Unboxed values as first class citizens in a non-strict functional language*

Simon L Peyton Jones and John Launchbury
Department of Computing Science, University of Glasgow G12 8QQ
{simonpj, jl}@dcs.glasgow.ac.uk

July 26, 1991

Abstract

The code compiled from a non-strict functional program usually manipulates heap-allocated *boxed* numbers. Compilers for such languages often go to considerable trouble to optimise operations on boxed numbers into simpler operations on their unboxed forms. These optimisations are usually handled in an *ad hoc* manner in the code generator, because earlier phases of the compiler have no way to talk about unboxed values.

We present a new approach, which makes unboxed values into (nearly) first-class citizens. The language, including its type system, is extended to handle unboxed values. The optimisation of boxing and unboxing operations can now be reinterpreted as a set of correctness-preserving program transformations. Indeed the particular transformations required are ones which a compiler would want to implement anyway. The compiler becomes both simpler and more modular.

Two other benefits accrue. Firstly, the results of strictness analysis can be exploited within the same uniform transformational framework. Secondly, new algebraic data types with unboxed components can be declared. Values of these types can be manipulated much more efficiently than the corresponding boxed versions.

Both a static and a dynamic semantics are given for the augmented language. The denotational dynamic semantics is notable for its use of *unpointed domains*.

1 Introduction

Most compilers have a phase during which they attempt to optimise the program by applying correctness-preserving transformations to it. Constant folding is a particular example of this sort of transformation; procedure inlining is another.

Functional languages are especially amenable to such transformation because of their simple semantics. Non-strict functional languages are nicest of all, because β-reduction (sometimes called unfolding in a transformational context) is always valid; in a strict language it is only valid if the body of the function being unfolded is guaranteed to evaluate the argument.

*This paper appears in the Proceedings of the 1991 Conference on Functional Programming Languages and Computer Architecture, Cambridge, Sept 1991.

Several researchers have begun to express more and more of the work of the compiler in the form of correctness-preserving transformations (see Section 11). Such an approach has obvious advantages. Firstly, each transformation can be proved to be correct independent of the others. When more of the compiler takes the form of such transformations, it is easier to prove the compiler correct. Secondly, each transformation exposes opportunities for other transformations. The more that is done within a transformation phase, the more chance there is for such beneficial interactions.

There is an important class of optimisations for non-strict languages which has so far been beyond the scope of program transformation. These all relate to the treatment of so-called *unboxed values*, which we introduce in the next section. Instead, this family of optimisations is generally implemented in an *ad hoc* manner in the code generator.

Following some preliminaries (Sections 2 and 3), this paper makes four main contributions:

- Most important, we show how the class of unboxed-value optimisations can formulated as correctness-preserving transformations (Section 4). Unboxed values are made first-class citizens, distinguished from their boxed counterparts by the type system. These transformations do not generate much better code than current compilers do; our intention is only to present the optimisations in a new and elegant way.
- We show how to apply the same idea to express and exploit the results of strictness analysis in a uniform way (Section 5).
- We show how the approach can be generalised to other algebraic data types (Section 6). Declaring and using such unboxed data types gives a substantial performance improvement.
- We provide a formal underpinning for the approach, by giving both a static semantics and a denotational dynamic semantics for the language extended with unboxed values (Sections 7, 8, and 9). The dynamic semantics has the desirable property that the semantic equations are unchanged from those for the language without unboxed values, despite the extra strictness of the extended language. This effect is achieved by using *unpointed domains*.

We conclude by discussing some language-design issues, reviewing related work, and mentioning some areas for further work.

2 The problem

Consider the following function definition:

```
double x = x + x
```

In a non-strict language, `x` may be unevaluated when `double` is called, in which case a pointer to a heap-allocated *closure* (or suspension) for `x` is passed to `double`. When `double` needs `x`'s value (in this case right away), it *evaluates* the closure. As a side effect of this evaluation, the closure of `x` is overwritten with its value. Any further attempts to evaluate `x` will succeed immediately, returning its value. This way of ensuring that unevaluated closures are evaluated at most once is called *lazy evaluation*.

Unfortunately, lazy evaluation tends to make arithmetic horribly inefficient if some care is not taken. In particular, in a naïve implementation numbers are always represented by a pointer to a heap-allocated object which is either an unevaluated closure, or is a "box" containing the number's actual value, which has now overwritten the closure. This means that a simple arithmetic operation, which would take a single machine instruction in a strict language, requires quite a long sequence of instructions: the two operands are fetched from their boxes, the operation performed, a new box allocated to contain the result, and the result placed in it.

The bit-pattern representing the value itself, on which the built-in machine instructions operate, is called an *unboxed value*. Unboxed values come in a variety of shapes and sizes: 32-bit integers, 64-bit integers, single and double-precision floating point numbers, and so on, are all unboxed values. A pointer to a heap-allocated box containing an unboxed value is called a *boxed value*. Clearly it is vastly more efficient to manipulate unboxed values than boxed ones.

Quite a lot can be done. For example, consider again the definition of `double` given above. A naïve compiler would compile code for `double` which would evaluate `x`, extract its unboxed value, then evaluate `x` again and extract the value again, then add the two, and box the result. A slightly cleverer compiler would realise that it already had `x`'s value in hand, and refrain from the second evaluation. As another example, consider the following definition:

```
f x y z = x + (y * z)
```

A naïve compiler might generate code to box the value of `y*z`, only to unbox it again right away. A cleverer compiler can elide these unnecessary operations.

