Using Hindley-Milner Type Inference to Optimise List Representation

Cordelia V. Hall,
Computing Science Dept,
Glasgow University,
Glasgow, Scotland,
cvh@dcs.glasgow.ac.uk

Abstract

Lists are a pervasive data structure in functional programs. The generality and simplicity of their structure makes them expensive. Hindley-Milner type inference and partial evaluation are all that is needed to optimise this structure, yielding considerable improvements in space and time consumption for some interesting programs. This framework is applicable to many data types and their optimised representations, such as lists and parallel implementations of bags, or arrays and quadtrees.

1 Introduction

Lists are a popular data structure among programmers using strongly-typed functional languages such as ML and Haskell[HdEtAl92]. For this reason, it is important to represent and use them as efficiently as possible.

There are a variety of optimisations on simple list representation, and it isn't difficult to think of new ones. The hard problem, and the one solved in this paper, is to automatically infer where an optimised representation can be used. The approach we take can be summarized as follows;

- Each pair of constructors in a simple list is replaced in a *compressed* list by one 'hydra' constructor with two heads. If the list has an odd length, then a single headed constructor appears at the end.
- Lists are treated as an abstract data-type (ADT). The new list ADT exports a simple list representation, and operations on both the simple and the compressed representations.
- A systematic transformation, guided by an analysis phase, transforms a user program written entirely in terms of simple lists into one which uses compressed lists in as many places as possible.

  The analysis and transformation propagates *local* constraints on list structure by using Hindley-Milner type inference on a list type modified to represent both compressed and simple lists.

  A two-stage process handles *global* constraint propagation. First, the type inference uses a standard transformation to convert the type-checked source program into a restricted form of the second order polymorphic lambda calculus. In this 'core language', each polymorphic function application receives extra type arguments which instantiate the polymorphic variables of the function's type. Some of these type arguments express local constraints on list structure.

  Partial evaluation then statically reduces these type applications, propagating global constraints to each instance of an operation on lists and replacing it with a reference to a version whose type expresses the same constraints.

benchmark	time
adder	82%
cosine transform	92%
fast fourier transform	88%
life	51%
peano	37%
sigma	66%
transitive closure	69%

Figure 1: Summary of results (optimised execution time divided by unoptimised execution time)

Given the appropriate pragmas, this framework is applicable to many data types and their optimised representations, such as lists and parallel implementations of bags [KuGl93], or arrays and quadtrees. Unlike many transformations, it is fully higher order.

The analysis and transformation has been implemented in the Glasgow Haskell compiler, and has significantly improved execution times on a number of small programs (see Figure 1) — up to a factor of two or more on list intensive programs. A later section of the paper gives additional statistics, such as total space allocation ratios and changes in code size.

The next section describes the basic ideas in more detail, and is followed by comparisons with related work. The

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

analysis and transformation are specified using type inference rules and a partial evaluation algorithm, followed by an extended example. Finally, we show that the transformation is safe and terminates, and give performance figures.

2 The idea

Compilers often perform an optimisation called 'loop unrolling', in which the loop body is substituted once for its call. This is applicable when a loop test is expensive and the body of the loop is small. It trades code size for the time and space needed to perform the loop test.

Here is an unrolled version of map [HlWs89], written in Haskell [HdEtAl92]. The list constructor *cons* is written :, and *nil* is [].

```
map f (x1:x2:xs)
 = f x1 : f x2 : map f xs
map f (x1:[])
 = f x1 : []
map f []
 = []
```

The optimised map constructs half as many thunks for recursive calls. We can make map even more efficient by giving it a compressed list type that builds only half the number of *cons* cells. This new list type contains a "hydra" constructor Cons2, a cons-cell with more than one head.

```
  data List2 a
    = Cons2 a a (List2 a)
    | Cons1 a
    | Nil
```

Here is map's definition using the compressed representation. We refer to this version as map_O_O, because it receives and returns an optimised list.

```
  map_O_O :: (a -> b) ->
             List2 a -> List2 b
  map_O_O f (Cons2 x1 x2 xs)
    = Cons2 (f x1) (f x2) (map_O_O f xs)
  map_O_O f (Cons1 x1)
    = Cons1 (f x1)
  map_O_O f Nil
    = Nil
```

Similarly, we write map_S_O for the version of map with type

```
(a -> b) -> List Simple a
        -> List u b
```

The version map_O_O is significantly more efficient than the original, but it isn't directly usable because it refers to List2. Instead, we develop a transformation that automatically inserts references to it at compile time.

Theoretically, there are four possible versions of map to work with, and these have the following types:

```
map_S_O :: (a -> b) ->
           [a]     -> List2 b
map_O_S :: (a -> b) ->
           List2 a -> [b]
map_O_O :: (a -> b) ->
           List2 a -> List2 b
map_S_S :: (a -> b) ->
           [a]     -> [b]
```

In fact, it seems reasonable to assume that each operation on lists will have 2^n versions if its type has n list types. However, this assumes that it is always a good idea to compress a list. Our goal is to compress lists only if that doesn't create extra work which might slow the program down.

It turns out that some functions cannot use List2 without performing extra work. For example, the append function (++) should not take an optimised list as its second argument. Suppose that its call was

```
append
   (Cons2 1 2 (Cons1 3))
   (Cons2 4 5 Nil)
```

The result would be

```
(Cons2 1 2
   (Cons2 3 4
      (Cons1 5 Nil)))
```

which means that the second argument had to be recopied. Restricting the second argument to be a simple list forces the result of append to be simple as well. We can still decide whether or not to optimise the first argument, as this list has to be traversed and rebuilt anyway.

