   Memory_Get_Immediate(PTR_type);
UNS_type        Memory_Get_Size(PTR_type);
BYT_type         Memory_Get_Tag(PTR_type);
NIL_type      Memory_Initialize(ADR_type,
                                UNS_type);
BLN_type    Memory_Is_Immediate(PTR_type);
PTR_type      Memory_Make_Chunk(BYT_type,
                                UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

Memory Manager interface (cont'd)

```
UNS_type       Memory_Available(NIL_type);
BLN_type        Memory_Claim(UNS_type);
PTR_type      Memory_Collect(PTR_type);
LNG_type   Memory_Get_Immediate(PTR_type);
UNS_type     Memory_Get_Size(PTR_type);
BYT_type      Memory_Get_Tag(PTR_type);
NIL_type    Memory_Initialize(ADR_type,
                              UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type    Memory_Make_Chunk(BYT_type,
                              UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

to be called when memory is short

Memory Manager interface (cont'd)

```
UNS_type      Memory_Available(NIL_type);
BLN_type          Memory_Claim(UNS_type);
PTR_type        Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type       Memory_Get_Size(PTR_type);
BYT_type        Memory_Get_Tag(PTR_type);
NIL_type     Memory_Initialize(ADR_type,
                               UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type     Memory_Make_Chunk(BYT_type,
                               UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

memory allocation using tag and size

Memory Manager interface (cont'd)

```
UNS_type      Memory_Available(NIL_type);
BLN_type          Memory_Claim(UNS_type);
PTR_type        Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type       Memory_Get_Size(PTR_type);
BYT_type        Memory_Get_Tag(PTR_type);
NIL_type     Memory_Initialize(ADR_type,
                               UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type     Memory_Make_Chunk(BYT_type,
                               UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

make a shortinteger using a C number

Memory Manager interface (cont'd)

```
UNS_type      Memory_Available(NIL_type);
BLN_type          Memory_Claim(UNS_type);
PTR_type        Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type       Memory_Get_Size(PTR_type);
BYT_type       Memory_Get_Tag(PTR_type);
NIL_type     Memory_Initialize(ADR_type,
                               UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type       Memory_Make_Chunk(BYT_type,
                               UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

various getters

Memory Manager interface (cont'd)

```
UNS_type      Memory_Available(NIL_type);
BLN_type          Memory_Claim(UNS_type);
PTR_type        Memory_Collect(PTR_type);
LNG_type  Memory_Get_Immediate(PTR_type);
UNS_type       Memory_Get_Size(PTR_type);
BYT_type        Memory_Get_Tag(PTR_type);
NIL_type     Memory_Initialize(ADR_type,
                               UNS_type);
BLN_type   Memory_Is_Immediate(PTR_type);
PTR_type     Memory_Make_Chunk(BYT_type,
                               UNS_type);
PTR_type Memory_Make_Immediate(LNG_type);
```

support for memory claims

Using implicit recursion

```
static VEC_type read_vector(NIL_type)
  { VEC_type vector;
    next_token();
    if (Token == RBR_token)
      vector = Main_Empty_Vector;
    else
      vector = build_vector(1);
    return next_token_and_return(vector); }
```

A GC needs access to the recursion stack!

example: reading a vector

start recursion
continue recursion
start backtrack
continue backtrack

```
static VEC_type build_vector(UNS_type Count)
  { EXP_type expression;
    VEC_type vector;
    expression = read_expression();
    if (Token == RPR_token)
      vector = make_VEC(Count);
    else
      vector = build_vector(Count + 1);
    vector[Count] = expression;
    return vector; }
```

Using explicit recursion

the recursion stack is accessible!

```
static VEC_type read_vector(NIL_type)
  { EXP_type expression;
    UNS_type count;
    VEC_type vector;
    next_token();
    if (Token == RBR_token)
      vector = Main_Empty_Vector;
    else
      { for (count = 0;
             Token != RBR_token;
             count++)
        { expression = read_expression();
          Stack_Push(expression); }
      Main_Claim(count);
      for (vector = make_VEC(count);
           count > 0;
           count--)
        vector[count] = Stack_Pop(); }
  return next_token_and_return(vector); }
```