As a final example, consider the following call to `double`:

```
double (p+q)
```

The straightforward approach is to build a closure for `p+q` and pass it to `double` which will evaluate it. But `double` is clearly going to evaluate its argument so it is a waste to allocate the closure, only for `double` to evaluate it, and discard it for the garbage collector to recover later. It would be much better for the caller to evaluate `p` and `q`, add their values, and pass `p+q` to `double` in unboxed form.

3 The Core language

We begin with a few preliminaries, to set the scene for our new proposals.

Our compilation route involves the following steps:

1. The source language is HASKELL (Hudak et al. [1990]), a strongly-typed, non-strict, purely-functional language. HASKELL's main innovative feature is its support for systematic overloading, but we do not discuss this aspect at all here.

2. HASKELL is compiled to the *Core language*. All HASKELL's syntactic sugar has been compiled out, type checking performed, and overloading resolved. Pattern-matching has been compiled into case expressions, each of which performs only a single level of matching. Boxing and unboxing are explicit, as described below.

3. Program analyses and transformations are applied to the Core language. In particular, the boxing/unboxing optimisations are carried out here.

4. The Core language is translated to the STG language, the abstract machine code for the Spineless Tagless G-machine, our evaluation model. The Spineless Tagless G-machine was initially presented in Peyton Jones & Salkild [1989], but the latter has been completely rewritten as a companion to this paper (Peyton Jones [1991]).

5. The STG language is translated to "Abstract C". This is just an internal data type which can be simply printed out as C code and thence compiled to native code with standard C compilers. Abstract C can also serve serve as an input to a code generator, thereby generating native code directly, but we have not yet implemented this route.

In this paper we only concern ourselves with the Core language, which is introduced in the next section. However, since the Core language does not have explicit algebraic type declarations, we borrow HASKELL's syntax for this purpose when required.

3.1 The syntax of the Core language

The abstract syntax of the Core language is given in Figure 1. The *binds* constituting a *prog* should define `main`, which is taken to be the value of the program.

The concrete syntax we use is conventional: parentheses are used to disambiguate; application associates to the left and binds more tightly than any other operator; the body of a lambda abstraction extends as far to the right as possible; the usual infix arithmetic operators are permitted; the usual syntax for lists is allowed, with infix constructor "`:`" and empty list `[]`; and, where the layout makes the meaning clear, we allow ourselves to omit semicolons between bindings and case alternatives.

Notice that the bindings in `let(rec)` expressions are all simple; that is, the left hand side of the binding is always just a variable. Function bindings are expressed by binding a variable to a lambda abstraction. However, we permit ourselves the small liberty in the concrete syntax of writing the arguments of function bindings to the left of the `=` sign.

Similarly, the patterns in `case` expressions are all simple; nested pattern matching has been compiled to nested `case` expressions.

Program	*prog*	$\rightarrow$	*binds*	
Bindings	*binds*	$\rightarrow$	$bind_1$; ... ; $bind_n$	$n \geq 1$
	bind	$\rightarrow$	*var* `=` *expr*	
Expression	*expr*	$\rightarrow$	$expr_1$ $expr_2$	Application
		|	`\` *var* `->` *expr*	Lambda abstraction
		|	`case` *expr* `of` *alts*	Case expression
		|	`let` *bind* `in` *expr*	Local definition
		|	`letrec` *binds* `in` *expr*	Local recursion
		|	*con*	Constructor
		|	*var*	Variable
		|	*literal*	
Literal values	*literal*	$\rightarrow$	*integer*	
		|	*float*	
Alternatives	*alts*	$\rightarrow$	$calt_1$; ... ; $calt_n$; `default ->` *expr*	$n \geq 0$
		|	$lalt_1$; ... ; $lalt_n$; *var* `->` *expr*	$n \geq 0$
Constructor alt	*calt*	$\rightarrow$	*con* $var_1 \ldots var_n$ `->` *expr*	$n \geq 0$
Literal alt	*lalt*	$\rightarrow$	*literal* `->` *expr*	$n \geq 0$

Figure 1: Syntax of the Core language

Here is an example program to illustrate these points:

```
fac n = case n of
          0  -> 1
          n' -> n * factorial (n-1)

main = fac 100
```

3.2 A semantics for the Core language

In this section we present a denotational semantics for well-typed Core language programs. We do so with a little more care than usual, because we want it to form a basis for the denotational semantics of unboxed types later. We need to do two things:

- Give a *model*, which defines a domain $\mathcal{D}[\![\tau]\!]$ for each type τ in the language.
- Give a *valuation function* $\mathcal{E}[\![e]\!]$ which, for every expression e of type τ, gives its value in the domain $\mathcal{D}[\![\tau]\!]$.

3.3 The model

Beginning with the model, we give the syntax of types τ:

$$\begin{array}{rcll} \tau & ::= & \alpha & \\ & | & \texttt{Int} & \\ & | & \texttt{Float} & \\ & | & \tau_1 \rightarrow \tau_2 & \\ & | & \chi_n\ \tau_1 \ldots \tau_n & (n \geq 0) \end{array}$$

Here, α is a type variable, and χ_n ranges over type constructors of arity n. These type constructors arise from algebraic data type declarations made by the programmer. For example, the declaration

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

introduces a type constructor `Tree` with arity 1. Lists, booleans, pairs, and other types which usually come "built in" are all regarded as examples of such algebraic data types.