Thus while eight versions of append are available, only two, which have the following types, are beneficial:

```
append_S_S_S :: [a]     -> [a] -> [a]
append_O_S_S :: List2 a -> [a] -> [a]
```

This demonstrates that the transformation must be able to decide where it should use compressed lists and where it should not. The goal is to select the most optimised version available, then settle for something worse when necessary. *Notice that it never has to coerce one type to another, which would be inefficient.* The compressed and simple types are completely separate.

2.1 Using type inference to distinguish between forms of the type

The transformation must infer where lists are constrained to be simple before determining which versions can be used. This is done in two stages, the first being Hindley-Milner type inference.

The compiler uses a list-type definition that includes a new field called a *selector field*. For example, the usual list-type, defined as

```
  data List a = Nil
              | Cons a (List a)
```

is extended as follows:

```
  data List t a = Nil t
                | Cons t a (List t a)
```

The extra type variable t is only relevant to the analysis.

Unification will instantiate this selector field if either a producer or consumer of a given list requires it to be simple, otherwise it will remain polymorphic. If it remains polymorphic, then both producer and all consumers can handle compressed lists, so the compiler is at liberty to substitute optimised versions of the functions. Thus there are only two forms that selector types can take on — the Simple type or a type variable.

The function map can take and produce any form of list, so we give it the type:

```
map :: (a -> b) ->
          List t a -> List u b
```

where t and u are unconstrained selector fields. The selector fields in this type can be instantiated in any one of four ways, allowing the transformation to select one based upon context.

However, we give append the type:

```
append :: List t a
           -> List Simple a
           -> List Simple b
```

where Simple is some predefined type known to the compiler. This indicates that the first argument to append may be either a simple list or a compressed list, but both its second argument and its result must be simple lists.

We need the full power of unification because constraints on list structure flow up and down the parse tree. For example, suppose we have the expression

```
map f (append xs ys)
```

When this is typechecked, the argument type of map f will be unified with the type of (append xs ys). This type, and the type of ys, are constrained to be simple by our type for append, and this in turn constrains the *argument* type of map.

On the other hand, the expression

```
append xs (map f ys)
```

shows that constraints can cause function *results* to be simple lists.

2.2 Propagating context information

In the same way that more than one version of map may be needed, so the transformation may have to generate more than one version of a user defined function. For example, suppose that the program defines the function

```
glue front middle back
 = append front
     (append middle back)
```

The type inference will assign glue the polymorphic type

```
glue :: List t a -> List u a
                  -> List Simple a
                  -> List Simple a
```

which allows for the possibility that the first two arguments may be compressed. It would be a mistake to assume that these *must* be compressed, because the function may be *used* in a context that constrains its arguments to be simple. For example, one of them may receive a value created by append. Thus in each application of glue, the transformation has to ensure that the application's context determines the appropriate version.

There is a simple way to do this. We can make the typechecker translate a program into the second order polymorphic lambda calculus in which types are manipulated directly, almost like values themselves. In particular, a polymorphic function will have a number of *type parameters.* An instance of the function is expressed as a *type application* in which the function receives the actual instance types, which are then bound to the type parameters within that function body.

For example, the polymorphic function glue is translated into the following form:

$$\begin{array}{l} \text{glue} \\ \qquad = \Lambda\ t. \\ \qquad\qquad \lambda\ \text{front middle back.} \\ \qquad\qquad\qquad \text{append}\ t\ \text{front} \\ \qquad\qquad\qquad (\text{append}\ t\ \text{middle back}) \end{array}$$

The type variable *t* is bound by the type lambda form Λ, and is in turn passed as an argument to the two instances of append, where it will take on the type of the list elements. Note, however, that when lists are being optimised, the definition of glue contains some extra polymorphism introduced by the selector field (in small caps here).

$$\begin{array}{l} \text{glue} \\ \qquad = \Lambda\ \textsc{sel1}\ t\ \textsc{sel2}. \\ \qquad\qquad \lambda\ \text{front middle back.} \\ \qquad\qquad\qquad \text{append}\ \textsc{sel1}\ t\ \text{front} \\ \qquad\qquad\qquad (\text{append}\ \textsc{sel2}\ t\ \text{middle back}) \end{array}$$

If glue is ever used in a context where its first argument, say, is unoptimised, then it will be applied to the type SIMPLE, which is bound to (and instantiates) the polymorphic type variable for this argument. Thus all that has to be done to propagate these contexts safely is to use partial evaluation with respect to type applications, creating new versions when necessary.

For example, the partial evaluation of glue in the application

$$\text{glue}\quad \textsc{Simple}\ \mathit{Char}\quad \textsc{tv}\quad \text{front middle back}$$

will be

$$\begin{array}{l} \lambda\ \text{front middle back.} \\ \qquad\qquad \text{append}\ \textsc{Simple}\ \mathit{Char}\ \text{front} \\ \qquad\qquad (\text{append}\ \textsc{tv}\ \mathit{Char}\ \text{middle back}) \end{array}$$

which is translated into

$$\begin{array}{l} \lambda\ \text{front middle back.} \\ \qquad\qquad \text{append\_S\_S\_S front} \\ \qquad\qquad (\text{append\_O\_S\_S middle back}) \end{array}$$

Here, partial evaluation has performed a compile-time beta-reduction of type applications (in practice, the type-lambda forms are left in place to enable other optimisations). The result is a version of glue in which all the information needed to select versions of append is now present.

3 Comparison with other work

There is an old LISP technique called cdr-coding which is related to this approach. Whenever a copy garbage collection occurred, list elements would be laid out in successive locations, and the cells tagged to indicate representation. This superseded technology selects list representation dynamically, whereas our approach allows static choices.

One question that might reasonably be asked is: can overloading of data types as defined by Jones [Jo93] handle our problem? Jones does this by defining a class of datatypes, and then instantiating the class type variable with the appropriate type.

