The Implementation of HashCaml

John Billings, Peter Sewell, Mark Shinwell and Rok Strniša

Computer Laboratory, University of Cambridge

http://www.cl.cam.ac.uk/users/pes20/hashcaml/

$Date: 2006-04-14 02:12:41 +0100 (Fri, 14 Apr 2006) $: $Rev: 773 $:

Contents

1 Introduction **4**
1.1 Overview of the compiler . . . 4

2 Language features and constructs **5**
2.1 Type-safe marshalling . . . 5
2.2 Runtime type representations for abstract types . . . 6
2.3 Type representations . . . 7
2.4 Names . . . 7
2.5 Other Acute constructs . . . 8
2.6 Compiler options . . . 8

3 Type-passing translation **10**
3.1 Value restriction . . . 10
3.2 Rewriting on typed syntax trees . . . 13
3.2.1 Translation of value identifiers . . . 13
3.2.2 Translation of non-recursive let bindings . . . 14
3.2.3 Translation of recursive let bindings . . . 15
3.2.4 Translation of record fields . . . 16
3.3 Rewriting on lambda-code . . . 17
3.4 Coercion wrapper insertion . . . 18
3.5 Insertion of discard wrappers . . . 19
3.6 Closing of lambda code . . . 20

4 AST normalization **21**
4.1 Overview . . . 21
4.1.1 Specifying different `myname` calculation modes . . . 21
4.1.2 Calculation of myname in ”hashed” mode . . . 21
4.1.3 Calculation of `myname` in ”fresh” mode . . . 22
4.1.4 Calculation of `myname` in ”cfresh” mode . . . 23
4.2 Implementation details of normalization . . . 23
4.2.1 Introduction . . . 23
4.2.2 Traversal and the `Normtree` . . . 23
4.2.3 Storing intermediate information . . . 24
4.2.4 Normalization of identifiers . . . 25
4.2.5 Normalization of paths . . . 25
4.2.6 Dependency upon external, pre- and sub- structures . . . 26
4.2.7 Generating expressions to compute `mynames` . . . 26
4.2.8 Example . . . 27
4.2.9 Aliasing (of functor parameters) . . . 28
4.2.10 Insertion of mynames into module signatures . . . 29
4.3 Debugging . . . 29
4.4 Normalization of functors . . . 29

4.4.1 Functor expressions 29
4.4.2 Normalisation of the parameter 30
4.4.3 Hash-parameter scope 30
4.5 Hash packages 30
4.6 Normalization of signatures 31

5 Type normalization **32**
5.1 Overview 32
5.2 Background: O'Caml type expressions 32
5.3 Background: O'Caml type declarations 33
5.4 Normalization for AST hashing (`Normtypedecl`) 35
5.4.1 Normalization algorithm 36
5.5 Normalization for type-passing (`Normtypes`) 39
5.5.1 Means of representing types 39
5.5.2 A note on recursive definitions 44
5.5.3 Normalization algorithm 44

6 Modifications to the O'Caml runtime **47**
6.1 Random myname generator (`byterun/random256.c`) 47
6.2 Hashing of structures (`byterun/hash256.c`) 47
6.3 Polymarshal (`byterun/polymarshal.c`) 47

7 The standard library **48**

Chapter 1

Introduction

This document describes the internals of the HashCaml implementation. HashCaml is a patch on the O'Caml bytecode compiler, runtime system and standard library that implements support for type-safe and abstraction-safe marshalling and associated features, summarised in Chapter 2 and presented in more detail in the paper [BSSS06]. The patch consists of a set of fairly small changes to the vanilla O'Caml source tree, plus additional C and O'Caml source files to the order of around 5,000 lines.

1.1 Overview of the compiler

The majority of the new code is contained in the source file `polymarshal/polymarshal.ml`. There is also a runtime support file `byterun/polymarshal.c` together with various changes scattered throughout the compiler.

In outline, the modifications made are as follows. These are listed in 'chronological' order with respect to a compilation session.

- Changes to permit extra compiler/toplevel flags.
- Changes to the lexer, parser and associated modules to accommodate new keywords.
- Changes to the type inference algorithm:
  - firstly, to cope with the new keywords;
  - secondly, to record information when inferring the types of value identifier nodes in order that their correct type schemes can be recovered later (see §3.2.1).
- A stage of rewriting inserted after type inference. This operates on *typed syntax trees* (as defined in `typing/typedtree.mli` etc).
- A stage of *coercion wrapper insertion* (see §3.4).
- A stage of rewriting on lambda-code (as defined in `bytecomp/lambda.mli` etc.) called from the `Translcore` module.
- A new runtime module written in C.

Chapter 2

Language features and constructs

This chapter, taken from the `HashCaml.README` included in the distribution, summarises the new constructs that HashCaml provides.

2.1 Type-safe marshalling

The O'Caml standard library module `Marshal` has been modified to support type-safe marshalling: attempts to unmarshal a value at the wrong type will cause a runtime exception. For example,

```
let s = Marshal.to_string  false  []
let b = true && (Marshal.from_string s 0)
```

executes successfully, whereas

```
let s = Marshal.to_string  17 []
let b = true && (Marshal.from_string s 0)
```

fails with the exception `Invalid_argument("Marshal type mismatch")`. (In O'Caml the second does not signal an error; it just silently continues with corrupted data. The old Marshal module that exhibits this behaviour is now available as the Oldmarshal module.)

To implement this, marshalled values now contain a runtime representation of the type at which they were marshalled, together with the value itself. The standard O'Caml marshaller is used underneath, so marshalling for code pointers and cyclic structures is as before.

Runtime type representations are 256-bit strings. For concrete types these are built by hashing the type structure, with very low probability of accidental collision. For abstract types the semantics is more subtle, and is outlined below. It is designed to give reasonable behaviour even for marshalling between non-identical programs: roughly, unmarshalling will succeed if the types involved have common definitions.

Marshalling and unmarshalling can be done within polymorphic functions, at types involving their generalised type variables. For example, in

```
let pair_and_marshal : 'a -> string
  = function x -> Marshal.to_string (x,x) []
let s = pair_and_marshal 17
let n = let (x1,x2)=(Marshal.from_string s 0) in x1+x2
```

the unmarshal succeeds correctly.

To implement this, runtime type representations are passed around dynamically. Loosely, extra lambda abstractions are introduced for each polymorphic generalisation point, and extra applications are introduced at each instantiation of a polymorphic identifier.

However, the dynamic type of marshalled values, and the dynamic type at unmarshal points, must not contain any (uninstantiated) type variables, otherwise a runtime exception will be raised. In other words, values cannot be marshalled polymorphically.

The current implementation naively passes types everywhere, which does incur a performance cost. There are many possibilities for optimization, eg to not pass types in the (frequent) cases in which they do not flow to a marshal or unmarshal point, or where at most one type will flow to such a point. There is also scope to memoise the type representation computations. We have not implemented any of these optimizations, or any optimization to the new compiler phases.

Bootstrapping of `ocamlc` is an order of magnitude slower than usual (much of this slowdown is likely to be due to the new phases), whilst the standard O'Caml test suite runs at almost the same speed when compiled with HashCaml as when compiled with vanilla O'Caml.

An SML'97-style value restriction is used rather than the (more liberal) O'Caml restriction.

The implementation is intended to cover all of the standard part of the language except marshalling of polymorphic variants and objects (which has not been addressed and may not work: a warning will be issued). The compiler can bootstrap itself.

2.2 Runtime type representations for abstract types

Runtime type representations for abstract types are built variously from pseudo-random numbers and hashes of module definitions, broadly following earlier work on the Acute language and its predecessor calculi.

In the implementation each structure (and therefore each compilation unit) has a hidden `myname` field containing a 256-bit value. This is used when calculating type representations for abstract types. For example the representation of some abstract type `M.t`, when viewed from another module `N`, will involve a hash of the string `"t"` and the `myname` value of `M`.

The `myname` value can be calculated in one of three ways, depending on what dynamic type equality semantics is desired:

1. as a hash of the abstract syntax tree of the module, up to alpha-conversion and type equality, and taking account of any module dependencies (”hashed mode”); or
2. as a random number determined at compile-time (”cfresh mode”); or
3. as a random number determined at module initialization time (”fresh mode”).

”hashed” mode ensures that if the implementation of the type verb+M.t+ changes, or if its associated invariants (enforced by functions in module `M`) change, then the type representations of `M.t` will be different from that before the change.

”cfresh” mode ensures incompatibility of such type representations between builds.

”fresh” mode ensures incompatibility of such type representations between instantiations of the module.

Hashed mode is the default. To specify ”fresh” or ”cfresh” mode, write `fresh struct` or `cfresh struct`, for example:

```
module M : sig type t end = fresh struct type t=int end
let x : typerep = rep ( M.t )

module M : sig type t end = cfresh struct type t=int end
let x : typerep = rep ( M.t )

module M : sig type t end = struct type t=int end
let x : typerep = rep ( M.t )
```

These annotations can also be used with functors. Note that although Objective Caml functors are applicative in the static type system, ”fresh” mode will cause the unique identifier for the functor body to be re-computed (with very high probability giving a different value) each time the functor is applied to a structure.

To set an entire implementation file (corresponding to a module of the same name) to ”cfresh” or ”fresh”, rather than the default of ”hashed”, insert `typemode cfresh` or `typemode fresh` as the *first* item in the source file, for example:

```
typemode fresh
type t=int

typemode cfresh
type t=int

type t=int
```

For abstract types with definitions that involve effects at initialisation time, ”fresh” should be used to guarantee that their invariants are preserved. The implementation permits any of the 3 options, however (it does not do any valuability analysis), as some applications need the more liberal ”hashed” or ”cfresh” semantics even in the presence of effects.

2.3 Type representations

There is a new built-in abstract type

```
typerep
```

of runtime type representations, with constructs `rep` to build the representation of a type, eg

```
let x : typerep = rep( int * int )
let s : string = Dyntype.string_of_typerep x
```

and the representation of the type of an arbitrary expression, eg

```
let x : typerep = dyntype( 3 )
```

Their behaviour is unspecified if the dynamic type contains uninstantiated type variables. Currently an exception is raised.

Usually the only interesting operation for type representations is to compare them for equality - they have no useful internal structure - but there is a new standard library module Dyntype containing the function

```
string_of_typerep : typerep -> string
```

2.4 Names

There is a new family of types `'a name`, represented as 256-bit values. Names can be generated in several ways:

1. Freshly: just write `fresh` (of type `'a name`). This produces 256-bit random values. For example:

   ```
   let x : int name = fresh
   ```

2. Using the module hashes produced for calculating type representations, for example

   ```
   module M = struct let f : int -> int = function z->z end
   let x : (int -> int) name = fieldname M.f
   ```

   which will produce a name based on the module `myname` value of `M` and the path `t`.

There is a special conditional for comparing names:

```
ifname e1 = e2 then e3 else e4
```

where e1 : 'a name, e2 : 'b name, e3 : 'c, e4 : 'c,
and e3 is typechecked in an environment where 'a and 'b are unified. For example

```
let x : int name = fresh
let y : bool name = fresh
let f x' y' = ifname x' = y' then [x';y'] else [x']
let a = f x y
let b = f x x
```

We antipate this to often be used in conjunction with an encoding of existentials, eg to manipulate collections of existential packages of names of channels (carrying t) and the pending values (of type t).

We also provide name coercions:

```
namecoercion(path1, path2, e)
```

where path1 and path2 are type constructor paths (and refer to type constructors of the same arity) and e has type [ty1 ... tyn] path1 name, yields a value of type [ty1 ... tyn] path2 name. For example:

```
type ('a,'b) t = Foo of 'a*'b
type ('c,'d) t' = Bar of 'c*'d*int
let x : (int,bool) t name = fresh
let y : (int,bool) t' name = namecoercion(t,t',x)
```

There is also a facility for constructing names from a pair of a type and a string:

```
hashname(t, s)
```

where t is a type and s is a value of string type. For example:

```
let x : int name = hashname(int, "foo")
```

2.5 Other Acute constructs

The prototype Acute language provided various additional features which are not supported by this implementation:

- dynamic rebinding to programmer-specified marks at unmarshal time (here, instead, rebinding takes place only to the standard library)
- explicit imports, version numbers and version constraints
- polytypic name support and swap operations
- thunkification of executing threads (and mutexes and cvars) into marshallable values
- marshalling of functions between non-identical builds (here the O'Caml marshaller is used internally, so functions can only be communicated (as function pointers) between identical builds).

