Appendix C

Supporting code for garbage collection

Contents

This appendix shows supporting code that can help with the Exercises in Chapter 4: visiting functions, scanning procedures, root-tracking code for the evaluator, and the implementation of the root stack.

C.1 Object-visiting procedures for mark-and-sweep collection

Section 4.4.2 presents a few procedures for visiting μScheme objects in a depth-first search. Here we present the remaining procedures.

To visit an expression, we visit its literal value, if any, and of course its subexpressions.

622 ⟨*ms.c* 622⟩≡ 623a▷

```
static void visitexp(Exp e) {
    switch (e->alt) {
    case LITERAL:
        visitvaluechildren(e->u.literal);
        return;
    case VAR:
        return;
    case IFX:
        visitexp(e->u.ifx.cond);
        visitexp(e->u.ifx.true);
        visitexp(e->u.ifx.false);
        return;
    case WHILEX:
        visitexp(e->u.whilex.cond);
        visitexp(e->u.whilex.body);
        return;
    case BEGIN:
        visitexplist(e->u.begin);
        return;
    case SET:
        visitexp(e->u.set.exp);
        return;
    case LETX:
        visitexplist(e->u.letx.el);
        visitexp(e->u.letx.body);
        return;
    case LAMBDAX:
        visitexp(e->u.lambdax.body);
        return;
    case APPLY:
        visitexp(e->u.apply.fn);
        visitexplist(e->u.apply.actuals);
        return;
    default:
        assert(0);
    }
}
```

visitexplist 624b

To visit a definition, we visit any expressions it contains.

623a ⟨*ms.c* 622⟩+≡ ◁622 623b▷

```
static void visitdef(Def d) {
    if (d == NULL)
        return;
    else
        switch (d->alt) {
        case VAL:
            visitexp(d->u.val.exp);
            return;
        case EXP:
            visitexp(d->u.exp);
            return;
        case DEFINE:
            visitexp(d->u.define.lambda.body);
            return;
        case USE:
            return;
        default:
            assert(0);
        }
}
```

To visit a root, we call the appropriate visiting procedure.

623b ⟨*ms.c* 622⟩+≡ ◁623a 624a▷

```
static void visitroot(Root r) {
    switch (r.alt) {
    case STACKVALUEROOT:
        visitstackvalue(*r.u.stackvalueroot);
        return;
    case HEAPLOCROOT:
        visitheaploc(*r.u.heaplocroot);
        return;
    case ENVROOT:
        visitenv(*r.u.envroot);
        return;
    case EXPROOT:
        visitexp(*r.u.exproot);
        return;
    case VALUELISTROOT:
        visitvaluelist(*r.u.valuelistroot);
        return;
    case DEFROOT:
        visitdef(*r.u.defroot);
        return;
    default:
        assert(0);
    }
}
```

visitvaluelist 624a

Here are procedures for visiting various lists.

624a ⟨*ms.c* 622⟩+≡ ◁623b 624b▷

```
static void visitvaluelist(Valuelist vl) {
    for (; vl; vl = vl->tl)
        visitvaluechildren(vl->hd);
}
```

624b ⟨*ms.c* 622⟩+≡ ◁624a

```
static void visitexplist(Explist el) {
    for (; el; el = el->tl)
        visitexp(el->hd);
}
```

C.2 Root-scanning procedures for copying collection

Section 4.5.3 presents a few procedures for scanning potential roots. We present the remaining procedures here. As explained in Section 4.5.3, these scanning procedures are hybrids. Like standard scanning procedures, they forward internal pointers to objects allocated on the μScheme heap, but because some potential roots are allocated on the C heap, these procedures use graph traversal to visit those. Almost all the forwarding is done by `scanvalue`, which is shown in chunk 196a. The procedures shown here do almost all graph traversal. They are therefore very similar to the visiting procedures in the previous section.

Scanning expressions means scanning internal values or subexpressions.

625 ⟨*copy.c* 625⟩≡ 626a▷

```
static void scanexp(Exp e) {
    switch (e->alt) {
    case LITERAL:
        scanvalue(&e->u.literal);
        break;
    case VAR:
        break;
    case IFX:
        scanexp(e->u.ifx.cond);
        scanexp(e->u.ifx.true);
        scanexp(e->u.ifx.false);
        break;
    case WHILEX:
        scanexp(e->u.whilex.cond);
        scanexp(e->u.whilex.body);
        break;
    case BEGIN:
        scanexplist(e->u.begin);
        break;
    case SET:
        scanexp(e->u.set.exp);
        break;
    case LETX:
        scanexplist(e->u.letx.el);
        scanexp(e->u.letx.body);
        break;
    case LAMBDAX:
        scanexp(e->u.lambdax.body);
        break;
    case APPLY:
        scanexp(e->u.apply.fn);
        scanexplist(e->u.apply.actuals);
        break;
    default:
        assert(0);
    }
}
```

scanexplist 627a

Scanning definitions means scanning their expressions.