Unfortunately, this does not work. It is easy to see why when we look at the Functor class, defined as

```
class Functor f where
 map :: (a -> b) -> f a -> f b
```

This has to be modified so that it can handle two different list types, one for the argument and an independent one for the result. That requires us to add another class variable, as in

```
class Functor f1 f2 where
 map :: (a -> b) -> f1 a -> f2 b
```

Unfortunately, overloading at more than one class type variable is often ambiguous, and unlikely to handle cases such as `zip`, which has two independent list arguments and returns a list result.

Shao, Reppy and Appel [ShReAp93] present an efficient representation for lists in ML which is slightly different from the one given here. The odd element appears at the beginning of the list, rather than the end, which produces a better version of `tail`. They use an algorithm based on refinement types to determine list parity, introducing optimised versions of *cons* when possible. Their results are also promising, and it would be interesting to compare our two methods for programs in ML. However, their approach applies to this problem only, and not to the general problem of introducing optimised representations for abstract data types.

Xavier Leroy [Le92] uses coercions `wrap` and `unwrap` and Hindley-Milner type inference to determine where unboxed types can be used. However, he observes that recursive data types require boxed elements at all times. The data type we propose in effect finesses this problem by unboxing all of the tails of the list represented by a hydra constructor. Leroy's approach also requires wrapping and unwrapping overhead whenever the list is constructed or accessed, which is not necessary in this framework.

Could deforestation [GiLaPJ93] largely or completely get rid of lists, making this work unnecessary? We have found that the two approaches are complimentary. In addition, current deforestation algorithms tend to degrade in the presence of higher order functions, and cannot handle sharing, whereas our technique compresses shared lists and is fully higher order.

Wadler's *views* [Wa87] are somewhat related to this problem. Both problems involve manipulating different representations of a data-type. However, the idea behind views is that it allows a variety of representations to be exported from an abstract data-type and used by the programmer at the source level, which is quite different from the approach given here.

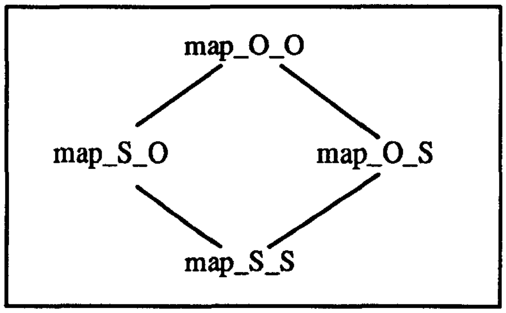

Figure 2: Lattice relating versions of `map`

4 The type inference rules

Type inference takes place after the program has been type checked, desugared, and after any analysis, such as strictness analysis, that supplies information it needs.

4.1 List operation versions

For each operation on lists, the transformation needs an 'optimal' version, one that uses the compressed data-type wherever possible. This is currently provided by the ADT library, but automatic version generation is also possible. Given the type of an optimal version, that library must also supply all versions with types below that type in the complete finite lattice induced by the Hindley-Milner polymorphic type ordering on the selector argument of the list type. For example, the most general type for `map` is

```
(a -> b) -> List t a -> List u b
```

The complete lattice of `map` versions appears in Figure 2.

If it isn't possible to use the 'best' version because not all versions below it exist, then a poorer 'optimal' version must be provided instead. For example, if the ADT could not provide `map_O_S`, then the best version would have to be `map_S_S`. However, none of the rest of the algorithm needs to be modified if this happens, as this worse version is still treated as the best available.

The reason for requiring a complete lattice of versions is that there must be a version available for every possible instantiation of the selector fields in the type of the best version. Initially the type inference rules use the best version type for each operation on lists, however the subsequent transformation must be able to instantiate that type freely.

4.2 The type environment and operations on it

The type inference rules take a program in the 'core language' of the compiler (Figure 3) and transform it into the second order polymorphic lambda calculus (Figure 4).

The core language is a standard intermediate language for functional compilers. It is the source and target language of the analysis and transformation. We assume that `fix` is a built-in function.

Extension of the type environment, Γ, is defined as

$$\Gamma_1 \bigoplus \Gamma_2 \;=\; \lambda\, i.\; i \in \mathbf{dom}\,\Gamma_2 \;\rightarrow\; \Gamma_2\; i, \Gamma_1\; i$$

The function `dom` returns the domain of an environment.

Expressions:

e	::=	v		variables
	|	con $e_1 \ldots e_n$,	$n \geq 0$	constructor applications
	|	e_1 e_2		application
	|	λ v e		lambda binding
	|	case e $(p_1, e_1) \ldots (p_n, e_n)$,	$n \geq 1$	case analysis
	|	let v e_1 e_2		let binding

Patterns:

p	::=	v		variable patterns
	|	con $p_1 \ldots p_n$,	$n \geq 0$	constructor patterns

Figure 3: The core language

Types:

τ	::=	α		type variables
	|	$\chi\ \tau_1 \ldots \tau_n$,	$n \geq 0$	constructor types
	|	$\tau_1 \rightarrow \tau_2$		function types
σ	::=	$\forall\ \alpha_1 \ldots \alpha_n.\tau$,	$n \geq 0$	polymorphic types

Expressions:

e	::=	v		variables
	|	con $e_1 \ldots e_n$	$n \geq 0$	constructor application
	|	e_1 e_2		application
	|	e $\tau_1 \ldots \tau_n$,	$n \geq 0$	type application
	|	$\Lambda\ \alpha_1 \ldots \alpha_n$. e,	$n \geq 0$	type lambda binding
	|	λ v e		lambda binding
	|	case e $(p_1, e_1) \ldots (p_n, e_n)$,	$n \geq 1$	case analysis
	|	let v e_1 e_2		let binding

Patterns:

p	::=	v		variable pattern
	|	con $p_1 \ldots p_n$,	$n \geq 0$	constructor pattern

Figure 4: The second order polymorphic lambda calculus

The usual initial type assignment, Γ_{init}, maps primitive functions and constructors to their types.