save

claim

allocate

retrieve

example: reading a vector

Injecting a Garbage Collector (cont'd)

```
static EXP_type evaluate_inline(EXP_type Expression,
                                UNS_type Frame_size)
  { bOD_type body_thread;
    VEC_type environment,
             frame;
    environment = Environment_Grow_Environment();
    frame = make_VEC(Frame_size);
    body_thread = (bOD_type)Thread_Poke(Continue_body,
                                        bOD_size);
    body_thread->env = Environment_Get_Environment();
    body_thread->frm = Environment_Get_Frame();
    Environment_Set_Environment_And_Frame(environment,
                                          frame);
    return evaluate_expression(Expression); }
```

example: inline evaluation of consequent /alternative in if

Injecting a Garbage Collector (cont'd)

local variables are used as cache for expression values

```
static EXP_type evaluate_inline(EXP_type Expression,
                                UNS_type Frame_size)
  { bOD_type body_thread;
    VEC_type environment,
             frame;
    environment = Environment_Grow_Environment();
    frame = make_VEC(Frame_size);
    body_thread = (bOD_type)Thread_Poke(Continue_body,
                                        bOD_size);
    body_thread->env = Environment_Get_Environment();
    body_thread->frm = Environment_Get_Frame();
    Environment_Set_Environment_And_Frame(environment,
                                          frame);
    return evaluate_expression(Expression); }
```

example: inline evaluation of consequent /alternative in if

Injecting a Garbage Collector (cont'd)

they are used without discrimination inside the body of the function

```
static EXP_type evaluate_inline(EXP_type Expression,
                                UNS_type Frame_size)
  { bOD_type body_thread;
    VEC_type environment,
             frame;
    environment = Environment_Grow_Environment();
    frame = make_VEC(Frame_size);
    body_thread = (bOD_type)Thread_Poke(Continue_body,
                                        bOD_size);
    body_thread->env = Environment_Get_Environment();
    body_thread->frm = Environment_Get_Frame();
    Environment_Set_Environment_And_Frame(environment,
                                          frame);
    return evaluate_expression(Expression); }
```

example: inline evaluation of consequent /alternative in if

Injecting a Garbage Collector (cont'd)

which interferes with GC

```
static EXP_type evaluate_inline(EXP_type Expression,
                                UNS_type Frame_size)
  { bOD_type body_thread;
    VEC_type environment,
             frame;
    environment = Environment_Grow_Environment();
    frame = make_VEC(Frame_size);
    body_thread = (bOD_type)Thread_Poke(Continue_body,
                                        bOD_size);
    body_thread->env = Environment_Get_Environment();
    body_thread->frm = Environment_Get_Frame();
    Environment_Set_Environment_And_Frame(environment,
                                          frame);
    return evaluate_expression(Expression); }
```

example: inline evaluation of consequent /alternative in if

Claiming Memory

```
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; }
    Main_Survival_Claim((REF_type)&expression,
                        Main_Default_Margin);
    return evaluate_expression(expression); }
```

example: evaluating a sequence

```
e Ignore)
ek();
expressions = sequence_thread->exs;
position    = sequence_thread->pos;
index = get_NBR(position) + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { Main_Survival_Claim((REF_type)&sequence_thread,
                        Main_Default_Margin);
    sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

Claiming Memory

consumes memory

```
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; }
    Main_Survival_Claim((REF_type)&expression,
                        Main_Default_Margin);
    return evaluate_expression(expression); }
```

example: evaluating a sequence

```
e Ignore)
ek();
  expressions = sequence_thread->exs;
  position    = sequence_thread->pos;
  index = get_NBR(position) + 1;
  expression = expressions[index];
  size_x = size_VEC(expressions);
  if (index < size_x)
    { Main_Survival_Claim((REF_type)&sequence_thread,
                          Main_Default_Margin);
      sequence_thread = (sEQ_type)Thread_Keep();
      position = make_NBR(index);
      sequence_thread->pos = position; }
  else
    Thread_Zap();
  return evaluate_expression(expression); }
```

Claiming Memory

```
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; }
    Main_Survival_Cla
                                  Main_Default_Margin);
    return evaluate_ex
```

example: evaluating a sequence

```
e Ignore)
ek();
expressions = sequence_thread->exs;
position    = sequence_thread->p
index = get_NBR(position) + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { Main_Survival_Claim((REF_typ
                         Main_Def
    sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

