A Compilation Strategy for the Habit Programming Language

The High Assurance Systems Programming Project (Hasp)
Department of Computer Science, Portland State University
Portland, Oregon 97207, USA

November 2010

1 Introduction

The Habit programming language is a dialect of Haskell [6] that has been designed to support the development of high quality systems software components such as operating system kernels and device drivers. The Habit report [7] explains the background and motivation for the language design and also provides technical material including an annotated grammar for the language, an overview of the standard environment, and an extended programming example to show how the language can be used in practice.

In this report, we document the strategy for compiling Habit that we are currently implementing in our prototype compiler. The high-level structure of the compiler is illustrated in Figure 1. Each of the boxes in this diagram represents a language/intermediate form that is used during the compilation process. The arrows between the boxes represent compiler *phases*, each of which implements a transformation between a pair of languages. The boxes labeled with Habit, Fidget, and GCMinor name specific languages in the compiler pipeline; the remaining labels are just brief attempts to characterize languages that have not yet been assigned a more formal name.

The left side of the diagram represents the *front-end* of the compiler: the parts most closely related to the Habit source language. The right side of the diagram represents the *back-end* of the compiler: the parts most focussed on the generation of executable machine code. The latter begins with the Fidget and GCMinor languages that have been developed within the HASP project [4], but also leverages the results of CompCert [3] (originally developed as a formally verified compiler for a large subset of C) to provide a high-assurance back-end. These components are described in depth in an accompanying report [8], so we focus our attention here on the front-end portion of the Habit compiler.

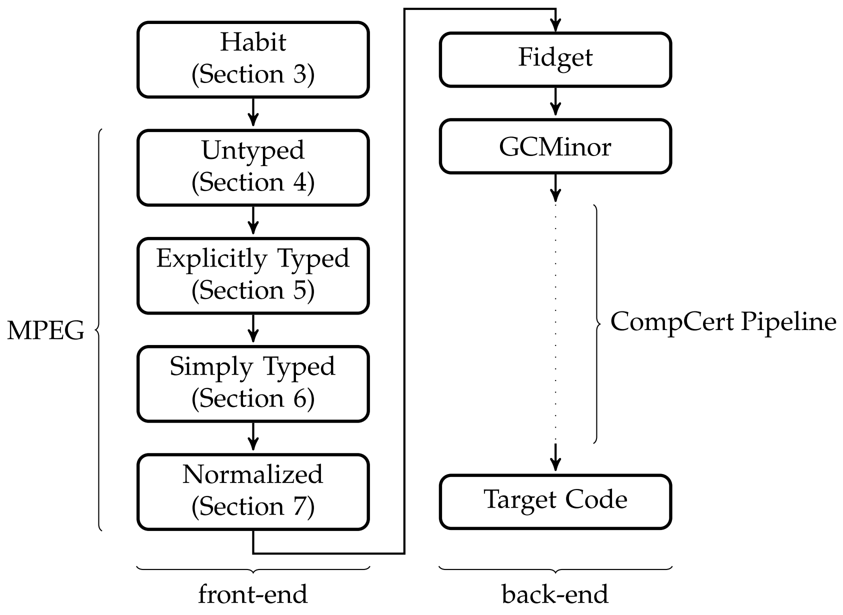

Figure 1: Structure of the Habit Compiler

Our implementations of the front end components do not yet offer anything approaching the strong guarantees of semantics (or even type) preservation as the back-end components. We have made some preliminary explorations to formalize the semantics for some aspects of the front-end languages, but significant work would be required to bring the front-end implementation up to something approaching the high-assurance standards of the back-end; we believe that continuing to harden the full compiler pipeline would be a worthwhile project for future work as the both the Habit language design and the compiler implementation continue to mature.

1.1 Report Outline

In the remaining sections of this report, we will use a running example to illustrate how the different phases shown in Figure 1 work together to enable compilation of Habit source code. The specific example that we choose is the priority set example from the Habit language report; Section 2 provides a brief overview of this example as context for the following sections.

Our tour of the compiler begins properly in Section 3 with some discussion of the (mostly standard) process of parsing Habit source code and constructing an initial abstract syntax tree. After this initial stage, Habit source code is desugared into the intermediate language that is described Section 4. We refer to this language as MPEG, which is an acronym for Matches, Patterns, Expressions, and Guards, the primary nonterminals of the language. In fact, other than Habit at the beginning of the pipeline, all of the intermediate languages that are used in the front-end are different flavors of MPEG. The initial form of MPEG is labeled as *Untyped* in the figure because it represents the source form of parsed Habit code fairly directly, before any significant static analysis, including type checking, has been performed.

As a result of type checking, we expect to obtain not only a claim that the input program is well-typed (or diagnostics explaining otherwise), but also a rewritten version of the code that captures inferred type information. This flavor of MPEG, described in Section 5, is referred to in the diagram as *Explicitly Typed* because it includes (limited forms of) type abstraction and application, as well as the corresponding evidence abstraction and application constructs that are used in a dictionary passing implementation of Haskell type classes [9].

Our compiler uses an implementation strategy, enabled but not required by the Habit design, in which all uses of polymorphism and overloading in source code are eliminated by translating the program into a collection of monomorphic definitions. In the figure, we use the label *Simply Typed* for the corresponding version of MPEG, described in more detail in Section 6, which can be generated from explicitly type code using an automated form of program specialization [2]. The programs in this variant of MPEG are superficially very similar to those of the initial untyped variant, except that they may include some code duplication as a result of specializing polymorphic definitions at distinct monomorphic types. In addition, each use of a polymorphic primitive is also replaced with a freshly generated identifier that represents a particular monomorphic instance of the primitive. By translating Habit code into this simply typed version of MPEG, we have the potential to eliminate run-time costs associated with the reliance on uniform/boxed representations or the use of dictionaries in the back-end of a conventional implementation of Haskell. On the other hand, this approach forces some restrictions on the language design, requires some form of whole-program compilation, and carries the risk of a dramatic increase in code size. Based on our past experience, and on the demands of the specific domain that Habit is targeting, we are not expecting to see substantial problems with this approach in practice, and we hope that it will yield some useful benefits and opportunities for back-end optimization.

Finally, in Section 7, we describe a process that leverages the algebraic laws of MPEG to rewrite simply typed code into an equivalent *normalized* form. These transformations eliminate matching of nested patterns, merge cases with common constructors, and identify points in pattern matching code that may require limited backtracking. Once this process is complete, most of the pattern matching sections of the code, although still expressed in simply typed MPEG, can be interpreted as more conventional `case` expressions. As a result, the code can be more readily translated, first into Fidget, and then after passing through the back-end components, into executable machine code.

2 The Priority Set Example

This section provides a brief summary of the *priority set* example from the Habit language report [7, Section 5]. This code provides a quick taste of Habit as well as the starting point of the running example that we will use to illustrate the various stages that are involved in compiling a Habit program.

The priority set example was originally derived from a fragment of `pork`, an implementation of the L4 microkernel written in C and x86 assembly language, where it is used to maintain a data structure that holds the set of all active thread priorities within the system. This set is implemented as a maximum heap data structure, which enables the kernel—in particular, the scheduler—to determine the priority of the highest priority runnable thread in constant time. It is important to ensure good performance for this code, parts of which are called many times a second from within the timer interrupt handler. As such, the priority set code is interesting as an example of performance-critical, systems-level programming that also has some interesting algorithmic content. On the other hand, this particular example makes less use of higher-order functions and algebraic datatypes than a typical Haskell program might do. These language features are supported in Habit, but their compilation is handled using standard techniques, and so it is not unreasonable to focus instead on more unusual aspects of the Habit compiler pipeline.

In the rest of this section, we will give the Habit code for the priority set example with relatively little comment or explanation; we refer the reader to the language report [7, Section 5] for a more in-depth treatment of this code, including a comparison with the original C implementation.

The key data structures in the priority set example are a pair of global arrays, `prioset` and `prioidx`, as well as a global variable, `priosetSize`, that records the number of elements in the priority set at any given time.

```
type NumPrio      = 256  -- Number of priority levels
type Priority     = Ix NumPrio

area priosetSize <- 0  :: Ref (Stored Unsigned)
area prioset <- noInit :: Ref (Array NumPrio (Stored Priority))
area prioidx <- noInit :: Ref (Array NumPrio (Stored (Ix NumPrio)))
```

From an implementation perspective, the compiler can infer size and alignment information for each of these three areas using the declared types. The details for these and any other areas that are defined in the program can then be passed to the back-end and used to assign an appropriate (static) address within available memory for each of the areas—including `priosetSize`, `prioset`, and `prioidx`—to ensure disjoint, appropriately aligned storage for each one.

Each of the area declarations also specifies an initializer expression, introduced by an <- symbol. At the source level, initializers are constructed as elements of abstract types of the form `Init a`. At the implementation level, however, these types are represented as computations in an implementation-defined monad whose limited set of primitives provide the expected methods for writing to memory. For example, the initializer for `priosetSize` can be translated in the obvious way as `writeRef priosetSize 0`, while the initializers for the two arrays are null actions. All of the resulting initialization code can then be collected together and executed at the beginning of a Habit program, before the `main` program begins, to provide the expected initialization semantics.

