# Unboxing using Specialisation<sup>\*</sup>

Cordelia Hall University of Glasgow Simon L. Peyton Jones University of Glasgow

Patrick M. Sansom University of Glasgow<sup>†</sup>

#### Abstract

In performance-critical parts of functional programs substantial performance improvements can be achieved by using *unboxed*, instead of boxed, data types. Unfortunately, polymorphic functions and data types cannot directly manipulate unboxed values, precisely because they do not conform to the standard boxed representation. Instead, specialised, monomorphic versions of these functions and data types, which manipulate the unboxed values, have to be created. This can be a very tiresome and error prone business, since specialising one function often requires the functions and data types it uses to be specialised as well.

In this paper we show how to automate these tiresome consequential changes, leaving the programmer to concentrate on where to introduce unboxed data types in the first place.

### 1 Introduction

Non-strict semantics certainly add to the expressive power of a language [8]. Sometimes the performance cost of this extra expressiveness is slight, but not always. It can happen that an inner loop of a program is made seriously less efficient by non-strictness. For example, consider the following fragment of a complex-number arithmetic package:

```
data Complex = Cpx Float Float
```

```
addCpx :: Complex -> Complex -> Complex
addCpx (Cpx r1 i1) (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
```

In a strict language, a complex number would be represented a pair of unboxed floating point numbers, and addCpx would actually perform the additions of the components. In a non-strict language such as Haskell [7], though, a complex number is a pair of pointers to possibly-unevaluated thunks. The function addCpx cannot actually force these thunks, since they might be bottom, but rather must build further thunks representing (r1+r2) and (i1+i2)respectively. If complex arithmetic is in the inner loop of the program, the performance penalty is quite substantial.

<sup>\*</sup>This paper appeared in Functional Programming, Glasgow 1994, Workshops in Computing, Springer Verlag, 1995.

<sup>&</sup>lt;sup>†</sup>Authors' address: Dept of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland. E-mail: {simonpj,sansom}@dcs.glasgow.ac.uk.

Let us suppose, then, that a profiler has focussed the programmer's attention on this arithmetic package. What can be done to make it more efficient? Peyton Jones & Launchbury [16] suggested that unboxed data types could be made "first class citizens", and that the programmer be allowed to declare data types involving them, thus:

#### data Complex = Cpx Float# Float#

Here, Float# is the type of unboxed floating-point numbers. (An immediate consequence is that the components of a complex number must be evaluated before the complex number itself is constructed, so that the representation is stricter than before.) Modifications of this kind can have a dramatic impact on the performance of some programs.

There is a catch, though. Suppose we wanted to transform a list of complex numbers to a list of their imaginary components. We might try to write:

```
imags :: [Complex] -> [Float#]
imags cs = [im | Cpx re im <- cs]</pre>
```

Unfortunately, we cannot form a list of unboxed floating point numbers, because both the size and the pointer-hood of a **Float#** differs from that of a pointer. Instead, a new data type must be declared for lists of **Float#**:

Alas, none of the usual list-manipulating functions (map, filter etc) work over LFloat, so new versions of them have to be defined, and so it goes on.

In general, Peyton Jones & Launchbury [16] put forward the restriction that: polymorphic functions (and data constructors) cannot be used at unboxed types. We use the term "creeping monomorphism" to describe the sad necessity to declare new functions simply because of this restriction. The goal of this paper is to lift the restriction, by automating the production of new versions of existing functions and data constructors.

Even the humble + function in addCpx is an example. When the Complex data type is changed, type inference will find that r1 and r2 are of type Float#. Since + is an overloaded function, with type

(+) :: Num a => a -> a -> a

Peyton Jones & Launchbury would prohibit + from being applied to a value of type Float#. However, if the restriction is lifted, and Float# is made an instance of class Num, then the code for addCpx will compile without modification.

Our goal is to allow the programmer to use profiling information [19] to improve run-time performance by making minimal changes to data type declarations and type signatures. The system we describe in this paper propagates these changes throughout the program, compiling specialised versions of polymorphic functions and constructors where they are now used at unboxed types.

We begin by describing our Core language (Section 2). We then examine the use of polymorphic functions and data types in the presence of unboxed values (Section 3), formalising Peyton Jones & Launchbury's unboxing restriction, describing the process of specialisation which enables us to relax this restriction (Section 3.1), and presenting a partial evaluator which performs this specialisation (Section 3.2). In Section 4 we discuss the practical implications, before presenting some preliminary results (Section 5) and discussing related work (Section 6).

### 2 The Core Language

Our source language is Haskell, but the language we discuss in this paper is the intermediate language used by our compiler, the Core language [15]. There are three reasons for studying this intermediate language. First, it allows us to focus on the essential aspects of the algorithm, without being distracted by Haskell's syntactic sugar. Second, Haskell's implicit overloading is translated into explicit function abstractions and applications so that no further special treatment of overloading is necessary. Third, the type abstractions and applications which are implicit in a Haskell program, are made explicit in the Core program. Thus, each polymorphic application which manipulates unboxed values can easily be identified by looking at the type arguments in the application. Our transformation identifies any applications involving unboxed types and replaces this with an appropriately specialised version of the function.