626a ⟨*copy.c* 625⟩+≡ ◁625 626b▷

```
static void scandef(Def d) {
    if (d != NULL)
        switch (d->alt) {
        case EXP:
            scanexp(d->u.exp);
            break;
        case DEFINE:
            scanexp(d->u.define.lambda.body);
            break;
        case VAL:
            scanexp(d->u.val.exp);
            break;
        case USE:
            break;
        default:
            assert(0);
        }
}
```

Scanning roots means calling the appropriate scanning procedures, with one exception; a HEAPLOCROOT, which is a pointer directly to an object on the heap, has to be forwarded.

626b ⟨*copy.c* 625⟩+≡ ◁626a 626c▷

```
static void scanroot(Root r) {
    switch (r.alt) {
    case STACKVALUEROOT:
        scanvalue(r.u.stackvalueroot);
        return;
    case HEAPLOCROOT:
        *r.u.heaplocroot = forward(*r.u.heaplocroot);
        return;
    case ENVROOT:
        scanenv(*r.u.envroot);
        return;
    case EXPROOT:
        scanexp(*r.u.exproot);
        return;
    case VALUELISTROOT:
        scanvaluelist(*r.u.valuelistroot);
        return;
    case DEFROOT:
        scandef(*r.u.defroot);
        return;
    default:
        assert(0);
    }
}
```

Scanning lists is straightforward.

626c ⟨*copy.c* 625⟩+≡ ◁626b 627a▷

```
static void scanvaluelist(Valuelist vl) {
    for (; vl; vl = vl->tl)
        scanvalue(&vl->hd);
}
```

627a ⟨*copy.c* 625⟩+≡ ◁626c
```
static void scanexplist(Explist el) {
    for (; el; el = el->tl)
        scanexp(el->hd);
}
```

C.3 Other supporting code

To control the size of the heap, we might want to use the μScheme variable &gamma-desired, as described in Exercises 5 and 15. This routine gets the value of that variable.

627b ⟨*loc.c* 627b⟩≡ 627e▷
```
int gammadesired(int defaultval, int minimum) {
    Value *gloc;

    assert(rootstacksize > 0 && rootstack[0].alt == ENVROOT);
    gloc = find(strtoname("&gamma-desired"), *rootstack[0].u.envroot);
    if (gloc && gloc->alt == NUM)
        return gloc->u.num > minimum ? gloc->u.num : minimum;
    else
        return defaultval;
}
```

627c ⟨*function prototypes for μScheme* 627c⟩≡ (125c) 627d▷
```
int gammadesired(int defaultval, int minimum);
```

To debug a collector, it can help to wrap values in calls to validate. Calling validate(v) returns v, unless v is invalid, in which case it causes an assertion failure.

627d ⟨*function prototypes for μScheme* 627c⟩+≡ (125c) ◁627c
```
Value validate(Value);
```

627e ⟨*loc.c* 627b⟩+≡ ◁627b
```
Value validate(Value v) {
    assert(v.alt != INVALID);
    return v;
}
```

While debugging the code in this Appendix, we used the following trick.

627f ⟨*untested example uses of* validate 627f⟩≡
```
static Value veval(Exp *e, Env *env);

Value eval(Exp *e, Env *env) {
    return validate(veval(e, env));
}

static Value veval(Exp *e, Env *env) {
    ⟨former implementation of eval (not shown)⟩
}
```

find	126b
rootstack	182b
rootstacksize	182b
strtoname	28d

C.4 Adding root tracking to the μScheme interpreter

The most difficult part of implementing the μScheme collectors is ensuring that all calls to `allocloc` satisfy the precondition that every pointer to a live, heap-allocated object is reachable from the root stack. The burden of establishing this precondition is distributed not only among those functions that call `allocloc` directly but also among functions that call other functions that might call `allocloc`, and so on. If calling a function f might result into a call to `allocloc`, we call f a "function that might allocate." In the μScheme interpreter, the functions that might allocate are `allocate`, `binary`, `bindalloc`, `bindalloclist`, `eval`, `evaldef`, `evallist`, `main`, `parse`, `parseexp`, `parselet`, `parseletvar`, `parselist`, `parsesx`, `primenv`, `readdef`, `readevalprint`, and `use`.