2.6 Compiler options

The compiler understands some new flags:

- -allowmynames Allow references to myname fields (not recommended)
- -nomlpoly Use relaxed value restriction (implies -nopolymarshal)

- `-nopolymarshal` Disable polymorphic marshalling (experts only)
- `-pmdebug` Print debugging info for polymorphic marshalling
- `-dnormtree` Print out normalized trees of compilation unit
- `-dnormtrans` Print debugging info for normalization

If a program is using polymorphic marshalling, all files that it makes use of must be compiled without `-nopolymarshal` as well (including the standard library, which is compiled with polymorphic marshalling by default).

Chapter 3

Type-passing translation

To effect the recovery of type information at runtime, HashCaml performs a certain translation on syntax trees. The guiding principle is to 'put in all the big lambdas' together with the corresponding type applications. In fact, these 'big lambdas' are just normal lambdas that accept type representations: type application is then treated as normal function application. We explain how this translation works after a brief diversion to the value restriction.

3.1 Value restriction

HashCaml enforces the SML'97-style value restriction. This is in contrast to vanilla O'Caml distributions, which use not only the Garrigue criterion[1] for generalization of expansive expressions, but also behave as follows:

- whether or not a conditional expression `if` e_1 `then` e_2 `else` e_3 is judged to be nonexpansive is independent of the condition e_1;
- an application expression will be deemed to be nonexpansive if the function part and all of the arguments are themselves nonexpansive. (Under the SML-style value restriction, an application expression is always expansive.)

Why does HashCaml enforce the SML-style value restriction? Simply because the scheme of type-passing becomes complicated and grotesque under the standard O'Caml restriction. We now examine why.

First off, the usual intuition with Garrigue's criterion is that a value whose type scheme involves a generalized type variable appearing only in covariant position cannot actually contain any value whose type is that type variable (in other words, the body of an expression yielding that value does not care about what the type variable is instantiated to). For example the empty list of type α `list` never contains any values of type α, whatever that may be instantiated to.

This intuition breaks down when considering the type-passing translation. It is indeed possible in that scenario to construct a function that is assigned a generalized type, with a type variable appearing only in covariant position, and yet needs to know what types that variable is being instantiated to. Here is one such function, given as part of a hypothetical toplevel session:

```
# let r = ref "";;
val r : string ref = {contents = ""}
# let f () = let x = [] in ((r := Marshal.to_string x []); x);;
val f : unit -> 'a list = <fun>
```

[1]That criterion generalizes in the usual way when the expression whose type is being generalized is non-expansive. If, however, the expression is expansive then it will generalize those type variables that occur only in covariant positions within the type of the expression.

```
..  3 :: (f ())  ..  "foo" :: (f ())  ..
```

Note that the function `f` has a generalized type containing a type variable appearing only in covariant position. Depending on what type is assigned to the application expression that invokes the function `f`, we need to have different type representations appearing for the marshal (and they should be sensible since the result can be observed via the reference `r`).

Now whilst it is indeed the case that no value of type τ appears inside the value bound to `f` when used at type `unit` $\rightarrow$ τ `list`, we still need to know what type `f` is being used at inside its body because the marshalling expression introduces a dependency on the corresponding (type representation) value identifier. In this case, it is therefore necessary to add a type representation lambda to the function `f` to correspond to the covariant-only type variable that is being generalized. A type-passing version of the above might look something like the following[2]. (We have abbreviated the marshal to just identify that it uses `rep` and hence depends on `tyrep_a`.)

```
# let r = ref "";;
val r : string ref = {contents = ""}
# let f tyrep_a () = let x = [] in ((r := .. .tyrep_a ..); x);;
val f : typerep -> unit -> 'a list = <fun>

..  3 :: (f rep(int) ())  ..  "foo" :: (f rep(string) ())  ..
```

So this leads us to believe that even covariant-only type variables that are generalized should be associated with type representation lambdas.

To refute this seemingly attractive proposition, we now consider the question of generalization of expansive expressions. (The reader will see that `f` above is being bound to a non-expansive expression – in that case a lambda abstraction.) First we consider what happens when the expansive expression evaluates to a non-functional value, as in the following code fragment.

```
let s = ref "" in
let id = fun y -> (s := Marshal.to_string y []; y) in
let r = ref 0 in
let x = (r := 1; id []) in
  (!r, !s)
```

Here, the identifier `x` receives the type scheme $\forall\alpha.\ \alpha$ `list`; it is assigned to a value that results from the evaluation of an expansive expression (the sequencing). We also need to pass a type representation into the function `id` since that has an (uncontroversial) generalized type. However we cannot get that type representation to the application point of `id` within the definition of `x`, because adding a type representation lambda to the value bound to `x` affects the evaluation order. Besides, we don't really want to add a lambda anyway: `x` is not a function, and although it may be used at one of many types, that use never affects the type representation passed into `id`. The solution, therefore, is presumably to never add any type representation lambdas when the type being generalized is not a function type; and furthermore, in cases such as this one where a type variable being generalized is used on the right-hand side of the `let`, dummy type representations should be added to be passed down as required. Hence the type-passing version of the above could perhaps be (abbreviating the marshal as above):

```
let s = ref "" in
let id = fun tyrep_a -> fun y -> (s := .. tyrep_a .. ; y) in
let r = ref 0 in
let x = (r := 1; id <dummy> []) in
  (!r, !s)
```

[2]The astute reader who has spotted that `x` has not been given a type representation lambda should wait a few paragraphs to find out why.

Now this may suffice for the case where we have generalization of the type variables that appear only in covariant position within the type of an expansive but non-functional expression. But what if the expression is of function type? In such cases, it could be applied multiple times, and type representations unfortunately need to be propagated through. We slightly alter the first example above to give the following code:

```
# let r = ref ""
val r : string ref = {contents = ""}
# let s = ref 0
val s : int ref = {contents = ""}
# let f = (s := 1; fun () -> let x = [] in ((r := Marshal.to_string x []); x))
val f : unit -> 'a list = <fun>

..  3 :: (f ())  ..  "foo" :: (f ())  ..
```

The Garrigue criterion will still generalize as shown above; this time, however, we cannot add a type representation lambda onto `f` because it would disrupt the side effect. The only option is, unfortunately, to assume within the body of `f` that the generalized type variable is always bound to a dummy typerep, just as we had in the previous code fragment. That then gives a type-passing version like this (no type representation lambda has been added to `f` and the invocation of `rep` on the type variable inside the marshal has yielded a dummy):

```
# let r = ref "";;
val r : string ref = {contents = ""}
# let s = ref 0;;
val s : int ref = {contents = ""}
# let f = (s := 1; fun () -> let x = [] in ((r := .. <dummy> ..); x));;
val f : unit -> 'a list = <fun>

..  3 :: (f ())  ..  "foo" :: (f ())  ..
```

Whilst this is unsatisfactory on many counts, it does mean that at least some type representation (albeit a dummy one) is communicated to the correct point; it also ensures that the side-effect is not disturbed. If such a scheme were to be implemented, care would need to be taken to record that the identifier `f` must never have any type representations applied to it (*viz.* the last line in the code above).

The only satisfactory thing to be said about this scheme is that it fits with the non-functional case: the summary for both cases is therefore that if an expansive expression is having its type generalized, whether that type is a function type or not, the corresponding type representation identifiers within the expression should be set to dummies and no type representation lambdas should be added. Thankfully such cases are, in practice, likely to be rare.

We now return to the O'Caml-style generalization of conditional expressions but assuming that everything else is as for SML'97. Consider the following code fragment:

```
let r = ref 0 in
let f = if (r := 1; true) then fun x -> Marshal.to_string x [] in
  !r
```

The identifier `f` is assigned the type scheme $\forall\alpha.\ \alpha \rightarrow \texttt{string}$, since even though the condition is not a nonexpansive expression, the body of the whole `if` expression is nonexpansive. However we cannot match this generalization with a type representation lambda inserted on the front of the value bound to `f` because such a lambda would delay the side effect. In general, the side effect might never happen or be repeated multiple times due to the erroneous translation. A plausible solution would be to adopt the same 'dummy' strategy as above: indeed, if the Garrigue criterion combined with that strategy is used then it subsumes this case anyway (the `if` is an expansive expression).

3.2 Rewriting on typed syntax trees

This section describes the rewrites that are performed on syntax trees whose types are defined in the `Typedtree` module.

3.2.1 Translation of value identifiers

The basic aim of the translation at value identifier nodes is to turn them into typerep applications if the identifier has a non-trivial type scheme. Value identifiers which are not 'regular' (such as ones representing C primitives) are never translated. In the future, when dealing with objects, this may well need to change: the other varieties of non-regular identifier all pertain to them.

Upon encountering a regular value identifier, we proceed as follows. The type scheme of the identifier is examined; if it is trivial then we just leave the node alone. Otherwise, we examine the type at which this occurrence of the value identifier is being used and determine the substitution which gets us from the type scheme to that type. This substitution will map each quantified type variable in the type scheme and can, given a canonical order on type variables, be directly translated into a type representation application.

In order to determine the substitution, we can allocate a number of fresh type variables (as many as there are quantified in the type scheme) and instantiate the scheme at those variables. Then the resulting type can be unified with the type at which the identifier is being used and the substitution read off by examining what the fresh type variables have been unified with.

For example if the identifier in question is called `x`, has the type scheme $\forall\alpha\beta.\ \alpha \to \beta \to \texttt{bool}$, and is being used at the type $\texttt{bool} \to \texttt{int} \to \texttt{bool}$, then we will emit

$$\texttt{x}\ R(\texttt{bool})\ R(\texttt{int})$$

where we write $R(\tau)$ for the code that constructs the *type representation block* that represents the type τ (see §5.5.1). For now it suffices to identify that, for any type variable α, $R(\alpha)$ contains a value identifier whose name is uniquely derived from the name of the type variable.

When emitting the typerep applications it is important (so that they match up with typerep abstractions added elsewhere) to ensure that they appear in a consistent order with respect to the type scheme parameters that they correspond to: this is why we sort using a canonical order. In the implementation this is the usual order on the natural numbers.

This procedure is significantly more complicated to implement than it might at first appear. The hardest problem relates to the fact that in the Objective Caml implementation there is actually no distinction between types and type schemes. In order to perform generalization, the system simply marks the appropriate type variables as being generalized (by setting their levels to `generic_level`). Therefore when examining the 'type scheme' of an identifier to determine the quantified variables, what we are actually doing is examining a plain type and just extracting those whose level is `generic_level`.

This all sounds well and good, but is not in itself a solution: since we examine the typed syntax trees after type inference has taken place, it is possible that the imperative generalizations give a false view of the world. Consider the following fragment:

```
let f x = dyntype(x) in (f "foo") :: (f 42) :: [];;
```

This should be translated to something like:

```
let f tyrep_a x = ... tyrep_a ... in (f String "foo") :: (f Int 42) :: [];;
```

writing `String` for $R(\texttt{string})$ and likewise for `Int`.

Suppose we are looking at the occurrence of the identifier `x` in the body of `f`. It is clear that this identifier has type scheme $\forall\emptyset.\ \alpha$. However inside the compiler this is not obvious. When typechecking the body of `f` the identifier would indeed have been assigned a type corresponding to the previous trivial type scheme. The difficulty is that after that body has been typechecked, generalization happens and that type variable is *imperatively updated* to be general. This is safe, because no uses of it occur outside of the function definition. So when we later come to look at the typed trees and come across that occurrence of x, it appears to have type scheme $\forall\alpha.\ \alpha$, which is erroneous.

