Moby Implementation Notes
(Version 0.9.18)

The Moby Project
moby.cs.uchicago.edu

Last revised: May 30, 2004

COPYRIGHT © 2003 The Moby Project

Contents

Chapter 1

An overview of the MOBY implementation

This chapter gives an overview of the main pieces of the MOBY implementation.

1.1 The MOBY compiler

1.2 The MOBY run-time system

1.3 Roadmap

Chapter 2

Lexing and parsing

2.1 The lexer

2.2 The prelude parser

2.3 The MOBY parser

The MOBY parser is implemented using ML-Yacc.

Chapter 3

The typed abstract syntax tree

This chapter describes the data structures used to represent MOBY types and the typed abstract syntax tree (typed AST).

Chapter 4

Environments

This chapter describes the various environments used in the type checking of a MOBY program. The code for these environments can be found in the `mobyc/Env` directory.

4.1 The environments

The `Env` structure collects together the various environment types. In addition, each type of environment has its own module that collects together the operations on the environment. Table 4.1 lists the environment type and their defining module. Envirnoments are layered; for example, a type environment contains a top-level environment that is used to resolve qualified names.

In the remainder of this section, we describe the rôle played by each type of environment.

Table 4.1: List of environments

SML type	**Support module**	Description
`Env.global_env`	`GlobalEnv`	Global environment
`Env.top_env`	`TopEnv`	Top-level environment
`Env.sig_env`	`SigEnv`	Signature environment
`Env.ty_env`	`TyEnv`	Type environment
`Env.exp_env`	`ExpEnv`	Expression environment

4.1.1 Global environments

4.1.2 Top-level environments

4.1.3 Signature environments

4.1.4 Type environments

4.1.5 Expression environments

4.2 Compilation environments

The environment library also supports *compilation environments*, which provide an interface to separately compiled modules (including the pervasive definitions).

Chapter 5

The MOBY typechecker

5.1 Checking Types

5.1.1 CompareTy

```
Typing/Util/compare-ty.sml
```

The CompareTy module is fairly complicated, and, I expect will be the source of many bug fixes. Also, experiments with the type system (such as changing the definition of subtype) will probably require modifying this file.

At last count, there are ten functions providing entry into the same set of mutually defined functions which make up the bulk of the module. Those that check for type-equality (compare, compareTyScheme, compareTyCon, compareExTy) return one of three values

- EQUAL if the types are currently equal
- CANUNIFY if there is an assignment to the metavars which would satisfy equality
- NOTEQUAL otherwise

The functions which check for a subtyping relationship between types (subtype, wsubtype, subtype-TyScheme, subtypTyCon) only return a boolean representing the equivalent of CANUNIFY/NOTEQUAL. (In other words, true if there is an assignment to the metavars which makes one type a subtype of the other, false otherwise. These functions could be modified to return a third value equivalent to equal, but there hasn't been the need for it.

The remaining two, unifyEq and unifySub, first call compare or subtype, and if unification is possible, do it. The boolean they return is whether unification was successful.

Most of the file consists of a number of mutually recursive functions, that compare a pair of types, influenced by the following environment structure:

```
type env =
```

```
datatype compare = EQUAL | CANUNIFY | NOTEQUAL
val compare  : (Types.ty * Types.ty) -> compare
val compareTyScheme : (Types.ty_scheme * Types.ty_scheme) -> compare
val compareTyCon : (Types.ty_con * Types.ty_con) -> compare
val compareExTy : (Types.extended_ty * Types.extended_ty) -> compare

val subtype  : (Types.ty * Types.ty) -> bool (* full subtyping *)
val wsubtype : (Types.ty * Types.ty) -> bool (* width-only subtyping *)
val subtypeTyScheme : (Types.ty_scheme * Types.ty_scheme) -> bool
val subtypeTyCon :  (Types.ty_con * Types.ty_con ) -> bool

val unifyEq  : (Types.ty * Types.ty) -> bool
val unifySub : (Types.ty * Types.ty) -> bool

val resolveCon : Types.ty_con -> Types.ty_con
```

Figure 5.1: Signature of the CompareTy module

```
{ eqn   : NameEq.eq_assum,
  eqv   : TyVarEq.eq_assum,
  meta  : T.meta_kind H.hash_table,
  metav : MetaEq.eq_assum,
  full  : bool,
  unify : bool,
  subn  : SubName.sub_assum
}
```

The components of this structure are:

eqn A set of assumptions about type names (i.e. names of type constructors) which are assumed to be equal. This is used to prove that two recursive types are equal – they are examined structurally, under the assumption that they are equal.

eqv A set of assumptions about type variables which are equal. This is used to handle alpha equivalence – the bodies of a parameterized type are compared under the assumption that the type parameters are equal.

meta This and the next item, metav, are used to represent functionally unifications to metavariables, without actually performing them destructively. This allows us to test whether two types may be unified without actually unifying them. The field meta is a hash table which contains unifications that would have been done. In the case of full unification, this structure used used at the end to actually commit the unification.

metav This is used so that metavariables are not unified to each other. Basically, it is an equivalence class of metavariables. It is not necessary for soundness, it is just merely an implementation of a soft of “union find” algorithm, so that there are not long chains of metavariables to traverse.

full Controls whether the subtyping relation is full subtyping or width only subtyping

unify This flag indicates whether we should actually try to do unifications or not. It is used in recursive calls to ask if types are equal, without modifying the unification structure already in place.

subn similar to eqn, a set of assumptions about what type constructors are subtypes of one another, used in proving subtyping relationships for recursive types.

Basically subtyping with unification is a constraint solving problem. Analogous to setting a metavar equal to a type in equality unification, subtyping unification adds upper and lower constraints to metavariables. The trick is to manage these constraints. Because meets and joins of types do not exist in moby, we must accumulate a list of constraints. As each constraint is added we must check it for consistency with the list, and if we eventually want to unify the metavariable with a type, we need to also check that that solution satisfies the constraints we have accumulated.

There are several alternatives to this constraint solving – constraints may be accumulated unchecked and then checked all at once during final unification with a type – but this may delay an error message from an error which causes an unsatisfiable constraint set.

Other issues about the constraint solving: We have to watch for cycles of metavariables in the constraints which would mean that they all should be set equal to each other. Cycles are searched for with the function occursBounds. We have to watch for bound that become tight (same type in upper and lower bounds) indicating that we can instantiate the metavariable with the type. Currently this is done only when unifying equal two metavariables with bounds (eqMetaMeta), but it could also be done as each new bound is added to the list (subMetaTy) or when unifying metavars in a subtyping relationship (subMetaMeta).

Another issue with subtyping deals simplifies overload resolution. It is convenient instead of unifying a variable to be a subtype of Int, to actually unify the variable with Int, as there are no subtypes of Int. The function canSubSuptype determines if sub or supertypes exist for a given type, and if they don't, falls over into unifying for equality.

5.1.2 ChkType

`Typing/chk-type.sml`

This section checks that a given type of the parse-tree is valid and return the corresponding internal type. The tricky thing here is to correctly calculate the variance information of a type variable so that it may be used in compare-ty to determine tycons in a subtyping relationship. Not only do we have to remember in the environment as we traverse the type whether we are in a covariant or contravariant position, we also need for each variable to remember the variance of where it was bound. This is because the variance of a variable is relative – if a type variable is bound contravariantly, and appears in a contravariant context, then with respect to the type, it appears covariantly.

5.1.3 ChkTyDcl

`Typing/chk-tydcl.sml`

This module checks the definition of the type constructors, and computes their internal representations. For Defs, fully abstract and enum types, recursion is not allowed, so the check is pretty much straightforward. For checking recursive tycons, the tycon must already be in the environment when called, so it is fetched from the environment and the mutable components of it are set based on the compilation of its definition.

5.2 Sig and Module Checking

These two parts are closely related, as the portion in chk-signature which is used to check class interface and type declarations is also used in chk-module. Because of this, the chk-signature implementation has a few extra parameters which only become relevant in the module checking setting. As signature checking is slightly simpler than module checking, I will cover that first.

5.2.1 ChkSignature

`Typing/chk-signature.sml`

The main goal of signature checking is to produce a sig_env, which is a mapping of atoms to their specifications.

For signatures which consist of a list of specifications, the order in which the signatures are checked is the main complication of the control flow. Because specifications may refer to any other specification in the signature, a spec may only be checked after entries for all of the specs it refers to (including possibly itself) have been entered into the environment. This is accomplished via a working environment (wkEnv) produced from the WorkingEnvFn. A wkEnv is parameterized by a datatype describing the specifications which could possibly be incomplete. The first pass of chkSpecList (called prechkSpecList) just enters each of the specs into the working environment. The next pass (called chkSpec) then begins the actual tracing of the spec. If the spec is allowed to be recursive, then it is just walked over (using findForwardRefs) and any references to other specs are added to the wkEnv. If, in findForwardRefs, all references added were already covered in this manner, the spec (and others mutually recursive with it) will appear in WE.nextGrp, called at the end of chkSpec. This group of mutually recursive specs is checked with chkRecSpecs (described later). If the spec may not be recursive, it is immediately checked using the current environment, and the Types.ty representation produced.

The function chkRecSpecs first sorts the specs so that any derived classes will appear after their parents in the list. This is important as the complete types of the parent must be available before the type of the child class can be synthesized.

Then insPspec first partially examines each spec and makes (incomplete) entries for each in the sigEnv and returns a list of finishing functions that check the bodies of the specs, and fill in the details. The important point, is that in checking the well-formedness of these types, the complete information about the type is not required. (Basically, all that is needed is that the tycon is bound to something and not a free reference.) The one caveat to this is in checking the well-formedness of type application in the presence of type bounds. Here, we need to see if the argument is a subtype of the bound, perhaps using an argument that is not fully defined. For this reason, we need to delay these bounds checks (as they are not critical in creating the semantic type) until after all of the types have been fully defined, and then go back and check that all of the bounds were satisfied.

The finishing functions returned by insPspec are then invoked – in chk-signature the results of the finishing functions are unimportant, but later in chk-module if a full class was checked, then information about that class is returned by chkRecSpecs, so that the term portion of the class may then be examined. (More on this later).

5.2.2 ChkModule

`Typing/chk-module.sml`

Module checking is very similar to signature checking. Just as types may be mutually recursive, so too functions and the makers of classes, so module checking includes a working env for types and a separate working environment for terms. These two interact with the checking of classes. A mod_env keeps track of these two working envs during type checking. The full components of the mod_env are in Figure 5.2.

Just as in chk-signature, the stages of checking are :

- insert nodes into the working env,
- trace the right hand side of each node, added edges to the env for all references to other nodes
- between tracing each node, check to see anything has been completed, and process that.

However, as the term part of classes cannot be completely checked before the type portion has been completely defined, we need to delay this operation. This delay is accomplished by not tracing a class immediately. Instead, when a group of recursive types is checked (through a call to ChkSignature.chkRecSpecs) the result is a list of classes which now have their types fully defined. These classes are then traced with the function chkClassDecl. The information returned by CS.chkRecSpecs includes not only the syntax of the class, but a finishing function for checking the term portion of the class. This finishing function is stored away until chkRecDcl finds the class again.