<default> determined by static analysis

Claiming Memory

```
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; }
    Main_Survival_Claim(REF_type)&expression,
                        Main_Default_Margin);
    return evaluate_expression(expression); }
```

example: evaluating a sequence

claim memory with guaranteed survivor

Claiming Memory

```
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; }
    Main_Survival_Claim((REF_type)&expression,
                        Main_Default_Margin);
    return evaluate_expression(expression); }
```

At least <default> cells available when called

example: evaluating a sequence

Claiming Memory

```
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 =

        sequence_thread->
        sequence_thread->
    Main_Survival_Claim((
                        M

    return evaluate_expre
```

example: evaluating a sequence

```
static EXP_type continue_sequence(EXP_type Ignore)
  { 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 = get_NBR(position) + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { Main_Survival_Claim((REF_type)&sequence_thread,
                        Main_Default_Margin);
    sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

Claiming Memory

```
static EXP_type evaluate_sequence(VEC_type Expressions)
  { sEQ_type sequence_thread;
    EXP_type expression;
    UNS_type size_x;
    xpressions);
    sions[1];
    =
    ->
    ->
    Main_Survival_Claim((
                        M
    return evaluate_expre
```

No guaranteed memory available when called

example: evaluating a sequence

```
static EXP_type continue_sequence(EXP_type Ignore)
  { 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 = get_NBR(position) + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { Main_Survival_Claim((REF_type)&sequence_thread,
                        Main_Default_Margin);
    sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

Claiming Memory

```
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 =

        sequence_thread->
        sequence_thread->
    Main_Survival_Claim((
                        M
    return evaluate_expre
```

```
static EXP_type continue_sequence(EXP_type Ignore)
  { 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 = get_NBR(position) + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { Main_Survival_Claim((REF_type)&sequence_thread,
                        Main_D
    sequence_thread = (sEQ_typ )Thread_Keep();
    position = make_NBR(index)
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

might require memory

example: evaluating a sequence

Claiming Memory

```
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 =

        sequence_thread->
        sequence_thread->
    Main_Survival_Claim((
                        M

    return evaluate_expre
```

example: evaluating a sequence

```
static EXP_type continue_sequence(EXP_type Ignore)
  { 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 = get_NBR(position) + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
    Main_Survival_Claim( REF_type)&sequence_thread,
                         ain_Default_Margin);
    sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

claim memory

Claiming Memory

```
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 =

        sequence_thread->
        sequence_thread->
    Main_Survival_Claim((
                        M

    return evaluate_expre
```

inherits claim

example: evaluating a sequence

```
static EXP_type continue_sequence(EXP_type Ignore)
  { 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 = get_NBR(position) + 1;
expression = expressions[index];
size_x = size_VEC(expressions);
if (index < size_x)
  { Main_Survival_Claim((REF_type)&sequence_thread,
                        Main_Default_Margin);
    sequence_thread = (sEQ_type)Thread_Keep();
    position = make_NBR(index);
    sequence_thread->pos = position; }
else
  Thread_Zap();
return evaluate_expression(expression); }
```

Claiming Memory (cont'd)

```
static EXP_type make_vector_native(VEC_type Vector)
  { EXP_type value;
    LNG_type raw_number;
    NBR_type size;
    UNS_type index,
             raw_size;
    VEC_type vector;
    raw_size = size_VEC(Vector);

    if (raw_number == 0)
      vector = Main_Empty_Vector;
    else
      { Main_Survival_Claim((REF_type)&value,
                            raw_number);
        vector = make_VEC(raw_number);
        for (index = 1;
             index <= raw_number;
             index++)
          vector[index] = value; }
    return vector; }
```

example:
make-vector

Claiming Memory (cont'd)

```
static EXP_type make_vector_native(VEC_type Vector)
  { EXP_type value;
    LNG_type raw_number;
    NBR_type size;
    UNS_type index,
             raw_size;
    VEC_type vector;
    raw_size = size_VEC(Vector);
```