The type assignment Γ_{list} maps list operations to the type of the best version for the operation, and the constructors Nil and Cons to the type List applied to Simple.

Thus the initial type assignment used by these rules is

$$\Gamma_0 = \Gamma_{init} \bigoplus \Gamma_{list}$$

4.3 Inference rules

The rules themselves appear in Figure 5. They are straightforward, and are similar to others given elsewhere [MiHa88].

There are three judgement forms. They all have a similar structure,

$$\Gamma \vdash \mathrm{e} : \tau \leadsto \mathrm{e}$$

in which Γ is the type assignment, e the expression being typed, and e the translation of that expression.

The first, $\vdash^{\mathrm{polyexp}}$, infers a polymorphic type for an expression, while the second, $\vdash$, infers a monomorphic type. The third, $\vdash^{\mathrm{pat}}$, infers a type, and a type assignment for bound pattern variables.

Notice that type-lambda forms may enclose `let`-bound function definitions, but never are enclosed themselves (unless surrounded by a `let`). The reason for this is that Hindley- Milner types do not permit universal quantification unless it takes place over all type variables in the type; in other words, it must be at the outermost level.

5 Partial evaluation

The partial evaluator, $\mathcal{T}$, takes a program in the second order polymorphic lambda calculus and converts it back into the core language, inserting the appropriate operation version names.

$$\frac{\Gamma\ \mathrm{v} \;=\; \sigma}{\Gamma \;\vdash^{\mathrm{polyexp}}\; \mathrm{v} : \sigma \;\leadsto\; \mathbf{v}}$$

$$\frac{\Gamma \;\vdash^{\mathrm{polyexp}}\; \mathrm{v} : \forall\alpha_1 \ldots \alpha_n.\tau \;\leadsto\; \mathbf{v}}{\Gamma \;\vdash\; \mathrm{v} : \tau[\tau_1/\alpha_1 \ldots \tau_n/\alpha_n] \;\leadsto\; \mathbf{v}\ \tau_1 \ldots \tau_n}$$

$$\frac{\begin{array}{llll} \Gamma\ \mathrm{con} & = & \chi\ \tau_1 \ldots \tau_n & \\ \Gamma & \vdash & \mathrm{e}_1 : \tau_1 & \leadsto \mathbf{e_1} \\ \ldots & & & \\ \Gamma & \vdash & \mathrm{e}_n : \tau_n & \leadsto \mathbf{e_n} \end{array}}{\begin{array}{llll} \Gamma & \vdash & \mathrm{con}\ \mathrm{e}_1 \ldots \mathrm{e}_n : \chi\ \tau_1 \ldots \tau_n : & \leadsto \mathbf{con\ e_1 \ldots e_n} \end{array}}$$

$$\frac{\begin{array}{llll} \Gamma & \vdash & \mathrm{e}_1 : \tau_1 \to \tau_2 & \leadsto \mathbf{e_1} \\ \Gamma & \vdash & \mathrm{e}_2 : \tau_1 & \leadsto \mathbf{e_2} \end{array}}{\begin{array}{llll} \Gamma & \vdash & \mathrm{e}_1\ \mathrm{e}_2 : \tau_2 & \leadsto \mathbf{e_1\ e_2} \end{array}}$$

$$\frac{\Gamma \bigoplus \{\mathrm{v} : \tau_1\} \;\vdash\; \mathrm{e} : \tau_2 \;\leadsto\; \mathbf{e}}{\Gamma \;\vdash\; (\lambda\ \mathrm{v}\ \mathrm{e}) : \tau_1 \to \tau_2 \;\leadsto\; \lambda\ \mathbf{v\ e}}$$

$$\frac{\begin{array}{llll} \Gamma & \vdash & \mathrm{e} : \tau_e & \leadsto \mathbf{e} \\ \Gamma & \vdash^{\mathrm{pat}} & \mathrm{p}_1 : (\tau_e, \Gamma_1) & \leadsto \mathbf{p_1} \\ \Gamma \bigoplus \Gamma_1 & \vdash & \mathrm{e}_1 : \tau & \leadsto \mathbf{e_1} \\ \ldots & & & \\ \Gamma & \vdash^{\mathrm{pat}} & \mathrm{p}_n : (\tau_e, \Gamma_n) & \leadsto \mathbf{p_n} \\ \Gamma \bigoplus \Gamma_n & \vdash & \mathrm{e}_n : \tau & \leadsto \mathbf{e_n} \end{array}}{\begin{array}{llll} \Gamma & \vdash & \mathrm{case}\ \mathrm{e}\ (\mathrm{p}_1, \mathrm{e}_1) \ldots (\mathrm{p}_n, \mathrm{e}_n) : \tau & \leadsto \mathbf{case\ e\ (p_1, e_1) \ldots (p_n, e_n)} \end{array}}$$

$$\frac{\begin{array}{llll} \Gamma & \vdash & \mathrm{e} : \tau & \leadsto \mathbf{e} \\ \forall\, i,\ 1 \leq i \leq n,\ \alpha_i \notin \Gamma & & & \end{array}}{\begin{array}{llll} \Gamma & \vdash^{\mathrm{polyexp}} & \mathrm{e} : \forall\alpha_1 \ldots \alpha_n.\tau & \leadsto \mathbf{\Lambda\alpha_1 \ldots \alpha_n.e} \end{array}}$$

$$\frac{\begin{array}{llll} \Gamma & \vdash^{\mathrm{polyexp}} & \mathrm{e}_1 : \sigma & \leadsto \mathbf{e_1} \\ \Gamma \bigoplus \{\mathrm{v} : \sigma\} & \vdash & \mathrm{e}_2 : \tau & \leadsto \mathbf{e_2} \end{array}}{\begin{array}{llll} \Gamma & \vdash & (\mathrm{let}\ \mathrm{v}\ \mathrm{e}_1\ \mathrm{e}_2) : \tau & \leadsto \mathbf{(let\ v\ e_1\ e_2)} \end{array}}$$