To circumvent this problem we adjust the type inference algorithm itself to record, whenever it hits a value identifier, which type variables in its type are marked as general at that time. We call such a record a *type variable identifier memo*. In the above case, the corresponding set of type variables would be empty, and so when we examine the typed trees after type inference we know that the identifier actually had type scheme $\forall\emptyset.\ \alpha$.

Similar problems apply when dealing with `let` expressions as we shall see in §3.2.2 and §3.2.3.

3.2.2 Translation of non-recursive let bindings

The basic procedure followed here is to equip let-bound values with typerep lambdas according to the type scheme of the value. This apparently simple procedure is made complicated by two things:

- the fact that let bindings can involve arbitrarily complicated patterns on the left-hand side;
- difficulties in identifying the type scheme of let-bound identifiers (in a similar vein as those encountered when translating value identifier nodes).

The first problem manifests itself in examples such as the following:

```
let (x, y) = (fun x -> x), (fun y -> y);;
```

What we need to do here is to insert one typerep lambda on the front of the values of the let-bound identifiers x and y. We know this because we have examined the types of x and y and noted that both types contain one generalized type variable.

In order to effect the addition of lambdas, we either need to decompose the pair on the right-hand side of the binding (before the match compiler is applied), or we need to rewrite after the match compiler. Since the right-hand side of such a binding could be arbitrarily complicated (although still a value if enforcing the SML 97-style value restriction), and the match compiler already has ways of decomposing such values, we choose the second option. This means that the majority of the rewrites performed on let expressions are applied at the lambda-code level rather than at the typed tree level. The exact transformation performed there is described in §3.3: nothing more happens at this stage.

The transformation on lambda code requires the type schemes of each identifier occurring on the left-hand side of a let binding, bringing us to the second difficulty identified above. As for value identifier nodes, we cannot simply use the post-type-inference type to determine which type variables are generalized. For example in the following example the identifier g has a trivial type scheme, whereas its type variables will be marked as general after type inference due to the generalization occurring at the outermost `let`.

```
# let f x = let g y = (x = y) in g x;;
val f : 'a -> bool = <fun>
```

To obtain the correct answers we adopt a similar procedure to before; nodes representing `let` expressions (and, correspondingly, value bindings in structures) in the typed syntax trees are augmented with a type variable memo that enables us to recover, after type inference, the correct generalization information which we require. This entails a reasonable number of small changes throughout the compiler.

Even though we defer discussion of the lambda-code rewrites until §3.3, we do give the following lambda-code, which is the output of the translation on the above fragment (when placed in a file `test.ml`).

```
(setglobal Test!
  (let
    (f/71 (function _tyrep_678/99999
            (function x/72 (
              let (g/73 (function y/74 (caml_equal x/72 y/74)))
                (apply g/73 x/72))))
       myname/83
         (hash256_checked
            (makeblock 0 "\019 .. \252"
                         (field 81 (global Pervasives!)))))
  (makeblock 0 f/71 myname/83)))
```

3.2.3 Translation of recursive let bindings

A further complication arises when the value identifier being considered is bound by a `let rec`. For example:

```
let rec f x n = if n = 0 then dyntype(x)
                         else f x (n - 1);;
```

The function `f` has type scheme $\forall\alpha.\ \alpha \to \texttt{int} \to \texttt{string}$ and so the occurrence of `f` on the right-hand side of the binding should be replaced by the application of a typerep to `f`. However applying the scheme of §3.2.1 for determining generalized variables at value identifier nodes fails: originally, the occurrence of `f` would have been assigned a trivial type scheme (because polymorphic recursion is not permitted in O'Caml) and the record of that fact would later be used to identify (incorrectly) `f` as having a trivial type scheme at the end of it all.

To understand how we solve this problem consider a general `let rec` expression where multiple, possibly mutually-recursive, identifiers are being bound at once, thus:

```
let rec x1 = ...
        ...
    and xn = ...
```

(Whether or not the `let rec` has a body makes no difference, for inside the body the scheme of §3.2.1 is correct.)

During type inference, type variable memos will be formed for each identifier `x1` through `xn` in the same manner as is done for non-recursive bindings (viz. §3.2.2). What we then do, when effecting the type-passing translation, is to first combine all those memos into one single one and then combine that with a current 'running memo' that is the result of this procedure happening at any outer `let rec` bindings. In this case, that combined memo enables us to identify which type variables in the types of `x1` through `xn` are those generalized by the `let rec` (rather than any enclosing expression). Also (by virtue of the 'running memo'), if our `let rec` expression is actually sitting inside the bindings of another such, the memo enables us to get the correct generalization information for value identifiers bound by that outer expression.

When delving inside the bindings of the `let rec`, then, we use the combined memo whenever we encounter a value identifier that is being recursively (monomorphically) bound by either the left-hand side of our current binding or (if we are nested inside the binding of another `let rec`) an outer left-hand side. Upon examining the memo we determine the correct type scheme of the recursively-bound identifier and thus we can insert correspondingly many type representation lambdas.

Note that since `let rec` expressions may only possess single variable patterns on their left-hand sides, we can perform all rewrites on recursive `let` expressions at the typed syntax tree level rather than on lambda-code.

Now going back to the example above:

```
let rec f x n = if n = 0 then dyntype(x)
                          else f x (n - 1);;
```

we can understand how the following lambda-code is generated as a result of this procedure:

```
(setglobal T!
  (letrec (f/71
    (function _tyrep_667/99999 x/72 n/73
      (if (== n/73 0)
        (let (_marshalled/83 x/72) (flattentyperep _tyrep_667/99999))
        (apply f/71 _tyrep_667/99999 x/72 (- n/73 1)))))
  (let (myname/82
    (hash256_checked
      (makeblock 0 "g\189 .. >\154" (field 81 (global Pervasives!))
                  (field 81 (global Pervasives!)))))
  (makeblock 0 f/71 myname/82))))
```

Observe how we have correctly identified the occurrence of `f` on the right-hand side as being a recursively-bound identifier.

3.2.4 Translation of record fields

In O'Caml record fields may be assigned a polymorphic type; during type inference the compiler correspondingly attempts to generalize the type of a value being put into a record field. Since the field's value may subsequently be used at multiple types, we equip it with as many type representation lambdas as are necessary to reflect the generalization that has happened. Correspondingly, when a record field is referenced, we add type representation applications as appropriate.

For example the following source text:

```
type record_t = { field : 'a . 'a list -> int }
let record = { field = List.length }
let l1 = ["foo"; "bar"]
let l2 = [1; 2]
let xs = (record.field l1) :: (record.field l2) :: []
```

yields the lambda-code below:

```
(setglobal T!
  (let
    (record/74 (makeblock 0 (
      function _tyrep_694/99999
        (apply (field 0 (global List!)) _tyrep_694/99999)))
     l1/75 [0: "foo" [0: "bar" 0a]]
     l2/76 [0: 1 [0: 2 0a]]
     xs/77
       (makeblock 0
         (apply (field 0 record/74) [0: "Ew\031 .. \023z9"] l1/75)
         (makeblock 0 (apply (field 0 record/74)
           [0: "E].\227 .. e\""] l2/76) 0a))
     myname/79
       (hash256_checked
         (makeblock 0 "\027 .. V5" (field 42 (global List!)))))
    (makeblock 0 record/74 l1/75 l2/76 xs/77 myname/79)))
```

3.3 Rewriting on lambda-code

Recall the following example from §3.2.2.

```
let (x, y) = (fun x -> x), (fun y -> y);;
```

When compiled with a standard Objective Caml compiler we obtain the following lambda-code:

```
(let
  (match/75 (makeblock 0 (function x/73 x/73) (function y/74 y/74))
   y/72 (field 1 match/75)
   x/71 (field 0 match/75))
  (makeblock 0 x/71 y/72))
```

It is straightforward, having examined the various types and determined which typerep lambdas need to be inserted (by means of the type variable identifier memos discussed in §3.2.2), to put them on the bindings for `x` and `y` in this code. However, the right-hand side of the original let binding has been moved into the lambda-code binding of the fresh identifier `match/75`. So in fact we need to determine *all* of the typerep abstractions which are needed across the right-hand side of the original bindings and add all of those onto the bindings for any auxiliary `match` identifiers such as this one. The individual bindings for `x` and `y` in this case only have one lambda added to them each, but they actually each apply *two* typerep arguments to `match/75`. If a particular typerep argument of a `match` identifier is not needed in a particular case, it is replaced by a dummy.

As an example consider the following fragment:

```
let (x, y) = (fun x -> dyntype(x)), (fun y -> dyntype(y))
```

The output of translation to lambda code is as follows (ignore the redundant `lets` binding the `_marshalled` identifiers; they exist for historical reasons):

```
(setglobal T!
  (let
    (match/86
      (function _tyrep_667/99999
        (function _tyrep_675/99999
          (makeblock 0
            (function x/73 (
              let (_marshalled/77 x/73) (flattentyperep _tyrep_667/99999)))
            (function y/74
              (let (_marshalled/78 y/74) (flattentyperep _tyrep_675/99999))))))
     y/72 (function _tyrep_675/99999
             (field 1 (apply match/86 _tyrep_667/99999 _tyrep_675/99999)))
     x/71 (function _tyrep_667/99999
             (field 0 (apply match/86 _tyrep_667/99999 _tyrep_675/99999)))
     myname/76 "h\230 .. \175\220")
    (makeblock 0 x/71 y/72 myname/76)))
```

Note that the bindings for `x` and `y` contain unbound type representation identifiers. These are caught at a later stage (after dumping of the above output) and replaced by dummy type representations, since they will never be used. It would be pleasing if a more satisfactory means of treating this could be produced.

Another outstanding issue at the moment, which makes the code unclear (and indeed necessitates de-shadowing of all lambda code), is that the translation is performed at every `let` node in the lambda code, rather than it being effected by starting at the top level of the lambda code and working downwards. An implementation along the latter lines would be desirable and more straightforward.

3.4 Coercion wrapper insertion

Coercion wrapper insertion pertains to a certain problem, which arises due to the fact that modules' interfaces may exhibit less general types than the actual implementation underneath. We start with a simple example. In the standard library is a function called `List.iter` declared thus:

```
let rec iter f = function
  [] -> ()
| a::l -> f a; iter f l
```

Due to a typechecking peculiarity, this has the inferred type

```
val iter : ('a -> 'b) -> 'a list -> unit
```

but is declared in the interface file as

```
val iter : ('a -> unit) -> 'a list -> unit
```

This means that when using type-passing, the code for `iter` has two typerep abstractions on the front of it. However, when an external user (viewing `iter` only via the interface) uses it, they think it only has one, and so it fails to work correctly. (One might think that in such scenarios the coercion is invisible; however, it can be observed as we shall see in the next example.)

Our analysis of this problem centres on the fact that

if an implementation has an interface then the interface must be compiled first.

Therefore, when compiling such an implementation, we can see if the interface is causing coercions. If it is, we proceed as follow. For each identifier `old` exposed in the interface in such a way as to coerce its type, we allocate a fresh identifier (call it `new`). We rename *all* occurrences of the identifier `old` to `new` throughout the structure body (including in the domains of type variable identifier memos) and then insert a wrapper function called `old` which accepts as many typerep arguments as there are type parameters exposed in the interface of that function and then calls the function `new`, supplying it with parameters as necessary from the called arguments and also those statically known from the interface coercion.

Here is a concrete example, which will clarify the situation. We have three source files comprising a program: `m.ml`, with accompanying interface `m.mli`, and `n.ml`. Remember that the `.mli` file must be compiled first.
The source of `m.ml` is:

```
let foo a b = (dyntype(a), dyntype(b))
```

The source of `m.mli` is:

```
val foo : 'a -> unit -> (typerep * typerep)
```

The source of `n.ml` is:

```
M.foo 42 ()
```

Because `foo` exhibits a coercion in the interface, we rename it to a fresh identifier (in this case `foo_61_body/66`) and insert a wrapper function called `foo`. This yields the following output of compiling `m.ml`:

```
(setglobal M!
  (let
    (foo/71
      (function _tyrep_664/99999
        (function _tyrep_669/99999
          (function a/72 b/73
```

```
          (makeblock 0 (let (_marshalled/76 a/72)
                              (flattentyperep _tyrep_664/99999))
                         (let (_marshalled/77 b/73)
                              (flattentyperep _tyrep_669/99999))
          ))))
     myname/75 "\002 .. \024")
    (makeblock 0
      (function _tyrep_721/99999
        (apply foo/71 _tyrep_721/99999 [0: "E\180 .. \161"]))
      myname/75)))
```

Note the way that the original foo, equipped with all its typerep arguments, has been renamed. Internal users will still use that version, whilst external users (viewing the function through the interface) will use the wrapper (the third and second lines from the bottom).