We define the domain corresponding to each type τ inductively, thus:

$$\begin{array}{lcl} \mathcal{D}[\![]\!] & : & \mathit{Monotype} \rightarrow (\mathit{Typevar} \rightarrow \mathbf{Dom}) \rightarrow \mathbf{Dom} \\ \mathcal{D}[\![\alpha]\!]\ \rho & = & \rho\ \alpha \\ \mathcal{D}[\![\texttt{Int}]\!]\ \rho & = & \{\text{The set of fixed-precision integers}\}_\perp \\ \mathcal{D}[\![\texttt{Float}]\!]\ \rho & = & \{\text{The set of fixed-precision floating-point numbers}\}_\perp \\ \mathcal{D}[\![\tau_1 \rightarrow \tau_2]\!]\ \rho & = & (\mathcal{D}[\![\tau_1]\!]\ \rho) \rightarrow (\mathcal{D}[\![\tau_2]\!]\ \rho) \\ \mathcal{D}[\![\chi_n\ \tau_1 \ldots \tau_n]\!]\ \rho & = & \chi_n\ (\mathcal{D}[\![\tau_1]\!]\ \rho) \ldots (\mathcal{D}[\![\tau_n]\!]\ \rho) \end{array}$$

The environment ρ maps type variables to domains; where the type τ has no free variables we write simply $\mathcal{D}[\![\tau]\!]$. Notice that the arrow on the left hand side of the fourth equation is part of the syntax of types, whereas on the right hand side it stands for the function space constructor (or functor) for domains. In just the same way, the χ_n stands for a domain constructor; so we must obviously say just how χ_n is defined for any given algebraic data type declaration. The general form of an algebraic data type declaration is as follows:

$$\texttt{data}\ \chi\ \alpha_1 \ldots \alpha_t\ =\ c_1\ \tau_{11} \ldots \tau_{1a_1}\ |\ \ldots\ |\ c_n\ \tau_{n1} \ldots \tau_{na_n}$$

where $t \geq 0$, $n > 0$ and $a_i \geq 0$. Corresponding to this declaration we give the following functor definition:

$$\begin{array}{lcl} \chi\ d_1 \ldots d_t & = & (s_1 + \ldots + s_n)_\perp \\ & & \begin{array}{rcll} \text{where } \rho & = & [\alpha_1 \mapsto d_1,\ \ldots, \alpha_n \mapsto d_n] & \\ s_i & = & \mathcal{D}[\![\tau_{i1}]\!]\ \rho \times \ldots \times \mathcal{D}[\![\tau_{ia_i}]\!]\ \rho & (1 \leq i \leq n) \end{array} \end{array}$$

The domain constructions are categorical product ($\times$) and sum ($+$), and lifting ($\cdot_\perp$); they are defined in Figure 2. This equation is more usually given using separated sum instead of categorical sum, and omitting the lifting, but the result is the same in either case, as is easily verified. The reason we choose this formulation is that it extends smoothly when we add unboxed types.

Lifting	$A_\perp$	$=$	$\{\perp\} \cup \{lift\ a \mid a \in A\}$
Categorical sum	$A + B$	$=$	$\{\langle 0, a\rangle \mid a \in A\} \cup \{\langle 1, b\rangle \mid b \in B\}$
Categorical product	$A \times B$	$=$	$\{\langle a, b\rangle \mid a \in A, b \in B\}$

Figure 2: Definitions of domain constructions

This construction gives rise to a mutually recursive set of functor definitions. These can be solved in the usual way to define a domain $\mathcal{D}[\![\tau]\!]$ for each type τ with no free variables. (See Smyth & Plotkin [1982] for a categorical account, or Schmidt [1986] for a more element-orientated treatment. The latter is particularly useful as it discusses unpointed (i.e. bottomless) domains, an aspect important later on.)

3.4 The semantic equations

That completes the description of the domains involved, so it remains only to give the definitions of the valuation functions. Figure 3, which gives these definitions, contains no surprises. The valuation function $\mathcal{P}[\![]\!]$ gives the meaning of programs, $\mathcal{E}[\![]\!]$ give the meaning of expressions, $\mathcal{B}[\![]\!]$ gives the meaning of groups of definitions, and $\mathcal{K}[\![]\!]$ (which is not further defined) give the meaning of literal constants.

The valuation function $\mathcal{E}[\![]\!]$ takes an *expression* and an *environment* and returns a *value*. We use **Env** for the domain of environments, and **Val** for the domain of values, defining them like this:

$$\begin{array}{lcl} \mathbf{Env} & = & \bigcup_\tau (var_\tau \to \mathcal{D}[\![\tau]\!]) \\ \mathbf{Val} & = & \bigcup_\tau \mathcal{D}[\![\tau]\!] \end{array}$$

The environment maps a variable of type τ to a value in the domain $\mathcal{D}[\![\tau]\!]$, and the domain of values is the union of all the $\mathcal{D}[\![\tau]\!]$. In these two equations τ ranges only over types with no free type variables.

We need a notation for writing values in the domains corresponding to algebraic data types. Since only binary sum and product were defined, such values should formally consist of nested pairs, but these are rather tiresome to write. Instead we will permit ourselves the liberty of writing such values in the flattened form:

$$\langle c, \epsilon_1, \ldots, \epsilon_n\rangle$$

where c is a constructor of arity n, and ϵ_i are values from the appropriate argument domains.