$$\frac{\Gamma\ \mathrm{v} \;=\; \tau}{\Gamma \;\vdash^{\mathrm{pat}}\; \mathrm{v} : (\tau, \{\mathrm{v} : \tau\}) \;\leadsto\; \mathbf{v}}$$

$$\frac{\begin{array}{llll} \Gamma\ \mathrm{con} & = & \chi\ \tau_1 \ldots \tau_n & \\ \Gamma & \vdash^{\mathrm{pat}} & \mathrm{p}_1 : (\tau_1, \Gamma^1) & \leadsto \mathbf{p_1} \\ \ldots & & & \\ \Gamma & \vdash^{\mathrm{pat}} & \mathrm{p}_n : (\tau_n, \Gamma^n) & \leadsto \mathbf{p_n} \end{array}}{\begin{array}{llll} \Gamma & \vdash^{\mathrm{pat}} & \mathrm{con}\ \mathrm{p}_1 \ldots \mathrm{p}_n : (\chi\ \tau_1 \ldots \tau_n, \Gamma^1 \bigoplus \ldots \bigoplus \Gamma^n) & \leadsto \mathbf{con\ p_1 \ldots p_n} \end{array}}$$

Figure 5: Type inference rules

5.1 The partial evaluation algorithm

More specifically, the rules examine each type application,

$$f\ \tau_1\ \ldots\ \tau_n$$

They distinguish between list operations and other functions as follows.

5.1.1 List operations

If f is a list operation, the sequence of type arguments is used to select the version.

We define a function provided by the compiler,

$$\Psi\ ::\ \mathtt{Name}\ \rightarrow\ [\tau]\ \rightarrow\ \mathtt{Name}$$

which maps the name of the best version of the list operation, labelled with this type argument sequence, into the correct version for this particular occurrence of the operation.

This function first forms a substitution s from the polymorphic type of the best version,

$$\forall \alpha_1\ \ldots\ \alpha_n\ .\ \tau$$

and the sequence of types from the application. However, the idea is to instantiate only those type variables corresponding to the selector arguments, since these are what determine the version required. So from s, Ψ creates another substitution s', that maps all type variables to themselves, unless the variable is bound to Simple.

$$s'\ =\ \{(\alpha,\alpha)\mid(\alpha,\tau)\ \in\ s, \tau \neq\ \mathtt{Simple}\}\ \cup$$
$$\{(\alpha,\mathtt{Simple})\mid(\alpha,\mathtt{Simple})\ \in\ s\}$$

For example, suppose that

```
[Int, Simple, Bool, tv]
```

is a type argument sequence for the operation map. The substitution created is

```
[(a,a),(t,Simple),(b,b),(u,u)]
```

When applied to map's optimal version type, it produces the type

```
(a -> b)
  -> List Simple a
  -> List u b
```

and so the version selected is map_S_0.

Thus the transformation must build substitutions, using

$$\sigma\ \in\ \mathtt{Subst}\ ::\ \mathtt{Tv}\ \rightarrow\ \mathtt{Ty}.$$

The function aps applies a substitution to a type.

The initial substitution is

$$\sigma_0\ =\ \lambda\ \alpha.unbound.$$

5.1.2 User defined functions

If the function f is user-defined, then the sequence of type arguments must be propagated into its body, eventually to applications of list operations.

The definition of f is partially evaluated with respect to these type arguments, which are always static values. If a version of f has already been partially evaluated with respect to the same type arguments, then no new partial evaluation takes place. Instead, a reference to that version is inserted.

5.2 The code and version environments

In order to create versions, the partial evaluation rules need to be able to retrieve function definitions. This is done using the *code environment*

$$\rho\ \in\ \mathtt{Env}\ ::\ \mathtt{Name}\ \rightarrow\ \mathtt{Exp}$$

which maps let bound variables to their definitions. The initial code environment is

$$\rho_0\ =\ \lambda\ n.\ unbound.$$

Function versions are uniquely identifiable by a type argument sequence, which is used as a label:

$$\mathtt{Label}\ ::\ [\mathtt{Ty}]$$

We write a labelled variable as $\mathtt{v}_{\overline{\tau}}$.

The versions of a particular function are stored in an environment

$$\varphi\ \in\ \mathtt{Vfun}\ ::\ (\mathtt{Name},\mathtt{Label})\ \rightarrow\ \mathtt{Exp}$$

When a let expression is translated back into the core language, φ provides the versions of the locally defined function. We alter the grammar of the source language slightly, changing only the let form, which is redefined as

$$\mathrm{let}\ \varphi\ \mathrm{e}$$

and adding labels to variables.

Versions for all let-bound functions in a particular scope are stored in a *version environment*

$$\pi\ \in\ \mathtt{VEnv}\ ::\ \mathtt{Name}\ \rightarrow\ \mathtt{Vfun}$$

which is altered using update, defined as

$$\mathtt{update}\ \pi\ \mathtt{v}\ \mathtt{v}_{\overline{\tau}}\ \mathtt{e}\ =\ \pi[((\pi\ \mathtt{v})[\mathtt{e}/\mathtt{v}_{\overline{\tau}}])/\mathtt{v}]$$

The initial version environment is

$$\pi_0\ =\ \lambda\ n.\ unbound.$$

5.2.1 The partial evaluator

The partial evaluator $\mathcal{T}$ takes an expression, a substitution, a code environment, a version environment and a tuple containing the operation type assignment Γ_{list} and the abstract data type function Ψ. Each rule returns an expression and the new version environment, threading around the accumulated information on versions already created.