At least <default> cells available when called

```
  if (raw_number == 0)
    vector = Main_Empty_Vector;
  else
    { Main_Survival_Claim((REF_type)&value,
                          raw_number);
      vector = make_VEC(raw_number);
      for (index = 1;
           index <= raw_number;
           index++)
        vector[index] = value; }
  return vector; }
```

example:
make-vector

Claiming Memory (cont'd)

```
static EXP_type make_vector_native(VEC_type Vector)
  { EXP_type value;
    LNG_type raw_number;
    NBR_type size;
    UNS_type index,
             raw_size;
    VEC_type vector;
    raw_size = size_VEC(Vector);

    if (raw_number == 0)
      vector = Main_Empty_Vector;
    else
      { Main_Survival_Claim((REF_type)&value,
                            raw_number);
        vector = make_VEC(raw_number);
        for (index = 1;
             index <= raw_number;
             index++)
          vector[index] = value; }
    return vector; }
```

allocate dynamic number of cells

example: make-vector

Claiming Memory (cont'd)

```
static EXP_type make_vector_native(VEC_type Vector)
  { EXP_type value;
    LNG_type raw_number;
    NBR_type size;
    UNS_type index,
             raw_size;
    VEC_type vector;
    raw_size = size_VEC(Vector);

    if (raw_number == 0)
      vector = Main_Empty_Vector;
    else
      { Main_Survival_Claim((REF_type)&value,
                            raw_number);
        vector = make_VEC(raw_number);
        for (index = 1;
             index <= raw_number;
             index++)
          vector[index] = value; }
    return vector; }
```

claim dynamic number of cells

example: make-vector

Claiming Memory (cont'd)

```
static EXP_type make_vector_native(VEC_type Vector)
  { EXP_type value;
    LNG_type raw_number;
    NBR_type size;
    UNS_type index,
             raw_size;
    VEC_type vector;
    raw_size = size_VEC(Vector);
```

value survives possible GC

```
    if (raw_number == 0)
      vector = Main_Empty_Vector;
    else
      { Main_Survival_Claim((REF_type)&value,
                            raw_number);
        vector = make_VEC(raw_number);
        for (index = 1;
             index <= raw_number;
             index++)
          vector[index] = value; }
    return vector; }
```

example: make-vector

Managing a root reference

```
static UNS_type Root_counter;

static REF_type Force_collect[Root_size];
```

```
enum { Root_size = 12 };
```

```
typedef EXP_type * REF_type;
```

```
NIL_type Main_Register(REF_type Reference)
  { if (Root_counter == Root_size)
      Main_Fatal_Error(TMR_error_string);
    Force_collect[Root_counter++] = Reference; }
```

Managing a root reference

```
static UNS_type Root_counter;

static REF_type Force_collect[Root_size];
```

make an inventory of references to locations
of values that need to survive a GC

```
enum { Root_size = 12 };
```

```
typedef EXP_type * REF_type;
```

```
NIL_type Main_Register(REF_type Reference)
  { if (Root_counter == Root_size)
      Main_Fatal_Error(TMR_error_string);
    Force_collect[Root_counter++] = Reference; }
```

Managing a root reference

```
static UNS_type Root_counter;

static REF_type Force_collect[Root_size];
```

```
enum { Root_size = 12 };
```

```
typedef EXP_type * REF_type;
```

size is statically assigned

```
NIL_type Main_Register(REF_type Reference)
  { if (Root_counter == Root_size)
      Main_Fatal_Error(TMR_error_string);
    Force_collect[Root_counter++] = Reference; }
```

Managing a root reference

```
static UNS_type Root_counter;

static REF_type Force_collect[Root_size];
```

```
enum { Root_size = 12 };
```

```
typedef EXP_type * REF_type;
```

all expression types are allowed

```
NIL_type Main_Register(REF_type Reference)
  { if (Root_counter == Root_size)
      Main_Fatal_Error(TMR_error_string);
    Force_collect[Root_counter++] = Reference; }
```

Managing a root reference

```
static UNS_type Root_counter;

static REF_type Force_collect[Root_size];
```

```
enum { Root_size = 12 };
```

```
typedef EXP_type * REF_type;
```

```
NIL_type Main_Register(REF_type Reference)
  { if (Root_counter == Root_size)
      Main_Fatal_Error(TMR_error_string);
    Force_collect[Root_counter++] = Reference; }
```

each module is responsible for
registering its GC-sensitive references

Managing a root reference (cont'd)

```
static VEC_type Root;
```

```
Root = make_VEC(Root_size);
```

```
    before_collect();
    Root = (VEC_type)Memory_Collect((PTR_type)Root);
    after = Memory_Available();
    if (after < Margin)
      Main_Fatal_Error(IMM_error_string);
    after_collect();
```