The initial environment ρ_{init} contains bindings for all the built-in functions. For example, it

$$
\begin{array}{l}
\mathcal{P}[\![\mathit{program}]\!] : \mathbf{Val} \\
\mathcal{P}[\![\mathit{prog}]\!] \;=\; \mathcal{E}[\![\texttt{letrec}\ \mathit{prog}\ \texttt{in main}]\!]\ \rho_{init}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}[\![\mathit{expr}]\!] : \mathbf{Env} \rightarrow \mathbf{Val} & & \\
\mathcal{E}[\![k]\!]\ \rho & = & \mathcal{K}[\![k]\!] \\
\mathcal{E}[\![x]\!]\ \rho & = & \rho\ x \\
\mathcal{E}[\![e_1\ e_2]\!] & = & (\mathcal{E}[\![e_1]\!]\ \rho)\ (\mathcal{E}[\![e_2]\!]\ \rho) \\
\mathcal{E}[\![\texttt{\\}x\texttt{->}e]\!]\ \rho & = & \lambda\epsilon.(\mathcal{E}[\![e]\!]\ (\rho \oplus \{x \mapsto \epsilon\})) \\
\mathcal{E}[\![\texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ b]\!]\ \rho & = & \mathcal{E}[\![b]\!]\ (\rho \oplus \{x \mapsto \mathcal{E}[\![e]\!]\ \rho\}) \\
\mathcal{E}[\![\texttt{letrec}\ \mathit{binds}\ \texttt{in}\ e]\!]\ \rho & = & \mathcal{E}[\![e]\!]\ (\rho \oplus \mathit{fix}\ (\lambda\rho'.\mathcal{B}[\![\mathit{binds}]\!]\ (\rho \oplus \rho'))) \\
\mathcal{E}[\![c]\!]\ \rho & = & \lambda\epsilon_1.\ldots.\lambda\epsilon_a.\langle c, \epsilon_1, \ldots, \epsilon_a\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ c_1\ x_{11}\ldots x_{1a_1}\ \texttt{->}\ e_1\texttt{;}\ \ldots\texttt{;}\ c_n\ x_{n1}\ldots x_{na_n}\ \texttt{->}\ e_n\texttt{;}\ \texttt{default ->}\ e_d]\!]\ \rho \\
\quad = \mathbf{case}\ E[\![e]\!]\ \rho\ \mathbf{of} \\
\qquad \begin{array}{lll}
[\,] & \bot & \rightarrow \bot \\
[\,] & \langle c_1, \epsilon_{11}, \ldots, \epsilon_{1a_1}\rangle & \rightarrow \mathcal{E}[\![e_1]\!]\ (\rho \oplus \{x_{11} \mapsto \epsilon_{11}, \ldots, x_{1a_1} \mapsto \epsilon_{1a_1}\}) \\
[\,] & \ldots & \\
[\,] & \langle c_n, \epsilon_{n1}, \ldots, \epsilon_{na_n}\rangle & \rightarrow \mathcal{E}[\![e_n]\!]\ (\rho \oplus \{x_{n1} \mapsto \epsilon_{n1}, \ldots, x_{na_n} \mapsto \epsilon_{na_n}\}) \\
[\,] & \mathit{else} & \rightarrow \mathcal{E}[\![e_d]\!]\ \rho
\end{array} \\
\qquad \mathbf{end} \\
\mathcal{E}[\![\texttt{case}\ e\ \texttt{of}\ k_1\ \texttt{->}\ e_1\texttt{;}\ \ldots\texttt{;}\ k_n\ \texttt{->}\ e_n\texttt{;}\ x\ \texttt{->}\ e_d]\!]\ \rho \\
\quad = \mathbf{case}\ E[\![e]\!]\ \rho\ \mathbf{of} \\
\qquad \begin{array}{lll}
[\,] & \bot & \rightarrow \bot \\
[\,] & \mathit{lift}\ k_1 & \rightarrow \mathcal{E}[\![e_1]\!]\ \rho \\
[\,] & \ldots & \\
[\,] & \mathit{lift}\ k_n & \rightarrow \mathcal{E}[\![e_n]\!]\ \rho \\
[\,] & \mathit{lift}\ \epsilon & \rightarrow \mathcal{E}[\![e_d]\!]\ (\rho \oplus \{x \mapsto \epsilon\})
\end{array} \\
\qquad \mathbf{end}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{B}[\![\mathit{binds}]\!] : \mathbf{Env} \rightarrow \mathbf{Env} \\
\mathcal{B}[\![x_1\texttt{=}e_1\texttt{;}\ \ldots\texttt{;}\ x_n\texttt{=}e_n]\!]\ \rho \;=\; \{x_1 \mapsto \mathcal{E}[\![e_1]\!]\ \rho, \ldots, x_n \mapsto \mathcal{E}[\![e_n]\!]\ \rho\}
\end{array}
$$

Figure 3: Denotational semantics of the Core language

contains the following binding for the addition function:

$$
\begin{array}{l}
\texttt{+} \mapsto \lambda x.\lambda y.\textbf{case}\ x\ \textbf{of} \\
\quad\quad [\!]\ \ \bot \quad\quad \to \bot \\
\quad\quad [\!]\ \ \mathit{lift}\ x' \to \textbf{case}\ y\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad [\!]\ \ \bot \quad\quad \to \bot \\
\quad\quad\quad\quad\quad\quad\quad [\!]\ \ \mathit{lift}\ y' \to \mathit{lift}(x' + y') \\
\quad\quad\quad\quad\quad\quad\quad \textbf{end} \\
\quad\quad \textbf{end}
\end{array}
$$

This definition also illustrates the semantic **case** construct which we use to discriminate among elements of a domain.