We use the following idea to ensure the precondition on `allocloc`: *every function that might allocate must ensure that its private potential roots are on the root stack, but potential roots that are shared with the calling function are the responsibility of the caller.* More precisely,

- If a function receives a pointer to a potential root of category A or B, it is up to the caller to ensure that this pointer is reachable from the root stack.
- If a function receives a potential root of category A, with no indirection, it is up to the function itself to ensure that this potential root is on the root stack. The reason is that C passes arguments by value, so for example, if we pass a `Value` to a function, that function gets a private copy, and to be sure that internal pointers are updated when objects on the heap move, that private copy must go on the root stack.

We call these requirements the *tracing invariant*; callees can assume the invariant, and callers must establish it before each call to a function that might allocate. It is not necessary to establish the tracing invariant before calling a function that doesn't allocate, and the tracing invariant is irrelevant for the implementation of such functions.

To make the μScheme interpreter respect the tracing invariant, we modify the evaluator `eval.c`, the environment-manipulation routines `env.c`, the primitives `prim.c`, and the parser `parse.c`, all of which contain functions that might allocate. The top-level interpreter in `scheme.c` has been carefully crafted not to require modification.

To highlight differences from Chapter 3, we show new or changed code in *`italic typewriter`* font.

C.4.1 The revised μScheme evaluator

The structure is as before.

628 ⟨*eval.c* 628⟩≡ 629a▷

```
#include "all.h"

⟨eval.c declarations 631d⟩
```

We don't need to worry about `e` or `env` here: the tracing invariant guarantees that `eval`'s arguments are already reachable from the roots.

629a ⟨*eval.c* 628⟩+≡ ◁628 632a▷
```
Value eval(Exp e, Env env) {
    switch (e->alt) {
    case LITERAL:
        ⟨evaluate e->u.literal and return 629b⟩
    case VAR:
        ⟨evaluate e->u.var and return 629c⟩
    case SET:
        ⟨evaluate e->u.set and return 630a⟩
    case IFX:
        ⟨evaluate e->u.if and return 633d⟩
    case WHILEX:
        ⟨evaluate e->u.while and return 634a⟩
    case BEGIN:
        ⟨evaluate e->u.begin and return 634b⟩
    case LETX:
        ⟨evaluate let expression and return 632b⟩
    case LAMBDAX:
        ⟨evaluate e->u.lambdax and return 630b⟩
    case APPLY:
        ⟨evaluate e->u.apply and return 631a⟩
    }
    assert(0);
    return falsev;
}
```

Values and variables

The LITERAL and VAR rules don't allocate, so there's no need to worry about roots.

629b ⟨*evaluate* `e->u.literal` *and return* 629b⟩≡ (629a)
```
return e->u.literal;
```

629c ⟨*evaluate* `e->u.var` *and return* 629c⟩≡ (629a)
```
if (find(e->u.var, env) == NULL)
    error("variable %n not found", e->u.var);
return *find(e->u.var, env);
```

The Assign rule does allocate because it calls `eval` to evaluate the right-hand side. Because C does not specify the order of evaluation, we cannot implement `set` using the simple assignment we use in chunk 129d. The risk is that the C compiler might call `find(e->u.set.name, env)` first. If this happens, the value returned is a pointer to an object allocated on the heap. But calling `eval` might trigger a garbage collection, objects on the heap would move, and the assignment would write over the wrong location. We avoid such a disaster by calling `eval` *first*, saving the result in `v`, and then calling `find`. We needn't push `v` on the root stack because `find` doesn't allocate.

630a ⟨*evaluate* `e->u.set` *and return* 630a⟩≡ (629a)

```
{
    Value v;

    if (find(e->u.set.name, env) == NULL)
        error("set unbound variable %n", e->u.set.name);
    v = eval(e->u.set.exp, env);
    return *find(e->u.set.name, env) = v;
}
```

Closures and application

Making a closure doesn't allocate, so there's no work to be done.