Managing a root reference (cont'd)

```
static VEC_type Root;
```

Root holds a vector of all registered values

```
Root = make_VEC(Root_size);
```

```
before_collect();
Root = (VEC_type)Memory_Collect((PTR_type)Root);
after = Memory_Available();
if (after < Margin)
  Main_Fatal_Error(IMM_error_string);
after_collect();
```

Managing a root reference (cont'd)

```
static VEC_type Root;
```

```
Root = make_VEC(Root_size);
```

```
before_collect();
Root = (VEC_type)Memory_Collect((PTR_type)Root);
after = Memory_Available();
if (after < Margin)
  Main_Fatal_Error(IMM_error_string);
after_collect();
```

this is the actual GC step

Managing a root reference (cont'd)

```
before_collect();
Root = (VEC_type)Memory_Collect((PTR_type)Root);
after = Memory_Available();
if (after < Margin)
  Main_Fatal_Error(IMM_error_string);
after_collect();
```

Managing a root reference (cont'd)

```
before_collect();
Root = (VEC_type)Memory_Collect((PTR_type)Root);
after = Memory_Available();
if (after < Margin)
  Main_Fatal_Error(IMM_error_string);
after_collect();
```

```
static NIL_type before_collect(NIL_type)
  { REF_type reference;
    UNS_type index;
    for (index = 0;
         index < Root_counter;
         index++)
      { reference = Force_collect[index];
        Root[index + 1] = *reference; }}
```

update Root vector from registered values

Managing a root reference (cont'd)

```
before_collect();
Root = (VEC_type)Memory_Collect((PTR_type)Root);
after = Memory_Available();
if (after < Margin)
  Main_Fatal_Error(IMM_error_string);
after_collect();
```

perform GC and update Root

Managing a root reference (cont'd)

```
before_collect();
Root = (VEC_type)Memory_Collect((PTR_type)Root);
after = Memory_Available();
if (after < Margin)
  Main_Fatal_Error(IMM_error_string);
after_collect();
```

verify availability

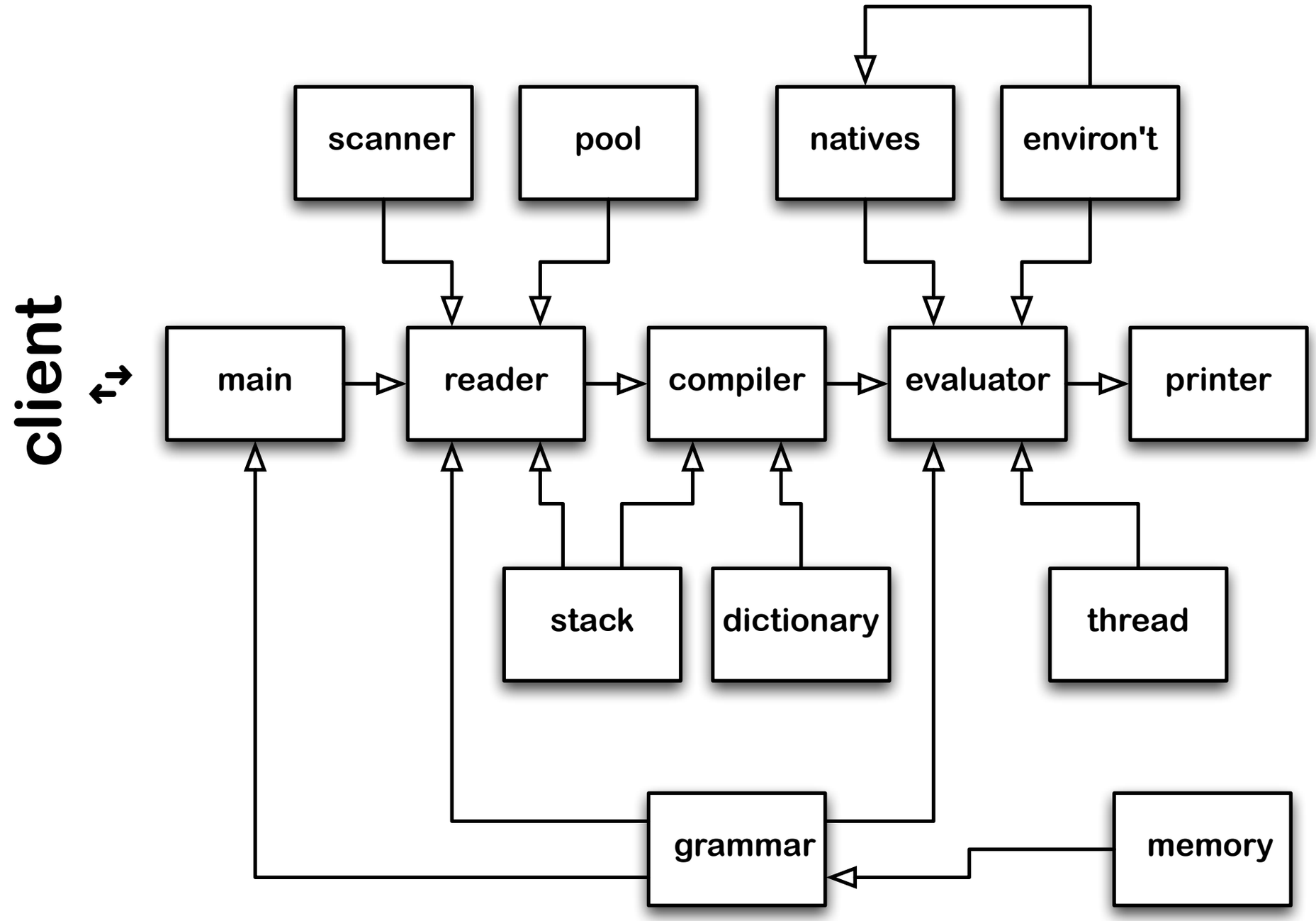
Managing a root reference (cont'd)

```
before_collect();
Root = (VEC_type)Memory_Collect((PTR_type)Root);
after = Memory_Available();
if (after < Margin)
   Main_Fatal_Error(IMM_error_string);
after_collect();
```