$$\mathcal{T}\ ::\ \mathtt{Exp}\ \rightarrow\ \mathtt{Subst}\ \rightarrow\ \mathtt{Env}$$
$$\rightarrow\ \mathtt{VEnv}\ \rightarrow\ (\mathtt{TyEnv},\mathtt{Adtf})$$

$$\mathcal{T}\ [[\mathbf{v}]]\ \sigma\ \rho\ \pi\ \iota \quad = \quad ([[\mathbf{v}_{[]}]], \pi)$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{T}\ [[\mathbf{con\ e_1 \ldots e_n}]]\ \sigma\ \rho\ \pi\ \iota} \\
= & \text{let} & (\mathbf{e}'_1, \pi^1) = & \mathcal{T}\ \mathbf{e_1}\ \sigma\ \rho\ \pi\ \iota \\
 & & \ldots & \\
 & & (\mathbf{e}'_n, \pi^n) = & \mathcal{T}\ \mathbf{e_n}\ \sigma\ \ \rho\ \pi^{n-1}\ \iota \\
 & \text{in} & & \\
 & & \multicolumn{2}{l}{([[\mathbf{con\ e'_1 \ldots e'_n}]], \pi^n)}
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{T}\ [[\mathbf{e_1\ e_2}]]\ \sigma\ \rho\ \pi\ \iota} \\
= & \text{let} & (\mathbf{e}'_1, \pi^1) = & \mathcal{T}\ \mathbf{e_1}\ \sigma\ \rho\ \pi\ \iota \\
 & & (\mathbf{e}'_2, \pi^2) = & \mathcal{T}\ \mathbf{e_2}\ \sigma\ \rho\ \pi^1\ \iota \\
 & \text{in} & & \\
 & & \multicolumn{2}{l}{([[\mathbf{e'_1\ e'_2}]], \pi^2)}
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{T}\ [[\mathbf{v}\ \tau_1 \ldots \tau_n]]\ \sigma\ \rho\ \pi\ \iota@(\Gamma_{\mathbf{list}}, \Psi)} \\
= & \multicolumn{3}{l}{\text{let}\ \tau'_1, \ldots, \tau'_n = \ \mathtt{aps}\ \sigma\ \tau_1, \ldots, \mathtt{aps}\ \sigma\ \tau_n} \\
 & \multicolumn{3}{l}{\qquad\quad \overline{\tau} \ \ = \ [\tau'_1, \ldots, \tau'_n]} \\
 & \multicolumn{3}{l}{\text{in}} \\
 & \multicolumn{3}{l}{\text{case}\ (\mathbf{v}\ \in\ \mathtt{dom}\ \Gamma_{\mathbf{list}})\ \text{of}} \\
 & \text{True} \rightarrow & \multicolumn{2}{l}{(\Psi\ \mathbf{v}\ \overline{\tau}, \pi)} \\
 & \text{False} \rightarrow & \multicolumn{2}{l}{\text{let}\ \ (\Lambda\ \alpha_1 \ldots \alpha_n.\ \mathbf{e})\ =\ \rho\ \mathbf{v}} \\
 & & \multicolumn{2}{l}{\qquad\ (\mathbf{e}', \pi') \qquad\quad = \mathcal{T}\ \mathbf{e}\ (\sigma[\tau'_1/\alpha_1]...[\tau'_n/\alpha_n])\ \rho\ \pi\ \iota} \\
 & & \multicolumn{2}{l}{\text{in}\ ([[\mathbf{v}_{\overline{\tau}}]],\ \mathtt{update}\ \pi'\ \mathbf{v}\ \mathbf{v}_{\overline{\tau}}\ \mathbf{e}')}
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{T}\ [[\mathbf{case\ e\ (p_1, e_1) \ldots (p_n, e_n)}]]\ \sigma\ \rho\ \pi\ \iota} \\
= & \text{let} & (\mathbf{e}', \pi') = & \mathcal{T}\ \mathbf{e}\ \sigma\ \rho\ \pi\ \iota \\
 & & (\mathbf{e}'_1, \pi^1) = & \mathcal{T}\ \mathbf{e_1}\ \sigma\ \rho\ \pi'\ \iota \\
 & & \ldots & \\
 & & (\mathbf{e}'_n, \pi^n) = & \mathcal{T}\ \mathbf{e_n}\ \sigma\ \rho\ \pi^{n-1}\ \iota \\
 & \text{in} & & \\
 & & \multicolumn{2}{l}{([[\mathbf{case\ e\ (p_1, e'_1) \ldots (p_n, e'_n)}]], \pi^n)}
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{T}\ [[\lambda\ \mathbf{v.e}]]\ \sigma\ \rho\ \pi\ \iota} \\
= & \text{let} & (\mathbf{e}', \pi^1) = & \mathcal{T}\ \mathbf{e}\ \sigma\ \rho\ \pi\ \iota \\
 & \text{in} & & \\
 & & \multicolumn{2}{l}{([[\lambda\ \mathbf{v.e'}]], \pi^1)}
\end{array}$$

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathcal{T}[[\mathbf{let\ v\ e_1\ e_2}]]\ \sigma\ \rho\ \pi\ \iota} \\
= & \text{let} & (\mathbf{e}', \pi^1) = & \mathcal{T}\ \mathbf{e_2}\ \sigma\ (\rho[\mathbf{e_1}/\mathbf{v}])\ \pi\ \iota \\
 & \text{in} & & \\
 & & \multicolumn{2}{l}{([[\mathbf{let}\ (\pi^1\ \mathbf{v})\ \mathbf{e}']],\ \pi^1[\bot/\mathbf{v}])}
\end{array}$$

Figure 6: Partial evaluation of core expressions

```
              → (Exp, VEnv)
```

The interesting rules are those handling variables and `let`.

Variables that are not applied to a series of types are lambda bound, in which case the variable is given a label indicating that it is monomorphic and returned.