4 The main idea: exposing unboxed types to transformation

We are now ready to present the key idea of the paper. The big problem with earlier approaches to the boxing issue is this: *there is no way to talk about unboxed values in the Core language.* An immediate consequence is that *evaluation of numbers is implicit.* For example, in the definition:

```
f x y = y - x
```

`x` and `y` must be evaluated before they can be subtracted, but this fact is implicit, as is the *order* in which `x` and `y` are evaluated.

This implicit evaluation is in contrast with the situation for algebraic data types. For example, the `length` function might be defined like this:

```
length xs = case xs of
            x : xs -> 1 + length xs
            []     -> 0
```

Here, the evaluation of `xs` is completely explicit. In general, it is precisely `case` expressions (and nothing else) which perform evaluation of data structures.

This suggests an obvious improvement: perhaps the evaluation of numbers can be done by `case` expressions as well. Suppose we wrote the definition of `f` like this:

```
f x y = case y of
        Int y# -> case x of
                  Int x# -> case (y# -# x#) of
                            t# -> Int t#
```

The idea is that the data type `Int`, of fixed-precision integers, is no longer primitive. Instead, we can imagine `Int` being declared in HASKELL like this:

```
data Int = Int Int#
```

That is, the `Int` type is an algebraic data type like any other, with a single constructor `Int`. (Here and elsewhere we will use the same name for the type and its constructor.) The `Int` constructor has one component, of type `Int#`, which is the primitive type of *unboxed* fixed-precision integers. A new primitive operator, `-#`, subtracts values of type `Int#`.

The outermost `case` expression in our new formulation of `f` serves to evaluate `y` and extract its unboxed component `y#`. The next `case` expression performs a similar function for `x`, giving `x#`. Next, the difference between `y#` and `x#` is computed and bound to `t#`, and finally, the function returns a value of type `Int`, obtained by applying the `Int` constructor to `t#`.

At first sight, the innermost `case` expression is rather curious. Why not just write the following instead?

```
Int (y# -# x#)
```

The reason we chose to use `case` for this purpose is to make explicit that the subtraction is performed before the result of the function is returned.

In general we will use a `#` sign to identify unboxed types, and to identify variables whose type is unboxed. This only serves to make the presentation clearer; the `#` characters are treated by the compiler in the same way as any other alphabetic character, as part of a name.

We note in passing that the transformation from the first form of `f` to the second can be carried out in a systematic way, merely by giving the following definition to the subtraction operator `-`, which was previously considered primitive:

```
(-) p q = case p of
          Int p# -> case q of
                    Int q# -> case (p# -# q#) of
                              t# -> Int t#
```

Now the passage from one version of `f` to the other is just a matter of β-reduction, unfolding the application of `-` to its two arguments.

Now that `Int` has been expressed in terms of a more primitive type, its constructors, the integers `0`, `1`, and so on, must be regarded as short for `Int 0#`, `Int 1#`, and so on, where `0#` and `1#` are the unboxed constants for zero and one. Similarly, pattern matching against integers becomes a two-stage process. The function

```
f 1 = e1
f 2 = e2
f n = en
```

is shorthand for

```
f (Int 1#) = e1
f (Int 2#) = e2
f n        = en
```

The latter will get translated by the pattern-matching compiler to

```
f n = case n of Int n# -> case n# of
                          1# -> e1
                          2# -> e2
                          default -> en
```

This has a straightforward operational interpretation. The outer `case` evaluates `n` and extracts its unboxed contents, `n#`. The inner `case` scrutinises `n#` and selects the appropriate alternative.

Now that we can express programs involving unboxed values, we can demonstrate how the optimisations mentioned in Section 2 can be re-interpreted as program transformations.

4.1 Avoiding repeated evaluation

Consider the expression:

```
x+x
```

A naïve implementation would evaluate and unbox `x` twice. Let us see what the expression looks like when we unfold the application of `+`, just as we unfolded `-` in the previous section. It becomes this:

```
case x of
Int x1# -> case x of
           Int x2# -> case (x1# +# x2#) of
                      t# -> Int t#
```

Now, it is a simple observation that the inner `case` is scrutinising the same value as the outer case. In general, the following transformation holds:

$$\begin{array}{l}\texttt{case}\ e\ \texttt{of}\ \ldots\texttt{;}\ c\ x_1 \ldots x_n\ \texttt{->}\ \ldots \texttt{case}\ e\ \texttt{of} \ldots c\ y_1 \ldots y_n\ \texttt{->}\ \mathit{body} \ldots\texttt{;}\ \ldots \\ \Longrightarrow \\ \texttt{case}\ e\ \texttt{of}\ \ldots\texttt{;}\ c\ x_1 \ldots x_n\ \texttt{->}\ \ldots \mathit{body}[x_1/y_1 \ldots x_n/y_n] \ldots\texttt{;}\ \ldots\end{array}$$

(There is a side condition: we assume that every binding site binds a distinct variable, so that the two occurrences of the expression e denote the same value, and so that *body* does not re-bind the x_i.) Applying this transformation to the expression we are studying, we get:

```
case x of
Int x1# -> case (x1# +# x1#) of
           t# -> Int t#
```

which expresses precisely the optimisation we were seeking.