The output of compiling n.ml is then:

```
(setglobal N!
  (seq (apply (field 0 (global M!)) [0: "E].\227 .. \""] 42 0a)
    (let
      (myname/72
        (hash256_checked
          (makeblock 0 "\012 .. \233kP" (field 1 (global M!)))))
      (makeblock 0 myname/72))))
```

Note how the typerep corresponding to the int type has been statically filled in.

When compiling modules containing other modules (or functors) nested inside them, we must be careful to insert the coercion wrappers at the end of the appropriate module to ensure that the relevant identifiers are in scope.

3.5 Insertion of discard wrappers

External functions (written in C) are never passed type representation blocks – if such a function needs the type information it must explicitly have an extra argument and an ML wrapper function that uses rep() to obtain the appropriate type. Such functions are likely to be polymorphic and higher-order, for example one that takes a polymorphic function as argument and then invokes it from C. Cases like this—of which only one has been identified in the compiler system itself—may need to explicitly pass type representation arguments back to ML functions. If the reader needs to do this they should note the order in which the type representation lambdas have been added to the ML function (using option -drawlambda) and insert explicit applications in their C code. Type representations are applied curried, not tupled.

In order to ensure that external functions are not passed type representation blocks, the compiler does not add type applications to values of kind Val_prim. The only case that might seem tricky is when a value identifier is assigned to be an external function (for example, given an external f, we might say let g = f). However, in all such scenarios the O'Caml compiler already eta-expands the primitive (presumably because application of it requires special treatment, which only needs to be inserted once—in the eta-expanded code—and not at every use point). In a similar manner, then, we allow the eta-expanded versions to be equipped with type representation lambdas (and since they are regular values, not Val_prim ones, they will get type representation blocks applied to them as usual). The lambda-code that consists of such type representation lambdas, with an eta-expanded primitive in the middle, is called a *discard wrapper* – so named because it accepts the relevant number of type representation lambdas and simply discards the arguments passed into them.

3.6 Closing of lambda code

When the lambda code has been generated there may be unbound type representation identifiers. These arise from two sources: the translation of §3.3 and situations where free type variables end up having their type representations calculated. Currently, a pass at bytecode generation time traps these unbound identifiers and converts them into dummy type representations. If they arose from the translation of §3.3 they will never be used; however if they arose from free type variables they might be used via a `dyntype()` or a `rep()`. Attempting to use either of these on a type representation block that contains dummies within it fails with a runtime exception `Invalid_argument "Dynamic type contains free type variables"`. This should probably be turned into a distinguished, predefined exception.

Chapter 4

AST normalization

4.1 Overview

Each structure (and therefore each compilation unit) contains a hidden field called `myname` that is a unique identifier for the module. This field cannot be accessed by the user without explicitly specifying the `-allowmynames` compiler flag.

4.1.1 Specifying different `myname` calculation modes

The different modes for calculating `myname` are ”hashed” (default, no keyword), ”fresh” (keyword `fresh`) and ”cfresh” (keyword `cfresh`). The keywords can appear in front of any `struct` keyword making the corresponding structure's `myname` of the selected mode. To select the mode for the top level (compilation unit's) structure other than the default ”hashed” mode, the keyword pair `typemode fresh` or `typemode cfresh` must appear as the first item in the file.

4.1.2 Calculation of myname in ”hashed” mode

The abstract syntax tree of a structure is subjected to a process of normalization before it is hashed to compute the unique identifier for that module.

The normalization process satisfies the following.

1. All identifiers that cannot be externally visible are assigned de Bruijn indices, e.g.

   ```
   let f x = x
   ```

   is normalized to

   ```
   let f [0] = [0]
   ```

   Here `f` is visible and remains unchanged, whereas `x` is not visible, so it is assigned a de Bruijn index.

2. The `myname` of a named structure is independent of that structure's name.

3. A module's `myname` depends on the `myname` values of modules it refers to. For example consider:

   ```
   module M = struct let _ = print_endline "HashCaml" end
   ```

Here `M.myname` depends on `Pervasives.myname`, since it uses its `print_endline` function. `Pervasives.myname` is thus included in the hash calculation that yields `M.myname`. (That calculation is not performed until runtime, since in general a module might depend on another that is specified as ”fresh”.)

4. A module’s `myname` depends on the `myname`s of its inner modules.

5. An inner module’s `myname` depends on the hash of the (compilation unit’s) prefix (all the structure items defined up to that module) iff the inner module uses the prefix, e.g.

```
module M = struct
  let x = 1
  module N = struct
    let y = x
  end
end
```

Here `N`’s `myname` depends on the hash of `module M = struct let x = 1`. In the following case, however, `N`’s `myname` is completely independent of its prefix:

```
module M = struct
  let x = 1
  module N = struct
    let y = 2
  end
end
```

6. The `myname` value inside a functor body depends on the `myname` of the functor argument iff the functor body uses it, e.g.

```
module F(P : sig val x : int end) = struct
  module Q = P
  let y = Q.x
end
module M = struct let x = 1 end
module N = F(M)
```

Here `N`’s `myname` depends on the `myname` of `M`, whereas in the following case it does not:

```
module F(P : sig end) = struct end
module M = struct end
module N = F(M)
```

During the process of normalizations, any type declarations or type expressions that are encountered are converted to a normalized form. This involves identifying any references to external modules and inserting appropriate `myname` dependencies, together with quotienting modulo alpha-conversion for type parameters.

Note that if any sub-structure, or any structure that a structure depends on, uses ”fresh” or ”cfresh” `myname` type mode, the structure will effectively also use ”fresh” or ”cfresh” type mode, respectively.

4.1.3 Calculation of `myname` in ”fresh” mode

The structure’s `myname`s will be assigned a pseudo-random 256bit string *at execution-time*.

4.1.4 Calculation of `myname` in "cfresh" mode

The structure's `mynames` will be assigned a pseudo-random 256bit string *compile-time*.

4.2 Implementation details of normalization

4.2.1 Introduction

The normalization is performed separately for each compilation unit, and is invoked from within `typing/typemod.ml`, by calling the `main` function in `hashing/normtrans.ml`. The function therefore receives a `Typedtree.structure` and returns an identical `Typedtree.structure`, but where all (sub-)structures contain an appropriately assigned `myname` field.

4.2.2 Traversal and the `Normtree`

The `myname` of a module depends mainly on the hash of the normalized structure of the module itself. The normalization is performed recursively on the original `Typedtree.structure` structure:

```
type structure = structure_item list

and structure_item =
    Tstr_eval of expression
  | Tstr_value of rec_flag * (pattern * expression * Ident.t tyvar_id_memo) list
  | Tstr_primitive of Ident.t * value_description
  | Tstr_type of (Ident.t * type_declaration) list
  | Tstr_exception of Ident.t * exception_declaration
  | Tstr_exn_rebind of Ident.t * Path.t
  | Tstr_module of Ident.t * module_expr
  | Tstr_recmodule of (Ident.t * module_expr) list
  | Tstr_modtype of Ident.t * module_type
  | Tstr_open of Path.t
  | Tstr_class of (Ident.t * int * string list * class_expr) list
  | Tstr_cltype of (Ident.t * cltype_declaration) list
  | Tstr_include of module_expr * Ident.t list

and ...
```

There is a translation function for each type of an (sub-)element of the `Typedtree.structure`, which calls other translation functions for each of its sub-elements. A translation function (prefix = `tr_`) takes a sub-element `E` of the original tree, and returns a tuple `(nE, E')`, where `nE` is a normalized version of `E`, and `E'` is the original `E`, where all (sub-)modules already contain appropriately assigned `mynames`. The normalized version of the original tree is defined in `hashing/normtree.ml`:

```
type nstructure = nstructure_item list

and nstructure_item =
    Nstr_eval of nexpression
  | Nstr_value of Asttypes.rec_flag * (npattern * nexpression) list
  | Nstr_primitive of ident_t * nvalue_description
  | Nstr_type of (string * Normtypedecl.ntype_decl) list
  | Nstr_exception of ident_t * (Normtypedecl.ntype list)
  | Nstr_exn_rebind of ident_t * npath
  | Nstr_module of ident_t * nmodule_expr
```

```
  | Nstr_recmodule of (ident_t * nmodule_expr) list
  | Nstr_modtype of ident_t * nmodule_type
  | Nstr_open of npath
  | Nstr_class of (ident_t * int * string list * nclass_expr) list
  | Nstr_cltype of (ident_t * Types.cltype_declaration) list
  | Nstr_include of nmodule_expr * ident_t list

and ...
```

The `Normtree` either uses the original type present in the `Typedtree` erasing some information from it during normalization, or it uses the corresponding normalized type, when the normalized tree requires to store different type of information to the original tree. Most used of the different types are `ident_t` and `npath`, which are described later.

4.2.3 Storing intermediate information

The normalization procedure needs to preserve some information throughout the translation of a (sub-)structure. This is done by using state represented by the type `Normtrans.state_t`, of which:

`seen_external` states whether the current sub-module references anything defined before it (in the same compilation unit),

`bcounter` is the de Bruijn counter for the current (sub-)module,

`bmap` is a map from identifiers in the current (sub-)module to their normalized form,

`old_bmap` is a map from identifiers in all the super-modules to their normalized form,

`functor_params` is a list of the visible identifiers of the functor parameters,

`local_module_params` is a list of identifiers of modules defined with `let module` construct,

`param_set` is a set that stores the expressions accessing `mynames` of structures (either internal or external to the compilation unit) referenced from within the code in the current structure, but *not* also from within its sub-structures,

`packages` is a list of lists of all the already normalized `Normtree.structure_items` in a form of a `Hashpackage.package`, such that each list stores items defined within a particular level, and where a `Hashpackage.package` stores `Normtree.structure_item`(s) with a corresponding `Hashpackage.hash_param_set`, which is a `param_set` for this `Normtree.structure_item`,

`mod_defs` is a list of names of immediate sub-structures already normalized, and

`extra_deps` is a list of paths of external modules used during normalization of types.

Most of the (sub-)structures use their own copy of state, but pass part of it to the state of its sub-structure and part to its super-structure. The normalization of the outermost module starts with the `null_state`, where all the above parts of the state are set to either `false`, `0` or `[]`, except for the `param_set`, which is initialized to `Hashpackage.empty_hash_param_set`. Every sub-module starts with the same `null_state`, with the exception of `old_bmap` and `packages`, which are assigned the combination `bmap` and `old_bmap` of the super-state and the `packages` of the super-state, respectively.

4.2.4 Normalization of identifiers

When speaking of identifiers, we do not only refer to something of type `string`, but rather of something of type `Ident.t`, which also distinguishes same `string` identifiers on different levels, by storing a `stamp`, which is unique within the same compilation unit. For this reason, we do not need to worry about variable hiding, as identifiers of variables always include their own `stamp`. Its type definition in `typing/ident.ml` is:

```
type t = { stamp: int; name: string; mutable flags: int }
```

We distinguish identifiers to those which also appear in the externally visible definitions (also referred to as *top identifiers*), and all the others (referred to as *normal identifiers*). All identifiers are normalized to `Normtree.ident_t`, however, top identifiers are normalized to its `Id_string` variant, which only stores the original identifier's `string`, whereas normal identifiers are normalized to `Normtree.ident_t`'s `Id_bruijn` variant, which stores the appropriately assigned de Bruijn index. The type of `ident_t` is therefore:

```
type ident_t =
  | Id_bruijn of int
  | Id_string of string
```

By splitting relevant parts of the tree (`Typedtree.structure` to `Normtree.structure`) translation, we know when a top identifier's definition is encountered. The mapping from that identifier to `Id_string` is then simply put in current sub-structure's `bmap`. On the other hand, when a normal identifier is encountered for the first time, i.e. its definition does not appear in either `old_bmap` or `bmap`, we map it to `Id_bruijn` with the current value of the de Bruijn counter `bcounter`. Having done the mapping, all the identifiers are then simply replaced by whatever they are mapped to in either `old_bmap` or `bmap`. Note that if the identifier is found in `old_bmap`, the flag `seen_external` is set to `true`, which consequently means that the current (sub-)structure's `myname` depends on its prefix.