The priority set code relies on the invariant that, if the `i`th entry of `prioset` holds the priority `p`, then the `p`th entry of `prioidx` holds the index `i`. This invariant is maintained by tunneling all updates to `prioset` through the following function:

```
-- Update priority set to save priority value prio at index i
prioSet       :: Ix NumPrio -> Priority -> M ()
prioSet i prio = do writeRef (prioset @ i) prio
                    writeRef (prioidx @ prio) i
```

The process of adding a new element to the priority set is handled by the `insertPriority` function, which increments `priosetSize` and then performs a heap repair operation to bubble the new priority in to an appropriate position:

```
insertPriority     :: Priority -> M ()
insertPriority prio = do s <- readRef priosetSize
                         writeRef priosetSize (s+1)
                         heapRepairUp (modIx s) prio

heapRepairUp :: Ix NumPrio -> Priority -> M ()
heapRepairUp i prio
  = case decIx i of
      Nothing -> prioSet 0 prio  -- at the root
      Just j  -> do let parent = j>>1
                    pprio <- readRef (prioset @ parent)
                    if pprio < prio then
                      prioSet i pprio
                      heapRepairUp parent prio
                    else
                      prioSet i prio
```

Removing a priority from the set is a little more complicated and requires locating the priority to be removed (using the values in `prioidx`) and then replacing it with the value, `rprio`, from the end of the priority set. After these changes have been made, we can restore the maximum heap invariant using a combination of a downward repair operation that finds an appropriate position for the relocated priority, `rprio`, within its subtree of the heap and then uses an upward repair to integrate that subtree with the rest of the heap. This process is captured in the following code:

```
removePriority     :: Priority -> M ()
removePriority prio = do s <- readRef priosetSize
                         writeRef priosetSize (s-1)
                         rprio <- readRef (prioset @ modIx (s-1))
                         if prio/=rprio then
                           i <- readRef (prioidx @ prio)
                           heapRepairDown i rprio (modIx (s-2))
                           nprio <- readRef (prioset @ i)
                           heapRepairUp i nprio

heapRepairDown :: Ix NumPrio -> Priority -> Ix NumPrio -> M ()
heapRepairDown i prio last
  = let u = unsigned i in
    case (2*u+1) <=? last of         -- Look for a left child
     Nothing -> prioSet i prio       --   i has no children
     Just l  ->                      --   i has a left child
       do lprio <- readRef (prioset @ l)
          case (2*u+2) <=? last of   -- Look for a right child
            Nothing ->               --   i has no right child
              if lprio > prio then
                prioSet i lprio
                prioSet l prio
              else
                prioSet i prio
            Just r  ->               --   i has two children
              rprio <- readRef (prioset @ r)
              if prio > lprio && prio > rprio then
                prioSet i prio
              else if (lprio > rprio) then
                prioSet i lprio      --   left is higher
                heapRepairDown l prio last
              else                   --   right is higher
                prioSet i rprio
                heapRepairDown r prio last
```

3 Parsing and Basic Static Analysis of Habit

The compilation of Habit code begins with a conventional parsing process that builds an abstract syntax tree structure whose form closely reflects the grammar of the Habit language. This tree structure will inevitably contain artifacts of the source language that are not relevant in later stages of the compilation process. For example, the use of infix operators in an expression like `1 + 2 * 3` cannot be resolved during parsing because of the way that operator fixities are determined in Habit. (Although we would not encourage it, the declaration of an operator fixity in the source code might not appear until after its use.) To handle issues like this, the parser will build a data structure that represents this particular example using a list of the three operand subexpressions for `1`, `2`, and `3`, and a list of the two operator subexpressions for `+` and `*`. Once parsing is complete, the front-end can build environment structures that describe the mapping from operator names to fixities at each point in the program, and then use these to determine whether the expression `1 + 2 * 3` should be interpreted as either `1 + (2*3)` or, with unconventional fixity settings, either as `(1+2)*3` or as an error if the declared fixities of `+` and `*` are incompatible.

More generally, a myriad of static checks and desugaring steps are required after parsing and before more substantive static analysis/type checking and compilation steps begin. Most of these are quite straightforward. For example, the front-end should check that: there is at most one definition for any given identifier in any given context; that all of the identifiers that are used in a program have a corresponding definition or declaration; that constructor functions are applied to the correct number of arguments in patterns; that all equations in a multiple-line definition of a function have the same number of left hand side arguments; and so on. Examples of simple desugaring include rewriting infix expressions like `(1+2)` into the corresponding applicative form `(+) 1 2`, or transforming functional notation in types into uses of type class predicates.

From an algorithmic perspective, the most interesting tasks that must be completed before type checking involve dependency analysis on sets of type and, independently, value declarations. The purpose here is to break a set of declarations into a list of binding groups, each of which represents either a single non-recursive definition or else a group of one or more (mutually) recursive definitions. For type declarations, this information can be used to detect illegal `class` or `type` definitions, and it is also needed as a prelude to kind inference. In a similar way, for value declarations, this information can be used to detect forms of recursive definition that are not permitted in a call-by-value language (for example, `x = x+1`), and it is also necessary as prerequisite to polymorphic type inference. These tasks can be accomplished by combining a traversal of the abstract syntax tree that computes the underlying dependency graph with a standard algorithm for computing its strongly connected components.

4 MPEG: Matches, Patterns, Expressions, Guards

It is possible to implement type checking on Habit code with only very light desugaring (such as resolving uses of infix operators, as described previously). Our compilation strategy, however, assumes a more aggressive approach that eliminates several other high-level constructs of Habit—including `case`, and `if` (but not `do`) expressions, guards and `where` clauses in function definitions, and several forms of compound pattern—by rewriting programs into an intermediate language called MPEG. This section provides an overview of MPEG (whose name is an acronym of its nonterminals: Matches, Patterns, Expressions, and Guards), and of the methods that are used to translate Habit into MPEG.

An early variant of MPEG was described formally as part of the background material in Diatchki's dissertation work [1, Chapter 6], but the MPEG label was not adopted until later, and there are some small differences in terminology (for example, the guards that we refer to here are known as qualifiers in Diatchki's text) as well as some details of the language (for example, Diatchki's version of MPEG combines lambda abstraction and matching in a single construct). The primary attractions of MPEG for compiling Habit are: that it is quite a simple language; that it enables a fairly direct embedding of Habit source code, including general support for pattern matching; and that it admits a rich collection of algebraic laws. The laws, in particular, are used in the compiler to transform MPEG programs into a normalized form that can be translated fairly directly into the `case` constructs that are used to support pattern in Fidget.

4.1 Syntax of (Untyped) MPEG

The grammar in Figure 2 shows the basic syntax of the language, including a possible representation for its abstract syntax in Haskell/Habit notation as well, as part of the comments, as the corresponding concrete syntax that we will use in the rest of this report. The types `I` and `C` appearing here represent identifiers and constructor functions, respectively. The grammar for expressions (`E`) describes a λ-calculus extended with syntax for constructing data values (`Cons`), embedding pattern matches (`Match`), and representing the monad bind operations corresponding to Habit's `do`-notation (`Bind`). Matches (`M`) encode decision trees that will eventually either commit to a particular answer (`Commit`) or else fail (`Fail`). Backtracking after failure is supported by chaining together alternatives using the *fatbar* operator (`:||:`). Progress through a match (`:=>:`) is described using guards (`G`), which allows evaluation (`Let`) and matching (`From`) to be interleaved in arbitrary ways. The syntax for patterns (`P`) is much simpler than Habit, comprising variables (`Var`), constructor patterns (`CPat`), and guarded patterns (`Guard`). The latter are not used in Habit but are included in MPEG because they provide a flexible mechanism for encoding more advanced pattern matching features. For example, a literal pattern, `l`, in Habit is implemented by `(n | n==l)` in MPEG (where `n` is a new variable), while a Habit pattern `v@p` translates to `(v | p<-v)`.

Finally, every `let` takes a list of declaration groups (`D`) arranged so that there are no forward references from one group to an identifier that is defined in a later group. Each of the declaration groups is either a non-recursive pattern binding (`NonRec`) or a collection of one or more mutually recursive bindings (`Rec`). Because Habit is a call-by-value language, all of the expressions in a recursive binding must be headed with at least one lambda (`Lam`). Note that the distinction between recursive and non-recursive bindings is made using the

```
data M = Fail        -- fail              match failure  -- Matches
       | Commit E    -- ^ e               commit result
       | M :||: M    -- m1 | m2           alternative
       | G :=>: M    -- g => m            guarded match

data P = Var I       -- i                 variable       -- Patterns
       | CPat C [P]  -- C p1 ... pn       constructor pattern
       | Guard P G   -- (p | g)           guarded pattern

data E = Id I        -- i                 identifier     -- Expressions
       | Cons C [E]  -- C e1 ... en       constructor
       | Lam I E     -- \i -> e           abstraction
       | Ap E E      -- e e'              application
       | Match M     -- { m }             match
       | LetE [D] E  -- let decls in e    local decls
       | Bind I E E  -- i <- e; e         monadic bind

data G = Let [D]     -- let decls         local decls    -- Guards
       | From P E    -- p <- e            pattern match

data D = Rec [(I,E)] -- {i1=e1;...}       recursive      -- Declarations
       | NonRec P E  -- p = e             non-recursive
```

Figure 2: Syntax for the (Untyped) MPEG Intermediate Language

dependency analysis mentioned in Section 3, and does not rely on annotations or explicit structure in the source code. For example, dependency analysis will identify `i=i+1` as a recursive definition, not as a non-recursive pattern binding, and hence, with no lambda on the right hand side, it will be flagged as an error, and not reconsidered as a non-recursive definition referencing some prior binding of `i` in an enclosing scope.

4.2 Transforming Habit Code into (Untyped) MPEG

The primary attraction of MPEG in the initial stages of the compiler pipeline is that it allows a fairly direct embedding of Habit source code using appropriate combinations of the MPEG primitives. For example, a Habit function that is defined by a sequence of equations

```
f p1 ... pn = e1
...
f q1 ... qn = ek
```

can be implemented using a lambda expression with `n` arguments whose body is a match with one alternative for each equation in the source:

```
f = \v1 -> ... \vn -> { ((p1 <- v1) => ... => (pn <- vn) => ^ e1)
                      | ...
                      | ((q1 <- vn) => ... => (qn <- vn) => ^ ek) }
```

The syntax of Habit allows each equation in a function definition to include boolean guards as well as local definitions that scope over all of the guards and right hand side expressions, as in the following:

```
f p1 ... pn | g1 = e1
            | ...
            | gk = ek
              where ....
```

The structure of an equation like this can be captured in a match that uses MPEG guards for pattern matching arguments as well as the Boolean guards, with the latter chained together as a sequence of alternatives:

```
(p1 <- v1) => ... => (pn <- vn) =>      -- match arguments
   (let ...) =>                         -- local definitions
         ( ((True <- g1) => ^ e1)       -- guarded right hand sides
         | ...
         | ((True <- gk) => ^ ek) )
```

Clearly these patterns extend to allow the translation of an arbitrarily complex function definition in Habit code into an MPEG definition of the form `f=e`.