4.2 Eliding redundant boxing operations

As a second example, consider the expression

```
x + (y * z)
```

As mentioned earlier, we want to avoid wrapping a box around the value of `y*z`, because it will immediately be unwrapped by the enclosing addition. As before, let us see what the expression looks like when the applications of `+` and `*` are unfolded. It becomes this:

```
case x of
Int x# -> case ( case y of
                  Int y# -> case z of
                            Int z# -> case (y# *# z#) of
                                      t1# -> Int t1#
                ) of
           Int w# -> case (x# +# w#) of t2# -> Int t2#
```

This expression does not look very promising, but it yields to the following well-known transformation:

$$\texttt{case (case } e \texttt{ of } p_1 \texttt{->} e_1 \texttt{;} \ldots \texttt{;} p_n \texttt{->} e_n \texttt{) of } alts$$
$$\Longrightarrow$$
$$\texttt{case } e \texttt{ of } p_1 \texttt{-> (case } e_1 \texttt{ of } alts\texttt{);} \ldots \texttt{;} p_n \texttt{-> (case } e_n \texttt{ of } alts\texttt{)}$$

(The same unique-binding side condition is necessary here too, to ensure that the meaning of *alts* is not changed by being moved into the scope of the patterns p_i.) Where the expression scrutinised by a `case` is itself a `case` expression, the `case`s can be interchanged, as shown. We call this the *case-of-case transformation*, for obvious reasons.

In general, there is a danger of duplicating code, because if the inner `case` has multiple alternatives, *alts* will be duplicated. Happily, in the expression we are transforming, the inner `case` has just one alternative. We can apply the transformation three times to give:

```
case x of
Int x# -> case y of
          Int y# -> case z of
                    Int z# -> case (y# *# z#) of
                              t1# -> case (Int t1#) of
                                     Int w# -> case (x# +# w#) of
                                               t2# -> Int t2#
```

Now another generally-useful transformation applies:

$$\texttt{case } (c\ x_1 \ldots x_n) \texttt{ of } \ldots\texttt{;}\ c\ y_1 \ldots y_n \texttt{ -> } e\ \texttt{;} \ldots$$
$$\Longrightarrow$$
$$e[x_1/y_1 \ldots x_n/y_n]$$

That is, a `case` expression which scrutinises a constructor applied to some variables, can be replaced by the appropriate alternative after suitable renaming. (In general, a constructor which is applied to arbitrary expressions can be transformed to one applied to variables by introducing some `let`-bindings.) Using this transformation on the inner `case` expression gives:

```
case x of
Int x# -> case y of
          Int y# -> case z of
                    Int z# -> case (y# *# z#) of
                              t1# -> case (x# +# t1#) of
                                     t2# -> Int t2#
```

This final form refrains from boxing `t1#` and then taking it apart, just as we hoped. Instead, the result of the multiplication is used directly in the addition.

5 Strictness analysis and unboxed calls

Much effort has been devoted in the literature to *strictness analysis*, which detects whether or not a function is strict (Hankin & Abramsky [1986]). For a sequential implementation the significance is that strict arguments can be evaluated before the call. In particular, strict numeric arguments can be passed unboxed. In the following section we show how the results of strictness analysis can be exploited within our transformational framework.

5.1 Exploiting the results of strictness analysis

As with other aspects of boxing and unboxing, the exploitation of strictness analysis is usually left to the hapless code generator. Let us see how it can be expressed in our extended language. Consider the factorial function with an accumulating parameter, which in HASKELL might look like this:

```
afac a 0 = a
afac a n = afac (n*a) (n-1)
```

Translated into the Core language, it would take the following form:

```
afac a n = case n of
           Int n# -> case n# of
                     0#  -> a
                     n#' -> afac (n*a) (n-(Int 1#))
```

Integer constants have been replaced by the `Int` constructor applied to the corresponding unboxed constant, and pattern-matching have been translated into `case` expressions.

Suppose that a strictness analyser informs us that afac is strict in both its arguments. Then without (as yet) doing anything further to its body, we can transform it into the following pair of definitions:

```
afac a n = case a of Int a# -> case n of Int n# -> afac# a# n#

afac# a# n# = let n = Int n#
                  a = Int a#
              in
              case n of
              Int n# -> case n# of
                        0#  -> a
                        n#' -> afac (n*a) (n-(Int 1#))
```

The "wrapper", `afac`, evaluates the arguments and passes them unboxed to the "worker", `afac#`. (Recall that the `#` annotations are merely present as cues to the human reader; the compiler treats `#` as part of a name, like any other letter.) The latter consists of a `let`-expression, whose bindings reconstruct the original arguments, and whose body is the unmodified body of the original `afac`.