5.2.3 ChkCompUnit

`Typing/chk-compunit.sml`

This part is pretty straightforward. If the compunit is a signature, just call chkSignature and insert it into the environment. If the unit is a module, all ChkModule.chkModExp to get its signature,

```
loc : Error.location,
    (* Where we are in the file, for error reporting *)

wkEnv : WE.working_env,
    (* working env for partially defined functions/classes *)

twkEnv : TWE.working_env,
    (* working env for partially defined types/classes *)

topEnv : TopEnv.env,
    (* The top env being constructed *)

publicEnv : TopEnv.env,
    (* The visible portion of the above *)

top_decls : AST.top_decl list ref,
    (* Top-level declarations (stored in reverse order) *)

class_finish : class_finish AtomMap.map ref,
    (* Functions for finishing off class declarations (see below) *)

makers : Atom.atom AtomMap.map ref
    (* the set of makers for each class name -- as functions refer
       to makers, but we do not have individual nodes for makers
       we need to be able to figure out what class a maker is in
       to add a reference to it in the wkEnv during tracing.   *)
```

Figure 5.2: Components of a mod_env

match it is necessary, and then insert that into the environment. Finally if the unit is a functor, first check the signatures of the parameters to the functor (adding them to the environment as we go as one signature may refer to a previous module parameter). Then, check the body of the functor. If the functor has a result signature, that is then matched and then the functor is added to the environment.

5.3 Signature Manipulation

5.3.1 ChkExport

`Typing/chk-export.sml`

In check export we examine the derived public signature from a module and check that it is well-formed. This only involves walking over all of the types in the signature, checking that the tycons are public tycons defined in the env. This type-walk is defined in the fcns, chkTy chkTs, chkETy, chkConName and chkTycUse, and chkConName actually looks up the tycon in the environment, and reports an error in its absence. It is important that the stamp of the con returned from looking a tycon up in the current env is checked, as the type may actually refer to a more distant, unreachable type. If the type is not found in the current environment, chkConName then calls TopEnv.slowFindTy, which is a hack to find tycons in scope, but not in the most local scope. As we don't keep the path info in the ty structure, we have to just search for the ty in all accessible modules.

5.3.2 MatchSig

`Typing/match-sig.sml`

Signature matching is checking that a given signature constraint (from here on referred to as the signature on the left) is satisfied or matched by a derived signature of a module (on the right). The key part is keeping track of the constraints: for instance in matching

```
LEFT                RIGHT

type T              type T = Int
type S = List(T)    type S = List(Int)
```

when we check to see if the type S on the left matches the type S on the right we need to do this in a context where T is Int. One possibility would be to gather all the constraints we go and solve them at the end, but we have already the order of dependencies. Another possibility would be to gather the current constraints in an environment and compare types based on that environment. The third (and implemented) option is augment the representation of abstract types so they can be destructively changed to look like the types of the right hand signature.

So, in order to do signature matching the representation of abstract types includes an extra mutable field called def. This field contains the tycon that the abstract ty has been set equal to. Because of this field, the resolve function in type-util.sml and and extra resolveCon function in compare-ty.sml need to expand an abstract con with the def field set into that tycon (this resolveCon function also turns a typeof into its objTy and eta-reduces a def constructor).

Like many of the other passes in this typechecker, signature matching of recursive decls is also done in two phases. The key thing is that a type on the left (in the given signature) must act the same as the type it is being matched to on the right. This allows compare-ty in the second phase to conclude it equal with its corresponding matched type. Therefore the def on a matched absTy is set to be the rhs ty so that in compare ty, the absTy will just be resolved to the rhs ty. For a tagtype, the only thing that needs to be set is the tag (its parent should be already matched). The second phase actually determines if the types should have been matched by looking at the bounds on an abstract ty, or just calling compareTyCon if the left type is not an abstract ty.

5.3.3 RenameSig

`Env/rename-sig.sml`

Signature renaming is used to create genericity in abstract types and tagtypes. The function SigEnv.generate will rename these components of a signature, and return a map mapping stamps to their new constructors. This map can be used in generate_result, to beta-reduce a functor application. On functor application the result signature may refer to types in the parameter signatures. In checking the application the parameter sig must be renamed before it can be matched to the sigs of the arguments. The results of this parameter sig renaming are used in the renaming of the result sig so that it may still refer to the correct types. Further more, as sigmatching destructively updates the abstract types, the transparent types will be automatically propagated through. For example... (I've added stamps to the types after their names)

$$\texttt{module M(N}:\{\texttt{type T0}\})\{\texttt{type T1} = \texttt{N.T0}\}$$

where the signature of the parameter N is $\{$ T $\mapsto$ C_Abs(id=0,def=NONE) $\}$ and the result signature of this functor is $\{$ T $\mapsto$ C_Def (id=1, def=T0) $\}$ and on the application of the functor M to N where N is

$$\texttt{module N}\{\texttt{type T3} = \texttt{Int}\}$$

the signature of N is first renamed to $\{$ T $\mapsto$ C_Abs(id=4,def=NONE) $\}$ with the map [0 $\mapsto$ T4].

On matching, the signature of N, T4 is updated to C_Abs(id=4,def=SOME Int). So when the result signature is renamed, it becomes $\{$ T $\mapsto$ C_Def (id=1,def=T4) $\}$

Renaming must be done in the given order of the signature, so that the deep copy involved in renaming will find the new versions of the types. It also must be done in two stages so that new nodes can be created for recursive objects first, before their bodies are copied.

The only tricky part is that the obvious deep copy, while safe, would destroy the sharing between the objty component of classes and typeof tycons. For this reason, we charge the class with putting the objty directly into the top-env in the first phase of copying, so that when the typeof constructors are copied, they may find this same objty.

Chapter 6

The BOL representation

The MOBY optimizer uses a low-level λ-calculus base intermediate representation called BOL.

6.1 BOL types

6.1.1 Kinds

6.2 BOL terms

The BOL representation is a normalized λ-calculus — each intermediate result is bound to a variable. We go beyond many such representations in that arguments are *always* variables (never constants). This design choice simplifies the implementation of rewriting, but it does add some additional structure to program representations.

A BOL expression is a BOL term labeled with a program point. The `E_Pt` constructor builds an expression from a program point and term. The meaning of the BOL terms is as follows:

`E_Let`(*xs*, *rhs*, *exp*)

`E_StackAlloc`(*x*, *sz*, *align*, *exp*)

`E_Fun`(*fbs*, *exp*)

`E_Cont`(*fb*, *exp*)

`E_AllocCheck`(*n*, *exp*)

`E_If`(x, exp_1, exp_2)
    If x is `True`, then evaluate exp_1, otherwise evaluate exp_2.

`E_Switch`(x, *cases*, *dflt*)

`E_Apply`(f, xs)

`E_Throw`(k, xs)

`E_Ret`(xs)
    Evaluates to the values of the variables xs. When this term is in a *tail* position of a function body, then the values are returned as the result of the function. Otherwise, the term is the right-hand-side of some let binding and the values will be bound to the left-hand-side variables of the let.

The right-hand side of a BOL let binding is represented by the `rhs` datatype, which has the following constructors:

xs = `E_Exp`(*exp*)
    Evaluate an expression *exp* and bind its results to the left-hand-side variables xs.

x = `E_Select`(i, y, ty)
    Select the ith element (zero-based) of the heap object bound to y. The BOL type ty specifies the layout of the object and is used to compute the offset of the ith element.

x = `E_Alloc`(ys)

x = `E_Wrap`(y)

x = `E_Unwrap`(y)

x = `E_IConst`(n)
    Binds x to the integer constant n. The precision of n is determined by the type of x, which may be an enumeration.

x = `E_SConst`(s)
    Binds x to the string constant s.

x = `E_FConst`(f)
    Binds x to the floatin-point constant f. The precision of n is determined by the type of x.

x = `E_BConst`(b)
    Binds x to the boolean constant b.

x = `E_Extern`(a, ty)

x = `E_Label`(l)

x = `E_Prim`(p)

x = `E_DictFieldSel`(y, f)

x = `E_DictMethSel`(y, m)

x = `E_FieldGet`(y, s)

x = `E_FieldPut`(y, s, z)

x = `E_MethGet`(y, s)

x = `E_CCall`(f, ys)

Chapter 7

Translation from typed AST to BOL

The translation phase is responsible for converting the typed abstract syntax tree produced by the type checker into BOL code. A major part of the translation process is compiling the high-level patterns of the typed AST to decision trees encoded in BOL. Pattern compilation is described in Chapter 8; in the chapter we describe the other aspects of translation.

The current translation scheme relies on the assumption that we have complete knowledge of the representationf of any imported module (including parameter modules). In the future, we plan to suport a *development mode* that allows modules to be compiled based on only the signatures of their antecedants. Development mode will require a more conservative treatment of features such as class linking.

7.1 The translation environment

7.2 Translating types

7.3 Exception handling

One of the responsibilities of the translation phase is to make the exception handling control-flow explicit. This goal is achieved by adding an *exception handler* parameter to each function.

7.4 Translating classes and objects

7.4.1 Class linkage

7.4.2 Translating methods

7.4.3 Translating makers

7.4.4 Translating `initially` clauses

A class definition may have an **`initially`** clause, which is an expression that gets invoked immediately after object initialization (see Section 7.4.7). We support **`initially`** clauses by translating them to a premethod-like function (*i.e.*, a function that takes **`self`** as its first argument). The name of the initially function is stored in the class's environment. When the class has a superclass, its initially function is responsible for invoking the superclass initially function first (if present). If the superclass has an initially function, but the subclass does not, then we use the superclass's initially function for the subclass.

7.4.5 Translating method dispatch

7.4.6 Translating field operations

7.4.7 Translating `new`

Chapter 8

Compiling MOBY patterns

This note desribes the algorithms used by the MOBY compiler to translate pattern matches to simpler form. Our approach is based on that of Pettersson [Pet92], but handle a richer language for specifying patterns.

8.1 Pattern matching in MOBY

Along with function application, pattern matching is one of the two mechanisms for control-flow in functional languages. In MOBY, we have enhanced the power of pattern matching over that found in SML in several ways:

1. *Or-patterns* allow two or more alternatives to be specified. For example, the pattern

   `(_::a::b::_ | a::b::Nil)`

   matches a list of length two or greater; if the list has length two, then the elements are bound to `a` and `b`, if the list has length greater than two, then the second and third elements are bound to `a` and `b`.

2. *When-clauses* allow a predicate to be associated with a rule.

3. *Abstract value constructors* [AR92]

4. *Tagtype constructors*

5. *Negative pattens* allow the programmer to specify the values that are not matched by a pattern. For example, assuming the color type has constructors `Red`, `Blue`, `Green`, the pattern

   `(x **isnot** Red)`

   has the same effect as

   `(x **is** (Green|Blue))`

   The subpattern of a negative pattern cannot have free variables.

8.2 Pettersson's algorithm

In this section, we describe our version of Pettersson's algorithm [Pet92].