The syntax of the Core language is given in Figure 1. It is an explicitly typed second-order functional language with (recursive) let, (saturated) data constructors, algebraic case and explicit boxing. This language combines the higher-order explicit typing of Core-XML [11] with the explicit boxing and type structure of Peyton Jones & Launchbury [16].

In this language the argument of an application is always a simple variable. A non-atomic argument is handled by first binding it to a variable using a letexpression. This restriction reflects the fact that a non-atomic argument must be bound to a heap-allocated closure before the function is applied. In the case of a strict, unboxed value ( $\sigma \in$  UnboxedType), the let-expression evaluates the bound expression and binds the result value before the function is applied.

A type normalised expression [6] is one that satisfies the following two conditions. 1) The type abstractions occur only as the bound expression of letexpressions; i.e. let  $x: \sigma = \Lambda \alpha_1 . \Lambda \alpha_2 . . . \Lambda \alpha_n . e_1$  in  $e_2$  which we will abbreviate with let  $x: \sigma = \Lambda \alpha_1 . . . \alpha_n . e_1$  in  $e_2$ . 2) Type applications are only allowed for variables; i.e.  $x \{\tau_1\} . . . \{\tau_n\}$  which we will write  $x \{\tau_1 . . . \tau_n\}$ . Henceforth we will assume that all expressions are type normalised.

### 2.1 Data constructors

In this second-order language a data constructor must be (fully) applied to both the type arguments of the data type and the value arguments of the data object being constructed. For example, the standard list data type

data List  $\alpha$  = Nil | Cons  $\alpha$  (List  $\alpha$ )

has two constructors, Nil and Cons. The implied constructor declarations might be expressed in the higher-order calculus as follows:

```
Expression
                                             e ::=
                                                                   x
                                                                   \lambda x : \tau.e
                                                                   e x
                                                                   \begin{array}{l} \texttt{let} \ x : \sigma = e_1 \ \texttt{in} \ e_2 \\ \Lambda \alpha . e \end{array}
                                                                    \begin{array}{l} \underset{e \in \{\tau\}}{e \in \{\tau\}} \\ C \in \{\tau_1 \cdots \tau_n\} x_1 \cdots x_a \\ \text{case } e \text{ of } \{C_j \ x_{j1} : \tau_{j1} \cdots x_{ja_j} : \tau_{ja_j} \rightarrow e_j\}_{j=1}^m \end{array} 
PolyType
                                           \sigma ::= \forall \alpha . \sigma \mid \tau
                                            \tau ::= \pi \mid v
MonoType
                                           \begin{aligned} \pi & ::= & \alpha \\ & | & \tau_1 \to \tau_2 \\ & | & \chi & \tau_1 \cdots \tau_n \end{aligned} 
{\rm BoxedType}
UnboxedType v ::= int# | float# | char#
                                                                  \chi# \tau_1 \cdots \tau_n
```

Figure 1: Core Language Syntax

```
 \begin{array}{ll} \texttt{Nil} & : \; \forall \alpha.\texttt{List} \; \alpha \\ & = \; \Lambda \alpha.[\texttt{Nil}] \\ \texttt{Cons} \; : \; \forall \alpha.\alpha \rightarrow \texttt{List} \; \alpha \rightarrow \texttt{List} \; \alpha \\ & = \; \Lambda \alpha.\lambda v_1 : \alpha.\lambda v_2 : \texttt{List} \; \alpha.[\texttt{Cons} \; v_1 \; v_2] \\ \end{array}
```

where [] indicates actual construction of the data object. Even though the constructor Nil has an arity of zero the higher-order constructor still requires a type parameter to indicate what type it is being used at, e.g. Nil {Int}. In general, a data declaration has the form

data  $\chi \alpha_1 \cdots \alpha_n = C_1 \tau_{11} \cdots \tau_{1a_1} \mid \cdots \mid C_m \tau_{m1} \cdots \tau_{ma_m}$ 

which gives rise to m higher-order constructors with the form

 $\begin{array}{rcl} \mathbf{C}_{j} & : & \forall \alpha_{1} \cdots \alpha_{n} . \tau_{j \, 1} \rightarrow \cdots \rightarrow \tau_{j \, a_{j}} \rightarrow \chi \ \alpha_{1} \cdots \alpha_{n} \\ & = & \Lambda \alpha_{1} \cdots \alpha_{n} . \lambda v_{j \, 1} : \tau_{j \, 1} . \cdots \lambda v_{j \, a_{j}} : \tau_{j \, a_{j}} . [\mathbf{C}_{j} \ v_{j \, 1} \cdots v_{j \, a_{j}}] \end{array}$ 

where n is the number of type parameters of the data type and  $a_j$  is the arity of the data constructor. Since the number of type parameters is determined by the arity of the type constructor,  $\chi$ , it is the same for all data constructors of that type.