Now we can go to work on the body of `afac#`. We unfold the definitions of `*`, `-`, and `afac` itself; and apply the transformations described earlier. One further related transformation is needed:

`let` x `=` $c\ x_1 \ldots x_n$ `in` $\ldots$(`case` x `of` $\ldots$; $c\ y_1 \ldots y_n$ `->` *body*; $\ldots$) $\ldots$
$\Longrightarrow$
`let` x `=` $c\ x_1 \ldots x_n$ `in` $\ldots(body[x_1/y_1 \ldots x_n/y_n]) \ldots$

This transformation is applied to the auxiliary `let` bindings for `a` and `n`, after which no uses of `a` and `n` remain, so the `let` bindings for them can be dropped. A few moments work should convince you that the result is this:

```
afac# a# n# = case n# of
              0#  -> Int a#
              n'# -> case (n# *# a#) of
                     a1# -> case (n# -# 1#) of
                            n1# -> afac# a1# n1#
```

Bingo! `afac#` is just what we hoped for: a strict, constant-space, efficient factorial function. Even the recursive call is made directly to `afac#`, rather than going via `afac`. Meanwhile, `afac` acts as an "impedance-matcher" to provide a boxed interface to `afac#`.

Now, much of the benefit of strictness comes from the fact that often the arguments are already partly or completely evaluated before the call. For example, consider the following call to `afac`:

```
if (x>10) then (afac 1 x) else 0
```

Here, `x` is already evaluated before we reach the call to `afac`. All we need to do to take advantage of this is to *always unfold calls to* `afac` (though not `afac#`!). This unfolding exposes the two evaluations which `afac` does. The transformations already discussed can then eliminate both of them, leaving a direct call to `afac#`.

To summarise, the results of strictness analysis can be uniformly incorporated into the transformation process by applying the following procedure to each strict function:

- Split the function into two: a *wrapper function* which evaluates the strict arguments and passes them to the *work function*. The work function uses let-bindings to reconstitute the original arguments, but is otherwise identical to the original function.
- Unfold all applications of the wrapper function wherever possible.
- Optimise using the transformations described earlier.

5.2 Strictness over non-numeric types

In fact, this is not quite the whole story. Firstly, a function can be strict in an argument whose type is a list. It is far from clear what the wrapper function should do in this case. Nor is it clear what the wrapper should do in the case of a function strict in a functional argument.

The most obvious cases where something useful can be done are these: *when the strict argument is of a data type which has just one constructor.* Then the wrapper can evaluate the object, extract its components, and pass them to the work function. (Actually, it is only necessary to pass the free variables of the work function's body to the work function. Components which are not used can be discarded by the wrapper.)

To take an example where the constructor has more than one component, consider the function

```
f t = case t of (x,y) -> (...)
```

It is clearly strict in its argument, which is a pair. Therefore we split it into two, thus:

```
f t = case t of (x,y) -> f# x y
f# x y  = let t = (x,y) in (...)
```

Now a call to `f` in which an explicit pair is given, thus

```
f (e1,e2)
```

will benefit from unfolding the wrapper for `f` and simplifying, giving

```
f# e1 e2
```

The example stands revealed as the standard currying transformation, another example of an *ad hoc* transformation appearing as a special case.

5.3 Unboxing results

The second addition to the strictness story can be seen by attempting the same transformation on the non-accumulating factorial function. Here is the original definition:

```
fac n = case n of
        Int n# -> case n# of
                    0#  -> Int 1#
                    n#' -> n * fac (n-(Int 1#))
```

Again, because `fac` is strict, we may split it into a wrapper and a worker, thus:

```
fac n = case n of Int n# -> fac# n#

fac# n# = case n# of
            0#  -> Int 1#
            n#' -> case (n# -# 1#) of
                     n1# -> case (fac# n1#) of
                              Int m# -> case (n# *# m#) of
                                          r# -> Int r#
```

The point is that `fac#` has type `Int#` $\rightarrow$ `Int`, not `Int#` $\rightarrow$ `Int#`. Hence `fac#` contains code for taking apart the boxed integer returned by the recursive call to `fac#` (the next to innermost `case` does this).

It looks as if a new boxed `Int` is constructed in the heap for each recursive call, and this will indeed be the case for many implementations[1].

Can we avoid this problem? Observing that the result of a function call is always required (or else the call would not have been made), we can modify the way in which the split into wrapper and work functions is made. If the result of the original function is a single-constructor type (as in this case), we can split like this:

```
fac n = case n of Int n# -> case (fac1# n#) of t# -> Int t#

fac1# n# = let n = Int n#
           in
           case (...original body of fac...) of
           Int r# -> r#
```

Now `fac1#` has type `Int#` $\rightarrow$ `Int#`, and transformation turns it into

```
fac1# n# = case n# of
```

[1] It happens not to be the case for the Spineless Tagless G-machine, which returns even apparently boxed integers (among other things) in a register. The reason for this is that the only reason a boxed integer is ever evaluated is to unbox it. Even so, returning an unboxed value is still slightly more efficient than returning a boxed one.

```
0#  -> 1#
n#' -> case (n# -# 1#) of
       n1# -> case (fac1# n1#) of m# -> n# *# m#
```

To conclude, the point of all this is not so much that any one alternative is obviously much better than any other, but that rather the new expressiveness offered by a language featuring unboxed types allows us to discuss and explore a wide range of options within a single uniform framework.

6 Generalising unboxed types

The presentation of Section 4 gave a definition for the `Int` type as an algebraic data type based on a primitive unboxed type, `Int#`. This is rather suggestive: can we generalise the idea to all algebraic data types, so that `Int` is not special, but rather a particular case of a general feature?

It turns out that we can indeed do so in two distinct ways:

- We can allow any constructor (and not just `Int`) to take arguments of unboxed type.
- We can allow any algebraic data type (and not just `Int#`) to be unboxed.