Many of the more advanced expression forms in Habit source code can also be desugared into combinations of more basic MPEG constructs. For example, a case expression of the form `case e of p1 -> e1; p2 -> e2` can be translated as:

```
{ v <- e => ((p1 <- v => ^ e1) | (p2 <- v => ^ e2)) }
```

where `v` is a new (temporary) variable. In a similar way, a conditional expression like `if e then t else f` can be translated as:

```
{ (v <- e) => ((True <- v => ^ t) | (False <- v => ^ f)) }
```

This is a consequence of interpreting an `if` expression as a special form of `case` expression, `case e of True -> t; False -> f`, and then applying the translation of case expressions described previously. Alternatively, however, because we know that True and False are the only constructors for the Bool type, we can actually use a more compact translation for if expressions using:

```
{ (True <- e => ^ t) | ^ f }
```

This approach can also be used to handle the compilation of && and || as in the following translations:

```
e1 && e2  =  { (True  <- e1 => ^ e2) | ^ False }
e1 || e2  =  { (False <- e1 => ^ e2) | ^ True  }
```

One feature of Habit that we do not attempt to desugar away completely in the translation into MPEG is do-notation. Instead, we translate the list of statements within each block using a nested sequence of monadic binds. For example, an expression like (do x1 <- e1; ... ; xn <- en; e) can be translated into MPEG as (x1 <- e1; (... (xn <- en ; e)...)). We use parentheses here to emphasize the construction of this particular MPEG expression but, in practice, we can often drop the parentheses and write (x1 <- e1; ... ; xn <- en; e), which, superficially, looks just like the source term. It is also possible to use an expression as a statement in a do-construct without specifying a variable to capture its result, as in an example like do e1; ... ; en; e. Expressions like this can also be translated using the monadic bind constructs of MPEG by picking an otherwise unused variable. We reserve the name _ for this purpose, and then we can translate the previous example as (_ <- e1; ... ; _ <- en; e).

4.3 The Priority Set Example in (Untyped) MPEG

To complete the introduction of MPEG, we will now show a translation of the Habit code from Section 2 into MPEG. For example, we can translate the previous definition of prioSet as follows:

```
prioSet :: Ix NumPrio -> Priority -> M ()
prioSet  = \i -> \prio ->
                   _ <- writeRef (prioset @ i) prio;
                   writeRef (prioidx @ prio) i
```

Strictly speaking, if we apply the patterns described previously in a purely mechanical fashion, then the resulting translation of prioSet would look more like the following:

```
prioSet = \a1 -> \a2 ->
             { (i <- a1)
```

```
                 => (prio <- a2)
                     => ^ (_ <- writeRef ((@) prioset i) prio;
                           writeRef ((@) prioidx prio) i) }
```

In this second version of the code, we have: dropped the type signature; eliminated the use of infix notation; wrapped the right hand side in a match; and added some guards whose only effect is to associate the names chosen for the lambda-bound parameters with the corresponding names in the original code. However, given the laws for manipulating MPEG programs that will be discussed in Section 7, we can eliminate the guards (by applying renaming substitutions to the commit expression) and then eliminate both the match and the commit terms (by using the law `{^e} = e`) to transform this second version of the code into the first, modulo the syntactic niceties of infix notation and the documentation value of the type signature. Internally, our compiler actually produces the unsimplified form of code that is exemplified by the second version of `prioSet` shown above. It would not be particularly difficult to include an additional simplification pass at this point in the compiler, but that is not necessary because all of these transformations, and others too, will be applied during the later normalization process (Section 7). To avoid unnecessarily complicating the presentation, however, we will give the translation for the rest of the priority set example by showing what the generated code looks like after these simplifying transformations have been applied.

For example, the MPEG versions of the `insertPriority` and `heapRepairUp` functions are as follows, while the longer code fragment for `removePriority` and `heapRepairDown` appears in Figure 3.

```
insertPriority :: Priority -> M ()
insertPriority  = \prio ->
  s <- readRef priosetSize;
  _ <- writeRef priosetSize (s + 1);
  heapRepairUp (modIx s) prio

heapRepairUp :: Ix NumPrio -> Priority -> M ()
heapRepairUp  = \i -> \prio ->
  { t1 <- decIx i =>
    (Nothing  <- t1 => ^(prioSet 0 prio))
  | ((Just j) <- t1 => ^(let parent = j >> 1 in
                         pprio <- readRef (prioset @ parent);
                         { (True  <- (pprio < prio) =>
                             ^(_ <- prioSet i pprio;
                               heapRepairUp parent prio))
                         | ^(prioSet i prio)}))}
```

```
removePriority   :: Priority -> M ()
removePriority    = \prio ->
  s <- readRef priosetSize;
  _ <- writeRef priosetSize (s - 1);
  rprio <- readRef (prioset @ (modIx (s - 1)));
  { (True  <- (prio /= rprio) =>
       ^(i <- readRef (prioidx @ prio);
         _ <- heapRepairDown i rprio (modIx (s - 2));
         nprio <- readRef (prioset @ i);
         heapRepairUp i nprio))
  | ^(return ())}

heapRepairDown :: Ix NumPrio -> Priority -> Ix NumPrio -> M ()
heapRepairDown  = \i -> \prio -> \last ->
  let u = unsigned i in
  { t1 <- ((2 * u) + 1) <=? last =>
    (Nothing <- t1 => ^(prioSet i prio))
  | (Just l  <- t1 =>
      ^(lprio <- readRef (prioset @ l);
        { t2 <- ((2 * u) + 2) <=? last =>
          (Nothing <- t2 =>
             ^{ (True <- (lprio > prio) =>
                 ^(_ <- prioSet i lprio;
                   prioSet l prio))
              | ^(prioSet i prio)})
        | (Just r  <- t2 =>
             ^(rprio <- readRef (prioset @ i);
               { (True <- { (True <- (prio > lprio)
                              => ^ (prio > rprio))
                          | ^ False } =>
                  ^(prioSet i prio))
               | ^{ (True <- (lprio > rprio) =>
                     ^(_ <- prioSet i lprio;
                       heapRepairDown l prio last))
                  | ^(_ <- prioSet i rprio;
                      heapRepairDown r prio last)}}))}))}
```

Figure 3: Translation of `removePriority` in (Untyped) MPEG

```
data M = ...                                       -- Matches

data P = Var I T         -- i :: t                 -- Patterns
       | CPat C [T] [P]  -- C{ts} p1 ... pn
       | ...

data E = LamId I         -- i                      -- Expressions
       | LetId I [T] [V] -- i{ts}{vs}
       | Cons C [T] [E]  -- C{ts} e1 ... en
       | Lam I T E       -- \(i :: t) -> e
       | ...
       | Bind I T T E E  -- (i :: t) <-{T} e; e

data G = ...                                       -- Guards

data D = Rec [(I,[I],[I],E)] -- {i1=\{ts1}{vs1}e1;...} -- Declarations
       | ...
```

Figure 4: Syntax for (Explicitly Typed) MPEG

5 Explicity Typed MPEG

Once the translation of from Habit to MPEG is complete, the compiler passes the resulting program to the type checker/type inference engine. One goal in this part of the pipeline is to identify and reject any input program that is not well-typed. A second goal is to generate a new representation of the program that is annotated with type information at the points where polymorphism or overloading are used. These annotations take the form of type and dictionary abstractions at the points where polymorphic/overloaded functions are defined, and type and dictionary applications at the points where polymorphic/overloaded functions are used. This method of capturing type information in the code for an implicitly typed functional language was first suggested by Mitchell and Harper [5] and by Wadler and Blott [9]. In the former, these techniques were used as an attempt to explain ML-style polymorphism using the constructs of System F, while in the latter they were employed as an implementation strategy, known as *dictionary passing*, for type class overloading.

5.1 Syntax of (Explicitly Typed) MPEG

The grammar fragments in Figure 4 show how the original MPEG grammar in Figure 2 is modified to include type annotations. Where it is necessary to make a distinction between this and other versions of MPEG, we refer to the resulting language here as *Explicitly Typed* MPEG. One important change that is made here is the explicit distinction between lambda-bound (`LamId`) and let-bound (`LetId`) identifiers. The former can only have a monomorphic type, but the latter, in general, can have polymorphic/overloaded types. To record the particular instance at which each occurrence of a let-bound variable is used, we annotate each occurrence with an appropriate list of types (`T`) and a second list of dictionary evidence (`V`). Lambda-bound variables are introduced, unsurprisingly, by lambda expression (`Lam`), but also by variables occurring in patterns (`Var`) or monadic bindings (`Bind`). In each of these three cases, we annotate the binding occurrence of the variable with the associated type; this information is then propagated through the remaining stages of the front-end so that it available for use during code generation in the back-end. For monadic bindings, we also include a second type parameter to record the associated monad, which must also be an instance of the standard `Monad` class.) Let-bound variables, however, can only be introduced as part of a recursive binding group (`Rec`), and each of these bindings includes not only the name of the item being defined, but also a list of type variables (one for each quantified variable in the type of the item) and a list of dictionary variables (one for each qualifying predicate in the type of the item). Constructor functions, appearing in patterns (`CPat`) and expressions (`Cons`), can also have polymorphic (but not qualified) types, so the structures that represent these constructs are also modified to include the list of types that is needed to identify a particular instance.