### 2.2 Well-formed expressions

We call an expression e well-formed under type assumption? if we can derive the typing judgement?  $\vdash e : \sigma$ . The typing rules are quite standard [11] and are not given here (but see Section 3).

#### 2.3 Notation

For notational convenience we abbreviate sequences such as  $x_1 \cdots x_n$  with  $\overline{x}$ , where  $n = \texttt{length } \overline{x}$ . This is extended to sequences of pairs which are abbreviated with paired sequences, e.g.  $x_{j1}: \tau_{j1} \cdots x_{ja_j}: \tau_{ja_j}$  is abbreviated with  $\overline{x_j}: \overline{\tau_j}$ . We use  $\equiv$  for syntactical identity of expressions.

### 3 Polymorphism and Unboxed Values

A pure polymorphic function is usually compiled by treating all polymorphic values in a uniform way. Typically all such values are required to be represented as a pointer to a heap allocated closure i.e. they must be boxed. For example, consider the permuting combinator C:

$$C f x y = f y x$$
(Haskell)  

$$C = \Lambda a \ b \ c.\lambda f : a \rightarrow b \rightarrow c.\lambda x : b.\lambda y : a.f \ y \ x$$
(Core)

To generate the code for C we must know the representation of the polymorphic values, x and y, being manipulated.<sup>1</sup> By insisting that such polymorphic values are always boxed we can compile code which assumes that such values are always represented by a single pointer into the heap.

It follows that a polymorphic function can only be used at boxed types, since the representation of an unboxed type violates the assumption above. We impose a restriction in the typing rules for expressions which prevents a polymorphic function being applied to an unboxed type.<sup>2</sup>

$$\frac{? \vdash e : \forall \alpha. \sigma}{? \vdash e \ \{\tau\} : \sigma[\tau/\alpha]} \ \tau \notin \text{UnboxedType}$$
(\*)

A similar restriction is imposed in the typing rule for data constructors. This prohibits the construction of polymorphic data objects with unboxed components, e.g. List Float#.

These restrictions cause the "creeping monomorphism", described in Section 1, since the programmer must declare suitable monomorphic versions of any polymorphic functions and data types used at unboxed types. This can be exceedingly tedious and error prone.

To address this problem we propose to relax the unboxing restriction (\*), allowing the programmer unrestricted use of unboxed values. During the compilation we undertake automatically to generate the necessary monomorphic function versions: converting the unrestricted program into one which satisfies the unboxing restriction (\*). We can then generate code which directly manipulates the unboxed values since their type, and hence their representation,

<sup>&</sup>lt;sup>1</sup>The representation information that is typically required is the size of a value and the position of any heap pointers (so that all roots can be identified during garbage collection). When more sophisticated calling conventions are used, such as passing arguments in registers, the actual type may also affect the treatment of a value. For example a boxed value may be passed in a pointer register, an Int# in an integer register, and a Float# in a dedicated floating point register.

<sup>&</sup>lt;sup>2</sup>This restriction is equivalent to "Restriction 1: loss of polymorphism" in Peyton Jones & Launchbury [16].

is known at compile time. For example, here is the monomorphic version of C which manipulates **Float#**s:

```
C' = \lambda f: \texttt{Float} \# \rightarrow \texttt{Float} \# \rightarrow \texttt{Float} \# . \lambda x: \texttt{Float} \# . \lambda y: \texttt{Float} \# . f \ y \ x
```

Since the code generator knows that x and y have type Float# it produces code which manipulates floating point numbers, instead of pointers. This is the only difference between the code produced for C and C'.

### 3.1 Specialisation

The transformation of program with unrestricted use of unboxed types into one which satisfies the unboxing restriction above is performed using a partial evaluator. The idea is to remove all type applications involving unboxed types by creating new versions of the functions being applied, specialised on the unboxed types. These specialised versions are created by partially evaluating the unboxed type applications.

Before launching into the definition of the partial evaluator itself, we give an overview of the algorithm. Each time a function (or constructor) is applied to a sequence of types, a new version of the function (or constructor), specialised on any unboxed types in the application, is created, unless such a version has already been created. For example, given the code<sup>3</sup>

```
append {Int#}
xs (map {[Int#] Int#}
(sum {Int#}) (append {[Int#]} yss zss)
```

a version of append, specialised at type Int#, is created. Given the definition of append:

append =  $\Lambda \alpha . \lambda xs : [\alpha] . \lambda ys : [\alpha] . e$ 

the specialised version, append\_Int#, is:

The name of the specialised version, append\_Int#, is constructed by appending the specialising type(s) to the original name.