630b ⟨*evaluate* `e->u.lambdax` *and return* 630b⟩≡ (629a)

```
if (duplicatename(e->u.lambdax.formals) != NULL)
    error("formal parameter %n appears twice in lambda",
          duplicatename(e->u.lambdax.formals));
return mkClosure(e->u.lambdax, env);
```

duplicatename	35d
env	629a
error	35b
eval	629a
find	126b
mkClosure	$\mathcal{A}$

Both primitive and user-defined functions can allocate, so before we can apply a function, we have to get the function value and the actual parameters on the root stack. To keep pushes and pops together, we save the result of application in the "return value" rv. This technique makes it easy to pop the root stack before returning the answer.

631a ⟨*evaluate* e->u.apply *and return* 631a⟩≡ (629a)
```
{
    Value rv;
    Value f = eval(e->u.apply.fn, env);
    Valuelist vl;

    pushroot(mkStackvalueroot(&f));
    vl = evallist(e->u.apply.actuals, env);
    pushroot(mkValuelistroot(&vl));

    switch (f.alt) {
    case PRIMITIVE:
        ⟨apply f.u.primitive, storing the result in rv 631b⟩
        break;
    case CLOSURE:
        ⟨apply f.u.closure, storing the result in rv 631c⟩
        break;
    default:
        error("%e evaluates to non-function %v in %e", e->u.apply.fn, f, e);
    }
    poproot(mkValuelistroot(&vl));
    poproot(mkStackvalueroot(&f));
    return rv;
}
```

We apply primitives exactly as in Chapter 3, except the result goes into rv instead of being returned directly.

631b ⟨*apply* f.u.primitive, *storing the result in* rv 631b⟩≡ (631a)
```
rv = f.u.primitive.function(e, f.u.primitive.tag, vl);
```

The tracing invariant requires that env be on the root stack before the call to eval.

631c ⟨*apply* f.u.closure, *storing the result in* rv 631c⟩≡ (631a)
```
{
    Env env;
    Namelist nl = f.u.closure.lambda.formals;
    checkargc(e, lengthNL(nl), lengthVL(vl));

    env = bindalloclist(nl, vl, f.u.closure.env);
    pushroot(mkEnvroot(&env));
    rv = eval(f.u.closure.lambda.body, env);
    poproot(mkEnvroot(&env));
}
```

bindalloclist	126c
checkargc	35c
env	629a
error	35b
eval	629a
evallist	632a
lengthNL	$\mathcal{A}$
lengthVL	$\mathcal{A}$
mkEnvroot	$\mathcal{A}$
mkStackvalueroot	$\mathcal{A}$
mkValuelistroot	$\mathcal{A}$
poproot	181a
pushroot	181a

Evallist

In evallist, we push v before calling evallist recursively.

631d ⟨*eval.c declarations* 631d⟩≡ (628 128c)
```
static Valuelist evallist(Explist el, Env env);
```

632a ⟨*eval.c* 628⟩+≡ ◁629a 634c▷

```
static Valuelist evallist(Explist el, Env env) {
    if (el == NULL) {
        return NULL;
    } else {
        Value v = eval(el->hd, env);      /* enforce uScheme's order of evaluation */
        Valuelist rv;
        pushroot(mkStackvalueroot(&v));
        rv = mkVL(v, evallist(el->tl, env));
        poproot(mkStackvalueroot(&v));
        return rv;
    }
}
```

Let, let*, and letrec

The implementations of all let expressions require assigning to `env`, then evaluating the body with the new `env`. By pushing `&env` on the root stack, we ensure that all the modified versions are visible on the root stack.

632b ⟨*evaluate let expression and return* 632b⟩≡ (629a)

```
{
    Value rv;

    pushroot(mkEnvroot(&env));
    switch (e->u.letx.let) {
    case LET:
        if (duplicatename(e->u.letx.nl) != NULL)
            error("bound name %n appears twice in let", duplicatename(e->u.letx.nl));
        ⟨extend env by simultaneously binding el to nl 633a⟩
        break;
    case LETSTAR:
        ⟨extend env by sequentially binding el to nl 633b⟩
        break;
    case LETREC:
        if (duplicatename(e->u.letx.nl) != NULL)
            error("bound name %n appears twice in letrec", duplicatename(e->u.letx.nl));
        ⟨extend env by recursively binding el to nl 633c⟩
        break;
    default:
        assert(0);
    }

    rv = eval(e->u.letx.body, env);
    poproot(mkEnvroot(&env));
    return rv;
}
```

duplicatename	35d
env	629a
error	35b
eval	629a
mkEnvroot	$\mathcal{A}$
mkStackvalueroot	$\mathcal{A}$
mkVL	$\mathcal{A}$
poproot	181a
pushroot	181a

Because bindalloclist allocates, the tracing invariant requires that vl be on the root stack.