5.2 Type Annotations in (Explicitly Typed) MPEG

To illustrate how the annotations of explicitly typed MPEG are used in a relatively simple and familiar case, consider the following example of Habit code:

```
data Maybe a = Nothing | Just a

maybe              :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing  = n
maybe n f (Just x) = f x

test = maybe False not (Just False)
```

Translating this definition first into Untyped MPEG, would produce code that looks something like the following:

```
data Maybe a = Nothing | Just a

maybe :: b -> (a -> b) -> Maybe a -> b
maybe  = \n -> \f -> \m -> { (Nothing <- m) => ^ n
                           | (Just x  <- m) => ^ (f x) }

test  = maybe False not (Just False)
```

As a prelude to type checking, we can extract the following types for the two constructors that are defined by the `data` definition (we write explicit `forall` quantifiers here to emphasize the use of polymorphism, but note that this is not actually valid Habit syntax):

```
Nothing :: forall a . Maybe a
Just    :: forall a . a -> Maybe a
```

Given this input, the type checker will now produce the following translation of the example into the notation of explicitly typed MPEG:

```
maybe :: forall a, b. b -> (a -> b) -> Maybe a -> b
maybe  = \{a,b}{} ->
            \(n :: b) -> \(f :: a -> b) -> \(m :: Maybe a) ->
               { (Nothing{a}        <- m) => ^ n
               | (Just{a} (x :: a)  <- m) => ^ (f x) }

test :: Bool
test  = maybe{Bool,Bool} False{} not{}{} (Just{Bool} False{})
```

In this code, every constructor has been annotated with a list of type parameters. As a result, we can determine, for example, that the occurrence of `Just` in the definition of `test` refers exclusively to a constructor for the `Maybe Bool` type, whereas the particular instance of `Just` that is used in the definition of `maybe` depends on the choice of the type parameter `a`. Even constructors like `False` for types that do not have any parameters must be annotated with an empty list. We emphasize also that the annotations on each constructor correspond to type annotations rather than specific instances. For example, the expression `Nothing{Maybe Bool}` represents the instance of the `Nothing` constructor for the type `Maybe (Maybe Bool)`, and not for the type `Maybe Bool` that is named by the parameter. This is important because it means that we can uniquely characterize each use of a polymorphic operator without having to write down its full type (or to allow for the possibility of writing down an invalid type).

The variables `maybe` and `not` that are used in the example above are both let-bound, so their uses must be annotated with two lists, one for type parameters and one for dictionary parameters. For these particular examples, there is no use of overloading and hence the second list will be empty in each case. Indeed, the `not` function does not even have a polymorphic type, which is why the associated list of type parameters is also empty in that case.

5.3 Dictionary Annotations in (Explicitly Typed) MPEG

To show how dictionary parameters are used in the explicitly typed variant of MPEG, we will consider the following example of Habit code uses some of the basic definitions from the standard environment:

```
class Width (n::nat)
instance Width 0     -- instances built in to the compiler
instance Width 1
instance Width 2
...

class Eq a where (==), (/=) :: a -> a -> Bool
instance Eq (Bit n)  if Width n where ...
instance Eq Unsigned            where ...

class ToUnsigned t where unsigned :: t -> Unsigned
instance ToUnsigned Unsigned           where ...
instance ToUnsigned (Bit n) if Width n where ...
```

In explicitly typed MPEG, each of the classes listed here corresponds to a type of dictionaries, and each of the instance declarations corresponds to a dictionary constructor function, as suggested by the following code:

```
type Width :: nat -> *
instWidth0 :: Width 0     -- instances built in to the compiler
instWidth1 :: Width 1
instWidth2 :: Width 2
...

type Eq :: * -> *
instEqBit      :: forall (n::nat). Width n -> Eq (Bit b)
instEqUnsigned :: Eq Unsigned

type ToUnsigned :: * -> *
instToUUnsigned :: ToUnsigned Unsigned
instToUBit      :: forall (n::nat). Width n -> ToUnsigned (Bit n)
```

For this presentation, we have used a syntax that looks much like regular Habit code, with names for each of the types and functions that go at least some way to suggesting what they represent In practice however, the mechanisms for describing dictionary construction can be cleanly separated from the rest of the language, and the compiler is free to choose arbitrary names for these entities, none of which can be referenced directly from Habit source code.

The intuition here is that the functions above can be combined to construct a *proof* that a given type is an instance of a particular class. Moreover, from an implementation perspective, the structure of a proof that is generated in this way can be used to determine the implementation of an overloaded operator at a particular instance of the class. Given the definitions above, for example, the identifier `instWidth2` serves as a proof that the natural number type `2` is an instance of the `Width` class, while the term `instEqBit{2} instWidth2` provides a proof that the `Bit 2` type is an instance of the `Eq` class.

As an example, the Habit expression `(\x -> x == B00)` has type `Bit 2 -> Bool` and translates to `(\x -> (==){Bit 2}{instEqBit{2} instWidth2} x B00)` in the explicitly typed version of MPEG. This particular example can be understood by starting with the most general type for the equality operator, instantiating the quantified type variable `a` to `Bit 2`, and then discharging the `Eq (Bit 2)` constraint using the proof given above, as suggested by the following typings:

```
(==)        :: forall a. Eq a => a -> a -> Bool
(==){Bit 2} :: Eq (Bit 2) => Bit 2 -> Bit 2 -> Bool
(==){Bit 2}{instEqBit{2} instWidth2}
            :: Eq (Bit 2) => Bit 2 -> Bit 2 -> Bool
```

The only difference between explicitly typed MPEG and what we have done here is that the tasks of instantiating quantifiers and discharging constraints must be done at the same time in explicitly typed MPEG, and cannot be broken into two separate steps.

In cases involving polymorphic functions, we may not be able to discharge a type class constraint completely, and this is why we include dictionary parameters as well as type parameters at the defining occurrence of each let-bound variable. To illustrate this, consider the following Habit function that tests to determine whether a bit vector is zero (in an unnecessarily complicated way) by converting it to an `Unsigned` integer and then performing the comparison:

```
isZero  :: Width n => Bit n -> Bool
isZero b = unsigned b == 0
```

This uses two overloaded functions: `unsigned` with type `Bit n -> Unsigned`, and `(==)` with type `Unsigned -> Unsigned -> Bool`. The second of these requires an instance `Eq Unsigned`, which can be obtained using the proof `instEqUnsigned`. The first assumes an instance `ToUnsigned (Bit n)`, which can be only proved by applying `instToUBit` to a proof of `Width n`. To accomplish the latter, we include a dictionary parameter to supply a proof of `Width n` in the translation:

```
isZero :: Width n => Bit n -> Bool
isZero  = \{n}{d::Width n} -> \(b :: Bit n) ->
            (==){Unsigned}{instEqUnsigned}
                    (unsigned{Bit n}{instToUBit d} b)
                    0{Unsigned}{NumLit 0 Unsigned}
```

As the last line of this example highlights, even numeric literals in Habit are overloaded, so they are also treated as let-bound identifiers with appropriate type and dictionary parameters.

5.4 The Priority Set Example in (Explicitly Typed) MPEG

We conclude our description of the explicitly typed version of MPEG by showing how it can be used to encode the priority set example that was originally described in Section 2 and translated into MPEG in Section 4.3. This particular section of code does not introduce any polymorphic or overloaded definition, so the most interesting aspect of the translation is the way in which it uses polymorphic primitives defined elsewhere. For reference purposes, we begin by listing the complete set of primitives that are used in the priority set example together with their most general types:

```
0        :: forall t. NumLit 0 t => t
1        :: forall t. NumLit 1 t => t
2        :: forall t. NumLit 2 t => t
(+)      :: forall a. Num a => a -> a -> a
(-)      :: forall a. Num a => a -> a -> a
(*)      :: forall a. Num a => a -> a -> a
unsigned :: forall t. ToUnsigned t => t -> Unsigned
(/=)     :: forall t. Eq t => t -> t -> Bool
(<)      :: forall t. Ord t => t -> t -> Bool
(>)      :: forall t. Ord t => t -> t -> Bool
(>>)     :: forall t. Shift t => t -> Unsigned -> t
decIx    :: forall n. Index n => Ix n -> Maybe (Ix n)
(@)      :: forall n a. Index n => Ref (Array n a) -> Ix n -> Ref a
modIx    :: forall n. Index n => Unsigned -> Ix n
(<=?)    :: forall n. Index t => Unsigned -> Ix n -> Maybe (Ix n)
return   :: forall m a. Monad m => a -> m a
readRef  :: forall m a b. (MemMonad m, ValIn a b) => Ref a -> m b
writeRef :: forall m a b. (MemMonad m, ValIn a b) => Ref a -> b -> m ()
```

Now we can begin the task of annotating the untyped MPEG code from Section 4.3 to include additional type information. While the overall structure of the code does not change in this process, it can be quite difficult to see that amidst all of the annotations, as illustrated by the following explicitly typed version of `prioSet`:

```
prioSet :: Ix NumPrio -> Priority -> M ()
prioSet  = \{}{} -> \(i :: Ix NumPrio) -> \(prio :: Priority) ->
       (_ :: ()) <-{M}
            writeRef{M, Stored (Ix 256), Ix 256}
                    {MemMonad M, ValIn (Stored (Ix 256)) (Ix 256)}
                     (prioset{}{} @{256, Ix 256}{Index 256} i) prio;
       writeRef{M, Stored (Ix 256), Ix 256}
               {MemMonad M, ValIn (Stored (Ix 256)) (Ix 256)}
              (prioidx{}{} @{256, Ix 256}{Index 256} prio) i
```

Note that we have written class constraints like `MemMonad M` in this example in place of the dictionary expressions that should be provided as parameters to overloaded functions. However, this is intended only as a notation for the corresponding dictionary expressions, which cannot be written down directly here without first naming the dictionary constructor functions for each of the instance declarations in the Habit standard environment.