When a function is applied to a boxed type, there is no need to specialise on that type argument since the polymorphic version, which assumes a boxed type will suffice. Consequently, we make the specialisation polymorphic in any boxed type arguments. For example, the application of **map** is only specialised on the second type argument, **Int#**, since the first type argument, **[Int#]**, is a boxed type.

```
 \begin{split} \mathtt{map} &= \Lambda \alpha \ \beta . \lambda \mathtt{f} : \alpha \to \beta . \lambda \mathtt{xs} : [\alpha] . e \\ \mathtt{map}_*\_\mathtt{Int} \# &= \Lambda \alpha_* . \mathtt{map} \ \{\alpha_* \ \mathtt{Int} \# \} \end{split}
```

<sup>&</sup>lt;sup>3</sup>For notational convenience we use the standard [] list notation, where  $[\alpha] \equiv \text{List } \alpha$ .

 $= \Lambda \alpha_* . (\Lambda \alpha \ \beta . \lambda \mathbf{f} : \alpha \to \beta . \lambda \mathbf{xs} : [\alpha] . e) \ \{\alpha_* \ \mathtt{Int\#} \}$ =  $\Lambda \alpha_* . (\lambda \mathbf{f} : \alpha \to \beta . \lambda \mathbf{xs} : [\alpha] . e) [\alpha_* / \alpha, \mathtt{Int\#} / \beta]$ =  $\Lambda \alpha_* . \lambda \mathbf{f} : \alpha_* \to \mathtt{Int\#} . \lambda \mathbf{xs} : [\alpha_*] . e [\alpha_* / \alpha, \mathtt{Int\#} / \beta]$ 

A \* is used to indicate a boxed type argument in which the specialised version remains polymorphic. This reduces the number of specialised versions created since all boxed type arguments will be treated as a \* type when determining the specialisation required. For example, the application map {Bool Int#} would also use the specialised version map\_\*\_Int#.

The applications can now be modified to use the specialised versions, with all unboxed types new removed from the application. The final version of the code for the example above is:

In summary the specialisation algorithm is:

while the unboxing restriction (\*) is not satisfied:

- 1. Find a type application,  $f\{\overline{\tau}\}$ , involving an unboxed type.
- 2. Create a suitably specialised version of f (if it does not already exist).
- 3. Use the specialised version at this application site, removing the unboxed types from the application.

Since all polymorphic values must be let-bound (see Figure 1), the definition of f, which has to be specialised, will always be visible in the enclosing scope.

Notice that the specialised versions must themselves be specialised since the substitution of the unboxed type over the body of the function may introduce further unboxed type applications. To ensure termination in the presence of recursive functions we rely on Hindley-Milner type inference having guaranteed that all recursive references occur at the same type; thus no new versions of a function will be created while specialising its body, since we must be creating the specialised version required.

#### 3.2 The partial evaluator

The specialisation algorithm is efficiently implemented using a partial evaluator.

The partial evaluator,  $\mathcal{T}$ , takes an expression with unrestricted use of unboxed types, and two environments: one containing the polymorphic **let**bindings and the other the specialised versions of those **let**-bindings which have been created so far. It returns a triple containing an equivalent expression which satisfies the unboxing restriction, a modified environment of specialised versions, and a set of specialised data types required.

The environments are partial maps with suitable domain and lookup functions. **BEnv** simply maps a variable name to its type and unrestricted expression. **SEnv** is a nested environment, mapping a variable name to an environment containing the specialised versions for that variable. The domain of this specialised environment is the variable name and the specialising types (a vector containing unboxed types and \*s), which uniquely identifies the specialised version. We use a subscript notation  $\mathbf{x}_{\overline{v}}$  to refer to the specialised version. We also use the notation  $\{\}$  for the empty environment and  $\rho[x \to v]$  to extend (or modify) an environment  $\rho$  with the mapping  $x \to v$ .

TSet is the set of specialised data types required by the expression. The partial evaluator does not explicitly specify the data type transformation — it just collects the data types required. These are subsequently given to the code generator which creates the required constructor functions directly from the data type specifications.

The partial evaluator is defined in Figure 2. The equations for simple variables (1),  $\lambda$ -abstraction (2), application (3), monomorphic let-binding (4), and **case** (8) are quite straightforward.

For a polymorphic let-binding (equation 5) let  $\mathbf{x} : \boldsymbol{\sigma} = \mathbf{e}_{\Lambda}$  in  $\mathbf{e}$  the body  $\mathbf{e}$  is evaluated using the following environments:  $\rho$  extended with the binding for  $\mathbf{x}$ ; and  $\delta$  extended with an empty set of specialised bindings for  $\mathbf{x}$ . (We assume that all bound variables have unique names.) The set of specialised bindings for  $\mathbf{x}$ , returned in the modified specialisation environment  $\delta^1$ , are then letbound and returned. For simplicity, we assume that the target form of the core language allows the set of specialised bindings to be bound in a single let.