8.2.1 Step 1 — pattern canonicalization

We are given as input a list of *rules*, where each rule consists of a tuple of patterns, an optional when clause, and an action. Typechecking ensures that each rule in the match has the same arity. Our representation for this is as follows:

```
datatype rule = RULE of (pat list * exp option * exp)
datatype match = M of rule list
```

The first step of the algorithm is to canonicalize the patterns in the match.

8.2.2 Step 2 — DFA construction

$$\begin{pmatrix} path_1 & \cdots & path_n & & \\ \downarrow & & \downarrow & & \\ p_{1,1} & \cdots & p_{1,n} & \Rightarrow & act_1 \\ \vdots & \ddots & \vdots & & \vdots \\ p_{m,1} & \cdots & p_{m,n} & \Rightarrow & act_m \end{pmatrix}$$

8.2.3 Step 3 — DFA optimization

8.3 Extensions to the basic algorithm

8.3.1 Or-patterns

8.3.2 When clauses

8.3.3 Abstract value constructors

8.3.4 Tagtype constructors

8.3.5 Negative pattens

Simple implementation is to translate to when clause, but we can do better in some situations.

8.4 Choosing the column

8.5 Related work

There have been a number of algorithms published for pattern matching in lazy languages ([**?**]), but we are only aware of a couple of descriptions of compiling pattern matching in strict languages.

The core of our approach is based on Pettersson's algorithm, but we have extended his approach to handle a much richer pattern language. Leroy [Ler90] describes an approach that is similar to Pettersson's, although without the DFA analogy.

Chapter 9

Optimization of the BOL representation

Most optimizations in the MOBY compiler are performed on the BOL representation described in Chapter 6. Currently, the optimizer performs the following phases:

- Initial census
- First contraction
- Eta splitting
- Second contraction
- Useless variable elimination
- Small constant copying
- Third contraction
- Cluster conversion
- Unit contraction
- Allocation checks

9.1 Variable substitutions

9.2 Census

The `Census` module provides code to compute information about the number of uses of variables. This information is used by the various transformation phases. Specifically, the census computes a *use count* for every BOL variable, which reflects the number of non-binding occurances of the variable, and an *application count* for every variable that is bound as a function or continuation.

The `Census` module also provides functions for adjusting the census data when code is eliminated or replicated.

9.3 Denesting

We often find it useful to introduce extra levels of block (or let) nesting when transforming BOL. Unfortunately, such extra structure hides data dependencies from simple analysis, such as that used by the contraction phase (see Section 9.4). For example, consider the following BOL fragment:

```
{
  let a0 = {
    let b1 = "blah"
    let x3 = {
      let y4 = { let z5 = True in (z5) }
      in y4
    }
    in x3
  }
  in a0
}
```

Without data-flow analysis, the fact that `a0` is **`True`** is obscured by the nesting of blocks. The *denesting* transformation flattens unnecessary block-nesting.[1] For example, the above fragment is reduced to the following by the denesting phase:

```
{
  let b1 = "blah"
  let z5 = True
  let y4 = z5
  let x3 = y4
  let a0 = x3
  in a0
}
```

The contraction phase can easily reduce this fragment to

```
{
  let z5 = True
  in z5
}
```

in one pass.

The denesting transformation is implemented in the `BOLUtil` module (file `bol-util.sml`) and takes time linear in the size of the expression. It can be either applied to an entire expression or just to the top of an expression.

For the Mandelbrot program (about 50 lines of MOBY code), applying the denesting transformation prior to contraction eliminated 80 let-float transformations and reduced the number of contraction iterations over two phases from 9+3 to 2+2.

[1]It is essentially a restricted form of the *let-floating* transformation described by Peyton Jones, Paqrtain and Santos [PPS96].

9.4 Contraction

The `Contract` module implements a collection of transformations that are guaranteed to *shrink* the size of the program.

9.5 Cross-module inlining

9.6 Eta splitting

The `EtaSplit` module applies η-expansion to escaping functions that have known call sites. This transformation is useful, because it turns known calls to escaping functions into known calls to known functions, which allows useless variable elimination and specializing of calling conventions. The η-expansion transformation is only applied to those escaping functions that have known call sites. Recall that escaping functions are those that either are marked as exported or have a use count that is greater than their application count. If a function has a non-zero application count and is escaping, then we split it into two functions.

9.7 Useless varible elimination

The `Useless` structure is responsible for removing variables that do not contribute to the computation (even though they may be used). This phase proceeds in two steps. First, we compute the set of useful variables of the program. A variable is defined to be *useful* if it satisfies one of the following conditions:

- If it is an applied function or thrown continuation.
- If it is the argument to an unknown or C function call.
- If it is the argument to a primitive operation that has a side effect.
- If it is the argument to a `FieldPut` operation.
- If it is returned from a function.
- If it is the argument to a right-hand side expression that computes a useful result.
- If it is the result of a right-hand side expression and the corresponding left-hand side variable is useful.
- If is the argument to a known function or continuation, and the corresponding parameter is useful.

The last three of these conditions propagate usefulness backwards through the dataflow and require that we iterate the analysis until a fixed point is reached.

Once the set of useful variables has been computed, we can transform the program to eliminate useless variable (*i.e.*, those not in the set). This transformation is implemented by walking the tree looking for binding occurances of useless variables and eliminating them.

9.8 Small constant copying

Because the BOL representation does requires that all operands be variables, normal optimizations may result in sharing of small constants (*i.e.*, binding a BOL variable to a small constant that is used in multiple contexts). This situation can result in poor code, since shared small constants will not be used as immediate operands in the generated machine instructions. A related problem is when a small constant appears as a free variable in a function. To address these problems, the constant copying phase replicates variables that are bound to small constants at the site of their use. For example, consider the following BOL code fragment:

```
let x : int = 1
fun f (y : int) {
  let t = y + x
  ret t
}
let z = f(x)
...
```

The constant copying phase will rewrite this fragment as follows:

```
fun f (y : int) {
  let x1 : int = 1
  let t = y + x1
  ret t
}
let x2 : int = 1
let z = f(x2)
...
```

9.9 Cluster conversion

The cluster phase takes care of both closure conversion and cluster merging, as well as the LCPS transformation [**?**]. It works by first performing a series of analyses to determine where CPS splits are needed, how to map functions to clusters, closure representation, *etc.*. It then uses the information collected during these analyses to drive a single transformation phase that produces the BOL cluster representation.

Chapter 10

Code generation

10.1 Calling conventions

10.1.1 Classifying functions

Functions are grouped into *clusters*, which are functions at the same lexical level that can share the same closure and stack frame. We classify functions based on an analysis of their call sites.

ESCAPING the function escapes and, thus, may have unknown call sites.

KNOWN all call sites for the function are known.

TAIL all call sites for the function are known tail calls.

LABEL all call sites for the function are tail calls from the function's cluster.

We define an *entry* to a cluster is a function that is not a LABEL; *i.e.*, it is called from outside the cluster.

10.1.2 Classifying call sites

10.1.3 Choosing a calling convention

The calling convention used at a call site depends on the target fucntion's classification, as well as the call-site's classification. The following table describes the analysis:

	Call-site class	
Function class	**Non-tail call**	**Tail call**
ESCAPING	Standard call	Standard tail-call
KNOWN	Known call	Known tail-call
TAIL	*n.a.*	
LABEL	*n.a.*	Goto

10.1.4 Notation

We use a simple C-like syntax for describing the calling conventions. The calling conventions are described in terms of a stack-based argument passing model; in practice some architecture-specific prefix of the arguments will be passed in registers. We may also use dedicated registers for the implicit parameters (*e.g.*, the exception handler continuation and the environement pointer). In the discussion below, we assume that we are calling a function with the following signature:

$$\texttt{\textbf{val} f : (}\tau_1\texttt{, ..., }\tau_n\texttt{) \textbf{->} (}\sigma_1\texttt{, ..., }\sigma_m\texttt{)}$$

In the code below, we use `f` to refer to the callee's closure. A function closure is a two-word record consisting of a *code pointer* (refered to by `cp`) and an *environment pointer* (refered to by `ep`). We use `sp` to refer to the stack pointer, `fp` to refer to the frame pointer, `exh` to refer to the current exception handler continuation, `arg[`i`]` to refer to the ith argument, and `res[`i`]` to refer to the ith result.

10.1.5 Escaping functions

For calls to escaping functions, the arguments and results are passed in uniform representation (*i.e.*, one word per value). There are two additional implicit arguments: the exception handler continuation and the function's environment pointer. We let $p = \max(n + 2, m)$ be the size of the argument/result area.

Upon function entry, an escaping function first saves the previous function's `fp` register and sets the `fp` to point to the location of the saved `fp`. It then allocates the stack frame. Figure 10.1 shows the stack layout.

Standard call

When $n + 2 < m$, we allocate $k = m - (n + 2)$ extra stack space in the calling convention.

```
cp = f[0]
ep = f[1]
*--sp = arg[1]
...
*--sp = arg[n]
*--sp = exh
*--sp = ep
sp -= k
call *cp
res[1] = sp[m-1]
...
res[m] = sp[0]
sp += m
```

When $n + 2 \geq m$, let $k = n + 2 - m$

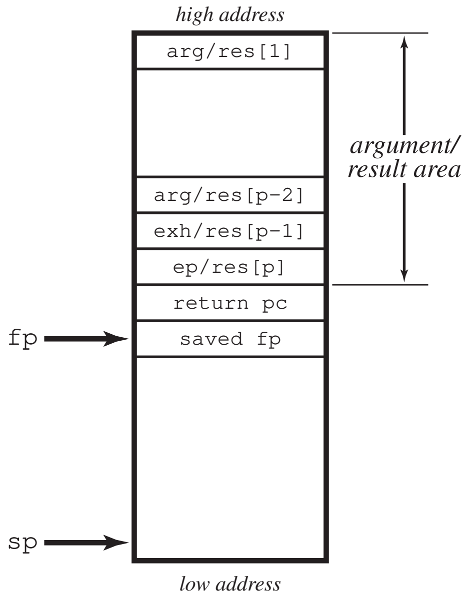

Figure 10.1: The standard stack-frame layout

```
cp = f[0]
ep = f[1]
*--sp = arg[1]
...
*--sp = arg[n]
*--sp = exh
*--sp = ep
call *cp
res[1] = sp[n+2]
...
res[m] = sp[k]
sp += n+2
```

Note that when `f` is a known function, we can call directly to `f`'s label and we do not need `cp`.

Standard tail-call

When a standard function call is made from a tail position, we need to discard the caller's stack frame prior to transfering control to the callee. We also need to setup the call so that the callee returns to the caller's caller. Let l be the argument arity of the caller (not including any implicit arguments). Note that MOBY's type system guarantees that the result arity of the tail call will be m.