633a ⟨*extend* env *by simultaneously binding* el *to* nl 633a⟩≡ (632b 131b)
```
{
    Valuelist vl = evallist(e->u.letx.el, env);
    pushroot(mkValuelistroot(&vl));
    env = bindalloclist(e->u.letx.nl, vl, env);
    poproot(mkValuelistroot(&vl));
}
```

In the implementation of let*, you might think that the result of eval(el->hd, env) would have to be on the root stack. But the copy in the caller is *dead*[1] as soon as it is passed to bindalloc, before any allocation can take place. When potential roots of category A are passed by value, it's the callee's responsibility to ensure that they are on the root stack.

633b ⟨*extend* env *by sequentially binding* el *to* nl 633b⟩≡ (632b 131b)
```
{
    Namelist nl;
    Explist el;

    for (nl = e->u.letx.nl, el = e->u.letx.el; nl && el; nl = nl->tl, el = el->tl)
        env = bindalloc(nl->hd, eval(el->hd, env), env);
    assert(nl == NULL && el == NULL);
}
```

To implement letrec, we have the same risk we had for set; in the second loop, we must force a particular order of evaluation for evallist and find. As before, by calling evallist first, we avoid having to manipulate the root stack, because find doesn't allocate and v is dead by the time we call eval again.

633c ⟨*extend* env *by recursively binding* el *to* nl 633c⟩≡ (632b 131b)
```
{
    Namelist nl;
    Valuelist vl;

    for (nl = e->u.letx.nl; nl; nl = nl->tl)
        env = bindalloc(nl->hd, mkNil(), env);

    vl = evallist(e->u.letx.el, env);
    for (nl = e->u.letx.nl;
         nl && vl;
         nl = nl->tl, vl = vl->tl)
        *find(nl->hd, env) = vl->hd;
}
```

bindalloc	126c
bindalloclist	126c
env	629a
eval	629a
evallist	632a
find	126b
istrue	127d
mkNil	$\mathcal{A}$
mkValuelistroot	$\mathcal{A}$
poproot	181a
pushroot	181a

Control flow

For the most part, control flow can ignore the root stack, since there are no intermediate values.

633d ⟨*evaluate* e->u.if *and return* 633d⟩≡ (629a)
```
if (istrue(eval(e->u.ifx.cond, env)))
    return eval(e->u.ifx.true, env);
else
    return eval(e->u.ifx.false, env);
```

[1] A variable or temporary is *dead* if its value can't possibly affect any future computation.

634a ⟨*evaluate* e->u.while *and return* 634a⟩≡ (629a)

```
while (istrue(eval(e->u.whilex.cond, env)))
    eval(e->u.whilex.body, env);
return falsev;
```

Begin repeatedly assigns to v, but v is not a root—at the time we call eval, v is dead.

634b ⟨*evaluate* e->u.begin *and return* 634b⟩≡ (629a)

```
{
    Explist el;
    Value v = falsev;
    for (el = e->u.begin; el; el = el->tl)
        v = eval(el->hd, env);
    return v;
}
```

Definitions

Most of the evaluation of definitions is as in Chapter 3.

634c ⟨*eval.c* 628⟩+≡ ◁632a 636a▷

```
Env evaldef(Def d, Env env, int echo) {
    switch (d->alt) {
    case VAL:
        ⟨evaluate val binding and return new environment 635a⟩
    case DEFINE:
        ⟨evaluate function definition and return new environment 635b⟩
    case EXP:
        ⟨evaluate expression, store the result in it, and return new environment 634d⟩
    case USE:
        ⟨read in a file and return new environment 635c⟩
    default:
        assert(0);
        return NULL;
    }
}
```

bindalloc	126c
env	629a
eval	629a
falsev	127b
find	126b
istrue	127d
mkStackvalueroot	$\mathcal{A}$
poproot	181a
print	33f
pushroot	181a
strtoname	28d

To evaluate a top-level expression, we don't need to do anything special to maintain the tracing invariant.

⟨*evaluate expression, store the result in* it, *and return new environment* 634d⟩≡ (634c 133e)

```
{
    Value v = eval(d->u.exp, env);
    Value *itloc = find(strtoname("it"), env);
    if (echo)
        print("%v\n", v);
    if (itloc == NULL) {
        pushroot(mkStackvalueroot(&v));
        env = bindalloc(strtoname("it"), v, env);
        poproot(mkStackvalueroot(&v));
        return env;
    } else {
        *itloc = v;
        return env;
    }
}
```

Because `val` changes our private copy of `env`, the tracing invariant requires that before we call `eval`, we must put `env` on the root stack.