The translation of `insertPriority` and `heapRepairUp` provides a slightly longer example of explicitly typed MPEG:

```
insertPriority :: Priority -> M ()
insertPriority  = \{}{} -> \(prio :: Priority) ->
 (s :: Unsigned)
  <-{M} readRef{M, Stored Unsigned, Unsigned}
               {MemMonad M, ValIn (Stored Unsigned) Unsigned}
          priosetSize{}{};
 (_ :: ())
  <-{M} writeRef{M, Stored Unsigned, Unsigned}
                {MemMonad M, ValIn (Stored Unsigned) Unsigned}
         priosetSize{}{}
         (s +{Unsigned}{Num Unsigned} 1{Unsigned}{NumLit 1 Unsigned});
 heapRepairUp{}{} (modIx{256}{Index 256} s) prio

heapRepairUp :: Ix NumPrio -> Priority -> M ()
heapRepairUp  = \{}{} -> \(i :: Ix NumPrio) -> \(prio :: Priority) ->
 { (t1 :: Maybe (Ix 256)) <- dec{256}{Index 256} i =>
   ((Nothing{Ix 256} :: Maybe (Ix 256)) <- t1
    => ^(prioSet{}{} 0{Ix 256}{NumLit 1 (Ix 256)} prio))
 | (((Just{Ix 256} (j :: Ix 256)) :: Maybe (Ix 256)) <- t1
    => ^(let parent = j >>{Ix 256}{Shift (Ix 256)}
                       1{Unsigned}{NumLit 1 Unsigned} in
         (pprio :: Ix 256)
           <-{M} readRef{M, Stored (Ix 256), Ix 256}
                       {MemMonad M, ValIn (Stored (Ix 256)) (Ix 256)}
                     (prioset{}{} @{256, Ix 256}{Index 256} parent);
         { (True{}  <- (pprio <{Ix 256}{Ord (Ix 256)} prio) =>
            ^((_ :: ()) <-{M} prioSet{}{} i pprio;
              heapRepairUp{}{} parent prio))
         | ^(prioSet{}{} i prio)}))}
```

Translations for `removePriority` and `heapRepairDown` into explicitly typed MPEG can be constructed in a similar way. However, we choose not to show them here; given the level of annotations in these examples, the result is more likely to contribute clutter than any deeper enlightenment!

6 Simply Typed MPEG

The annotations in explicitly typed MPEG contribute significant text to the examples shown in the previous section. However, the information that they provide is an essential input to the next stage of the compiler whose purpose is to eliminate polymorphic and overloaded definitions completely by a process of specialization. The basic idea is to compute the set of all instances at which each polymorphic function is used, and then generate a separate, monomorphic implementation for each one. To illustrate this process, suppose the the following function definition appears in a Habit source program:

```
f  :: (NumLit 0 b) => a -> (Bool, b, a)
f y = let id x = x in (id True, id 0, id y)
```

In addition, suppose that there are calls to `f` elsewhere in the program, one of which is required to produce a result of type `(Bool, Unsigned, Unsigned)`, and another that is required to produce a result of type `(Bool, Bit 4, Unsigned)`. In this scenario, specialization will replace the two calls of `f` with calls to two distinct specialized versions of `f`, each of which also requires some specialized versions of the locally defined `id` function:

```
f1  :: Unsigned -> (Bool, Unsigned, Unsigned)
f1 y = let id1  :: Bool -> Bool
           id1 x = x
           id2  :: Unsigned -> Unsigned
           id2 x = x
       in  (id1 True, id2 (0::Unsigned), id2 y)

f2  :: Unsigned -> (Bool, Bit 4, Unsigned)
f2 y = let id3  :: Bool -> Bool
           id3 x = x
           id4  :: Bit 4 -> Bit 4
           id4 x = x
           id5  :: Unsigned -> Unsigned
           id5 x = x
       in  (id3 True, id4 (0::Bit 4), id5 y)
```

While this transformation eliminates polymorphic definitions, there is clearly a risk that it could significantly increase the size of compiled programs as a result of code duplication. It is certainly possible to construct examples that exhibit this kind of behavior, but previous experience in this area [2] suggests that such problems are unlikely to occur in practice. Those results were obtained by considering general purpose functional programs, and we suspect that the likelihood of problems is even lower within the domain of systems software that Habit is targeting. In addition, there is an intuitive argument that functions generally tend to become more constrained, and hence less polymorphic, as code size increases. As such, the functions with the highest degree of polymorphism in a given program are also likely to be among the smallest functions in that program, and hence are also the most likely to be inlined by an optimizing compiler, either resulting in a similar degree of code duplication, or else exposing more opportunities for optimization. The previous example certainly illustrates this: it is not hard to believe that a simple inlining phase could reduce the code for the definitions of `f1` and `f2` to the following compact form (which might then be inlined at the original call sites for `f`):

```
f1  :: Unsigned -> (Bool, Unsigned, Unsigned)
f1 y = (True, 0::Unsigned, y)

f2  :: Unsigned -> (Bool, Bit 4, Unsigned)
f2 y = (True, 0::Bit 4, y)
```

One reason for including specialization as part of the Habit compiler pipeline is to provide the back-end with a language whose type system is much simpler than (and to some degree independent of) Habit's type system. Another motivation is to allow representation decisions in the back-end to be made on a case-by-case basis instead of requiring the use of uniform representations. As we gain experience with Habit, of course, we will learn whether these benefits can be realized in some effective manner, and we will also find out whether our speculation about the low risk of code explosion will hold out in practice.

6.1 The Specialization Algorithm

This section provides a brief overview of how specialization is implemented. The basic algorithm uses three tables, each of which stores pairs of the form `(n, i{ts}{vs})`, where `i{ts}{vs}` identifies a particular instance of `i` in the explicitly typed MPEG code, and `n` is the name that will be used to represent the corresponding implementation in the generated code that is generated. The three tables record *generated*, *primitive*, and *requested* specializations. The generated and primitive tables are initially empty and the requested table begins with a single entry of the form `(main, main{}{})` corresponding to the program entry point. Of course, the requested table could just as easily be initialized to contain a set of (monomorphic) functions if wanted to compile something like a library that has multiple entry points. From this point, specialization proceeds by: removing an item `(n, i{ts}{vs})` from the required table; adding that same item to the generated table; generating a definition for `n` in the output program by specializing the code for `i`; and then repeating these steps until the required table is empty. It is particularly important that we add items to the generated table before attempting to construct the corresponding specialized code or else even a simple recursive definition will cause the specialization algorithm to loop. (Programs that require the use of a single variable at infinitely many distinct instances would also cause the specialization algorithm to loop, but they are already excluded in the definition of Habit [7, Section 3.3.3].)

In the process of generating a specialized version of the (explicitly typed) code for `i`, we may encounter uses of let-bound variables of the form `f{ts}{vs}`. If `f` is a primitive, then we check for an entry `(p, f{ts}{vs})` in the primitives table, adding a new entry with a freshly generated identifier `p` if no previous occurrence has been found, and then use `p` in place of `f{ts}{vs}` in the specialized version of the code. The same basic pattern is used for functions `f` that are not primitives (i.e., that are defined elsewhere in the program). In this case, however, we must check both the generated and requested tables to determine whether a specialized version of `f{ts}{vs}` has already been requested, and any new instances that we find must be queued up for subsequent processing by adding them to the requested table.

Once this process completes, the result is a specialized version of the original program with `main` as its entry point. From the final version of the primitives table, we can also extract a description of all of the primitive function instances that the program uses; we must rely on the back-end to provide appropriate implementations in each case.

In practice, the algorithm described here requires extensions to handle the specialization of local definitions and numeric literals as well as the elimination of dictionary arguments. These are mostly straightforward and so will not give further details here.

6.2 Syntax of (Simply Typed) MPEG

We refer to the version of MPEG that is produced as a result of specialization as *Simply Typed* MPEG because every program in this language has a uniquely determined (monomorphic) type. The grammar fragments in Figure 5 highlight the changes between simply typed MPEG and the explicitly typed variant that was presented in Figure 4. As the figure shows, the simply typed version of MPEG does not require the type abstractions and applications (or the dictionary abstractions and applications) on let-bound variables, but it still retains the type annotations on constructor functions and on lambda-bound variables. One other small change in the simply typed version of MPEG is that we have dropped the typing annotation on the arrow in the monadic bind construct. Whereas the `<-` symbol in the untyped and explicitly typed versions of MPEG denoted a generic monadic bind operator (i.e., for any instance of the `Monad` class), the same symbol in simply-typed MPEG represents the bind operation for a specific *base* monad (the `M` monad in the preceding examples) that encapsulates the functionality of the execution platform. If the specialization algorithm encounters an explicitly typed monadic bind `(x :: t <-{m} e1; e2)` that uses a different monad, then it will be replaced with an expression of the form `(bind e1' (\(x :: t) -> e2'))` where `bind` is the name that is used to refer to

```
data M = ...                                          -- Matches

data P = Var I T         -- i :: t                    -- Patterns
       | CPat C [T] [P]  -- C{ts} p1 ... pn
       | ...

data E = Id I            -- i                         -- Expressions
       | Cons C [T] [E]  -- C{ts} e1 ... en
       | Lam I T E       -- \(i :: t) -> e
       | ...
       | Bind I T E E    -- (i :: t) <- e; e

data G = ...                                          -- Guards

data D = Rec [(I,E)]     -- {i1=e1;...}               -- Declarations
       | ...
```

Figure 5: Syntax for (Simply Typed) MPEG

the `>>={m}{Monad m}` operator in the specialized version of the program. This is a simple technique for preserving the structure of computations over the base monad through the front end without relying on the use of specific optimization techniques (specifically, inlining), but it may become unnecessary as the compiler evolves to include stronger and more general optimization phases.