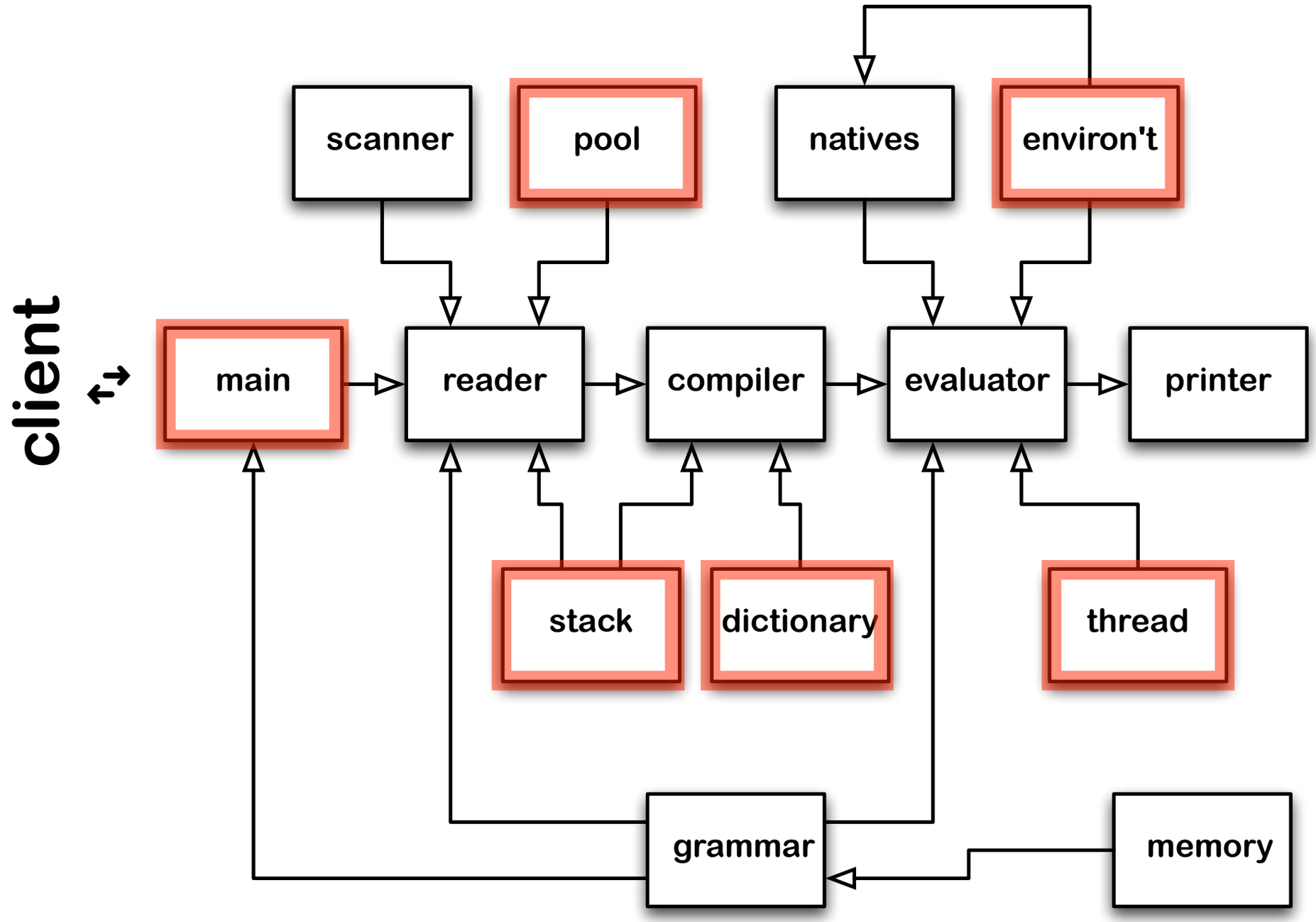
```
static NIL_type after_collect(NIL_type)
  { EXP_type expression;
    REF_type reference;
    UNS_type index;
    for (index = 0;
         index < Root_counter;
         index++)
      { reference = Force_collect[index];
        expression = Root[index + 1];
        *reference = expression; }}
```