A polymorphic application (equation 6) is replaced with an application of an appropriately specialised version of the binding. The auxiliary function spectys (Figure 3) determines:

- $\overline{v}$ : the unboxed types on which the binding must be specialised. A \* type indicates that the specialised version is still polymorphic in that type parameter.
- $\pi$ : the boxed types the specialised version remains polymorphic in. These correspond to the \* types in  $\overline{v}$ .

The specialised version  $\mathbf{x}_{\overline{v}}$  is then applied to the remaining boxed type arguments  $\overline{\pi}$  and returned. The auxiliary function **specfn** (Figure 3) is used to extend the environment  $\delta$  with a newly created specialisation (if it does already contain it). The original binding is extracted from  $\rho$  and the specialising types substituted for the corresponding type variable. (The usual alpha substitution to avoid capture is assumed.) The partial evaluator is then applied to the specialised body,  $\mathbf{e}'$ , in an environment,  $\delta'$ , extended with the specialisation being created. This ensures termination in the presence of recursion since recursive references will assume that the required specialisation already exists (see Section 3.1).

Finally, a constructor application (equation 7) is replaced with an application of an appropriately specialised version of the constructor and returned with a specification of the specialised data type required. The global environment? maps constructors to their data type.

 $\mathcal{T} \llbracket \mathbf{x} \rrbracket \rho \ \delta = (\llbracket \mathbf{x} \rrbracket, \delta, \{\})$ (1) $\mathcal{T} \llbracket \lambda \mathbf{x} : \tau . \mathbf{e} \rrbracket \rho \ \delta$ (2) $= \begin{array}{ccc} & \mathbf{\hat{e}'} & (\mathbf{e}', \delta^1, \gamma^1) &= \mathcal{T} \mathbf{e} \ \rho \ \delta \\ & \mathrm{in} & ([[\lambda \ \mathbf{x} : \tau . \mathbf{e}']], \ \delta^1, \ \gamma^1) \end{array}$  $\mathcal{T} [[\mathbf{e} \mathbf{x}]] \rho \delta$ (3) $\begin{array}{rcl} \operatorname{let} & (\mathbf{e}', \delta^1, \gamma^1) &= & \mathcal{T} \mathbf{e} \ \rho \ \delta \\ \operatorname{in} & ([[\mathbf{e}' \ \mathbf{x}]], \ \delta^1, \ \gamma^1) \end{array}$ =(4)(5) $\mathcal{T} \llbracket \mathbf{x} \{ \overline{\tau} \} \rrbracket \rho \ \delta$ (6)let  $(\overline{v}, \overline{\pi})$ = spectys  $\overline{ au}$ =in case  $\mathbf{x}_{\overline{v}} \in \operatorname{dom}(\delta \mathbf{x})$  of True  $\rightarrow$  ([[ $\mathbf{x}_{\overline{\mathbf{v}}} \{\overline{\pi}\}$ ]],  $\delta$ , {})  $\begin{array}{rcl} \text{False} & \rightarrow & \text{let} & (\delta^1, \gamma^1) & = & \texttt{specfn} \ \mathbf{x} \ \overline{v} \ \rho \ \delta \\ & & \text{in} & ([[\mathbf{x}_{\overline{\mathbf{v}}} \ \{\overline{\pi}\}]], \ \delta^1, \ \gamma^1) \end{array}$  $\mathcal{T} \llbracket \mathbf{C} \{ \overline{\tau} \} \mathbf{\overline{x}} \rrbracket \rho \delta$ (7) $\begin{array}{rcl} & & & \\ \operatorname{let} & & (\overline{v}, \overline{\pi}) & = & \operatorname{spectys} \overline{\tau} \\ & \chi & = & ? \mathbf{C} \end{array}$ = $\chi = \{ \mathbf{C} \\ ([[\mathbf{C}_{\overline{\mathbf{v}}} \{ \overline{\pi} \} \ \overline{\mathbf{x}}]], \ \delta, \ \{ \chi_{\overline{\mathbf{v}}} \})$ in(8) $\begin{array}{lll} (\mathbf{e}'_n, \delta^m, \gamma^m) &= & \mathcal{T} \; \mathbf{e}_n \; \rho \; \delta^{m-1} \\ ([[\mathbf{case} \; \mathbf{e}' \; \mathbf{of} \; \{ \mathbf{C}_{\mathbf{j}} \; \overline{\mathbf{x}_{\mathbf{j}}} \colon \overline{\tau_{\mathbf{j}}} \to \mathbf{e}'_{\mathbf{j}} \}_{\mathbf{j}=\mathbf{1}}^{\mathbf{m}}]], \; \delta^{\mathbf{m}}, \gamma^{\mathbf{0}} \cup \cdots \cup \gamma^{\mathbf{m}}) \end{array}$ in