A function applied to zero or more types is either a list operation or is defined by the program. If it is an operation, it will be in the domain of the type assignment for operations, in which case the appropriate version is found by Ψ. Otherwise, its definition is retrieved, the substitution is extended with a binding for each type variable bound by that definition, and it is partially evaluated. Finally, the version environment is updated with the new definition.

6 An example

In this section, we show what happens during compilation to this small program:

```
inc x        =    x + 1

inclist xs   =    map inc xs

main resps
             =    list (AppendChan stdout
                  (show
                  (inclist
                  (inclist
                  (1:2:[])))))
```

Initially, the compiler inserts references to the best versions for each list operation, and converts lists appearing in the user program into applications of `Cons` and `Nil` to `Simple`.

```
inc x        =    x + 1

inclist xs   =    map_O_O inc xs

main resps
             =    list (AppendChan stdout
                  (show
                  (inclist
                  (inclist
                  (Cons SIMPLE 1
                  (Cons SIMPLE 2
                        (Nil SIMPLE))))))
```

When Hindley-Milner type inference takes place, it converts the program into the second order polymorphic lambda calculus (we've abstracted away from Haskell overloading here!):

```
inc      =    λ x. x + 1

inclist
         =    Λ SEL1 SEL2. λ xs.
                   map_O_O Int SEL1 Int SEL2 inc xs
main resps
         =    list (AppendChan stdout
              (show
              (inclist TV1 TV2
              (inclist SIMPLE TV3
              (Cons SIMPLE Int 1
              (Cons SIMPLE Int 2
                    (Nil SIMPLE))))))
```

Notice that one of the instances of `inclist` is applied to a simple list, while the other is applied to the result of the first call to `inclist`. This last list can be compressed, while the first cannot, so the compiler must generate versions.

Partial evaluation starts from the main program, creating a version for `inclist` which expects a simple list of integers. When it encounters `inclist` again, it creates a new version which expects a compressed list.

```
inc        =    λ x. x + 1

inclist_S_O
           =    λ xs.
                     map_S_O inc xs

inclist_O_O
           =    λ xs.
                     map_O_O inc xs

main resps
           =    list (AppendChan stdout
                (show
                (inclist_O_O
                (inclist_S_O
                (1:2:[])))))
```

In practice, the constant lists were created by versions of a special list operation, so in fact the final program does not contain any constructors with selector fields.

7 Termination and safety

In this section, we prove that the transformation terminates and is safe. Termination is always an issue when a transformation creates versions of recursive programs, but Hindley-Milner type inference allows this to be controlled without modifying the partial evaluation algorithm.

Theorem 1 *$\mathcal{T}$ creates a finite number of function versions.*

Proof: By structural induction. The interesting case is `letrec` (here, `let` and `fix`). Hindley-Milner type inference forces all recursive references to a given function to have the same monomorphic type, thus sub-recursive references will not be partially evaluated. □

A 'safe' program is one in which no function expecting one list representation receives the other one instead. The transformed program will refer to functions using two distinct types for lists, so it is sufficient to prove that the entire analysis and transformation produces a well-typed program when given one.

Lemma 1 *If the initial program* e *is well-typed, then so is* e′, *where* $\Gamma_0 \vdash e : \tau \rightsquigarrow e'$.

Proof: This is a standard [MiHa88], well-understood translation. □

Lemma 2 *If the translation* e′ *is well-typed, then*

$$(\mathcal{T}\ [[e']]\ \sigma_0\ \rho_0\ \pi_0\ (\Gamma_{\mathbf{list}}, \Psi)) \downarrow 1$$

is well-typed.

Proof: By Lemma 1, all type-lambda expressions in e′ are well-typed. Mitchell and Harper give rules typing expressions in the second order polymorphic lambda calculus [MiHa88], including the following type inference rule for type applications (TAPP):

$$\frac{\Gamma \quad \rhd \quad M : \Pi t : U_1.\sigma}{\Gamma \quad \rhd \quad M\ \tau : [\tau/t]\sigma}$$

Thus, compile-time beta reduction of type applications preserves the original typing.
□

Theorem 2 *Let* e *be well-typed and let* e″ *be*

$$(\mathcal{T}\ [[e']]\ \sigma_0\ \rho_0\ \pi_0\ (\Gamma_{\mathbf{list}}, \Psi)) \downarrow 1$$

where

$$\Gamma_0 \vdash e : \tau \rightsquigarrow e'.$$

Then e″ *is well-typed.*

Proof: The transformation receives a well-typed program, e. By Lemmas 1 and 2, each type application referring to a list operation contains the correct monomorphic type for each selector field in the type of the best version for that operation. When creating a substitution, Ψ partitions these types into two equivalence classes: one containing only `Simple` and the other containing all the remaining monomorphic types (which will be type variables). It then selects a version by forming a one-to-one mapping between equivalence classes and list types, ensuring that e″ is well typed. □

8 Some figures for benchmarks

The benchmark table in Figure 7 gives the following figures:

- the execution time taken by the optimised program divided by the time taken by the unoptimised program,
- the total number of bytes allocated by the optimised program divided by the total number of bytes allocated by the unoptimised program,
- the number of extra versions of user-defined functions created by the transformation.

These programs were compiled by the Glasgow Haskell compiler, version 0.16, modified to optimise lists using this transformation. They were executed on a Sun SPARCstation 1 with 28M of RAM, using a two-space garbage collector. The list representation used was unrolled 5 times, rather than 2.

In estimating the value of the transformation on lists, the ratios expressing space consumption are probably the most useful, since timings can vary from machine to machine, and Sun SPARCstation 1 machines executing functional programs behave poorly when cache misses occur and the cache is direct mapped [HaBuHo93].

no.	benchmark	time	space	versions
1)	`adder`	82%	90%	10
2)	`cosine transform`	92%	93%	0
3)	`fast fourier transform`	88%	98%	1
4)	`life`	51%	89%	3
5)	`peano`	37%	78%	0
6)	`sigma`	66%	87%	0
7)	`transitive closure`	69%	77%	1

Figure 7: Percentages for the optimised list transformation implemented in *glhc*, Version 0.16, executing on a Sun 4/25.