6.3 The Priority Set Example in (Simply Typed) MPEG

We conclude our presentation of simply typed MPEG by showing specialized code for the priority set example. Because there are no polymorphic definitions in this program, the structure of the specialized version is very similar to the untyped version from Section 4.3, with no code duplication. However, as we saw in Section 5.4, the code uses some primitive functions from the Habit standard environment, each of which is replaced in the specialized code with a freshly generated name that represents a particular monomorphic instance of that primitive. For example, the code for `prioSet` translates into the following simply typed version with `writeRef1` and `at1` standing in for specific instances of the `writeRef` and `(@)` primitives, respectively:

```
prioSet :: Ix NumPrio -> Priority -> M ()
prioSet  = \(i :: Ix NumPrio) -> \(prio :: Priority) ->
             (_ :: ()) <- writeRef1 (at1 prioset i) prio;
             writeRef1 (at1 prioidx prio) i
```

```
insertPriority :: Priority -> M ()
insertPriority  = \(prio :: Priority) ->
  (s :: Unsigned) <- readRef2 priosetSize;
  (_ :: ())       <- writeRef2 priosetSize (plus1 s one1);
  heapRepairUp (modIx1 s) prio

heapRepairUp :: Ix 256 -> Ix 256 -> M ()
heapRepairUp  = \(i :: Ix 256) -> \(prio :: Ix 256) ->
  { (t1 :: Maybe (Ix 256)) <- decIx1 i =>
    (Nothing{Ix 256} <- t1
        => ^(prioSet zero1 prio))
  | ((Just{Ix 256} (j :: Ix 256)) <- t1
        => ^(let parent = shr1 j one1 in
             (pprio :: Ix 256) <- readRef1 (at1 prioset parent);
             { (True{} <- lt1 pprio prio =>
                 ^((_ :: ()) <- prioSet i pprio;
                   heapRepairUp parent prio))
             | ^(prioSet i prio)}))}
```

Figure 6: Translation of `insertPriority` in (Simply Typed) MPEG

The remaining code for the priority set example is split between Figure 6 (for the implementation of `insertPriority`) and Figure 7 (for the implementation of `removePriority`), and the complete list of primitive instances that are referenced in this code is given in Figures 8 and 9.

7 Normalized MPEG

The original translation from Habit to untyped MPEG that was described in Section 4 follows the structure of the source code quite closely, for example, creating a separate match for each equation in the definition of a function. Although it is possible to compile the resulting MPEG programs directly, this does not typically produce very efficient code. For example, consider the following definition of the familiar `filter` function:

```
filter                     :: (a -> Bool) -> List a -> List a
filter p Nil                = Nil
filter p (Cons x xs) | p x  = Cons x rest
                     | True = rest
                       where rest = filter p xs
```

After converting this to MPEG, type checking, and specializing this code to a particular instance (say, lists of values of some type `T`) as described in the pre-

```
removePriority   :: Ix 256 -> M ()
removePriority    = \(prio :: Ix 256) ->
  (s :: Unsigned) <- readRef2 priosetSize;
  (_ :: ())       <- writeRef2 priosetSize (sub1 s one1);
  (rprio :: Ix 256) <- readRef1 (at1 prioset (modIx1 (sub s one1)));
  { (True{}  <- neq1 prio rprio =>
       ^((i :: Ix 256) <- readRef1 (at1 prioidx prio);
         (_ :: ()) <- heapRepairDown i rprio (modIx1 (sub1 s two1));
         (nprio :: Ix 256) <- readRef1 (at1 prioset i);
         heapRepairUp i nprio))
  | ^(return1 ())}

heapRepairDown :: Ix 256 -> Ix 256 -> Ix NumPrio -> M ()
heapRepairDown
 = \i :: (Ix 256) -> \prio :: (Ix 256) -> \last :: (Ix 256) ->
  let u = unsigned1 i in
  { (t1 :: Maybe (Ix 256)) <- leq1 (plus1 (mul1 two1 u) one1) last =>
    (Nothing{Ix 256} <- t1 => ^(prioSet i prio))
  | (Just{Ix 256} (l :: Ix 256) <- t1 =>
      ^(lprio <- readRef1 (at1 prioset l);
        { (t2 :: Maybe (Ix 256))
            <- leq1 (plus1 (mul1 two1 u) two1) last =>
          (Nothing{Ix 256} <- t2 =>
             ^{ (True{} <- gt1 lprio prio =>
                 ^((_ :: ()) <- prioSet i lprio;
                   prioSet l prio))
               | ^(prioSet i prio)})
        | (Just r{Ix 256}  <- t2 =>
             ^((rprio :: Ix 256) <- readRef1 (at1 prioset i);
               { (True{} <- { (True{} <- (gt1 prio lprio)
                                   => ^ (gt1 prio rprio))
                             | ^ False{} } =>
                  ^(prioSet i prio))
               | ^{ (True{} <- gt1 lprio rprio =>
                     ^((_ :: ()) <- prioSet i lprio;
                       heapRepairDown l prio last))
                  | ^((_ :: ()) <- prioSet i rprio;
                      heapRepairDown r prio last)}})))}))}
```

Figure 7: Translation of `removePriority` in (Simply Typed) MPEG

```
-- Numeric Literals:
zero1     :: Ix 256
zero1      = 0{Ix 256}{NumLit 0 (Ix 256)}
one1      :: Unsigned
one1       = 1{Unsigned}{NumLit 1 Unsigned}
two1      :: Unsigned
two1       = 2{Unsigned}{NumLit 2 Unsigned}

-- Arithmetic Operators:
plus1     :: Unsigned -> Unsigned -> Unsigned
plus1      = (+){Unsigned}{Num Unsigned}
sub1      :: Unsigned -> Unsigned -> Unsigned
sub1       = (-){Unsigned}{Num Unsigned}
mul1      :: Unsigned -> Unsigned -> Unsigned
mul1       = (*){Unsigned}{Num Unsigned}

-- Numeric Coercions:
unsigned1 :: Ix 256 -> Unsigned
unsigned1  = unsigned{Ix 256}{ToUnsigned (Ix 256)}

-- Numeric Comparisons:
neq1      :: Ix 256 -> Ix 256 -> Bool
neq1       = (/=){Ix 256}{Eq (Ix 256)}
lt1       :: Ix 256 -> Ix 256 -> Bool
lt1        = (<){Ix 256}{Ord (Ix 256)}
gt1       :: Ix 256 -> Ix 256 -> Bool
gt1        = (>){Ix 256}{Ord (Ix 256)}

-- Shift Operators:
shr1      :: Ix 256 -> Unsigned -> Ix 256
shr1       = (>>){Ix 256}{Shift (Ix 256)}

-- Index Operators:
decIx1    :: Ix 256 -> Maybe (Ix 256)
decIx1     = decIx{256}{Index 256}
at1       :: Ref (Array 256 (Stored (Ix 256)))
               -> Ix 256 -> Ref (Stored (Ix 256))
at1        = (@){256, Stored (Ix 256)}{Index 256}
modIx1    :: Unsigned -> Ix 256
modIx1     = modIx{256}{Index 256}
leq1      :: Unsigned -> Ix 256 -> Maybe (Ix 256)
leq1       = (<=?){256}{Index 256}
```

Figure 8: Primitive Instances for the Priority Set Example, Part 1

```
-- Monadic Operators:
return   :: forall m a. Monad m => a -> m a
return1   = return{M, ()}{Monad M}

readRef  :: forall m a b. (MemMonad m, ValIn a b) => Ref a -> m b
readRef1  = readRef{M, Stored (Ix 256), Ix 256}
                   {MemMonad M, ValIn (Stored (Ix 256)) (Ix 256)}
readRef2  = readRef{M, Stored Unsigned, Unsigned}
                   {MemMonad M, ValIn (Stored Unsigned) Unsigned}

writeRef :: forall m a. MemMonad m => Ref a -> ValIn a -> m ()
writeRef1 = writeRef{M, Stored (Ix 256), Ix 256}
                    {MemMonad M, ValIn (Stored (Ix 256)) (Ix 256)}
writeRef2 = writeRef{M, Stored Unsigned, Unsigned}
                    {MemMonad M, ValIn (Stored Unsigned) Unsigned}
```

Figure 9: Primitive Instances for the Priority Set Example, Part 2

ceding sections, we would obtain code that looks something like the following (omitting type annotations so as to minimize clutter):

```
filter1  = \q -> \ys ->
                { ((p <- q) => (Nil <- ys) => ^ Nil)
                | ((p <- q)
                    => ((Cons x xs) <- ys)
                         => (let rest = filter1 p xs)
                             => ((True <- p x)  => ^(Cons x rest))
                              | ((True <- True) => ^rest)) }
```

A naive implementation of this code, once supplied with values for the parameters `q` and `ys`, would start with the first alternative, binding `p` to the value of `q`, and then testing to see if `ys` is `Nil`. But if that test fails, then the implementation would backtrack to the second alternative, once again binding `p` to the value of `q`, and then testing to see if `ys` is a `Cons` value. Clearly, this process duplicates work (for example, by binding `p` two times), and performs unnecessary work (for example, by testing for a `Cons` value when this could be inferred from the fact that the previous test for `Nil` had failed).

Fortunately, each of the flavors of MPEG that are described in this report satisfies a collection of algebraic laws that can be applied in a specific manner to eliminate problems like the ones illustrated in the `filter1` example. For example, one rule tells us that there is actually no need to bind a variable with the value of another variable: we can just use a compile-time substitution instead:

$$(v \texttt{ <- } w) \texttt{ => } m \quad = \quad [w/v]m \tag{1}$$

Applying this to the code for `filter1` eliminates the two `(p <- q)` guards:

```
filter1 = \q -> \ys ->
              { ((Nil <- ys) => ^ Nil)
              | (((Cons x xs) <- ys)
                    => (let rest = filter1 p xs)
                        => ((True <- q x)  => ^(Cons x rest))
                         | ((True <- True) => ^rest)) }
```

A further benefit of this rewrite is that the body of `filter1` now contains a match of the form `((Nil <- v) => m1) | ((Cons x xs <- v) => m2)`, which can easily be recognized, for the purposes of translating into Fidget, as a conventional `case` expression of the form: `case v of Nil -> e1; Cons x xs -> e2`.