The simplest case is when $l = n$ and we do not have to move the return PC. Furthermore, since the call is a tail call, the exception handler continuation that was passed in to the caller will be the same as is passed to the callee, so we do not have to overwrite that stack location. These properties lead to the following code sequence:

```
cp = f[0]
ep = f[1]
fp[n+3] = arg[1]
...
fp[4] = arg[n]
fp[2] = ep
sp = fp
fp = *sp++
jump *cp
```

When $l > n$, let $k = l - n$.

```
cp = f[0]
ep = f[1]
fp[l+3] = arg[1]
...
fp[k+4] = arg[n]
fp[k+3] = exh
fp[k+2] = ep
fp[k+1] = fp[1]
sp = fp
fp = *sp
sp += (k+1)
jump *cp
```

10.1.6 Known functions

For calls to known functions, the arguments and results can be passed in unboxed representation. As with escaping functions, there are two additional implicit arguments: the exception handler continuation and the function's environment pointer.

Known call

Known tail-call

10.1.7 Goto

10.1.8 C function calls

The basic code generation for C function calls is handled using the MLRISC `C_CALLS` interface. This library supports the underlying ABI's argument-passing convention. The MOBY compiler supports annotations on C functions that can be used to specialize the calling convention. These annotations are introduced in the prototype declarations for the C functions in the MBX files and are described below.

may-callback

This annotation marks the C function as one that might *call back* into MOBY code prior to its own return (*e.g.*, the main event loop of a GUI library). If this annotation is present, then we must save and restore the allocation pointer across the C call in such a way as to allow any MOBY callbacks to have access to the allocation pointer. We achieve this goal by saving the allocation pointer in the host task's descriptor prior to the call and then restoring it on return.

```
basePtr = allocPtr & MASK;
taskPtr = basePtr->taskPtr;
taskPtr->allocPtr = allocPtr;
   ... call C function ...
allocPtr = taskPtr->allocPtr;
```

The wrapper code for the MOBY callback gets the allocation pointer from the task structure on entry and saves it on exit (see Section 10.1.9 below).

moby-lib

This annotation marks the C function as requiring access to the task descriptor of its host task. In addition to the saving and restoring of the allocation pointer (as in the case of the **may-callback** above), we pass the current task pointer as an extra argument to the C function.

pure

The pure annotation does not affect the calling convention, although it should not appear in conjunction with the "**may-callback**" annotation. Instead, it is used to mark C functions as having no visible side effects, which allows the optimizer more freedom (see Chapter 9). Examples of pure C functions include math library functions, such as `sin` and `cos`, as well as system calls, such as `gettimeofday` and `access`, that query the system for information.

10.1.9 C callable functions

The code generator also generates wrapper code for C-callable functions.

10.2 Leaf-procedure optimization

After register allocation, we check to see if the cluster is both a *leaf* procedure and does not need allocate stack space (either for stack objects or spill locations). If these two conditions are met, then we rewrite the instructions to not allocate or use a new stack frame. For example, the MOBY function

```
fun inc (r : Ref(Int)) -> () { *r := *r + 1 }
```

is compiled into the following instruction sequence on the IA32:

```
inc:
        pushl   %ebp
        movl    %esp, %ebp
        movl    16(%ebp), %eax
        incl    (%eax)
        leave
        ret     $12
```

By applying the leaf-procedure optimization, we can reduce it to the following:

```
inc:
        movl    12(%esp), %eax
        incl    (%eax)
        ret     $12
```

This optimization works by first rewriting the entry and exit code sequences and then rewriting each block to replace frame-pointer based addressing with stack-pointer based addressing. The one subtlety is that MLRISC considers any basic block that has non-local control flow (*i.e.*, because of a **`throw`**) to be an exit block. Thus, the exit sequence rewriting has to recognize such cases and ignore them. The leaf-procedure optimization can be disabled with the `--no-leaf-opt` command-line option.

Eventually, we may want to apply this optimization to a larger class of procedures. First, we could support stack allocation by making the operand rewriting more sophisticated (this is essentially like using a virtual frame pointer). We might also apply the optimization to procedures that make tail calls.

10.3 Generating PC maps

Note: It may make more sense for this information to be in the run-time chapter. This section doesn't really discuss how the compiler generates the PC map information. That would be useful information to have written down, though

The actions that a run-time service performs may depend on the state of the threads in the system. For example, we prefer not to trace dead variables during garbage collection. If the relevant portions of the state are available statically (that is, they are a function of the program counter), then the run-time system can combine two techniques to acquire the state:

- Use the program counter as an index into a compiler-generated table containing relevant information.
- Examine the instruction stream and the values of dedicated registers to determine information not contained in the table.

One important design goal is that we want to be able to interrupt threads *at any instruction* without preventing run-time services, such as garbage collection, from running. Achieving this goal allows us to hide the latency of a page fault by running other threads. It is not generally possible in systems with so-called "gc safe-points".

In this section, we describe the division of responsibility between the run-time system and the compiler, as well as the binary encoding of the PC-map table. Activation-record layout and code generation for memory allocation are closely related subjects, but we assume they are described in detail elsewhere. First, we describe the overall organization of the table. Second, we give the detailed binary encoding. Third, we explain how we expect the run-time system to use this information. At this time, PC maps are only implemented for the IA-32 architecture; much of what follows is architecture-specific.

10.3.1 Table Organization

There is one PC-map table per file. Each module's initialization code calls a run-time routine to register the file's table. (Because a file can have multiple modules but only one PC map, the run-time routine detects if a table has already been registered. This feature has not been tested, though.) The table itself is in a read-only data section.

A file table has three levels. (You could call it a "table of tables of tables", but each level contains information other than just pointers to tables.) The top level is also called the *cluster level*. The middle level is also called the *common level*. The bottom level is also called the *uncommon level*.

A cluster-level entry contains:

- The address at which the cluster begins. (By begins, we mean the lowest address; it need not be the entry point.)
- The number of bits that is sufficient to describe the difference between the beginning address of the cluster and any other address in the cluster. That is, if the cluster's code occupies b bytes, then this number must be at least $\mathrm{ceil}(\log_2 b)$. We call this number the "offset bits".[1]
- The address at which the common-level table for this cluster begins.
- The number of words between the frame pointer and the top of the activation record for this cluster.
- The number of words between the frame pointer and the beginning of the floating-point spill section of the activation record.
- The number of floating-point spill slots for the activation record.

In general, the cluster level describes the activation record, the encoding of lower-level tables, and the location of lower-level tables.

A common-level entry is for a particular address (program counter value). It contains:

[1] In practice, the compiler does not know instruction lengths, so it (very) conservatively estimates b to be 32 times the number of instructions.

- The difference between the address and the beginning of the cluster. (That is, the offset.)
- The address at which the associated uncommon-level table begins. (This address should probably be changed to an offset in the future.) This uncommon table contains entries for PC values between this common-level entry's value and the *previous* one. That is, it is for instructions preceding the common one.
- Static liveness information for local variables at this PC value. Specifically, a bit vector indicates which registers and stack slots may be used before they are defined (i.e., which locations are live).

An uncommon-level entry is usually for a particular PC value. It contains:

- The length of the instruction at this PC.[2] (In the future we hope to have the run-time recompute this information by parsing the instruction stream.)
- Whether this PC value is in an allocation sequence. That is, whether an object is partially initialized.
- Two lists of local variables (registers or stack slots) called "plus" and "minus". Given the liveness information for after the instruction, subtracting the minus and adding the plus gives the liveness information for before this instruction.

Note that the only way to find the uncommon-level entry for a particular PC value is to start at the beginning of the correct uncommon-level table and iterate through the entries subtracting instruction lengths until the correct PC value is obtained.

For allocation sequences, the liveness information is the same for the entire sequence since the run-time system should reset the program counter to the beginning of the sequence. Therefore, there is only one entry for the entire sequence; the length is the length of the sequence.

We now give some brief justification for using three different levels, each with its own encoding. The top-level table encodes the activation-record layout. Since it's entries are a fixed size, binary search is fast.

The division between common and uncommon entries is even more important. For most PC values, we never expect the run-time system to examine the information for this value. The run-time system needs the information only if a thread is currently at this PC or it is a return address on the call stack. So, currently we classify the ends of basic blocks and instructions after calls as common, all other PC values are uncommon.

Ends of basic blocks are actually not common, but it is easier to emit them into the common table since their liveness information is not closely related to the liveness information for the following instruction. Perhaps in the future we should segregate block ends from return addresses.

There are two rare situations where a return address is in the uncommon table instead of the common table. The first is after a call instruction that is the last instruction in a basic block. The compiler could detect this case, but it is a pain to do, so it is unimplemented. The second is after a

[2] It is quite frustrating that the IA-32 has variable-length instructions.

call instruction that immediately precedes an allocation. In this case, the entry cannot be a common one, because only uncommon entries can describe allocation sequences. These cases are both rare. A call instruction is not followed by a stack-pointer adjustment only if the call has no arguments (including an exception handler or environment pointer) or results. In the second case, we also need a heap-limit check between a call and an allocation unless we can bound the amount of memory allocated during the call.

All the entries in a particular middle-level table have the same size, so binary search is easy there. For the bottom-level table, table size is more important than lookup time, so entries have variable length and linear processing is necessary. Moreover, by keeping the uncommon entries in separate tables, we hope to have better paging behavior.

10.3.2 Binary Encoding

Location Numbering

To record the liveness of a register or stack slot, we use a mapping from registers and stack slots to small integers. The non-dedicated registers `eax, ebx, ecx, edx, esi` are represented by 0, 1, 2, 3, 4, respectively. The 4-byte (non floating point) stack slot in the lowest address (shallowest on the stack) is represented by 5. The next shallowest slot gets 6, and so on. This encoding is a bit wasteful since we use numbers for the slots containing the saved frame pointer and return address, yet these slots are never garbage-collection roots.

Cluster Level

The cluster-level table begins with one word that is the number of cluster entries in the table. The entries are sorted with respect to the beginning addresses of the clusters they describe.

Each cluster-level entry takes 3 words. The first word is the address of the associated common-entry table. The second word is the beginning address for the cluster. The third word contains four separate one-byte fields. The first byte is an 8-bit integer indicating the number of offset bits used in the common-entry table. The second byte is an 8-bit integer indicating the number of words above the frame pointer in the activation record. The third byte is an 8-bit integers indicationg the number of words between the frame pointer and the floating-point spill slots. The fourth byte is an 8-bit integer indicating the number of floating-piont spill slots. (Each slot is 12 bytes.)

Hence, the encoding fails if there are more than 253 argument/results/reserved slots, more than 255 word spill slots, or more than 255 floating-point spill slots.

A final word in the cluster-level table is a label which follows the last common-level table.

To give some intuituion for this layout, consider finding the correct cluster for a PC value for a table starting at address `pcmap`. The first cluster's lowest address is at `[pcmap+8]` and the last cluster's lowest address is at `[pcmap+8+3*[pcmap+0]]`. Each entry is 12 bytes, so binary search is now straightforward. Suppose such a search determines that the cluster containing the PC value has its lowest address at `[pcmap+i]`. Then the common-level table for the cluster is exactly between `[pcmap+i−1]` and `[pcmap+i−1+12]`. (The extra label at the end of the table makes this fact hold even for the last cluster.)

Common Level

Note: This level's encoding should change soon. And the current implementation is incomplete as explained below.

Because the cluster-level table specifies the size of a common-level table, the common-level table contains only common-level entries. The entries are in sorted order.