4.2.5 Normalization of paths

A path, e.g. `List.map`, is a reference to something external of the current module. Its type is found in `typing/path.ml`:

```
type t =
    Pident of Ident.t
  | Pdot of t * string * int
  | Papply of t * t
```

We translate `paths` to `npaths`, the definition of which is found in `hashing/normtree.ml`:

```
and npath =
    Bruijn_param of ident_t
  | Hash_param of Hashpackage.hash_param * string
```

In case a simple identifier is encountered (`Pident`), we normalize it as a normal identifier (`tr_id`) putting the result in `Bruijn_param`. In case of a compound structure (`Pdot`) of our `path`, e.g. `List.map`, however, we take the following steps:

1. get the `path'` to the containing module, i.e. `List` in our example,
2. locate the offset `vd` of the `myname` field within that module using the local environment (`env : Env.t`) using `get_offset env path'`,
3. create an expression `path_expr` accessing the field with offset `vd` in the module,

4. add path_expr to state's param_set using Hashpackage.add_hash_param, which also returns the sequence number param (see §4.5) of the hash parameter added to the set,and

5. return (insert into the normalized tree) the Hash_param containing param and the name of the member being accessed, i.e. map in our example.

Note that a reference to a member of a module, which has been opened, will be converted to a Pdot, as if there were no open statements and all accesses were made by specifying a full path, e.g. a reference to max with module Pervasives implicitly open results in a path Pervasives.max being seen during normalization. This conversion happens earlier in the compilation process.

The last case of the path dealing with the functor applications (Papply) is not supported yet.

4.2.6 Dependency upon external, pre- and sub- structures

As mentioned before, the myname of a structure depends on many things, including:

1. the (hash of the) normalized version (Normtree.structure) of the structure itself (including the sub-structures),

2. the mynames of modules, which are referenced within the structure itself, but *not* also within the sub-structures,

3. the mynames of immediate sub-structures (this is how their external dependencies are included), and

4. the (hash of the) normalized version of the prefix of the structure (within the same compilation unit), and the mynames of the modules it references.

All of the above are then hashed into a single 256bit long string. A lot of myname calculation is performed at execution-time, when it is required; however, when possible, the computation (or part of it) is performed at compile-time. Since the myname of a structure depends on mynames of its sub-structures, it is easiest to put the myname field at the end of each structure. However, this way we always have to determine the offset of the myname field as shown in the previous section.

4.2.7 Generating expressions to compute mynames

When the normalisation of a (sub-)structure is complete, the following steps are performed:

1. combine the Hashpackage.packages of normalized Typedtree.structure_items of the current structure (head of packages in state_t) into a single package P of same type by using the Hashpackage.combine_packages function,

2. add P to the packages of the super-module's state,

3. add expressions accessing mynames of immediate sub-structures (stored in mod_defs) to hash_param_set of P,

4. if the sub-module depends on its prefix (seen_external is true), combine each list in the list of lists of Hashpackage.packages of normalized Typedtree.structure_items of the prefix (tail of packages) into a single list of packages Q of the same type, then generate_code for each package in Q, obtaining a Hashpackage.hash_param_set H, which is added to P using Hashpackage.add_hash_params_to_package, and

5. add a myname field at the end of the structure, and

6. depending on the myname type mode of the compilation unit (optionally specified as the first word of a file), the value of the myname is an expression, which:

   - hashes P, if myname type mode is left unspecified,

- generates a pseudo-random 256bit `string` at execution-time, if myname type mode is set to ”fresh”, or
- generates a pseudo-random 256bit `string` at compile-time, if myname type mode is set to ”cfresh”.

4.2.8 Example

```
module M = struct
  let x i = i          (* package_x *)
  let y = (x 7) + 29   (* package_y *)
  module N = struct
    let w = max        (* package_w *)
    let z = y          (* package_z   [external dependency detected] *)
  end                  (* package_N *)
  let k = 4            (* package_k *)
end                    (* package_M *)
```

At end of scanning N, we have the following relevant part of the state:

```
seen_external = true
param_set = [(Pervasives.myname, "max")]
packages = [[package_z; package_w]; [package_y; package_x]]
mod_defs = []
```

where

```
package_x = ([let x = fun Id_bruijn (0) -> Id_bruijn (0)], [])
package_y = ([let y = ((Id_string ("x")) 7) + 29], [])
package_w = ([let w = Hash_param (0, "max")], [(0, Pervasives.myname)])
package_z = ([let z = Id_string ("y")], [])
```

As described above, we take the following steps:

1. `P = ([package_zw], [])`

   where

   ```
   package_zw = ([let z = Id_string ("y"); let w = Hash_param (0, "max")],
                 [(0, Pervasives.myname)])
   ```

2. `M.packages = [package_zw; package_y; package_x]`

3. *(this step is skipped, since module N has no sub-modules)*

4. ```
   Q = [package_yx]
   H = [generate_code package_yx]
   P = ([let z = Id_string ("y"); let w = Hash_param (0, "max")],
        [H; (0, Pervasives.myname)])
   ```

   where

   ```
   package_yx = ([let y = ((Id_string ("x")) 7) + 29;
                  let x = fun Id_bruijn (0) -> Id_bruijn (0)], [])
   H = [compile-time-hash package_yx]
   ```

5. add `myname` to the end of module `N`

6. `N.myname = runtime-hash P`
   ```

At end of scanning M, then, we have the following relevant part of the state:

```
seen_external = false
param_set = []
packages = [[package_k; package_zw; package_y; package_x]]
```

where

```
package_x = ([let x = fun Id_bruijn (0) -> Id_bruijn (0)], [])
package_y = ([let y = ((Id_string ("x")) 7) + 29], [])
package_zw = ([let z = Id_string ("y"); let w = Hash_param (0, "max")],
              [(0, Pervasives.myname)])
package_k = ([let k = 4], [])
```

Again, we take the same steps as before:

1. `P = ([package_M], [])`

   where

   ```
   package_M = ([let k = 4;
                 [let z = Id_string ("y"); let w = Hash_param (0, "max")];
                 let y = ((Id_string ("x")) 7) + 29;
                 let x = fun Id_bruijn (0) -> Id_bruijn (0)],
                 [(0, Pervasives.myname)])
   ```

2. *(this step is skipped, since there is no super-module)*
3. `P = ([package_M], [N.myname])`
4. *(this step is skipped, since* `M` *does not depend (and has no) prefix*
5. add `myname` to the end of module `M`
6. `M.myname = runtime-hash P`

4.2.9 Aliasing (of functor parameters)

A functor's application's myname only depends on the myname of its parameter's myname if the functor's body actually uses the parameter *or* its alias anywhere. An example:

```
module F(U : sig val x : int end) =
  struct
    module L = U
    let y = L.x
  end
```

The lambda code for this looks like:

```
(setglobal T!
  (let
    (F/77
      (function U/73
        (let
          (L/74 U/73
           y/75 (field 0 L/74)
           myname/80
             (hash256_checked
```

```
                (makeblock 0 "\\\170\187 .. \237\191"
                  (field 1 L/74) "G\170\024 .. \221\135ZZ")))
           (makeblock 0 L/74 y/75 myname/80)))
     myname/93 "+\250\146 .. \198\179|G;\027p\029")
    (makeblock 0 F/77 myname/93)))
```

As you can see, the functor's myname correctly includes the first field of `L`, which is actually `U.myname`. This is all done automatically by the original compiler.

4.2.10 Insertion of mynames into module signatures

With `hashing/transig.ml` within `typing/typemod.ml`.

4.3 Debugging

Compiling with option `-dnormtrans`, the compiler will output lots of information about the various aspects of the normalization of the tree. With option `-dnormtree`, the compiler outputs the most of the resulting normalized tree using the `hashing/npretty.ml`.

4.4 Normalization of functors

4.4.1 Functor expressions

An O'Caml functor is defined as one of module expression descriptions:

```
type module_expr_desc =
    ...
  | Tmod_functor of Ident.t * module_type * module_expr
    ...

and module_expr =
  { mod_desc: module_expr_desc;
    mod_loc: Location.t;
    mod_type: module_type;
    mod_env: Env.t }
```

Therefore, the definition contains the name of the functor parameter (`Ident.t`), the type of the functor parameter (`module_type`), and the body of the functor (`module_expr`).

- `Ident.t` – the name of the functor parameter.
- `module_type` – the type of the functor parameter.
- `module_expr` – the body of the functor containing:
  - `module_expr_desc` – the description of the functor body.
  - `Location.t` – the location of source code for the functor body.
  - `module_type` – the type of the functor body.
  - `Env.t` – the environment of the functor body.

4.4.2 Normalisation of the parameter

Say we have the following example:

```
module Foo = struct
  module type T = sig val x : int end
  module F(P : T) = struct
    module P' = P
    module M = struct
      let _ = P.x
      let _ = P'.x
    end
  end
end
```

Because the functor's parameter `P` is local, the path `P.x` needs to be normalised (de Bruijn-ed):

```
module Foo = struct
  module type T = sig val x : int end
  module F(0 : T) = struct
    module P' = 0
    module M = struct
      let _ = 0.x
      let _ = P'.x
    end
  end
end
```

The de Bruijn counter is reset to 0 every time a new structure is entered, *or just before every functor's parameter*.

Note, however, that aliases of the parameter are not normalised, since they can be accessed from outside the functor, i.e. `F(X).P'` where `X` is a module of type `T`.

4.4.3 Hash-parameter scope

Normally, a path such as `P.x` generates a hash parameter `(P.myname, "x")`, which is used when calculating myname of any module that encloses the path (contains the expression). In our example above, the hash parameter should, therefore, be used for calculating the myname of `M`, `F` and `Foo`. However, `P` is not in scope at the end of `Foo` (where its myname is calculated), therefore the expression `P.myname` would generate an error. For this reason, we use hash parameters in the enclosing module only if they are in its scope. In our example, this means that hash parameter `(P.myname, "x")` is not used when calculating myname of `Foo` — `Foo`'s myname still holds all the information about the enclosed functor, since it contains the hash of its normalised tree.

Note that the same rule holds for aliases of functor's parameter, e.g. `P'`, since `(P'.myname, "x")` would generate an error outside functor `F`.

4.5 Hash packages

We mentioned before that structure's myname depends on the mynames of the structures, which the original structure uses. This is done by including expressions accessing these mynames into the expression computing the myname. Hash package (`hashing/hashpackage.ml`) is used to store the dependencies, insert appropriate links in the `Hashpackage.hash_param` normalized tree, and finally generate the expression, which computes structure's myname.

A brute-force approach would insert an expression for each external dependency found in the original tree. Instead, a package (`Hashpackage.hashpackage`) stores this dependencies in a map, where dependencies (expressions accessing mynames of structures being used) are mapped to indices. Whenever a dependency is begin added (`Hashpackage.add_hash_param`), it is added (if not already present in the map) and the corresponding index is returned. This index is then inserted into the normalized tree where the dependency arose from, so that no information is lost. This way only one expression is generated for each distinct dependency, which drastically speeds up the runtime computation of mynames.

Note that each (sub-)structure stores a separate hash package, which logs all of the (sub-)structure's external dependencies. The super-structure gets those dependencies by including expressions accessing mynames of its sub-structures into the expression for computing its own myname — in a sense, the super-structure depends on its sub-structures.

4.6 Normalization of signatures

Signatures are normalized in much the same way as the rest of the typed tree structure. The definition of a normalized signature can also be found in `hashing/normtree.ml`.