Another law of MPEG tells us that matching a simple pattern and an expression that have the same outermost constructor will always succeed::

$$(C\,p_1\ \ldots\ p_n \texttt{ <- } C\,e_1\ \ldots\ e_n) \texttt{ => } m \quad = \quad (p_1 \texttt{ <- } e_1) \texttt{ => } \ \ldots \texttt{ => } (p_n \texttt{ <- } e_n) \texttt{ => } m \quad (2)$$

As a special case, this law can be applied to further simplify the code for `filter1`, eliminating the guard `(True <- True)` to give the following code:

```
filter1 = \q -> \ys ->
              { ((Nil <- ys) => ^ Nil)
              | (((Cons x xs) <- ys)
                    => (let rest = filter1 p xs)
                        => ((True <- q x)  => ^(Cons x rest))
                         | ^ rest) }
```

After this rewrite, the last two lines have been reduced to a match of the form `((True <- e) => m1) | m2)`, which, this time, can be recognized as a `case` expression with a default branch of the form `case e of True -> e1; _ -> e2`.

As a result of using the laws mentioned above, we have now transformed the original MPEG code into a form that can easily be recognized and translated into a very simple functional program with the following structure:

```
filter1 = \q -> \ys ->
              case ys of
                 Nil  -> Nil
                 Cons x xs -> let rest = filter1 p xs
                              in case q x of
                                   True -> Cons x rest
                                   _    -> rest
```

At first glance, this may look embarrassingly like the original Habit definition! Is it possible that, after all this work, we have only managed to produce a mildly desugared version of the original code? Of course, the answer is no. Although the starting and end points in this particular example look superficially similar, what we have actually demonstrated here is a general method for translating arbitrarily complex Habit programs into a very simple functional language, ready for the back-end, with none of the complexities of equational definitions, guards, rich pattern matching mechanisms, or the complications and overheads of a polymorphic type system.

In the following subsections, we describe additional laws that can be used, generalizing the `filter1` example above, to transform arbitrary MPEG code into a particular *match normal form*. The latter is a subset of the simply typed MPEG language that can be directly recast as an input for the back-end.

7.1 Commitment and Failure

The syntax of matches in MPEG has four constructs: alternatives (written using the `|` operator); guarded matches (written using the `=>` operator); commits (written using `^`); and the `fail` construct that represents match failure.

As the name suggests, a commit signals the end of a successful pattern match, at which point any unexplored alternatives can be abandoned:

$$\texttt{\^{}}e \mathrel{\texttt{|}} m \quad = \quad \texttt{\^{}}\, e \tag{3}$$

Matches are embedded in expressions using the `{_}` notation, which provides a simple exception handling mechanism that allows backtracking through multiple alternatives if a pattern match failure occurs. If the matching process reaches a commit, however, then matching terminates successfully, and computation can proceed with the result that was committed:

$$\{\, \texttt{\^{}}e \,\} \quad = \quad e \tag{4}$$

On the other hand, if the all of the alternatives in a match are exhausted, then execution of the whole program will terminate immediately; in this respect, pattern match failure is not a catchable exception:

$$\{\, \texttt{fail} \,\} \quad = \quad \bot \tag{5}$$

Because it can never lead to a successful match, any occurrence of `fail` in a list of alternatives can be eliminated using one of the following identity laws:

$$\texttt{fail} \mathrel{\texttt{|}} m \quad = \quad m \qquad\qquad m \mathrel{\texttt{|}} \texttt{fail} \quad = \quad m \tag{6}$$

In a similar way, a guarded failure is guaranteed to fail:

$$g \mathrel{\texttt{=>}} \texttt{fail} \quad = \quad \texttt{fail} \tag{7}$$

Using the laws above, we can rewrite an arbitrary match into an equivalent form that does not mention `fail`; the only exception to this is the match `fail` itself, which cannot be further simplified.

The translation of Habit source code into MPEG does not actually introduce any occurrences of `fail`, but it is still possible to find uses of `fail` that are inserted as a result of program optimization or simplification. The following law, for example, shows how `fail` is introduced when there is an attempt to match one constructor against a different one:

$$(C_1\ p_1\ \ldots\ p_i \texttt{ <- } C_2\ e_1\ \ldots\ e_j) \texttt{ => } m \quad = \quad \texttt{fail} \tag{8}$$

7.2 Case Blocks

We refer to a match of the following form as a *case block* for `v`:

```
  (C1 x1 ... <- v) => m1
| (C2 y1 ... <- v) => m2
| ...
| (Ck z1 ... <- v) => mk
```

In words, a case block for `v` is just a sequence of alternatives, each of which is guarded by a pattern match on the variable `v` with a simple constructor pattern whose arguments are all variables. As indicated in the earlier discussion, case blocks are of particular interest here because they correspond directly to a simple form of `case` expression, with a straightforward operational interpretation: check the tag of the value in `v`, and then perform a multiway branch to the code for the associated match.

Note that we do not need to worry about how the sequence of alternatives in case block is parenthesized because fatbar is associative:

$$m_1 \mid (m_2 \mid m_3) \quad = \quad (m_1 \mid m_2) \mid m_3 \tag{9}$$

In general, we can assume that each of the matches in a case block has a distinct constructor. If a given case block does not already have this property, then it can be established in two stages. In the first stage, we can apply the following law that allows us to reorder pairs of matches guarded by distinct constructors.

$$\begin{aligned} &((C_1\ p_1\ \ldots\ p_i \texttt{ <- } e_1) \texttt{ => } m_1) \mid ((C_2\ q_1\ \ldots\ q_j \texttt{ <- } e_2) \texttt{ => } m_2) \\ &\qquad = \quad ((C_2\ q_1\ \ldots\ q_j \texttt{ <- } e_2) \texttt{ => } m_2) \mid ((C1\ p_1\ \ldots\ p_i \texttt{ <- } e_1) \texttt{ => } m_1) \end{aligned} \tag{10}$$

Specifically, we can use this law to reorder the matches in a case block so that all matches with a particular constructor appear together in a group of adjacent matches. In the second stage, we can use the following (distributivity) law to merge matches that are guarded by the same constructor pattern[1]:

$$(g \texttt{ => } m_1) \mid (g \texttt{ => } m_2) \quad = \quad g \texttt{ => } (m_1 \mid m_2) \tag{11}$$

[1]This law is actually more general than we need here because it allows an arbitrary common guard, and is not restricted to matching against constructor patterns.

In practice, of course, the fact that two matches are guarded by patterns with the same constructor may not be enough to apply Law (11) directly; we also need to make sure that the same variable names are used in each one. This property, however, is easily established by applying a renaming substitution:

$$(C\ u_1\ \ldots\ u_i \texttt{ <- } e) \texttt{ => } m \quad = \quad (C\ v_1\ \ldots\ v_i \texttt{ <- } e) \texttt{ => } [v_1/u_1, \ldots, v_n/u_n]m \qquad (12)$$

Note that, if m_1 and m_2 are case blocks for the same variable `v`, then we can use the laws given above to rewrite the expression $m_1 \texttt{ | } m_2$ as a single case block for that variable. This observation allows us to restrict our attention to *maximal* case blocks, meaning that we only need to consider expressions like $m_1 \texttt{ | } m_2$ with adjacent case blocks if they are for different variables.

In theory, we could permute the matches in a case block in an arbitrary manner. Our implementation, however, is careful to arrange the matches so that the constructors are listed in the same order in which they are introduced in the Habit source code. For example, if the source matches on a sequence of constructors in the order `C`, `C`, `D`, `C`, `D`, `E`, then the corresponding case block will have three matches with the constructors appearing in the order `C`, `D`, `E`. This allows the programmer to include some performance hints in their code (put the fast path first), although we do not guarantee that this information will be used in the back-end. More importantly, it is important to preserve the ordering of constructors in matches over bitdata types when there is a particular bit pattern that could match against multiple constructors. In particular, this means that we cannot use Law (10) with bitdata types if there is confusion (in the technical sense [1, Chapter 8]) between the constructors C_1 and C_2.