Figure 2: The partial evaluator  $\mathcal{T}$ 

spectys  $\overline{\tau}$ = let  $\overline{v} = [v \mid \tau \leftarrow \overline{\tau}, v = \text{if } \tau \in \text{BoxedType then } * \text{else } \tau]$  $\overline{\pi} = [\tau \mid \tau \leftarrow \overline{\tau}, \ \tau \in \text{BoxedType}]$  $(\overline{v}, \overline{\pi})$ in specfn  $\mathbf{x} \ \overline{v} \ \rho \ \delta$  $(\forall \overline{\alpha}. \tau, \Lambda \overline{\alpha}. \mathbf{e})$ = let  $= \rho \mathbf{x}$ = length  $\overline{\alpha}$ n=  $\mathbf{e} [(\text{if } v_i \neq * \text{ then } v_i \text{ else } \alpha_i)/\alpha_i]_{i=1}^n$ =  $\mathcal{T} \mathbf{e}' \rho \delta'$  $\mathbf{e}'$  $(\mathbf{e}^{\prime\prime},\delta^1,\gamma^1)$  $= \delta[\mathbf{x} \to (\delta \mathbf{x})[\mathbf{x}_{\overline{v}} \to (\sigma, \mathbf{e}_{\Lambda})]]$  $= [\alpha_i \mid i \leftarrow [1..n], v_i \equiv *]$  $= \mathbf{\Lambda} \overline{\alpha_*} . \mathbf{e}''$  $\overline{\alpha_*}$  $\mathbf{e}_{\Lambda}$  $\tau'$ =  $\tau$  [(if  $v_i \not\equiv *$  then  $v_i$  else  $\alpha_i$ )/ $\alpha_i$ ]\_{i=1}^n  $= \forall \overline{\alpha_*}.\tau'$  $\sigma$  $(\delta^1, \gamma^1)$ in

Figure 3: Specialisation functions

### 4 Practical Considerations

The specialiser described above interacts with a number of other language features including: overloading, and separate module compilation. We address these issues and discuss the process of introducing unboxing below.

### 4.1 Overloaded functions

In Haskell many primitive functions, such as comparison and addition, are overloaded. This allows these operations to be applied to a number of different types. For example, addition belongs to the class Num

> class Num a where (+) :: a -> a -> a ...

which has instances for types such as Int, Integer, Float and Complex. Each of these instances provides a definition of the function which is called when it is used at that type. For example:

```
instance Num Int where
  (+) x y = plusInt x y
  ...
```

Since an overloaded function can now be applied to an unboxed type (it was prohibited by the unboxing restriction (\*) before), it makes sense to introduce new instances declarations for these unboxed types. For example:

```
instance Num Int# where
  (+) x y = plusInt# x y
  ...
```

This allows us to manipulate unboxed values in the same way as we manipulate their boxed counterparts, greatly reducing the code modifications required when introducing unboxing. It also overloads the literals, allowing us to write 1 instead of 1#, where an Int# is required.

### 4.2 Character I/O

In Haskell I/O is often a major performance bottleneck. One reason for this is that the I/O operations read and write strings i.e. [Char]. Given the ability to manipulate unboxed values directly, it would be nice to extend the I/O system to provide the ability to read and write strings containing unboxed characters i.e. [Char#], as well. One approach would be to introduce a parallel set of I/O operations, such as appendChan#, which read and write [Char#]. This would give the programmer the ability to choose unboxed I/O if desired.

Unfortunately these unboxed I/O operations must be used explicitly (since they require the **#** in the name). We are currently exploring an alternative approach which overloads the original I/O operations, enabling them to output lists of Char or lists of Char#.

### 4.3 Separate module compilation

In a language with separate module compilation type information flows from the defining module to the importing module. However, specialisation requires information about the use of a function to flow from the importing module back to the defining module. For example, consider the module structure:

```
module Tree (Tree(..), maptree) where
data Tree k a = Leaf k a | Branch k (Tree a) (Tree a)
maptree :: (a->b) -> Tree k a -> Tree k b
maptree f t = ...
module Use where
import Tree (Tree, maptree)
unbox_inttree :: Tree Int# Int -> Tree Int# Int#
unbox_inttree inttree = maptree int_to_int# inttree
```

In this example, module Use requires the maptree\_\*\_Int#\_Int# version of the imported function maptree. However, since Use imports Tree, Tree must be compiled before Use. When we compile Tree there is no requirement to create the maptree\_\*\_Int#\_Int# version of maptree since we have no information about module Use. When we subsequently compile Use we are faced with the problem that the required version of maptree has not been created.

One simple solution is to place this responsibility on the programmer: requiring them to request any specialised versions, which are not automatically generated, using pragmas. For example:

The SPECIALISE pragma is converted to the corresponding second-order type application: maptree {Int Int# Int#}. This is then processed by the partial evaluator and the specialised versions maptree\_\*\_Int#\_Int# produced. The specialised versions of the Tree data type: Tree\_Int#\_\* and Tree\_Int#\_Int#, will also be created.