Chapter 5

Type normalization

5.1 Overview

Broadly speaking, there are two parts of the compiler that require the conversion of O'Caml types and type declarations to normalized forms. They are:

1. the `Normtrans` module that calculates the hashes of modules' abstract syntax trees;
2. the `Polymarshal` module that performs the type-passing translation.

The normalization in these two cases is performed by the `Normtypedecl` and `Normtypes` modules, respectively.

The two applications require types and type declarations to be normalized in different ways. When calculating the hashes of modules' abstract syntax trees, we are simply interested in normalizing type declarations in such a way that we work up to α-conversion and collect the names of any external modules on which those declarations depend. Similarly, when normalizing types in this scenario, we are just interested in the external modules on which they depend.

The process of type normalization invoked by the `Polymarshal` module in order to calculate type representations that are to be passed around at runtime works differently. In this scenario, when faced with a type such as `int t` (say `'a t` is a variant type), we wish to include any known declaration of the type `t` into the normalized type. Including the declaration like this ensures that the runtime type representations of types with the same names but different declarations are distinct.

It is not obvious why the procedure of type normalization invoked from `Normtrans` does not need to perform this additional inclusion of the declarations of types. The reason why it does not is because it already happens elsewhere: prefix hashing ensures that the declaration of any type defined in the same module (or a previous substructure thereof) will already have been included in the hash; similarly, if the type is defined in an external module then the `myname` field of that module will be included in the hash.

The differences can be summarized by saying that the procedure used by `Normtrans` hashes individual types and type declarations before combining them all together with any external `myname` fields, whereas that used by `Polymarshal` takes a type as a standalone entity and returns a type representation encompassing all of the type declarations on which that particular type depends.

5.2 Background: O'Caml type expressions

An O'Caml type expression is represented by the following data structure:

```
type type_expr =
  { mutable desc: type_desc;
```

```
    mutable level: int;
    mutable id: int }

and type_desc =
    Tvar
  | Tarrow of label * type_expr * type_expr * commutable
  | Ttuple of type_expr list
  | Tconstr of Path.t * type_expr list * abbrev_memo ref
  | Tobject of type_expr * (Path.t * type_expr list) option ref
  | Tfield of string * field_kind * type_expr * type_expr
  | Tnil
  | Tlink of type_expr
  | Tsubst of type_expr
  | Tvariant of row_desc
  | Tunivar
  | Tpoly of type_expr * type_expr list

and row_desc =
    { row_fields: (label * row_field) list;
      row_more: type_expr;
      row_bound: type_expr list;
      row_closed: bool;
      row_fixed: bool;
      row_name: (Path.t * type_expr list) option }

and row_field =
    Rpresent of type_expr option
  | Reither of bool * type_expr list * bool * row_field option ref
  | Rabsent

and abbrev_memo =
    Mnil
  | Mcons of Path.t * type_expr * type_expr * abbrev_memo
  | Mlink of abbrev_memo ref

and field_kind =
    Fvar of field_kind option ref
  | Fpresent
  | Fabsent

and commutable =
    Cok
  | Cunknown
  | Clink of commutable ref
```

5.3 Background: O'Caml type declarations

An O'Caml type declaration is represented by the following data structure:

```
type type_declaration =
  { type_params: type_expr list;
    type_arity: int;
    type_kind: type_kind;
```

```
      type_manifest: type_expr option;
      type_variance: (bool * bool * bool) list }

  and type_kind =
      Type_abstract
    | Type_variant of (string * type_expr list) list * private_flag
    | Type_record of (string * mutable_flag * type_expr) list
                   * record_representation * private_flag
```

The `type_kind` field identifies the basic variety of type being defined. The possible interpretations are:

- `Type_abstract` – an abstract type, including those predefined abstract types such as `int`.
- `Type_variant` – a standard sum type (not a polymorphic variant). In this case there is extra data encoded in the type kind consisting of:
  1. a list of constructor descriptions, each consisting of the name of the constructor and a list containing the types of its parameters;
  2. a flag identifying whether the type is public or private.
- `Type_record` – a record type. The extra data in this case consists of:
  1. a list containing information about the record type's components: for each there is the label, a flag identifying whether it is mutable, and the type of the component;
  2. a flag specifying whether the record type's values are to be laid out in the normal fashion or the optimized fashion used for floating-point records;
  3. a flag identifying whether the type is public or private.

The `type_params` field gives the type parameters of the type constructor being defined. I think these are always type variables.

The `type_arity` field specifies the number of type parameters to the type constructor being defined.

The `type_manifest` field is only not `None` in the case where the type declaration is of the form of that for `t'` below:

```
# type t = C1 of int | C2 of string;;
type t = C1 of int | C2 of string
# type t' = t = C1 of int | C2 of string;;
type t' = t = C1 of int | C2 of string
```

In this case, the argument to `type_manifest` would correspond to the equality with the type `t` specified in the second declaration. (See case 4 "Re-exported variant type or record type: an equation, a representation" of `http://caml.inria.fr/pub/docs/manual-ocaml/manual016.html`.)

The `type_variance` field is a list of entries specifying the variance properties of each type parameter. Each entry consists of three entries that indicate whether the corresponding type parameter occurs in covariant, contravariant and weakly contravariant[1] positions respectively on the right-hand side of the declaration. Such variance annotations may also be present for abstract types (they can be supplied by the user, for example `type +'a t`).

[1] I believe that a type variable is *weakly contravariant* if it occurs on the left of a function arrow, even if the particular occurrence of the variable is in fact positive. The necessity for such information seems to be due to a principality-of-type-inference issue identified on page 15 of Garrigue's "Relaxing the value restriction" paper.

5.4 Normalization for AST hashing (`Normtypedecl`)

The target datatypes are currently:

```
(* Normalized type declarations. *)
type ntype_decl = NTDabstract of string
                | NTDvariant of (string * (ntype list)) list
                | NTDrecord of (string * Asttypes.mutable_flag * ntype) list
                | NTDabbreviation of ntype

(* Normalized types. *)
and ntype = NTarrow of ntype * ntype
          | NTtuple of ntype list
          | NTctor_param of int
          | NTconstructed of type_constructor_info * (ntype list)
          | NTunsupported
          | NTvar of int
          | NTunivar of int
          | NTpoly of int * ntype

(* Information about type constructors used within normalized types. *)
and type_constructor_info =
  TCabbreviation of ntype                (* abbreviation *)
| TCbuiltin of string                    (* built-in abstract tycon *)
| TClocal of string                      (* abstract tycon in same module
                                            or superstructure *)
| TCexternal of string * string          (* external abstract tycon *)
| TCbeing_defined of string              (* (recursive) reference to
                                            a tycon being defined in
                                            the current type
                                            declaration(s) *)
| TCthrough_functor of int * string      (* something like H.t where
                                            H is an in-scope functor
                                            argument *)
```

Type declarations are classified into one of the following categories:

Declaration of abstract type. The name of the type is stored.

Declaration of variant type. The textual name of each constructor along with the normalized types of its arguments are stored.

Declaration of record type. The textual name of each field along with its mutable/immutable flag and its normalized type is stored.

Declaration of type abbreviation. The normalized type to which the abbreviation expands is stored.[2] (Data constructor `NTDabbreviation`.)

A type constructor is classified into one of the following categories:

Type abbreviation. The normalized type to which the abbreviation expands is stored. (Data constructor `TCabbreviation`.)

Built-in type constructor. The textual name of the type constructor is stored. (Data constructor `TCbuiltin`.)

[2] What should actually happen is that abbreviations should be fully expanded before this normalization happens, in order that the hashing of ASTs is modulo type abbreviations. However this is not currently implemented.

Locally-defined type constructor. Applies if the type constructor is defined in the current module at an earlier point to where we are currently yet at the same textual level. (Currently it is not used for type constructors previously-defined in a substructure of the current module.) The textual name of the type constructor is stored. (Data constructor `TClocal`.)

Externally-defined type constructor. Applies if the type constructor is defined in an external module or a previously-defined substructure of the current module. The module name and the remaining textual path of the type constructor is stored. (Data constructor `TCexternal`.)

Recursive references. Used to signify a recursive reference to a type constructor being defined by the current type declaration being normalized. The textual name of the type constructor in question is stored. (Data constructor `TCbeing_defined`.)

Through-functor-argument references. Used to represent a type constructor defined in a module that is an in-scope functor argument (for example, it would be used in the body of a functor when normalizing `H.t` where `H` is an in-scope functor argument). The de Bruijn index of the functor argument (as assigned by `Normtrans`) and the textual name of the type constructor minus the functor argument prefix is stored. (Data constructor `TCthrough_functor`.)

Normalized types themselves are classified as follows.

Type variables (`NTvar`). The argument is an integer stamp. Note that (unlike in `Normtypes`) this stamp is *not* the type variable stamp used in type inference. Instead it is one assigned in a canonical manner by the normalization procedure. This ensures that the normalized type value is independent of the order in which type variables were created by the rest of the compiler.

Bound univars (`NTunivar`). The de Bruijn index of the univar is stored.

Function types (`NTarrow`). The normalized forms of the argument and result types are stored.

Tuple types (`NTtuple`). The normalized forms of the types of the tuple's components are stored.

Bound type constructor parameters (`NTctor_param`). The de Bruijn index of the type constructor parameter is stored. When normalizing a type declaration (say that of a variant type `'a t`) we may encounter the parameters of the type constructor being defined (`'a` in this case). Such parameters are to be treated as bound variables up to α-conversion and are hence represented by de Bruijn indices.

Constructed types (`NTconstructed`). The information about the type constructor, as described above, and the normalized forms of the argument types are stored.

Polytypes. The number of bound univars and the normalized form of the body type are stored.

Unsupported types (`NTunsupported`). Used for any varieties of types not listed above.

5.4.1 Normalization algorithm

We may either start by normalizing a type declaration or a type. We treat these two cases separately, although the first of course relies on the second.

5.4.1.1 Normalization of type declarations

The inputs are:

- the environment to be used;
- a list of identifiers to be treated as the names of locally-defined type constructors (to be used in the case where they will not exist in the environment, such as during signature normalization);

- a list of identifiers to be treated as in-scope functor arguments;
- a list of (identifier, type declaration) pairs giving the type constructors being declared and their associated declarations to be normalized.

During the process two maps are kept, both from type stamps (the `id` fields) to integers. One is used to assign de Bruijn indices to the parameters of type declarations whilst the other is used to assign de Bruijn indices to the bound univars of polytypes.

For each type declaration we proceed as follows.

At the start of the process both maps are cleared. Then as many fresh type variables as there are parameters to the current declaration are allocated and assigned de Bruijn indices. (The fresh variables are going to be used to instantiate the parameters of the type declaration before we traverse it, to avoid any possibility of name clashes.) We then case split according to the variety of declaration.

- If the declaration is that of an abstract type declaration, we simply construct an `NTDabstract` node containing the name of the type constructor being declared.
- If the declaration is that of a concrete type declaration, it is either declaring a variant or record type. These yield values whose outermost constructor is `NTDvariant` or `NTDrecord`, respectively. In either case, we instantiate the declaration at the fresh type variables. Then, if we have a variant, we normalize each constructor declaration by applying the procedure in §5.4.1.2 to each argument type; in the other case, where we have a record, we proceed similarly to apply §5.4.1.2 to the type of each field. The constructor declarations and record fields are then sorted by name to ensure that the eventual hash value is independent of the order of the constructors.
- If the declaration is that of a type abbreviation, we take the type to which the abbreviation expands and substitute the fresh type variables for any occurrences of the declaration's parameters throughout it. We then follow the procedure in §5.4.1.2 to normalize the type.

5.4.1.2 Normalization of types

The inputs are:

- the environment to be used;
- a list of identifiers to be treated as the names of locally-defined type constructors (to be used in the case where they will not exist in the environment, such as during signature normalization);
- a list of identifiers to be treated as in-scope functor arguments;
- the type to be normalized.

During the process we keep a map from type stamps to type variable stamps. This is used to assign the contents of `NTvar` nodes in a canonical manner that ensures that the resulting normalized type is independent of the order in which the rest of the compiler allocates type variables. Initially this map will be empty.