635a ⟨*evaluate* `val` *binding and return new environment* 635a⟩≡ (634c 133e)

```
{
    Value v;

    if (find(d->u.val.name, env) == NULL)
        env = bindalloc(d->u.val.name, mkNil(), env);
    pushroot(mkEnvroot(&env));
    v = eval(d->u.val.exp, env);
    poproot(mkEnvroot(&env));
    *find(d->u.val.name, env) = v;
    if (echo) {
        if (d->u.val.exp->alt == LAMBDAX)
            print("%n\n", d->u.val.name);
        else
            print("%v\n", v);
    }
    return env;
}
```

The definition we get from `mkVal` need not be a root because its components are reachable from `d`, and according to the tracing invariant, `d` is guaranteed to be reachable.

635b ⟨*evaluate function definition and return new environment* 635b⟩≡ (634c 133e)

```
if (duplicatename(d->u.define.lambda.formals) != NULL)
    error("formal parameter %n appears twice in definition of function %n",
          duplicatename(d->u.define.lambda.formals), d->u.define.name);
return evaldef(mkVal(d->u.define.name, mkLambdax(d->u.define.lambda)), env, echo);
```

Like `val`, `use` modifies our private copy of `env`.

635c ⟨*read in a file and return new environment* 635c⟩≡ (634c 133e)

```
{
    FILE *fin;
    const char *filename;

    filename = nametostr(d->u.use);
    fin = fopen(filename, "r");
    if (fin == NULL)
        error("cannot open file \"%s\"", filename);
    pushroot(mkEnvroot(&env));
    readevalprint(defreader(filereader(filename, fin), 0), &env, 1);
    poproot(mkEnvroot(&env));
    fclose(fin);
    return env;
}
```

bindalloc	126c
defreader	31f
duplicatename	35d
echo	634c
env	634c
error	35b
evaldef	634c
fclose	$\mathcal{B}$
filereader	31c
find	126b
fopen	$\mathcal{B}$
mkEnvroot	$\mathcal{A}$
mkLambdax	$\mathcal{A}$
mkNil	$\mathcal{A}$
mkVal	$\mathcal{A}$
nametostr	28d
poproot	181a
print	33f
pushroot	181a
readevalprint	636a

The tracing invariant requires that `readevalprint` put `d` on the root stack, but it guarantees that `envp` is already on the root stack. Because we push `&d`, not `d`, we needn't pop and push at every assignment.

636a ⟨*eval.c* 628⟩+≡ ◁634c

```
void readevalprint(Defreader reader, Env *envp, int echo)
{
    Def d;

    pushroot(mkDefroot(&d));
    d = NULL; /* don't allow garbage on stack during first call to [[readdef]] */
    while ((d = readdef(reader)))
        *envp = evaldef(d, *envp, echo);
    poproot(mkDefroot(&d));
}
```

The code above has `&d` on the root stack even during the call to `readdef`, when `d` is not live. If `readdef` triggers a garbage collection, garbage in `t` (from quoted lists) won't be reclaimed until the next collection cycle.

C.4.2 Revised code for primitives

When we create an environment to hold primitives, we have to keep that environment on the root stack.

636b ⟨*prim.c* 636b⟩≡

```
Env primenv(void) {
    Env env = NULL;
    pushroot(mkEnvroot(&env));
    #define xx(NAME, TAG, FUNCTION) \
        env = bindalloc(strtoname(NAME), mkPrimitive(TAG, FUNCTION), env);
    #include "prim.h"
    #undef xx
    poproot(mkEnvroot(&env));
    return env;
}
```

mkDefroot 𝒜
mkEnvroot 𝒜
poproot 181a
pushroot 181a
readdef 31e

The only primitive that allocates is cons. There are three roots.

- Values v and w must be roots in case the copying collector needs to update their internal pointers.
- The location allocated to hold v needs to be a root during the allocation of a fresh location to hold w.

637a ⟨*return cons cell containing* v *and* w 637a⟩≡ (138b)

```
{
    Value *l;
    Value rv;

    pushroot(mkStackvalueroot(&v));
    pushroot(mkStackvalueroot(&w));
    l = allocate(v);
    pushroot(mkHeaplocroot(&l));
    rv = mkPair(l, allocate(w));
    poproot(mkHeaplocroot(&l));
    poproot(mkStackvalueroot(&w));
    poproot(mkStackvalueroot(&v));
    return rv;
}
```

C.4.3 Revised environment-manipulation routines

Most environment code is unchanged, but we need a new version of bindalloclist.