The existence of all specialised versions created is recorded in the module's interface. If any specialised versions required by an importing module are not in the interface an error message is generated and the programmer has to add the appropriate specialise pragma to the declaring module and recompile. Unfortunately, the amount of programmer intervention and recompilation required is very unsatisfactory. To reduce these overheads, we plan to develop a scheme which automatically propagates the SPECIALISE pragmas back to the appropriate source modules and only recompiles once.

### 4.4 Introducing unboxing

In a lazy language, the programmer has to be careful when introducing unboxing, since an unboxed value is also strict. It is only safe to introduce unboxing where the implied strictness does not cause the program to bottom. This is normally not a problem, since the programmer is usually aware of the strictness implications.

We suggest that the programmer ensure that any intended unboxing is made explicit by introducing data type declarations with unboxed components or explicit type signatures for unboxing polymorphic data types. For example, the intention to unbox the list of prime numbers could be specified using the type signature:

#### primes :: [Int#]

After introducing this unboxing signature type errors may occur where the unboxed data structure is created and used.<sup>4</sup> In modifying the code to correct these type errors the programmer has to introduce explicit unboxing/boxing coercions at the "boundaries" of the unboxed values. We believe this is a "good thing", since the programmer is forced to identify these boundaries and consider the strictness implications. If the unboxing does cause the program to bottom the boundaries can be moved or the unboxing modifications abandoned.

It remains to be seen what the practical overheads of introducing this form of explicit unboxing are. However, we believe that when performance is an issue, and resources are allocated to improving  $it^5$ , it is essential that the programmer has access to language features, such as this, which enable them to optimise the execution.

### 5 Preliminary Results

We have not yet completed the implementation of the specialiser. However, we do have some preliminary results for programs in which we have introduced

 $<sup>^4</sup>$ These boundary type errors will not occur where the unboxed values are created/used by overloaded functions which have instances for the unboxed type (see Section 4.1).

<sup>&</sup>lt;sup>5</sup>We would also avocate that such improvements are carefully directed at the actual hotspots identified by an execution profiler [18].

Program	Brief Description	Unboxing Modifications		
clausify	converts logical formula to their	unbox the character symbols		
	clausal form [17,18]			
life	list based implementation of	unbox the integers used in the		
	Conway's Life algorithm [2]	board representation		
pseudoknot	floating point intensive molecu-	unbox all integers and floating		
	lar biology application [5]	point numbers		

	#lines	#lines modified/added				
Program	code	Unbox	Boundary	Specialise	Overload	Speedup
clausify	112	1	3	25	3	$1.25 \mathrm{x}$
life	75	1	1	42	9	1.06x
pseudoknot	3146	3	1	10	$2366^{6}$	$4.42 \mathrm{x}$

Figure 4: Preliminary results

some unboxing and performed the necessary specialisation by hand. These are summarised in Figure 4. The Unbox column reports the modifications required to introduce the unboxing while the Boundary column reports the modifications required to coerce data at the boundaries of the unboxing (see Section 4.4). The Specialise and Overload columns report the modifications which we expect to be automated (either by specialisation or as a result of extending the class operations to the unboxed types). The small number of changes required to introduce the unboxing is very encouraging.

### 6 Related Work

Other treatments of polymorphism in the presence of unboxed values fall into two categories. The first automatically introduces coercions which box/unbox values when they are passed to/from a polymorphic function [6,10,13,20]. The costs of creating and manipulating boxed values is only incurred when polymorphic code is used. This has the unfortunate consequence that it penalises the performance of polymorphic code, since unboxing is not possible.

The second approach compiles each polymorphic function in such a way that it can manipulate unboxed values. This is done by passing enough additional information at runtime to describe the representation of the values being manipulated [4,14]. This scheme also penalises the performance of polymorphic code since it must interpret the representation information. It has the unfortunate property that the performance penalty is paid even when the code is manipulating boxed values.

In contrast, our scheme generates specialised versions of polymorphic functions and data types which directly manipulate unboxed values. The performance of polymorphic code is not penalised since the polymorphism is removed precisely where it would impose a performance penalty. It also enables arbitrary data types to contain unboxed components. Traded off against this is the

 $<sup>^{6}</sup>$ The large number of modifications for pseudoknot were due to the amount of literal data (about 70% of the program) which had to by unboxed by added #s.

resulting code expansion and the difficulties associated with separate module compilation.

In a strict language, such as ML, both boxed and unboxed values have the same semantics. Consequently, the approaches to unboxing in strict languages focus on *automatically* unboxing values, because doing so is always possible when the type is known at compile time [4,6,10,14,20]. In a non-strict language, such as Haskell, unboxed values can only be introduced if we can be sure that the implied strictness will not change the behaviour of the program. Rather than relying on the often poor results of a strictness analyser, we ask the programmer to indicate where the unboxing is to be introduced. A similar approach is taken by Nocker & Smetsers [13]. They require the programmer to introduce explicit strictness annotations which they then use to safely unboxed values.