Each entry consumes an integral number of words. The first word is the address of the associated uncommon-level table. (A future change involves making this an offset so that it can normally consume less space.) The second word contains the PC offset for this entry, that is, the difference between the address of this common instruction and the beginning of the cluster. The second word *also* contains a bit vector encoding which registers and stack slots are live. The "offset bits" field of the associated cluster entry gives the information that the run-time system needs to extract these two fields correctly.

We expect that most clusters can fit the pc offset and bit vector in one word. However, it is likely that some cannot. Allowing longer entries is not yet implemented, but it will be very soon. Until then, the pc offset is in the low-order bits of the one shared word. The compiler raises an error if more than one word is needed.

Uncommon Level

The entries in an uncommon level table describe a sequence of instructions preceding the instructions that the associated common level entry describes. The entries are in "backwards order" — the first entry is for the instruction just before the common-entry instruction, the second entry is the for the instruction just before the first-entry instruction, etc. Allocation sequences are logically one instruction (i.e., one entry); they are discussed below.

For non-allocation sequences, each entry has a variable length of at least one byte. The first byte contains four independent pieces of information:

- The highest bit is unset to indicate that this entry is not for an allocation sequence.
- The next highest bit is set iff the "plus" list for the entry is non-empty.
- The next highest bit is set iff the "minuss" list for the entry is non-empty.
- The low 5 bits encode an unsigned integer representing the length of the instruction. (The encoding fails if an instruction can span more than 32 bytes, including `nop` bytes introduced by `.align` directives.)

The rest of the entry consists of the plus and minus lists, if necessary. If a list is non-empty, it contains the integer-encoding for each slot in the list followed by an "all 1s" sequence to terminate the list. The number of bits per list element is minimal; a list element may cross any byte boundary.

The minimal number of bits is $\text{ceil}(\log_2(5 + s + 1))$ where s is the number of slots in the activation record (number below the frame pointer plus number above the frame pointer).

Although the lists have no internal alignment, each uncommon-table entry begins on a byte boundary.

Allocation-sequence entries are very similar. The differences are:

- The highest bit of the first byte is set.
- We use 13 bits to encode the length instead of 5. Hence an allocation sequence can span up to 8Kb. The 13 bits (5 in the first byte and 8 in a mandatory second byte) are interpreted *big-endian*). Because IA-32 is little-endian, generating the correct assembly directives requires some acrobatics.

10.3.3 Run-time Use

Having explained what is encoded and how, we sketch how we expect the run-time system to use the information. We assume that the run-time needs to perform a garbage collection and that it has access to the program counters of all live threads. The system needs to create an appropriate collection of garbage-collection roots and leave all threads in appropriate states for continued execution.

Obviously, the system must index into the PC-map tables to get the information for the correct instruction. Since the method for doing so is fairly obvious from the encoding, we do not discuss it further. It is how the information is used that is more interesting.

An obligation of the code generator is to ensure that allocation sequences can be re-started (rolled back) simply be resetting the program counter to the beginning of the sequence. So if the run-time learns that a thread was in the middle of an allocation sequence, it needs only to calculate where the address of the beginning of the sequence.

Finding garbage-collection roots seems straightforward at first. The live variables for the current activation record are a function of the PC-map tables and the current program counter. From the activation-record layout information we can find the return address and use that to determine the live variables in the caller's frame. Iterating, we can "crawl the stack" finding all of the thread's roots.

Problems arise because we strive to support garbage collection at *any* program point. Hence the shallowest frame can be in a temporarily weird state and the run-time plus the tables must account for such strangeness. Specifically, a thread could be in a function prologue, a function epilogue, a call sequence (some arguments pushed on the stack), a return sequence (the results not yet popped off the stack), or in a tail-call optimized version of one of the above.

The real source of the strangeness is that the frame pointer, the stack pointer, or both are not where they usually are. So hopefully, the run-time system can take the difference between the two and use this difference to determine the correct state of the frame. For example, suppose we are in a call or return sequence. Then the difference should be too high and we could conservatively treat the extra slots (the arguments or results to the callee) as live.

A standard function epilogue has only one strange position, between the `leave` and `ret` instructions. In this case, the simplest solution may be for the run-time system to check the instruction stream for this case. Similarly, function prologues use instructions that do not appear elsewhere.

Tail calls may be more problematic. Currently, their code generation is broken for other reasons, so we will re-examine the tail-call case later.

The problem with all these "corner cases" is that they make rather brittle assumptions about code generation. The alternative, however, seems to be including a lot of extra bloat in the PC-map tables. In the long run, it is probably easiest to manage this bloat. For example, we could use one uncommon-level bit to indicate "this is a corner case" and then a few bits to indicate how far off the frame pointer is from where it should be.

Chapter 11

PC maps

It is necessary for the compiler to communicate information about the liveness and types of registers and stack-frame slots to the run-time and garbage collector. We use *PC maps*, which are a mapping from code addresses to liveness and type information, for this purpose. The compiler generates the PC maps (see Section 11.2) based on a static analysis of the code and the run-time system uses the static information from the PC map, along with the actual values in the live registers and stack-frame slots, to determine heap roots etc.

The original design and implementation of PC maps was done by Dan Grossman (Cornell University), this chapter describes a revised design.

11.1 The PC map representation

Each object file produced by the MOBY compiler contains a PC map fragment for the code in that file. The run-time system constructs the global PC map from these fragments (see Section 11.3). A PC map fragment has three levels. The top level is called the *cluster level*, the the middle level is also called the *block level*, and bottom level is also called the *instruction level*. The instruction level is not generated is only generated for multi-threaded code (see Section **??**).

11.1.1 The cluster level

A cluster-level entry contains the following fields:

`_start`
: The address at which the cluster begins.[1]

`_end`
: The address at which the cluster end.[2]

[1]By begins, we mean the lowest address; it need not be the entry point.
[2]By begins, we mean the highest address; it need not be the entry point.

```
typedef struct {
    addr_t              _start;
    addr_t              _end;
    pcmap_block_tbl_t   *_blkTbl;
#ifdef MOBY_MULTITHREADED
    pcmap_delta_t       *_deltaTbl;
#endif
    uint32_t            _frameSz;
    uint32_t            _fpSpillOffset;
    uint32_t            _fpSpillSz;
} pcmap_cluster_t;

typedef struct {
    uint32_t            _nClusters;
    pcmap_cluster_t     _clusters[1];
} pcmap_t;
```

Figure 11.1: Cluster-level PC map data structures

`_blkTbl`
: The address at which the block-level table for this cluster begins.

`_deltaTbl`
: This field contains the address of the start of the instruction-level liveness-delta information. It is not present in the single-threaded version of the run-time system.

`_frameSz`
: The number of words between the frame pointer and the top of the activation record for this cluster.

`_fpSpillOffset`
: The number of words between the frame pointer and the beginning of the floating-point spill section of the activation record.

`_fpSpillSz`
: The number of floating-point spill slots for the activation record.

In general, the cluster level describes the activation record, the encoding of lower-level tables, and the location of lower-level tables.

11.1.2 The block level

For each cluster in the object file, there is a block-level table that contains liveness and type information at specific code addresses. A block-level entry contains the following fields:

`_pcOffset`
: A 16-bit byte offset from the cluster-level `_start` address that specifies the address of "block."

```
typedef struct {
    uint16_t                _pcOffset;
#ifdef MOBY_MULTITHREADED
    uint16_t                _deltaTblOffset;
#endif
    uint8_t             _blockKind;
    uint8_t                _live[LIVE_VEC_SZ];
} pcmap_block_t;

typedef struct {
    uint32_t            _nEntries;
    pcmap_block_t       _entries[1];
} pcmap_block_tbl_t;
```

Figure 11.2: Block-level PC map data structures

`_deltaTblOffset`
: A 16-bit word offset from the cluster-level `_deltaTbl` address to the instruction-level liveness-delta information for this block. It is not present in the single-threaded version of the run-time system.

`_live`
: A liveness vector that specifies the state of the general-purpose registers and spill slots at the given program point.

Since the `_pcOffset` and `_deltaTblOffset` fields are limited to 16-bits, it is possible that some clusters will be too large to describe. In such cases, we plan to introduce multiple cluster entries.

11.1.3 The instruction level

For multithreaded MOBY programs, we want to support preemption at any instruction. Our approach to this problem is based on Shivers *et al.*'s design for "atom heap transactions" [SCM99].

For each instruction, there is a PC-map delta, which is represented by a variable-length sequence of one or more bytes. The first byte is a header byte, which is divided into four count fields as follows:

bits	`purpose`
0-1	number of "ignore" indices
2-3	number of "pointer" indices
4-5	number of "pointer or enum" indices
6-7	number of "any" indices

Following the header are the sequences of indices, first for slots to ignore, then for pointer slots, *etc.*. A slot index is represented by one or two bytes. For slot indices in the range $0 - 127$, one byte is used to directly represent the index, while for a slot index i in the range $128 - 32767$, we store $128 + (i/256)$ in the first byte and $i \mod 256$ in the second byte.

11.2 Generating the PC map

11.3 Interpreting the PC map

Chapter 12

The MOBY run-time system

12.1 Run-time representations

This section describes the run-time representations of various kinds of MOBY data values. We use C-like type and expression syntax to describe these representations, where the C type `MobyWord_t` is the unsigned integer type that is the same size as a pointer.

12.1.1 Function closures

Function closures are represented by a two-word record object: the first word holds the function's code addressand the second word holds the function's environment pointer. Note that the environment pointer is not always a pointer — if the function's environment consists of a single value that has uniform representation, then the environment pointer will be that value, and if the environment is empty, then the environment pointer will be zero.

12.1.2 Objects and method suites

Objects are represented as a pair of the object's *data pointer* and the object's *class ID* (see Figure 12.1. The object's data pointer points to a record whose first field is a pointer to the object's *method suite*, and whose other fields contain the object's fields. Every object generated from a particular class shares the same method suite. The object's class ID is used as a key when dynamically searching for the field or method slot for a given label.

When executing code inside a method body, we do not need the class ID to locate slots, since the slot offsets are static w.r.t. the particular class. Therefore, we represent **`self`** by just the pointer to the object's data inside methods (and when we pass **`self`** to a method's body during dispatch).

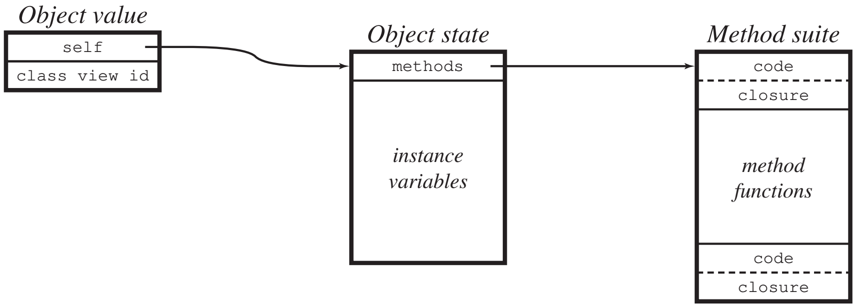

Figure 12.1: Object representation

12.1.3 Tagtypes

Tag types are implemented using a scheme described by Reppy and Riecke [RR96]. For each declared tagtype, there is a statically allocated descriptor with the following layout:

```
typedef struct MOBY_tagtype_desc *MOBY_tagtype_t;

struct MOBY_tagtype_desc {
    MobyWord_t          _depth;
    MOBY_tagtype_t      _id[N];
    char                _name[];
};
```

where `N` is the depth value for the tagtype. The address of the tagtype's descriptor serves as its unique ID. The `_id` field of the descriptor holds the IDs of the ancestors of the tagtype, with `X->_id[X->_depth-1]` holding `X`, for any tagtype descriptor `X`. Following the ancestor IDs is a null-terminated string, which is used by the run-time system to report the name of exceptions.