The programs themselves varied, but were between 10 to 100 lines long. Figures for the entire Haskell 'nofib' suite would be more convincing, however the implementation is not yet ready for it. For example, it does not handle imported modules, and manipulating simple lists still trips over the tricky code structure that can be expected at core level (these are restrictions incurred only by the prototype - there is no fundamental reason why this technique can't be used with modules). However, several smaller benchmarks were written in the classic functional style that this transformation currently supports.

- `adder` implements a combinational binary adder circuit;
- `cosine transform`is part of Rex Page's Fourier benchmark in the 'spectral' section of the Glasgow Haskell compiler `nofib` suite [Pa92];
- `fast fourier transform` is another part of the same benchmark;
- `life` is John Launchbury's implementation of Conway's Life [Gr83], also in the 'spectral' section of the `nofib` suite;
- `peano` performs multiplication using lists to represent the natural numbers;
- `sigma` adds up a series of numbers;
- `transitive closure` implements a transitive closure algorithm based on the usual inductive definition. It is one of the programming assignments for a course on functional programming.

The number of versions indicated by the table is the number of additional user-defined function versions created by partial evaluation.

While some of these test cases were intended to push the representation as hard as possible to see what it could do under supposedly ideal circumstances (`peano`,`sigma`), others needed to do a substantial amount of filtering (`transitive closure`), or did odd things with list elements such as shift them to the left or right (`life`), or had many components which appeared in different contexts (`adder`), or contained cyclic structures which sometimes required that the whole structure be made simple (`cosine transform`, `fast fourier transform`). Handling cyclic structures is still a research problem, and requires intervention by hand at times to avoid black holes.

It is worth noting that the strictness analysis used was aggressive — lists were unrolled if they contained no tails that were undefined, which is much stronger than the usual tail strictness requirement. This worked well in practice, but probably will hurt programs with space leaks. On the other hand, ML programs do not require strictness analysis at all, and so they should do particularly well under this transformation.

9 Conclusion and further work

We have given a general framework for integrating operations over unoptimised data types with versions that use an optimised implementation. This has been successfully applied to operations on lists, and we expect it to apply to a number of other data types.

For example, a parallel implementation of bags [KuGl93] has been suggested as providing a new way to take advantage of implicit list parallelism in functional languages. Programmers import functions that handle bags, such as `map` and `foldr`, and then use them where sequential lists are not required by the program. The implementation of these functions, which is imperative, is hidden within the abstract data type. It should be possible to infer coercions between bags and lists using this framework if the programmer can supply pragmas identifying associative and commutative functions for `foldr`. The compiler would have to receive versions for list operations that have unoptimised types if the operation sequentially accesses a list. For example, the `nth` function, which retrieves the nth element of a list, would have a type that forces its argument to be unoptimised.

10 Acknowledgements

John Launchbury contributed much to the presentation of the basic ideas, and made other useful comments, as did the conference referees.

Dennis Howe, John T. O'Donnell, Simon Peyton Jones, Andre Santos and Phil Wadler all provided helpful feedback on this and earlier drafts.

References

[Gr83] Gardner, M., Wheels, Life and Other Mathematical Amusements, W.H.Freeman and Company, New York, 1983.

[GiLaPJ93] Gill, A, J. Launchbury and S.L. Peyton Jones, "A short cut to deforestation", *Proc. Functional Languages and Computer Architecture*, Copenhagen, DK, Springer-Verlag, (1993).

[HlWs89] Hall, C. V. and D. S. Wise, Generating function versions with rational strictness patterns, *Science of Computer Programming* 12 (1989) 39-74.

[HdEtAl92] Hudak, P., S. L. Peyton Jones, and P. Wadler, editors, Report on the Programming Language Haskell, version 1.2, *ACM Sigplan Notices*, 27(5), 1992.

[HaBuHo93] Hammond, K., G. L. Burn and D. B. Howe, Spiking Your Caches, In K. Hammond and J. T. O'Donnell, eds., proceedings of *Functional Programming, Glasgow 1993.*

[Jo93] Jones, M. P., A system of constructor classes: overloading and implicit higher-order polymorphism, *Proc. Functional Languages and Computer Architecture*, Copenhagen, DK, Springer-Verlag, (1993).

[KuGl93] Kuchen, H. and K. Gladitz, Parallel Implementation of Bags, *Proc. Functional Languages and Computer Architecture*, Copenhagen, DK, Springer-Verlag, (1993).

[Le92] Leroy, X. Unboxed objects and polymorphic typing, in *Proc. Principles of Programming Languages*, New Mexico, USA, Springer-Verlag, (1992).

[MiHa88] Mitchell, J.C. and R. Harper, The Essence of ML, *Proc. Principles of Programming Languages*, San Diego, California, Springer-Verlag, (1988).

[Pa92] Partain, W., The nofib benchmark suite of Haskell programs. In J. Launchbury and P. Sansom, eds., *Functional Programming, Glasgow 1992*, Springer-Verlag, (1992).

[PJnLn91] Peyton Jones, S. L. and J. Launchbury, Unboxed values as first class citizens, *Proc. Functional Languages and Computer Architecture*, Boston, Springer-Verlag, (1991).

[ShReAp93] Shao, Z. J. H. Reppy and A. W. Appel, Unrolling Lists, December 3, 1993, to appear in *Proc. LISP and Functional Programming, (1994).*

[Wa87] Wadler, P. Views: A way for pattern matching to cohabit with data abstraction, *Proc. Principles of Programming Languages*, Munich, Germany, Springer-Verlag, (1987).