This version of bindalloclist needs to push its env because it modifies it. Pointers returned by earlier calls to bindalloc must be reachable from the root stack at subsequent calls to bindalloc.

637b ⟨*env.c* 637b⟩≡ 637c▷

```
Env bindalloclist(Namelist nl, Valuelist vl, Env env) {
    pushroot(mkEnvroot(&env));
    for (; nl && vl; nl = nl->tl, vl = vl->tl)
        env = bindalloc(nl->hd, vl->hd, env);
    poproot(mkEnvroot(&env));
    return env;
}
```

For bindalloc, val is a parameter passed by value, so we have a fresh copy of it. It contains Value* pointers, so you might think it needs to be on the root stack for the copying collector (so that the pointers can be updated if necessary). But by the time we get to allocate, our copy of val is dead—only allocate's private copy matters. It is therefore safe to reuse the implementation from Chapter 3.

allocate	126d
bindalloc	604c
env	604c
malloc	$\mathcal{B}$
mkEnvroot	$\mathcal{A}$
mkHeaplocroot	$\mathcal{A}$
mkPair	$\mathcal{A}$
mkStackvalueroot	$\mathcal{A}$
name	604c
newenv	604c
poproot	181a
pushroot	181a
val	604c

637c ⟨*env.c* 637b⟩+≡ ◁637b

```
Env bindalloc(Name name, Value val, Env env) {
    Env newenv = malloc(sizeof(*newenv));
    assert(newenv != NULL);

    newenv->name = name;
    newenv->loc  = allocate(val);
    newenv->tl   = env;
    return newenv;
}
```

C.4.4 The revised parser

Since `parse` can call `parsesx` (among other things), which allocates, we need to make sure all our intermediate `Exp*`s and `Explist*`s are on the root stack when we call another parse routine. Most of the parser does not change; we show only the changed parts.

In `parselist`, we push `e` before calling `parselist` recursively, just as in `evallist`.

638 ⟨*parse.c* 638⟩≡

```
Explist parselist(Parlist pl) {
    Exp e;
    Explist el;

    if (pl == NULL)
        return NULL;

    e = parseexp(pl->hd);
    pushroot(mkExproot(&e));
    el = mkEL(e, parselist(pl->tl));
    poproot(mkExproot(&e));
    return el;
}
```

mkEL	$\mathcal{A}$
mkExproot	$\mathcal{A}$
parseexp	616c
poproot	181a
pushroot	181a

In parseexp, we have just one small change: If we call parselist to parse arguments, we put the resulting Explist el on the root stack.

639 ⟨*parseexp* LIST *and return* 639⟩≡ (616c)

```
{
    Parlist pl;              /* parenthesized list we are parsing */
    Name first;              /* first element, as a name (or NULL if not name) */
    Explist el;              /* remaining elements, as expressions */
    Exp rv;                  /* result of parsing */

    pl = p->u.list;
    if (pl == NULL)
        error("%p: empty list in input", p);

    first = pl->hd->alt == ATOM ? pl->hd->u.atom : NULL;
    if (first == strtoname("lambda")) {
        ⟨parseexp lambda and put the result in rv 617b⟩
    } else if (first == strtoname("let")
           ||  first == strtoname("let*")
           ||  first == strtoname("letrec")) {
        ⟨parseexp let and put the result in rv 640a⟩
    } else if (first == strtoname("quote")) {
        ⟨parseexp quote and put the result in rv 618d⟩
    } else {
        el = parselist(pl->tl);
        pushroot(mkExplistroot(&el));
        if (first == strtoname("begin")) {
            ⟨parseexp begin and put the result in rv 619d⟩
        } else if (first == strtoname("if")) {
            ⟨parseexp if and put the result in rv 619e⟩
        } else if (first == strtoname("set")) {
            ⟨parseexp set and put the result in rv 620a⟩
        } else if (first == strtoname("while")) {
            ⟨parseexp while and put the result in rv 619f⟩
        } else {
           ⟨parseexp application and put the result in rv 619c⟩
        }
        poproot(mkExplistroot(&el));
    }
    return rv;
}
```

error	35b
mkExplistroot	$\mathcal{A}$
poproot	181a
pushroot	181a
strtoname	28d

When we parse a let expression, we parse the body first. The whole let expression goes on the root stack while we parse the bindings.