We can test if a tagtype value matches the constructor `X` as follows:

```
((tag->_depth >= X.depth) && (tag->_id[tag->_depth] == X))
```

where `tag` is the value's ID.

12.1.4 Arrays and vectors

We use a two-level representation of arrays, strings, and vectors.

12.2 Runtime system data structures

12.3 Garbage collection

We are using a modified version of Morrisett and Smith's *Mostly-copying Collector* (MCC) [SM97]. Although we have maintained much of the algorithm's structure, we have made liberal modifications

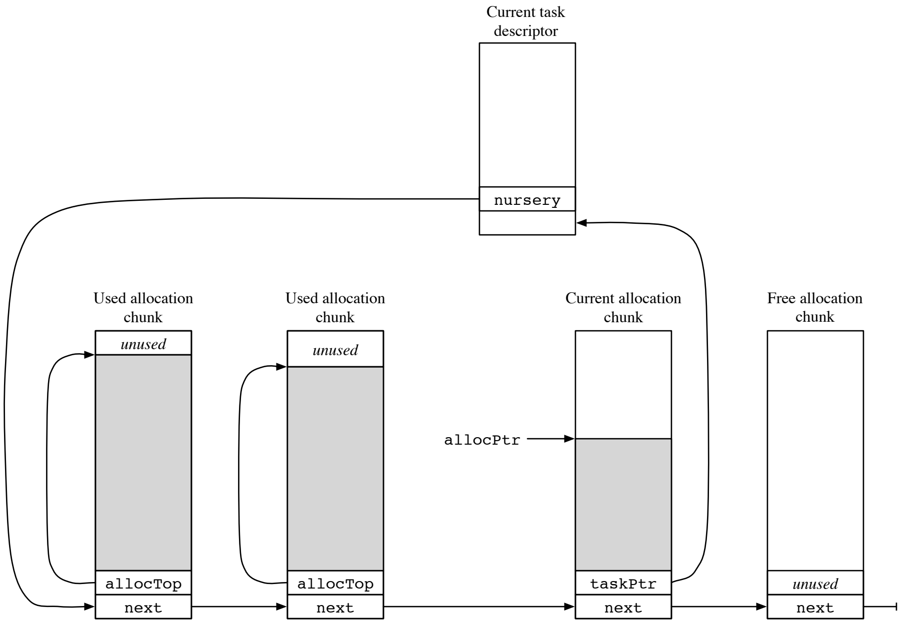

Figure 12.2: Per-task data structures

to the source code.

In this section, we describe only the single-threaded version of the garbage collector. By single-threaded, we mean the version that works only for Moby programs that do not spawn additional threads. Both versions of the collector run in one thread (they are not parallel) and must run while all mutator threads are stopped (they are not incremental). The next section explains how the multi-threaded version is different. Because the multi-threaded version is less stable, we currently have two source trees, one for each version. In the future, we hope to merge them, turning off expensive synchronization operations for single-threaded programs.

MCC has conservative objects and record objects. (It also has arrays and large objects.) Both objects have 32-bit headers. Conservative objects' headers just record the size of the object; we do not know which of the fields hold pointer values. Record objects' headers have a bitmask — a 1 means a field is definitely a pointer, a 0 means a field is definitely not a pointer. Unfortunately, this encoding means an object with just one ambiguous field must be given a conservative header, so all fields are considered ambiguous. So in the future, we intend to change the header encoding for record objects to use two bits per object field. Then one of the four bit patterns can indicate that a particular field is ambiguous.

The glue code to transfer control between the mutator and the collector and back is new and subtle, so we discuss it in detail. (It is also rather different for multi-threaded programs, so read the next section for that case.) The biggest difference from the original MCC is that the allocation pointer does not stay in a global variable.

At program initialization time, the Moby runtime calls `gcInit`. This function gets an initial heap, breaks it up into blocks, breaks the blocks up into pages, etc. Unlike the original MCC, it does not establish an allocation pointer. (In MCC terminology, the entire space for collectable objects is called the *heap*. It is broken up into contiguous-memory segments called *blocks* and the blocks are broken up into *pages*. Large objects can span pages, but not blocks. Small objects do not span pages. Pages can be used for mutator objects or for system information about other pages. Page size is a power of 2 (currently 8K for Moby) and pages are aligned on the same boundary.

`gcNewAllocPtr` returns a pointer to a fresh to-space page. The mutator is then responsible for allocating off this pointer. It should not let the pointer leave the page.

Each module's initialization code registers its statically allocated pointers by calling `MOBY_add_staticseg` so that they will be scanned during garbage collection. The collector assumes the values in such a segment are *unambiguous* pointers, so they can be small values, but they cannot be integers that could appear to be pointers in the Moby heap. Therefore, the compiler segregates top-level bindings by pointerness. This segregation works because top-level bindings are monomorphic, but it may make functors harder to compile. We may want a third category for the segregation for ambiguous pointers.

Because we expect allocation performance to be crucial for Moby applications, the compiler reserves a register (`edi` on the IA-32) to hold the allocation pointer while Moby code is running. The Moby code does heap-limit checks to ensure that there is room for ensuing allocations on the current page. (The placement of these checks is currently broken because it does not take into account that function calls can perform an arbitrary amount of allocation. That's the compiler's problem.)

When a heap-limit check fails, the mutator calls `MOBY_callgc`, written in assembly using a special calling convention. Basically, the function stores all the registers on the stack and then calls the C routine `MOBY_do_gc`. It is not necessary to preserve all the registers; indeed, the multi-threaded version does something more sophisticated. In other words, this assembly code should be improved for the single-threaded version.

The C routine then moves the registers from the stack into a global data structure and transfers control to the MCC routine `gcAllocSpace`. (Again, all this stuff is cleaner in the multi-threaded version, but we refrain from cleaning up the single-threaded case for fear of introducing bugs.) This routine marks the rest of the old page as unused, grabs a new page, and returns a pointer onto this new page. Upon return, `MOBY_callgc` moves this pointer into register `edi` and restores all the other registers.

The slow path, of course, is if the fast path described above has been taken enough times that a full garbage collection is warranted. The biggest difference in collection between Moby and the original MCC is how the roots are scanned. Therefore, the `scanRoots` function is re-implemented in the Moby run-time.

`scanRoots` first crawls the stack and then scans the registered static segments. *This order is very important.* MCC assumes that all ambiguous pointers are scanned before any unambiguous pointers. Otherwise, we may move an object that must be pinned. The order works because our current information about the stack requires us to treat all live locations as ambiguous pointers. In the future, we might put information in the PC map to indicate that some slots were live, unambiguous pointers. *Doing this naively will break the collector.* After all, it would break the invariant that all ambiguous roots are scanned first. The simplest fix is probably to maintain a stack of unambiguous roots and do them after all stack walking is done. Another option is two stack walks.

Stack-crawling must know where to stop. We currently use a straightforward kludge: Moby code is called from C code, so we will reach this C code's activation frame before "falling off" the deep-end of the stack. We can detect the C code because Moby frames have return addresses that appear in the PC map. So once a return address lookup fails, we stop. A more robust solution is to record the deep-end of the Moby stack. The trick here is to get it exactly right when transferring control to Moby code; some hand-written assembly may be necessary. Also, a more complicated solution will be necessary if we allow C code to "callback" into Moby code.

When we are done with the collection, we put the allocation pointer on a fresh page and return it. Note that the multi-threaded version does not move any allocation pointers during collection.

Because Moby code keeps its allocation pointer in a register, it is inconvenient to have foreign code allocate off of this pointer. Therefore, the collector maintains another allocation pointer, called `specialAllocPtr` in a global variable. Calls to malloc-replacement functions, such as `gcMalloc`, use this pointer to do allocation. The collector just needs to put this pointer onto a fresh page after collection is complete.

12.4 Multithreading

Note: the multithreaded version works fine for single-threaded programs, so we encourage using it unless bugs are encountered or performance demands otherwise. (The degraded performance would involve managing some unneeded mutexes, maintaining some unneeded global state, and enduring some unneeded function call indirections.) In fact, this situation is the only one that has been tested. That is, we have not yet invoked multiple threads with enough allocation to trigger a collection.

Currently, each Moby thread runs in its own Posix thread (pthread). Some day we hope to multiplex multiple Moby threads in a pthread, but we ignore that complication here and just focus on the garbage-collection issues. (A major reason we temporarily abandoned this multiplexing is that the current Linux pthread implementation determines thread identity by masking the stack pointer. Newer versions have an implementation that does not rely on register `esp`.) Be warned that by default, these threads have only 2 megabyte stacks.

For each thread, there is a Moby context (`moby_exec_ctxt_l_t`) structure. All the contexts are kept in a doubly-linked list that is reachable through the global `moby_runtime_info`. Each thread also keeps a pointer to its own context in thread-local-data (using the key `MobyCtxtKey`). But wait, there's more. Concerned that accessing thread-local-data is too expensive, we also put a pointer to the context at the bottom of the page into which the context's allocation pointer currently points. So we can mask the allocation pointer to get the context pointer. (To be precise, we write a record object with a header indicating one non-pointer field and then the context pointer. It is a non-pointer in the sense that it does not point into the Moby heap.) When a thread gets a new page for allocating, the context pointer must be copied into that page.

Access to the global MOBY runtime information is guarded by a single, global lock. We need the lock only for module initialization, thread creation, thread destruction, and (full-blown) garbage collection, so we don't expect this to be a major bottleneck. However, when we support lighterweight threads (much lighter than a pthread), we probably should strive not to need hold the global lock just to create and destruct such a lightweight thread.

We now describe the fields of a `moby_exec_ctxt_l_t` structure:

```
typedef struct moby_exec_ctxt_l {
  greg_t      saved_alloc_ptr;
  stk_ptr_t   saved_stack_ptr;
  stk_ptr_t   saved_frame_ptr;
  code_ptr_t  saved_ret_addr;
  ucontext_t * uctxt;
  int         gc_instigator;

  moby_thd_status_t thd_status;
  pthread_t         pthread;

  MOBY_closure_t *  closure;

  struct moby_exec_ctxt_l * prev;
  struct moby_exec_ctxt_l * next;
} moby_exec_ctxt_l_t;
```

The `prev` and `next` fields establish a doubly-linked list. Code modifying this list should hold the global MOBY info lock.