The use of partial evaluation to produce specialised code is not new. Hall [3] uses partial evaluation of special type arguments to create specialised versions which produce and consume an optimised list representation, while both Augustsson [1] and Jones [9] use partial evaluation to eliminate the overheads of overloading: creating versions which are specialised on their dictionary arguments.<sup>7</sup> Jones [9] also proposes an overloaded implementation of data types which cause the specialisation of overloading to specialises the data types as well. Unfortunately this requires a more powerful system of type classes.

### 7 Future Work

Our immediate goal is to complete the implementation of the specialiser, and develop a scheme for automatically propagating information about the required specialisations back to the declaring module. This should enable us to experiment with the unboxing of large programs by examining the practicalities of introducing the unboxing and the performance improvements which result.

We also plan to experiment with monomorphisation in general. By modifying the definition of **spectys** (Figure 3) the partial evaluator can be directed to introduce an arbitrary degree of monomorphisation. For example, if we define **spectys**  $\overline{\tau} = (\overline{\tau}, [])$  we get a completely monomorphic program. Our intention is to explore the practical benefits of optimisations which require monomorphic code to produce good results.

## Bibliography

- L Augustsson, "Implementing Haskell overloading," Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 1993, 65-73.
- [2] M Gardner, "Wheels, Life and Other Mathematical Amusements," W.H. Freeman and Company, New York, 1993.

<sup>&</sup>lt;sup>7</sup>Within our compiler we also use our partial evaluator to eliminate overloading by specialising on all overloaded type arguments, in addition to any unboxed type arguments. Care must be taken to ensure that the dictionary argument(s), introduced by the translation into the Core language, are also eliminated.

- [3] CV Hall, "Using Hindley-Milner type inference to optimise list representation," Conference on Lisp and Functional Programming, Orlando, Florida, June 1994.
- [4] R Harper & G Morrisett, "Compiling Polymorphism Using Intensional Type Analysis," Technical Report CMU-CS-94-185, School of Computer Science, Carnegie Mellon University, Sept 1994.
- [5] PH Hartel et al., "Pseudoknot: a float-intensive benchmark for functional compilers," in Proc Sixth International Workshop on the Implementation of Functional Languages, Norwich, JRW Glauert, ed., University of East Anglia, Norwich, Sept 1994.
- [6] F Henglein & J Jorgensen, "Formally optimal boxing," 21st ACM Symposium on Principles of Programming Languages, Portland, Oregon, Jan 1994, 213-226.
- [7] P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson, "Report on the functional programming language Haskell, Version 1.2," ACM SIGPLAN Notices 27 (5), May 1992.
- [8] John Hughes, "Why functional programming matters," The Computer Journal 32(2), April 1989.
- MP Jones, "Partial evaluation for dictionary-free overloading," Research Report YALE/DCS/RR-959, Dept of Computer Science, Yale University, April 1993.
- [10] X Leroy, "Unboxed objects and polymorphic typing," 19th ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, Jan 1992, 177-188.
- [11] JC Mitchell & R Harper, "On the type structure of Standard ML," ACM Transactions on Programming Languages and Systems 15(2), April 1993, 211-252.
- [12] R Morrison, A Dearle, RCH Conner & AL Brown, "An ad-hoc approach to the implementation of polymorphism," ACM Transactions on Programming Languages and Systems 13(3), July 1991, 342–371.
- [13] E Nocker & S Smetsers, "Partially strict non-recursive data types," Journal of Functional Programming 3(2), April 1993, 191–217.
- [14] A Ohori & T Takamizawa, "A polymorphic unboxed calculus and efficient compilation of ML," Research Institute for Mathematical Sciences, Kyoto University, Japan, 1994.
- [15] SL Peyton Jones, CV Hall, K Hammond, WD Partain & PL Wadler, "The Glasgow Haskell compiler: a technical overview," Joint Framework for Information Technology (JFIT) Technical Conference Digest, Keele, March 1993, 249-257.
- [16] SL Peyton Jones & J Launchbury, "Unboxed values as first class citizens," Conference on Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, Sept 1991.
- [17] C Runciman & D Wakeling, "Heap profiling of lazy functional programs," Journal of Functional Programming 3(2), April 1993, 217-245.

- [18] PM Sansom, "Execution profiling for non-strict functional languages," PhD thesis, Research Report FP-1994-09, Dept of Computing Science, University of Glasgow, Sept 1994.
- [19] PM Sansom & SL Peyton Jones, "Time and space profiling for non-strict, higher-order functional languages," 22nd ACM Symposium on Principles of Programming Languages, San Francisco, California, Jan 1995.
- [20] PJ Thiemann, "Unboxed values and polymorphic typing revisited," in Proc Sixth International Workshop on the Implementation of Functional Languages, Norwich, JRW Glauert, ed., University of East Anglia, Norwich, Sept 1994.