640a ⟨*parseexp let and put the result in* rv 640a⟩≡ (639 617a)
```
Letkeyword letword;
Par letbindings;

if (first == strtoname("let"))
    letword = LET;
else if (first == strtoname("let*"))
    letword = LETSTAR;
else if (first == strtoname("letrec"))
    letword = LETREC;
else
    assert(0);

if (lengthPL(pl->tl) != 2)
    error("%p: usage: (%n (letlist) exp)", p, first);

letbindings = nthPL(pl->tl, 0);
if (letbindings->alt != LIST)
    error("%p: usage: (%n (letlist) exp)", p, first);

rv = mkLetx(letword, NULL, NULL, parseexp(nthPL(pl->tl, 1)));

pushroot(mkExproot(&rv));
parseletbindings(p, letbindings->u.list, rv);
poproot(mkExproot(&rv));
```

allocate	126d
error	35b
first	639
lengthPL	$\mathcal{A}$
mkExproot	$\mathcal{A}$
mkHeaplocroot	$\mathcal{A}$
mkLetx	$\mathcal{A}$
mkList	$\mathcal{A}$
mkNil	$\mathcal{A}$
mkPair	$\mathcal{A}$
mkStackvalueroot	$\mathcal{A}$
nthPL	$\mathcal{A}$
parseexp	616c
parseletbindings	618c
parsesx	619a
pl	639
poproot	181a
pushroot	181a
rv	639
strtoname	28d

To implement quoted S-expressions, we have to maintain the root stack much as we did to implement cons.

⟨*parsesx* LIST *and return* 640b⟩≡ (619a)
```
if (p->u.list == NULL)
    return mkNil();
else {
    Value v, w, *l;

    v = parsesx(p->u.list->hd);
    pushroot(mkStackvalueroot(&v));
    w = parsesx(mkList(p->u.list->tl));
    pushroot(mkStackvalueroot(&w));
    l = allocate(v);
    pushroot(mkHeaplocroot(&l));
    v = mkPair(l, allocate(w));
    poproot(mkHeaplocroot(&l));
    poproot(mkStackvalueroot(&w));
    poproot(mkStackvalueroot(&v));
    return v;
}
```

C.5 Implementing the root stack

The internal stack is, as mandated by the interface, a pointer to an array of Roots. We grow it when necessary.

641a ⟨*root.c* 641a⟩≡ 641b▷
```
#include "all.h"

Root *rootstack;              /* points to base of the root stack */
int rootstacksize;            /* number of items on the stack now */
static int rootstackmax;      /* max number of items the stack can hold */
```

Private variable `rootstackmax` contains the size of the `rootstack` array (the maximum number of roots it can hold), while `rootstacksize` contains the number of roots actually on the stack.

We grow the root stack slowly, by increments of 16.

641b ⟨*root.c* 641a⟩+≡ ◁641a 641c▷
```
void pushroot(Root r) {
    if (rootstacksize == rootstackmax) {
        rootstack = realloc(rootstack, sizeof(rootstack[0]) * (rootstackmax + 16));
        assert(rootstack != NULL);
        rootstackmax += 16;
    }
    rootstack[rootstacksize++] = r;
}
```

Popping a root requires a check that the roots match. We compare pointers by casting to `void*`.

641c ⟨*root.c* 641a⟩+≡ ◁641b 641d▷
```
void poproot(Root r) {
    assert(rootstacksize > 0);
    rootstacksize--;
    assert(rootstack[rootstacksize].alt == r.alt &&
           (void*)rootstack[rootstacksize].u.envroot == (void*)r.u.envroot);
}
```

To reset the root stack, we just reset `rootstacksize`.

641d ⟨*root.c* 641a⟩+≡ ◁641c
```
void resetrootstack(void) {
    rootstacksize = 0;
}
```

C.6 Placeholders for exercises

realloc $\mathcal{B}$

641e ⟨*private declarations for copying collection* 641e⟩≡ (194a)
```
static void collect(void);
```

641f ⟨*copy.c* **[[prototype]]** 641f⟩≡
```
/* you need to redefine these functions */
void collect(void) { (void)scanroot; assert(0); }
void printfinalstats(void) { assert(0); }
```

642 ⟨*ms.c* **[[prototype]]** 642⟩≡

```
/* you need to redefine these functions */
void printfinalstats(void) {
  (void)nalloc; (void)ncollections; (void)nmarks;
  assert(0);
}
```