The `closure` field initially holds a pointer to the closure that a thread first executes. This field is necessary because the closure lives in the Moby heap: If a garbage collection occurs between the time the thread is created and when it starts running, then the closure field must be treated as a root and updated by the collector. Once the collector detects that the Moby thread has started running, it assigns `NULL` to this field.

The `pthread` field is what you think it is. When the garbage-collection instigator stops the world, it does so by walking through the list of contexts and sending a signal to each pthread encountered, *except for itself*.

The `gc_instigator` field is set only for the thread that instigated the garbage collection. The stackwalk must treat this thread differently for a couple of reasons. First, we can find the callee-save registers and Moby stack portion in known stack offsets. Second, the `uctxt` field is not correct because we do not signal ourselves (see below). Now that I think about it, we don't need to maintain this state because the fragment

```
pthread_equal(pthread_self(),ctxt->pthread)
```

should suffice. (This fragment works because the collection runs in the instigator's thread.)

The `thd_status` field should be deleted. We need something like this field when we do our own scheduling, but the operating system currently does all of the scheduling for us.

The uctxt field points to space that is allocated during thread-creation and destroyed during thread-destruction. The signal handler for an interrupted-for-garbage-collection thread copies the uctxt provided to it by the system into this field. (The signal handler kludges are described in more detail below.) Then on restarting, we somehow use this information to perform a setcontext. (This function is currently *unimplemented* in Linux, but we have some ideas on workarounds.) Note that this field is not correct for the instigating thread.

The saved_* fields are written by the trampoline code that is used to let Moby code call C code. In the single-threaded case, Moby code can call C code directly, especially because Moby has strictly fewer callee-save registers (namely, none). Also, for the time being we might assume that the C code is not allocating into the Moby heap, so we are assured that a garbage collection will not occur while executing C code. (If it can occur, we probably need trampoline code because C code does not use MOBY_callgc.) But in the multi-threaded case, another thread could instigate a garbage collection. We use the saved information to determine the Moby portion of the stack and the allocation pointer. Then the stack portion above (shallower) than the Moby portion is treated conservatively.

The trampoline code is in the file MOBY-ccall.S. The compiler uses the multithreaded flag (-t) to decide to generate C function calls to go through this code instead of directly to C. Also, the multithreaded version of MOBY_callgc actually calls the trampoline code, and *relies on its implementation*. Specifically, it stores the callee-save registers (ebx and esi) on the stack. Later, the stack-walker for the instigator will use the saved_stack_ptr field to find these values. This trick avoids the need to scan the C portion of the instigator's stack.

Now we can describe how garbage collection occurs. First we describe the synchronization and signalling. Then we describe how the actual collection process is different from the signal-threaded case.

Recall that most calls to gcAllocSpace take the fast path of finishing up the old page, grabbing a free page, writing the context-pointer record into it, and returning a pointer. These last three steps constitute a critical section for two reasons. First, we cannot have concurrent access to the free list (for grabbing a free page). Second, we cannot have a full-blown collection occur before the context-pointer record is written and the allocation pointer updated. Hence the call to collect is also in this critical section. We protect gcNewAllocPtr with the same lock because it also manipulates the list of free pages. In the future, we intend to use a (faster?) spinlock for this code because we do not expect much contention.

Once a thread decides to do a full-blown collection, it is the instigator. No other thread can also be instigating because of mutual exclusion. To stop the world, the instigator sends a signal to every other thread. It then waits for all of the threads to receive the signal, save appropriate state in their context, and wait to be restarted. The synchronization to do this stopping stage is essentially the mutex encoding of a counting semaphore. The synchronization for restarting is essentially the inverse — we set a start flag and broadcast to the other threads. (These other threads check the flag before stopping in case collection happens "really fast.") *None of this synchronization has been tested, of course.*

Note that the description above implies that signal handlers are performing synchronization operations. The pthreads implementation forbids doing so and uses scary words like "deadlock" and "unpredictable" to dissuade us. Therefore, we pull a dirty trick. After the handler copies the ucontext into its thread's context record,[1] it updates the system's ucontext's program counter so that the handler will return somewhere else, specifically the C function `stop_for_gc`.

Therefore, `stop_for_gc` must not take any arguments! Also, no function can ever have data at negative offsets from the stack pointer! We access the context pointer using thread-local data and do the synchronization described above. Upon restart, we use `setcontext` along with the saved `ctxt` field to restore things how they were. *The memory management of uctxt here is quite subtle.* Why don't we pass `setcontext` the actual field from the context record, not a copy? If `setcontext` frees this memory (it probably doesn't), the next collection will have a memory error. If `setcontext` does not free the memory, then we have a race condition: Another collection might occur before `setcontext` is done reading the `uctxt` field. When I think about what would happen as a result, my head feels fuzzy. The other option is to pass a copy of the `uctxt` to `setcontext`. But now we have a memory leak if `setcontext` does not free the memory. So we need to stack allocate this copy. Particularly subtle is that this structure is not flat; the pointer field must also refer to a stack-allocated object. But if `setcontext` does free the memory, we're hosed because it is stack-allocated. *The bottom line is we need to ensure that passing a stack-allocated parameter to `setcontext` is the right thing to do.*

Once the world is stopped, the actual garbage collection is much like the single-threaded case. Of course, we may have more than one stack to crawl and only one stack actually instigated the garbage collection. Starting a stack crawl is difficult because of pre-emption and is probably bug-ridden. There are five cases:

- The thread is the instigator. The top of the Moby portion of the stack is in the context record. The callee-save registers are in a known position. The C portion of the stack contains no roots.
- The thread is running Moby code. The context record is not accurate. We may be in an allocation sequence; if so we roll back the program counter because the already-initialized fields may have moved.
- The thread is running C code. The context record is accurate. Treat the C portion of the stack conservatively; doing so ensures we trace the callee-save registers.
- The thread has not started running yet. Trace the `closure` field only. The stack has no Moby roots.
- The thread is in the trampoline code. How to find the necessary roots depends on the exact value of the program counter. *This case is unimplemented.*

The other main difference in the multi-threaded collector is that we do not adjust any allocation pointers. For this policy to be correct, it suffices to pin the pages into which allocation pointers point. So if there are n contexts, we will pin an additional $n + 1$ pages (we still have the `specialAllocPtr`). The obvious disadvantage is less compaction. The advantage is that allocation pointers are not volatile; they are never changed by another thread. Changing them wouldn't

[1] We use `pthread_getspecific` for this operation; the pthread implementation suggests this operation is safe in a handler even though the documentation does not specifically permit it.

be so bad if they always lived in register `edi`. But with foreign calls and concurrent run-time calls, it gets very difficult to always track where a context holds its "true" allocation pointer. For example, imagine if another thread is in `gcAllocSpace` when it is pre-empted. Of course, we could have larger critical sections instead. Pinning allocation-pointer pages turned out to be a simple solution and a pleasing interaction with the mostly-coping concept.

Note that by not moving allocation pointers, it would be fairly simple to avoid the need for rollback. All we need to do is trace the already-initialized fields as roots. However, the real advantage of rollback comes when we have multiple Moby threads per context, all *sharing* an allocation pointer. Here, rollback is a duty of the thread scheduler anyway.

There are two unfortunate properties of MCC that may prove problematic for us. The first is that conservative objects (and, by extension, objects with ambiguous pointer fields) are maintained in an explicit data structure so that collection only has to traverse the pointer graph once. (This distinction is the primary difference between MCC and Bartlett's collector.) Therefore, allocating such objects is likely to be too complicated to do with in-line Moby code. Also the accessing the explicit data structure must by synchronized. *BUG: The current code does not synchronize access to the conservative objects queue.* A better solution, though, is probably to have a separate queue per context, thus avoiding the need for synchronization.

The second is that part of determining if an ambiguous pointer is valid involves testing whether it points to the beginning of an object. To detect object beginnings, the collector must walk from the beginning of a page to the beginning of the object. Every ambiguous pointer that is a pointer suffers this walk. Two changes suggest that this overhead will be much worse for us than for the original MCC. The first is that we use a larger page size. The second is that we intend, at least at first, to consider all polymorphic fields to be ambiguous pointers. (MCC's original mutator created conservative objects only in C code.)

The MCC code base has some support for using bitmasks to determine object beginnings. (So for each word — or double-word if objects are so aligned — have one bit indicating whether it is the beginning of an object.) I think these are a good idea, but the MCC code indicates that this feature is currently broken. It's probably worth fixing and/or finding out why it wasn't used. Another option, of course, is to be more conservative and conclude an ambiguous pointer might be a pointer as soon as it points into the Moby heap with correct alignment. The disadvantage here is more false pointers.

Chapter 13

The MOBY compilation environment

The MOBY compiler is a batch compiler that processes one file at a time. At the same time, it implements a language that requires intermodule typechecking. To support intermodule typechecking, as well as the propagation of other kinds of information between compilation units, the MOBY compiler relies on a *persistant* compilation environment. In the current implementation, the compilation environment is stored in MBI files and the persistence is provided by the host file system.

13.1 Groups and the filemap

MOBY programs are organized into groups of modules and signatures (sometimes these groups are called libraries). For each group, there is a *filemap* that maps the signature and module names of the group to its source file. Effectively, the filemap serves as the top-level index for the group.

A filemap is a text file with the following simple syntax:

FileMap
  ::= (*UnitSpec FileName* `;`)*

UnitSpec
  ::= **`module`** *Identifier*
  | **`signature`** *Identifier*

While it is possible to create and edit a filemap using a text editor, they are usually generated by the **moby-depend** tool.

13.2 MBI files

MBI files are binary files that contain information about previously compiled MOBY code and MOBY libraries. Information contained in an MBI file includes:

- pre-compiled MOBY signatures.
- the typed AST for functors.
- the public interface to a precompiled MOBY module.
- the BOL representation types for abstract MOBY types.
- the BOL code for MOBY functions.
- overloading specifications.

An MBI file is structured as an indexed collection of varying-length records. At the beginning of the file is a header that provides summary information about the MBI file (*e.g.*, target architecture, is multi-threading supported, *etc.*). Following the header is a dictionary that maps *stamps* to the records that make up the content of the MBI file. Each dictionary entry contains three pieces of information: the name of the entry, the kind of the entry, and its byte offset in the file.

[[ASDL]]

[[give a list of different entry kinds]]

For example, consider the following small MOBY module:

```
module M {
    datatype T { C of Int }
    val x : T = C(5)
  }
```

It can be represented by the following MBI file:

Stamp	Name	Kind	Contents
0	M	DK_MODULE	decls: [1, 5, 6]
1	T	DK_TYPE	TD_DATA: owner: 0 cons: [(C, [T_CON(3, [])])]
2	Int	DK_XTYPE	owner: 3
3	Int	DK_XMODULE	owner: 4
4	moby-basis	DK_LIBRARY	guid: "..."
5	C	DK_CON	DC_DATA: stamp: 1 id: 0
6	x	DK_VAL	owner: 0 ty: T_CON(1, [])

The first entry in this MBI file is for the module M. It contains a list of the names declared in the module (this list includes the constructor C, which is part of T's declaration). The second entry is for the datatype T; this entry contains the type's owner (M) and a list of the constructors. For each constructor, we have the constructor name with its list of argument types. In this case, there is one constructor (`C`) with an argument type of `Int`, which is described by the third entry. The `Int` type is defined in another MBI file, so it's kind is `DK_XTYPE` and it's entry contains the stamp of its owner, which is the externally defined `Int` module. The fourth entry describes the `Int` module as being defined in the `moby-basis` library, which is described by the fifth entry. For the `moby-basis` library, the MBI file records is GUID (Globally Unique ID). The full name of the `Int` type is then the pair of its library's GUID and the path `Int.Int`. These full names are globally unique and are used to preserve sharing between modules that might reference the same external types. The sixth entry describes the constructor `C`; the entry's contents lists the stamp of the owning datatype and the index of the constructor in the datatype's list of constructors. The seventh, and last, entry describes the value identifier `x`. Its contents includes the owner (`M`) and the type of the identifier.