We also keep maps from constructor parameter type stamps to de Bruijn indices, and from univar type stamps to de Bruijn indices, as in §5.4.1.1.

We proceed by recursion down the structure of the type. The various cases are as follows.

Type variables. Check to see if the type variable's stamp is mapped by the constructor parameter map. If so, then we emit an `NTctor_param` node. If not, check the type variable stamp map to see if this type variable has been seen before. If so, we use the corresponding canonical stamp in the map to form an `NTvar` node. Failing that, we assign a new canonical stamp, update the map and form a corresponding `NTvar` node.

Univars. Form an `NTunivar` containing the de Bruijn index of the univar, as identified from the univar map.

Polytypes. Save the univar map, update it to include new de Bruijn indices for the bound univars of this polytype, and normalize the body. Then restore the univar map and return an `NTpoly` value containing the number of bound univars and the normalized body.

Function types. Form an `NTarrow` node containing the normalized forms of the argument and result types.

Tuple types. Form an `NTtuple` node containing the normalized forms of the types of the tuple's components.

Constructed types. First normalize the arguments to the constructed type. Then check to see if the type constructor is one whose type declaration we are currently normalizing; if it is, return an `NTconstructed` node encapsulating a `TCbeing_defined` node and the normalized arguments. Otherwise, check to see if the type constructor is listed in the list of type constructors to be treated as locally-defined. If it is, return a `NTconstructed` node encapsulating a `TClocal` node and the normalized arguments.

In other cases, we proceed by looking up the type declaration in the environment. If it cannot be found, we check to see if the path of the type constructor contains a dot. If it does, we check to see if the part of the path to the left of the leftmost dot is that of an in-scope functor argument. If it is, the de Bruijn index of that functor argument is packaged up with the remainder of the path in a `TCthrough_functor` node that is then wrapped inside an `NTconstructed` node together with the normalized arguments to the type constructor. If the check for functor arguments fails, we issue a warning and return `NTunsupported`. (This should really be an assertion failure, but the use of the `include` keyword would trigger it. To be fixed in the future: some careful analysis of how to cope with `include` is needed.)

In all other cases, we have now found the type declaration in the environment. Just like when directly normalizing a type declaration, as many fresh type variables as there are parameters to the current declaration are allocated and assigned de Bruijn indices. (The fresh variables are going to be used to instantiate the parameters of the type declaration before we traverse it, to avoid any possibility of name clashes.) We then check to see if the declaration is that of a type abbreviation; if it is, we substitute the fresh type variables for the declaration's parameters throughout the type to which the abbreviation expands, normalize the resulting type, and then return a `TCabbrevation` node encapsulated (together with the normalized arguments to the type constructor) in an `NTconstructed` node[3].

If the declaration is not a type abbreviation, first check to see if it is one of the type declarations from the initial environment. If it is, then return a `TCbuiltin` node encapsulated (together with the normalized arguments to the type constructor) in an `NTconstructed` node. Otherwise, we need to examine the type constructor's path to determine which module the declaration lies in. There are three cases:

Path of length one. The type constructor is defined at the same scoping level in the current module, or in a superstructure. Simply emit a `TClocal` node encapsulated (together with the normalized arguments to the type constructor) in an `NTconstructed` node.

Path of length two or more. Split the path at the leftmost dot, since we have to emit a `myname` reference. Record the part of the path to the left of the leftmost dot so that we can cause `Normtypes` to emit that reference. Then return a `TCexternal` node (whose arguments are the name of the external module whose `myname` we are going to depend on, together with the rest of the original type constructor path) and wrap it (together with the normalized arguments to the type constructor) in an `NTconstructed` node.

[3]As identified previously, what should actually happen is that abbreviations should be fully expanded before this normalization happens, in order that the hashing of ASTs is modulo type abbreviations.

Application path. (The `Papply` case.) This is an assertion failure, since such a path can never be that of a type constructor (it is always the path of a module).

Links to other types (`Tlink` and `Tsubst`). Return the normalized form of the type being linked to.

Anything else. Issue a warning and return `NTunsupported`.

5.5 Normalization for type-passing (`Normtypes`)

5.5.1 Means of representing types

Inside the compiler and its output files, types are represented in the following ways:

- *normalized type trees* – the initial output from the type normalization algorithm;
- *flattened normalized types* – a list representation of normalized type trees;
- *optimized flattened normalized types* – flattened normalized types after having a certain optimization applied, which we explain below;
- *type representation blocks* – the runtime version of optimized flattened normalized types (code to construct these may contain free value identifiers and references to `mynames` as described later);
- *type representations* – type representation blocks that have been flattened and passed through a hash function.

Each representation above is computed from the variety immediately above it in the list, if one such exists. We now examine each variety in turn.

5.5.1.1 Normalized type trees

Normalized type trees are values of the following datatype.

```
type ntype = NTvar of int
           | NTunivar of int
           | NTarrow of ntype * ntype
           | NTtuple of ntype list
           | NTctor_param of int
           | NTconstructed of string * (Longident.t list) * (ntype list)
           | NTisorecursive_loop of string
           | NTpoly of int * ntype
           | NTother
           | NTloop
```

The various constructors and their arguments are as follows.

Type variables (`NTvar`) The argument is an integer stamp specifying which type variable is being represented. This is the same as used during type inference (the `id` field of the type). Recall that in `Normtypedecl` we assign stamps to normalized type variables in a canonical order to guarantee a certain independence: here, we do not need to do this since values of type `ntype` are subsequently transformed in such a way that the `NTvar` nodes actually turn into value identifier references – the stamps themselves do not get into the final hash.

Bound univars (`NTunivar`) The argument is a de Bruijn index. Bound univars (those type variables bound by the quantification in a polytype) are to be treated up to α-conversion and are hence assigned de Bruijn indices.

Function types (`NTarrow`) The arguments are the argument and result types in the usual manner.

Tuple types (`NTtuple`) The arguments are the types of the components in the usual manner.

Bound type constructor parameters (`NTctor_param`) The argument is a de Bruijn index. When normalizing a type declaration (say that of a variant type `'a t`) we may encounter the parameters of the type constructor being defined (`'a` in this case). Such parameters are to be treated as bound variables up to α-conversion and are hence represented by de Bruijn indices.

Constructed types (`NTconstructed`) The arguments are:

1. the SHA-256 hash of the normalized type declaration of the type constructor being applied;
2. a list of modules on which that normalized type declaration depends (all of these modules' `myname` values must be hashed into the eventual type representation);
3. the normalized forms of the types at which the type constructor is being applied.

The normalized form of the type declaration in question is the hash of a value of type `ntype_decl`, as follows:

```
type ntype_decl = NTDexternal_abstract of string
                | NTDbuiltin_abstract of string
                | NTDlocal_abstract of string
                | NTDvariant of (string * (ntype list)) list
                | NTDrecord of
                    (string * Asttypes.mutable_flag * ntype) list
                | NTDmanifest of ntype
```

The various cases here are:

External abstract type constructors (`NTDexternal_abstract`) Used for type constructors defined in an external module (and also in substructures of the current module previous to the current point being normalized[4]). The path of the type constructor, minus the portion of the path to the left of the leftmost dot, is stored as a string. For example the path `M.t` turns into `NTDexternal_abstract "t"` and `M.N.t` turns into `NTDexternal_abstract "N.t"`. The portion to the left of the leftmost dot corresponds to the outermost of the modules surrounding the definition of the type constructor (in this case `M`); rather than storing this module name inside the value of type `ntype_decl` it is emitted separately so that the appropriate `myname` dependency can be added. (Note that the `myname` values cannot be included directly in the hash because they may not be calculated until runtime.)

Built-in abstract type constructors (`NTDbuiltin_abstract`) This case is used for abstract type constructors defined in the initial environment (*viz.* `typing/predef.ml`). The textual name of the type constructor is stored.

Local abstract type constructors (`NTDlocal_abstract`) This is used for abstract type constructors defined at the current scoping level in the current module or in a superstructure thereof. The textual name of the type constructor is stored.

Variant types (`NTDvariant`) The textual name of each data constructor is stored, together with a list containing the normalized forms of the types of its arguments.

Record types (`NTDrecord`) Declaration of a record type. For each field of the record, the field name, mutable/immutable flag and the normalized form of the argument type are stored.

[4]This means that a reference to a type constructor defined inside a previously-defined substructure of the current module will yield a type representation dependent on the entire contents of that module.

Type abbreviations (`NTDmanifest`) Declaration of a type abbreviation. The normalized form of the type to which the abbreviation expands is stored[5].

Recursive occurrences of type constructors (`NTisorecursive_loop`) When normalizing a type declaration we may encounter occurrences of the type constructor being defined. Such recursive occurrences turn into nodes of this data constructor; the argument is the textual form of the type constructor.

Polytypes (`NTpoly`) The number of bound univars and the normalized form of the body is stored. (Occurrences of the univars inside the body will turn into `NTunivar` nodes.)

Other loops (`NTloop`) Used if an actual loop is found in the type structure. (It is not clear how these can arise.)

Unsupported types (`NTother`) Used for types of any variety not listed above.

5.5.1.2 Flattened normalized types

The idea of a flattened normalized type is to obtain a primarily textual, flat representation of a normalized type tree: such a representation takes less space and is easier to manage than a tree structure. These representations cannot just be strings because of two things: firstly, there may be type variable nodes in the normalized type tree and these must eventually turn into dependencies on identifiers corresponding to those type variables; secondly, there may be external module references that have been identified during production of the normalized type tree and these must also be preserved so the relevant `myname` dependencies can eventually be emitted.

Therefore a flattened normalized type is represented as a list of `flat_ntype_entry` values, which themselves have the form:

```
type flat_ntype_entry = FNtyvar of int
                      | FNmyname of Env.t * Longident.t
                      | FNstring of string
```

Such flattened types are generated from normalized type trees by traversing the trees and assigning mainly textual codes to the various nodes. The actions are as follows, depending on the type of node encountered.

Type variables. The emitted flattened normalized type is a singleton list containing an `FNtyvar` node, whose argument holds the stamp from the corresponding `NTvar` node.

Univars. The emitted flattened normalized type is a singleton list whose element is an `FNstring` holding the de Bruijn index of the univar prefixed by `U`.

Function types. The emitted flattened normalized type is a list whose elements are as follows (in order):

1. a textual string `FNstring "F("`;
2. the `flat_ntype_entry` values obtained by flattening the normalized form of the argument type;
3. a textual string `FNstring ")("`;
4. the `flat_ntype_entry` values obtained by flattening the normalized form of the result type;
5. a textual string `FNstring ")"`.

Tuple types. The emitted flattened normalized type is a list whose elements are as follows (in order):

[5] What should actually happen is that abbreviations should be fully expanded before this normalization happens, just as it should in `Normtypedecl` as previously noted.

1. a textual string `FNstring "T"`;
2. for each component of the tuple type,
   (a) a textual string `FNstring "("`;
   (b) the `flat_ntype_entry` values obtained by flattening the normalized form of the component's type;
   (c) a textual string `FNstring ")"`.

Bound type constructor parameters. A list containing an `FNstring` holding the de Bruijn index of the parameter prefixed by `C` is emitted.

Polytypes. The emitted flattened normalized type is a list whose elements are as follows (in order):

1. a textual `FNstring` formed by concatenating the character `Y`, the number of bound univars in the polytype, and an underscore, in that order;
2. the `flat_ntype_entry` values obtained by flattening the normalized form of the body type.

Constructed types. Firstly, any external dependencies that were identified during the production of the normalized type tree are collected together and turned into `FNmyname` nodes referencing the appropriate modules and environments. (The arguments to such a node are the environment in which the `myname` lookup may be performed, together with the module prefix – so if the prefix is `M` then a reference to `M.myname` will be emitted.)

Having done that the emitted list is then:

1. the string node `FNstring "E"`;
2. another `FNstring` node holding the hash of the type declaration (as stored in the `NTconstructed` node);
3. the list of `FNmyname` nodes from above;
4. for each argument of the constructed type,
   (a) a textual string `FNstring "("`;
   (b) the `flat_ntype_entry` values obtained by flattening the normalized form of the argument's type;
   (c) a textual string `FNstring ")"`.