update registered values from Root vector

Managing a root reference (cont'd)



Managing a root reference (cont'd)



affected modules

A fully functional Slip interpreter

Slip – Debugger Console

```
[Session started at 2010-05-03 13:13:09 +0200.]
GNU gdb 6.3.50-20050815 (Apple version gdb-1346) (Fri Sep 18 20:40:51 UTC 2009)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin".tty /dev/ttys000
Loading program into debugger…
sharedlibrary apply-load-rules all
Program loaded.
run
[Switching to process 20505]
Running…
cpSlip/c version 9
>>>(begin
  (define (QuickSort V Low High)
    (define Left Low)
    (define Right High)
    (define Pivot (vector-ref V (quotient (+ Left Right) 2)))
    (define Save 0)
    (while (< Left Right)
      (while (< (vector-ref V Left) Pivot)
        (set! Left (+ Left 1)))
      (while (> (vector-ref V Right) Pivot)
        (set! Right (- Right 1)))
      (if (<= Left Right)
        (begin
          (set! Save (vector-ref V Left))
          (vector-set! V Left (vector-ref V Right))
          (vector-set! V Right Save)
          (set! Left (+ Left 1))
          (set! Right (- Right 1)))))
    (if (< Low Right)
      (QuickSort V Low Right))
    (if (> High Left)
      (QuickSort V Left High)))
  (define V (make-vector 1000 0))
  (define Low 0)
  (define High (- (vector-length V) 1))
  (define x 0)
  (define y 1)
  (while (<= x High)
    (vector-set! V x y)
    (set! x (+ x 1))
    (set! y (remainder (+ y 4253171) 1235711)))
  (QuickSort V Low High)
  (vector-ref V High))
   ---
   ---before = 27 after = 247659 total = 250000 time = 0.000214
   ---
   ---
   ---before = 29 after = 247565 total = 250000 time = 0.000214
   ---
   ---
   ---before = 31 after = 247391 total = 250000 time = 0.000221
   ---
   ---
   ---before = 29 after = 247608 total = 250000 time = 0.000200
   ---
   ---
   ---before = 25 after = 247335 total = 250000 time = 0.000220
   ---
   ---
   ---before = 28 after = 247445 total = 250000 time = 0.000208
   ---
1233085
>>>
```

GDB: Running… Succeeded

Memory claims are cross-cutting

allocation claim