[[BOL]]

[[Overloading]]

13.3 MBX files

This section, describes the syntax of MBX files, which are an ASCII representation of MBI files, and which are used to implement "native" MOBY modules. The **gen-mbi** tool translates MBX files to their binary MBI representation and the **dump-mbi** prints a textual representation of an MBI file.

13.3.1 Basic structure of an MBX file

An MBX file consists of one or more *units*. There are two kinds of units: a *module*, which defines the interface and possibly implementation of a MOBY compilation unit; and a *pervasive unit*, which defines top-level bindings and overload schemas.

MBXFile
 ::= (MBXUnit)$^{+}$

MBXUnit
 ::= **`%module`** *ModuleId Guid* **`{`** Decl* **`}`**
 | **`%pervasive`** *Guid*^{opt} **`{`** PervDecl* **`}`**

Decl
 ::= TypeDecl

PervDecl
 ::= OverloadDecl

13.3.2 Overloading specifications

OverloadDecl
::= **%overload** *ValId* **:** *TyScheme*
| **%bind** *ValId* **:** *TyScheme* **=** *ValSpec*

13.3.3 Specifying types

TypeDecl
::= **%type** *TypeId* *TyParams*^{opt} **=** **%prim** *BOLType*
| **%type** *TypeId* *TyParams*^{opt} **=** *Type*
| **%enumtype** *TypeId* **{** *NullaryDataCons* **}**
| **%datatype** *TypeId* *TyParams*^{opt} **{** DataCons **}**
| **%objtype** *TypeId* *TyParams*^{opt} **{** ObjMemberList **}**
| **%tagtype** *TypeId* *TyParams*^{opt}
| **%tagtype** *TypeId* *TyParams*^{opt} **%of** OneOrMoreTys
| **%tagtype** *TypeId* *TyParams*^{opt} **%extends** NamedTy
| **%tagtype** *TypeId* *TyParams*^{opt} **%of** OneOrMoreTys **%extends** NamedTy

13.3.4 Specifying BOL types

The BOL representation has a simple (and incomplete) type system that is described in Section 6.1. BOL types are the primitive types in the MOBY compiler and are very close to machine representation. We use MBX files to establish a binding between abstract MOBY types, such as `Int`, and their underlying machine representation as described by a BOL type. The syntax of BOL types in an MBX file is as follows:

BOLType
::= **%any**
| **%unit**
| **%bool**
| **%int8**
| **%int16**
| **%int32**
| **%int64**
| **%integer**
| **%enum (** *Num* **,** *Num* **)**
| **%float**
| **%double**
| **%extended**
| **%ptr** *BOLHeapType*
| **%seq** *BOLHeapType*

BOLHeapType
  ::= **%object**
  | **%closure**
  | **%struct** *Num* **(** *BOLFieldType* (**,** *BOLFieldType*)* **)**

BOLFieldType
  ::= *Num* **:** *BOLType* (**[** *Num* **]**)^{opt}

13.3.5 Value bindings

ValueDecl
  ::= **%val** *ValId* **:** *TyScheme* (**=** *PrimValue*)^{opt}
  | **%val** *ValId* **:** *TyScheme* **=** *ValSpec*

PrimValue
  ::= **%prim** *PrimOp*
  | **%prim** *BOLFunction*

ValSpec
  ::= Path^{opt} *ValId*

The form "**%prim** *PrimOp*" is syntactic sugar for a function definition that applies *PrimOp* to its arguments. For example, the definitions

```
%val add : (Int, Int) -> Int = %prim I32Add
%val div : (Int, Int) -> Int = %prim I32Div
```

are expanded into the following:

```
%val add : (Int, Int) -> Int =
        %prim add (a : %int32, b : %int32, ex) {
          %let c : %int32 = I32Add(a, b)
          %return (c)
        }
%val div : (Int, Int) -> Int =
        %prim div (a : %int32, b : %int32, ex) {
          %let c : %int32 = I32Div(a, b, ex)
          %return (c)
        }
```

Note that in the case of the `add` function, the exception handler continuation is ignored, whereas it is passed as an argument to the `I32Div` operator in the case of the `div` function. This specialization is supported by comparing the arity of the MOBY type on the lhs with the arity and signature of the operator on the rhs.

13.3.6 Specifying BOL code

BOLFunction
  ::= *Var Params BOLBlock*

BOLBlock
  ::= **{** *BOLStmt* **}**

BOLExpr
  ::= *BOLBlock*
  | *TailExpr*

BOLStmt
  ::= *TailExpr*
  | **%let** *Binding BOLStmt*
  | **%fun** *BOLFunction* (**%and** *BOLFunction*)* *BOLStmt*
  | **%cont** *BOLFunction BOLStmt*
  | **%stack** *Var* **(** *Int* **,** *Int* **)** **:** *BOLType BOLStmt*

TailExpr
  ::= **%if** *Var* **%then** *BOLExpr* **%else** *BOLExpr*
  | **%switch** *Var Case*$^+$ (**%default** **:** *BOLExpr*)^{opt}
  | **%call** *Var* **(** *Var* (**,** *Var*)* **)**
  | **%ccall** *CFun Convention* **(** *Var* (**,** *Var*)* **)**
  | **%throw** *Var* **(** *Var* (**,** *Var*)* **)**
  | **%return** *Terms*

Case
  ::= **%case** *Num* **:** *BOLExpr*

Binding
  ::= *Params* **=** *BOLExpr*
  | *Param* **=** **#** *Num Var*
  | *Param* **=** **%alloc** **(** *BOLField* (**,** *BOLField*)* **)**
  | *Param* **=** *BOLTerm*
  | *Param* **=** **%meth** *Var* **.** *Label*
  | *Param* **=** **%field** *Var* **.** *Label*
  | *Param* **=** **%meth** *Var*
  | *Param* **=** **%field** *Var*
  | *Param* **=** **%meth** *Var* **@** *Var*
  | *Param* **=** **%field** *Var* **@** *Var*
  | **_** **=** **%field** *Var* **@** *Var* **:=** *Var*

BOLTerm
  ::= *PrimOp BOLTerms*
  | *Var*
  | *Constant*

BOLTerms
  ::= **(** (*BOLTerm* (**,** *BOLTerm*)*)^{opt} **)**

BOLField
  ::= *Num* **:** *Var*

Params
  ::= **_**
  | *ParamList*

ParamList
  ::= *Param*
  | **(** *Param* (**,** *Param*)* **)**

Param
  ::= *Var* (**:** *BOLType*)^{opt}

13.3.7 The `comp-env` library

The `comp-env` library (located in `src/comp-env`) provides low-level support for reading and writing MBI files.

13.3.8 The gen-mbi tool

13.3.9 Future directions

The current scheme does not allow generation of object files from MBX specifications; all code can only be accessed via inlining. This restriction may be too limiting for supporting the MOBY foreign interface.

Chapter 14

The code-generator generator

We use a tool for generating significant portions of the MOBY compiler's BOL optimizer and code generator.

14.1 The specification language

Specification
  ::= *Header* Spec^{+}

Spec
  ::= *ForBinding** *Operators*
  | *ForBinding** *RewriteRules*
  | *ForBinding** *Translation*

ForBinding
  ::= **`for`** **<** *Id* (**`,`** *Id*)* **>** **`in`** **`{`** *ExpansionGrp* (**`,`** *ExpansionGrp*)* **`}`**

ExpansionGrp
  ::= **<** *Expansion* (**`,`** *Expansion*)* **>**

Expansion
  ::= *Id*
  | *String*

RewriteRules
  ::= **`rules`** *RewriteRule*$^{+}$

RewriteRule
  ::= *ForBinding** *Pattern* *WhenClause*^{opt} **`=>`** *Term* *WhereClause*^{opt}

Pattern
  ::= *Operator* **(** *Pattern* (**,** *Pattern*)* **)**
  | **?** *Id*
  | **_**

Term
  ::= *Operator* **(** *Term* (**,** *Term*)* **)**
  | **!** *Id*

Operator
  ::= *Id*
  | **<** *Id* **>**

WhenClause
  ::= **when** *Expression*

WhereClause
  ::= **where** (*Id* **=** *Expression*)+

Expression

14.2 Generating the rewriting rules

For example, the following three rewrite rules:

```
IAdd32(IConst ?a, IConst ?b) => IConst(!c) where c = CA.iadd32(!a, !b)
IAdd32(a, IConst 0) => a
IAdd32(IConst 0, b) => b
```

are translated to the following SML code:

```
PRIM(IAdd32(x1, x2)) => (case (bind x2)
   of (IConst b) => (case (bind x1)
         of (IConst a) => let
              val c = CA.iadd32(a, b)
              in
                dec x1; dec x2; OK(IConst c)
              end
          | a => if CA.eq(CA.iconst_0, b)
                then (dec x2; OK a)
                else FAIL
        (* end case *))
    | _ => (case (bind x1)
         of (IConst a) => if CA.eq(CA.iconst_0, a)
              then (dec x1; OK b)
              else FAIL
          | => FAIL
        (* end case *))
  (* end case *))
```

In this code, the `bind` function maps variables to their binding expression and the function `dec` decrements the argument usage count of a variable.

Chapter 15

The gen-mbi tool

The **gen-mbi** is used to compile an ASCII representation of MBI files to a binary representation. A description of the tool can be found in Section 13.3.8; here we describe its implementation.

15.1 Overview

15.2 Parsing

15.3 Translation

15.4 Output

Bibliography

[AR92] William E. Aitken and John H. Reppy. Abstract value constructors: Symbolic constants for standard ml. Technical Report TR 92-1290, Department of Computer Science, Cornell University, June 1992. A shorter version appears in the proceedings of the "ACM SIGPLAN Workshop on ML and its Applications," 1992.

[Ler90] Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA, February 1990.

[Pet92] Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In *Fourth International Conference on Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 258–270, New York, NY, October 1992. Springer-Verlag.

[PPS96] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 1–12, May 1996.

[RR96] John H. Reppy and Jon G. Riecke. Simple objects for SML. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 171–180, May 1996.

[SCM99] Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, September 1999.

[SM97] Frederick Smith and Greg Morrisett. Mostly-copying collection: A viable alternative to conservative mark-sweep. Technical Report 97-1644, Department of Computer Science, Cornell University, August 1997.