Recursive occurrences of type constructors. A list containing an `FNstring` holding the name of the type constructor prefixed by `I` is emitted.

Loops. A list containing just `FNstring "P"` is emitted.

Unsupported types. A list containing just `FNstring "O"` is emitted.

5.5.1.3 Optimized flattened normalized types

Flattened normalized types often have contiguous runs of `FNstring` nodes due to the way in which the flattening algorithm operates. An optimized flattened normalized type is one where there are never any adjacent `FNstring` nodes: such a value is formed by traversing a flattened normalized type and collecting contiguous runs of `FNstring` nodes into single `FNstring` nodes holding the strings from the original nodes concatenated together.

5.5.1.4 Type representation blocks

Type representation blocks are the runtime representations of optimized flattened normalized types. Such blocks are heap-allocated with tag zero and have fields containing:

- constant strings (corresponding to `FNstring` nodes); or
- pointers to other type representation blocks.

Pointers to other type representation blocks arise due to the evaluation of code emitted to correspond to `FNmyname` and `FNtyvar` nodes inside an optimized flattened normalized type. For when emitting code for `FNmyname`, we emit a value identifier (such as `M.myname`), and when emitting code for `FNtyvar`, we do the same (in this case the identifier would be something like `_tyrep_1000`). At runtime, these identifiers will be 'pointing at' other type representation blocks, and thus the second case in the list above arises.

5.5.1.5 Type representations

Type representations are computed from type representation blocks. This computation happens only when the code generated for a `dyntype` or `rep` is executed. That generated code calls a C primitive `flatten_typerep_block` (defined in `hash256.c`) that takes a type representation block as argument.

The purpose of that primitive is to flatten the type representation block into a single string and then call the SHA-256 hash function on the result. To perform the flattening, a pass is made over the type representation block, concatenating adjacent strings together. If one of the fields of a type representation block turns out to be a pointer to another such block (as it will be if the type representation block is parameterized on another – for example if it represents the type $\alpha \times$ `int` – or if if the type representation block is parameterized on a `myname`), then that pointer is followed to continue the flattening; at the end of that we continue at the previous level, etc.

For example this block (displayed as the output of `-drawlambda` and abbreviated):

```
(makeblock 0 "T(" _tyrep_664/99999 ")(E ... )")
```

flattens to the string `"T(XXX)(E ...  )"` where `XXX` is the result of flattening the value of the identifier `_tyrep_664/99999` at runtime.

The flattening algorithm is implemented in C as noted above; it makes use of `setjmp()` and `longjmp()` to simulate exception handling.

5.5.1.6 Compositionality

The means of representing types are carefully constructed to ensure that compositionality is maintained. For example, given a type that does not contain any type variables and neither any external references, one might think that it is more efficient to simply hash the corresponding normalized type tree when the user compiles their code. This is likely to be incorrect, however, because if that (hashed) type representation could be substituted into some (non-hashed) larger one (say the type under consideration here is the instantiation of a type variable in a larger type) then we may lose compositionality.

Consider for example the code fragment:

```
let f1 x = dyntype (x, 42)
let f2 x = dyntype x

let a = f1 10
let b = f2 (10, 42)
```

The values of a and b should clearly be equal; however, if hashing happens too early then this will not be the case because the result of hashing the type representation obtained by substituting the (hashed) representation of int into the (non-hashed) representation of $\alpha \times$ int will not be the same as the (hashed) representation of int $\times$ int.

This problem could be solved by inserting lots of calls to the hash function (one at each point where two type representations are combined into another); compositionality would then be restored. However this will lead to an excessive number of calls to the hash function; we believe the solution described earlier that is used in the compiler to be far more efficient.

5.5.2 A note on recursive definitions

Before we embark on the details of the normalization algorithm, we remark on the procedure that will be used to normalize references to type constructors that are involved in mutually-recursive declarations. For example suppose we have the following:

```
type t1 = C of int | D of t2
 and t2 = E of int | F of t1
```

and we are normalizing a reference to the type t1. The intuition here is that we should get the hash of *both* type declarations paired with the string t1. Our algorithm computes something equivalent, but it is not really quite the same. This arises from the way in which the O'Caml compiler handles mutually-recursive declarations: they are stored as separate type declarations rather than in one structure. (The values corresponding to those type declarations aren't mutually recursive either: for example the t2 in the declaration of t1 is simply referenced via its path, and so one needs to perform environment lookups to retrieve further type declarations in order to traverse the 'recursive' loop.)

What we would do for the above is to note that we are examining the declaration of t1 on a stack and then delve inside it, normalizing its parameters. Upon reaching the occurrence of the type t2 we start a similar procedure. When we then reach the occurrence of t1 within the constructor argument of F, we attempt to normalize it but identify that we are already delving inside the declaration of t1. A recursive loop has therefore been identified. We are therefore not hashing the declarations of t1 and t2 *together*, but rather identifying when we are going to go in a cycle and noting that in the value to be hashed.

5.5.3 Normalization algorithm

The normalization algorithm (normalize_type and normalize_type_rec in Normtypes) is somewhat of a mess, having evolved together with our understanding of the O'Caml type structure, and could do with rewriting. The newer Normtypedecl module, already described, is significantly better in this regard. However whilst the code is untidy the algorithm behind it, explained forthwith, is believed to be sound (with the possible exception of not handling loops correctly).

The inputs are:

- the environment to be used;
- the type to be normalized.

Throughout the run of the compiler the following state is kept in the Normtypes module.

- A map from type expressions to normalized type expressions (keys compared using physical equality).

Additionally the following state is kept whilst normalizing a particular type.

- A map from constructor parameter type stamps to de Bruijn indices.
- A map from univar type stamps to de Bruijn indices.

- A map from type declarations to hashes of normalized type declarations and their associated external dependencies. This is highly important for efficient compilation as it saves recomputation of the normalized forms of type declarations.
- A set of type expressions (compared using physical equality) that we are already involved in examining.
- A set of type constructor paths whose declarations we are currently examining (used for detection of recursive loops in type declarations).

Upon entry to the algorithm, we check some special cases. If the type to be normalized is a type variable, we immediately generate the corresponding normalized type tree and convert it to the lambda code to generate a type representation block which is then returned. If the incoming type is a link to another (`Tlink` or `Tsubst` cases), we follow the link immediately. Otherwise, we check the type expression map to determine if we have seen the type before: if we have, we return the corresponding normalized type found as the data part of the map entry.

In all other cases we initialize the constructor parameter map, univar map, type declaration map and type constructor path set to empty. We then proceed by recursion down the structure of the type to generate a normalized type tree. In parallel with the generation of the normalized type tree, we also collect together all of the external module dependencies for the type being normalized. We leave the propagation of such information through the recursive procedure implicit in the description below in order to simplify matters as much as possible; it is clearly visible in the code, often using `List.fold_left` to combine dependency sets.

The normalized type tree is then flattened (using the external dependency information to produce the appropriate `FNmyname` nodes) to an optimized flattened normalized type tree and then converted to the lambda code to generate a type representation block.

The recursive procedure to produce the normalized type tree works as follows. First we check to see if the current type expression being examined has been seen before; if so we return an `NTloop` node. Otherwise, we note that we have visited this type expression, and case split; afterwards, we remove that note to show that that type expression is no longer in the process of being traversed. The various cases are as follows.

Type variables. Check to see if the type variable's stamp is mapped by the constructor parameter map. If so, then we emit an `NTctor_param` node. Otherwise, we emit an `NTvar` node encapsulating the type variable's stamp.

Univars. Form an `NTunivar` containing the de Bruijn index of the univar, as identified from the univar map.

Polytypes. Save the univar map, update it to include new de Bruijn indices for the bound univars of this polytype, and normalize the body. Then restore the univar map and return an `NTpoly` value containing the number of bound univars and the normalized body.

Function types. Form an `NTarrow` node containing the normalized forms of the argument and result types.

Tuple types. Form an `NTtuple` node containing the normalized forms of the types of the tuple's components.

Constructed types. First normalize the arguments to the constructed type. Then check to see if the type constructor is one whose type declaration we are currently normalizing; if it is, return an `NTisorecursive_loop` node containing the path of the type constructor.

In other cases, we find the type declaration in the environment (using the path of the type constructor) and proceed by looking up the type declaration to see if we already have a hash for it. If we do, we return an `NTconstructed` node containing that hash, the external dependencies (also stored in the map), and the normalized arguments to the type constructor.

Otherwise, we note that we are currently examining the type declaration. Just like in `Normtypedecl`, as many fresh type variables as there are parameters to the current declaration are allocated and assigned de Bruijn indices. (The fresh variables are going to be used to instantiate the parameters of the type declaration before we traverse it, to avoid any possibility of name clashes.) We then check to see if the declaration is that of a type abbreviation; if it is, we substitute the fresh type variables for the declaration's parameters throughout the type to which the abbreviation expands, normalize the resulting type, and then return a `NTDmanifest` node encapsulated (together with the normalized arguments to the type constructor) in an `NTconstructed` node.

If the declaration is not a type abbreviation, and is that of an abstract type, first check to see if it is one of the type declarations from the initial environment. If it is, then return an `NTDbuiltin_abstract` node encapsulated (together with the normalized arguments to the type constructor) in an `NTconstructed` node. Otherwise, we need to examine the type constructor's path to determine which module the declaration lies in. There are three cases:

Path of length one. The type constructor is defined at the same scoping level in the current module, or in a superstructure. Simply emit an `NTDlocal_abstract` node encapsulated (together with the normalized arguments to the type constructor) in an `NTconstructed` node.

Path of length two or more. Split the path at the leftmost dot, since we have to emit a `myname` reference. Record the part of the path to the left of the leftmost dot so that we can cause the flattening procedure to emit a `myname` reference. Then return a `NTDexternal_abstract` node, containing the rest of the original type constructor path, and wrap it (together with the normalized arguments to the type constructor) in an `NTconstructed` node.

Application path. (The `Papply` case.) This is an assertion failure, since such a path can never be that of a type constructor (it is always the path of a module).

If the declaration is not a type abbreviation, and is also not that of an abstract type, then it must be that of a variant or record type. In this case, we first substitute the fresh type variables for the declaration's parameters throughout the constructor argument types or record field types respectively. Then we either normalize the constructor argument types or the record field types as appropriate, eventually yielding a value whose outermost constructor is `NTDvariant` or `NTDrecord` respectively. This value is then wrapped up together with the arguments to the normalized type constructor in an `NTconstructed` node. As for `Normtypedecl`, the constructor declarations or record fields should be sorted by name to ensure that the eventual hash value is independent of the order of the constructors—currently this is not done.

Links to other types (`Tlink` and `Tsubst`). Return the normalized form of the type being linked to.

Anything else. Issue a warning and return `NTother`.

Chapter 6

Modifications to the O'Caml runtime

6.1 Random myname generator (`byterun/random256.c`)

The C function `random256` takes no arguments, and returns a random 256bit long O'Caml string. It is generated using the pseudo-random number generator found in C's standard library (`stdlib.h`). When first invoked, the generator's seed is set (`srandom`) using O'Caml's own random seed. The 32byte (256bit) string is then generated by concatenating 8 4byte long `ints` generated with the random number generator (`random`).

This function is used for generating random myname values at compile-time and runtime for ”cfresh” and ”fresh” myname type modes, respectively.

6.2 Hashing of structures (`byterun/hash256.c`)

6.3 Polymarshal (`byterun/polymarshal.c`)

Chapter 7

The standard library

The O'Caml standard library is largely unchanged. The original `Marshal` module is now available as `Oldmarshal`. The `Marshal` module itself has of course been modified to support type-safe marshalling.

There is a new standard library module `Dyntype` providing a single function for converting values of type `typerep` into strings.

Bibliography

[BSSS06] John Billings, Peter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml, April 2006. Submitted for publication. http://www.cl.cam.ac.uk/users/pes20/hashcaml.