As we have seen, the guards at the beginning of each match in a case block have a particularly simple form, requiring variables for each argument of the constructor function on the left, and a single variable on the right of the `<-` symbol. If the expression `e` in the guard `(p <- e)` is not a variable, then we can use the following law to break the guard into two pieces: a `let` construct that binds the value of the expression to a freshly generated variable `v`, and second guard `(p <- v)` that performs matching:

$$(p \texttt{ <- } e) \texttt{ => } m \quad = \quad (\texttt{let } v = e) \texttt{ => } (p \texttt{ <- } v) \texttt{ => } m \qquad (13)$$

There are also laws that we can use to decompose any pattern in a guard that does not have the simple form that is required for a case block. A nested constructor pattern, for example, can be rewritten as a simple match on the outermost constructor with subsequent guards to match each component:

$$\begin{aligned}&(C\ p_1\ \ldots\ p_n \texttt{ <- } e) \texttt{ => } m \\ &\qquad = \quad (C\ v_1\ \ldots\ v_n \texttt{ <- } e) \texttt{ => } (p_1 \texttt{ <- } v_1) \texttt{ => } \ \ldots \texttt{ => } (p_n \texttt{ <- } v_n) \texttt{ => } m \qquad (14)\end{aligned}$$

Guarded patterns can be eliminated by using the following law that moves the guard from a pattern to the associated match:

$$((p \texttt{ | } g) \texttt{ <- } e) \texttt{ => } m \quad = \quad (p \texttt{ <- } e) \texttt{ => } g \texttt{ => } m \qquad (15)$$

$$
\begin{array}{lcl}
\mathcal{M}[\![m_1 \texttt{ | } m_2 \texttt{ | } \ldots \texttt{ | } m_n]\!]k & = & \texttt{let}\ \texttt{k}_n\ \texttt{()}\ =\ \mathcal{B}[\![m_n]\!]k \\
 & & \qquad \ldots \\
 & & \qquad \texttt{k}_1\ \texttt{()}\ =\ \mathcal{B}[\![m_1]\!]\texttt{k}_2 \\
 & & \texttt{in}\ \texttt{k}_1\texttt{()}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{B}[\![\texttt{(}p_1 \texttt{ <- } v\texttt{) => } m_1 \texttt{ | } \ldots \texttt{ | (}p_n \texttt{ <- } v\texttt{) => } m_n]\!]k & & \\
 & = & \texttt{case}\ v\ \texttt{of} \\
 & & \quad p_1 \texttt{ -> } \mathcal{M}[\![m_1]\!]k \\
 & & \quad \ldots \\
 & & \quad p_n \texttt{ -> } \mathcal{M}[\![m_n]\!]k \\
 & & \quad \texttt{\_} \ \texttt{ -> } k\texttt{()} \\
\mathcal{B}[\![\texttt{(let}\ d\texttt{) => } m]\!]k & = & \texttt{let}\ d\ \texttt{in}\ \mathcal{M}[\![m]\!]k \\
\mathcal{B}[\![\texttt{\^{}}\ e]\!]k & = & e
\end{array}
$$

Figure 10: Compilation of Match Normal Forms

Finally, if the pattern in a matching guard is a simple variable, then the guard can never fail, and we can use the following law to rewrite the pattern matching guard as a `let` guard:

$$(v \texttt{ <- } e) \texttt{ => } m \quad = \quad (\texttt{let}\ v = e) \texttt{ => } m \tag{16}$$

7.3 Match Normal Form for MPEG Code

We will say that a match m is in *(match) normal form* if:

- $m = \texttt{fail}$; or
- $m = (m_1 \texttt{ | } \ldots \texttt{ | } m_n)$ for some $n \geq 1$ and some matches $m_1, \ldots, m_n$ such that each m_i is either:
  - a (maximal) case block in which each match on the right of its outermost guard is also in (match) normal form; or
  - an evaluation block, that is a match of the form `let` d `=>` m for some match m that is in (match) normal form; or
  - a commit of the form `^e`, but only in the case $i = n$;

Using the laws for MPEG as indicated in the previous sections, we can convert an arbitrary match into an equivalent version that is in match normal form, and then, using the compilation schemes in Figure 10, translate the match into code for a simple functional language with only basic pattern matching support. There are two compilation functions here: $\mathcal{M}[\![m]\!]k$ takes a match m in normal form as its argument, while $\mathcal{B}[\![m]\!]k$ compiles a single block (i.e., the individual alternatives in a normal form match). In both cases, the additional

k parameter specifies a *failure continuation*, which is used to specify how pattern match failure should be handled. For example, we can translate an MPEG expression of the from `{`m`}` using $\mathcal{M}[\![m]\!]$`abort`, where `abort` is a specific failure continuation, provided by the run-time system, that terminates program execution.

In the compilation schemes shown in Figure 10, we use thunks (i.e., functions taking a trivial `()` argument) to ensure that later alternatives in a match are not evaluated prematurely. In practice, the back-end can avoid the overhead of creating and entering thunks in these cases, translating invocations of failure continuations into direct jumps instead.

Some optimizations to the compilation schemes are possible in cases where we can determine that the continuation parameter for a match will never be used. For example, if the list of patterns in a case block covers all of the constructors for a specific algebraic datatype, then we can omit the default branch, `_ ->` k`()`, of the generated `case` construct. More generally, it is useful to provide a syntactic test for determining that a particular match *cannot fail*:

- A commit, `^`e, cannot fail.
- A case block, `(`p_1 `<-` v`) =>` m_1 `| ... | (`p_n `<-` v`) =>` m_n, cannot fail if the patterns $\texttt{p}_1, \ldots, \texttt{p}_n$ include cases for all of the constructors of a particular algebraic datatype and if all of the matches $m_1, \ldots, m_n$ cannot fail.
- An evaluation block, `(let` d`) =>` m, cannot fail if the match m cannot fail.
- A match, m_1 `|` m_2 `| ... |` m_n, cannot fail if any of the components m_i cannot fail.

To see how this can be used in practice, notice that if $i < n$ in the last case, then we can perform dead code elimination by removing the subterm m_{i+1} `| ... |` m_n from the match without changing its meaning.

Of course, this is only an approximation for determining whether a match is guaranteed to succeed: by inserting an evaluation block between two incomplete but complementary case blocks, it is not difficult to construct a match that does not satisfy the conditions for a match that cannot fail, even though it will actually succeed in practice. Such examples, however, are mostly pathological, and the definition above is appealing because it is safe, relatively simple, and works well in practice.

7.4 The Priority Set Example, Normalized

Following the pattern of previous sections, we end our discussion of normalized MPEG by showing how these ideas work in the context of the priority set example from Section 2. For the most part, however, the simply typed MPEG code in Figures 6 and 7 is already in match normal form, so there is not very much for us to do! The only exception has to do with the use of guards of the form `(p <- e)` where `e` is an expression and not just a variable as is required in a case block, a small detail that is easily rectified by applying Law (13).

For the purposes of explaining how the techniques described in this section apply to the priority set example, we will focus on the following match expression, which appears at the end of the code for `removePriority` in Figure 7:

```
{ (True{} <- { (True{} <- (gt1 prio lprio)   -- ** **
                  => ^ (gt1 prio rprio))
             | ^ False{} } =>
   ^(prioSet i prio))
| ^{ (True{} <- gt1 lprio rprio =>           -- **
      ^((_ :: ()) <- prioSet i lprio;
        heapRepairDown l prio last))
   | ^((_ :: ()) <- prioSet i rprio;
       heapRepairDown r prio last)}}
```

There are three examples of guards that do not have the necessary syntactic form for a case block in this code, as indicated by the `**` annotations on the right hand side. These, however, can be removed by applying Law (13), as mentioned previously, introducing three new local variables `v1`, `v2`, and `v3`:

```
(let v1 = { (let v2 = gt1 prio lprio) =>
             (True{} <- v2
                => ^ (gt1 prio rprio))
          | ^ False{} }) =>
{ (True{} <- v1 =>
   ^(prioSet i prio))
| ^{ (let v3 = gt1 lprio rprio) =>
     (True{} <- v3 =>
      ^((_ :: ()) <- prioSet i lprio;
        heapRepairDown l prio last))
   | ^((_ :: ()) <- prioSet i rprio;
       heapRepairDown r prio last)}}
```

If we apply the compilation schemes from Section 7.3 to this example, we obtain the following code, introducing three failure continuations `k1`, `k2`, and `k3` (but note, however, that the complete match cannot fail):

```
let v1 = let v2 = gt1 prio lprio
         in let k1 () = False
            in case v2 of
                 True -> gt1 prio rprio
                 _    -> k1 ()
```

```
in let k2 () = let v3 = gt1 lprio rprio
               in let k3 () = (_ :: ()) <- prioSet i rprio;
                              heapRepairDown r prio last
                  in case v3 of
                       True -> (_ :: ()) <- prioSet i lprio;
                               heapRepairDown l prio last
                       _    -> k3 ()
   in case v1 of
        True -> prioSet i prio
        _    -> k2 ()
```

In this particular example, because they are called only once from the default branch of a `case` expression, it is safe to inline each of the failure continuations to obtain the following equivalent version with an increase in code size:

```
let v1 = let v2 = gt1 prio lprio
         in case v2 of
              True -> gt1 prio rprio
              _    -> False
in case v1 of
      True -> prioSet i prio
      _    -> let v3 = gt1 lprio rprio
              in case v3 of
                   True -> (_ :: ()) <- prioSet i lprio;
                           heapRepairDown l prio last
                   _    -> (_ :: ()) <- prioSet i rprio;
                           heapRepairDown r prio last
```

We include this version of the code here because the control flow is easier to see once the failure continuations have been removed. However, as mentioned previously, we would expect the back-end to eliminate the code for constructing and entering failure continuation thunks. As a result, the two versions of this program shown here would already yield essentially the same target code, even if the back-end does not implement a more general version of inlining.

References

[1] Iavor Sotirov Diatchki. *High-Level Abstractions for Low-Level Programming*. PhD thesis, OGI School of Science & Engineering at Oregon Health & Science University, May 2007.

[2] Mark Jones. Dictionary-free overloading by partial evaluation. Technical Report YALEU/DCS/RR-959, Department of Computer Science, Yale University, April 1993.

[3] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[4] Andrew McCreight, Tim Chevalier, and Andrew Tolmach. A certified framework for compiling and executing garbage-collected languages. In *Proceedings of ICFP 2010, the 15th ACM SIGPLAN International Conference on Functional Programming*, Baltimore, MD, September 2010.

[5] J. C. Mitchell and R. Harper. The essence of ML. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*, January 1988.

[6] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report*. Cambridge University Press, 2003.

[7] The High Assurance Systems Programming Project (Hasp). The Habit Programming Language: The Revised Preliminary Report. Technical report, Department of Computer Science, Portland State University, Portland, Oregon, USA, November 2010.

[8] Tim Chevalier and Caylee Hogg. The Fidget Intermediate Language. Technical report, Department of Computer Science, Portland State University, Portland, Oregon, USA, November 2010.

[9] Philip Wadler and Stephen Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the Sixteenth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76, Austin, TX, USA, January 1989.