6.1 Constructors with unboxed components

Consider the following (conventional) definition of a data type for complex numbers and a corresponding addition function:

```
data Cpx = Cpx Int Int  -- Real and imaginary parts

addCpx (Cpx r1 i1) (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
```

But, since all constructors are non-strict, what `addCpx` will do is to build a `Cpx` box containing pointers to an unevaluated closure for `r1+r2` and another for `i1+i2`. Now this may be exactly what the programmer wanted — it allows values such as `Cpx` $\perp$ $\perp$, for example — but the implementation cost is heavy compared with a strict language which would simply build a `Cpx` box containing the *values of* `r1+r2` and `i1+i2`.

The discussion of Section 4 suggests the following alternative:

```
data UCpx = UCpx Int# Int#

addUCpx (UCpx i1 r1) (UCpx i2 r2)
= case (i1 +# i2) of
  t1# -> case (r1 +# r2) of
         t2# -> UCpx t1# t2#
```

Now the components of the `UCpx` constructor are *unboxed* integers, and are therefore computed before the constructor is built.

6.2 Arbitrary unboxed algebraic data types

An *enumeration type* is an algebraic data type whose constructors all have zero arity. For example

```
data Boolean = False | True
data Colour = Red | Green | White | Blue
```

Objects of type `Boolean` and `Colour` are boxed; yet it makes sense to think of an unboxed boolean or colour, represented as one of a suitable set of bit-patterns. This observation provokes the question: can we allow the programmer to declare new *unboxed* enumeration types, perhaps like this:

```
data unboxed Colour# = Red# | Green# | White# | Blue#
data unboxed Boolean# = False# | True#
```

It would certainly be more efficient to represent a value of type `Colour#` or `Boolean#` than to represent one of type `Boolean` or `Colour`. As usual the `#` annotations are present only to clarify the presentation; it is the `unboxed` keyword which indicates that the enumeration should be unboxed.

The idea can be generalised further, by allowing *any* algebraic data type to be declared `unboxed`. For example:

```
data unboxed UPair# a b = UPair# a b
data unboxed UCpx# = UCpx# Int# Int#
data unboxed Maybe# a = Just# a | Nothing#
```

Each of these declarations has a natural operational reading. The type of unboxed pairs, `UPair`, is represented by a pair of pointers. These pointers are actually carried around together, rather then placing them in a heap-allocated box and carrying around a pointer to this box. Similarly the `UCpx#` type is represented by a pair of unboxed integers.

The `Maybe#` type is a little different, because it has more than one constructor. It can be represented by a bit to distinguish one constructor from the other, together with enough words to contain the components of any of the constructors (one, in this case). The implementation would be trickier here, and the efficiency gains might be less clear cut. Since the sole purpose of these unboxed types is to improve efficiency, it is not clear whether it is worth implementing them in full generality.

Recursive unboxed types are even less plausible. For example:

```
data unboxed UTree a = ULeaf a | UBranch (UTree a) (UTree a)
```

Here, the size of an unboxed tree would be variable, and it is hard to imagine how it could ever be implemented efficiently. Accordingly, we impose the rule that there must be at least one boxed type involved in any recursive loop of types (see Section 7.3).

7 The semantics of unboxed types

So far we have motivated the introduction of unboxed types by showing a number of ways in which they can be useful. It is now time to give the idea some formal foundation.

We do so in two steps:

- We discuss the modifications necessary to the type system; that is, the static semantics (Section 8).
- We modify the dynamic semantics of the Core language to take account of unboxed values, and discuss the safety of program transformations in the presence of unboxed values (Section 9).

Before we begin this sequence, we discuss an important caveat. We are trying to make unboxed values into first-class citizens, but it turns out that we must make three significant restrictions on their use.

The first two of these restrictions certainly make writing programs involving unboxed types less convenient. But remember that we are talking here about the *Core* language, rather than about the *source* language in which the programmer writes. There are several steps we can take to reduce the impact of these restrictions on the programmer, by inserting implicit evaluations and coercions, but it is better to deal with one issue at a time. We therefore concentrate now on giving a semantics for the restricted language, while in Section 10, we discuss the language-design question.

7.1 Restriction 1: loss of polymorphism

Unboxed objects are dangerous beasts: if the garbage collector should ever treat one as a pointer to a heap object, the entire system might crash. Furthermore, the size of an unboxed object may vary with its type. An `Int#` object may be the same size as a pointer, but a `LongInt#` object would be larger, as would a 64-bit floating point number and an unboxed pair.

One approach would be to add tag bits to distinguish boxed and unboxed objects, and give size and layout information for unboxed objects. This would be intolerably slow. For example the simple function

```
head (x:xs) = x
```

would have to test the tag on the value `x` to find out how many bytes to move into the result location(s). This approach would largely obviate one of the major benefits of strong typing, namely the performance advantage of working with untagged data (Appel [1988]).

Accordingly, we assume that no tag bits distinguish boxed from unboxed objects, relying solely on the type system to keep them apart. It follows immediately that:

> **Restriction 1.** *Polymorphic functions cannot manipulate unboxed values.*

7.2 Restriction 2: explicit evaluation

Whenever an unboxed value is stored in a data structure or passed to a function, it must first be evaluated. Whilst this could be left implicit, we prefer to make it explicit. The main reason for doing so is that it allows the dynamic semantics to be much more straightforward, and ensures that most existing program transformations remain valid (Section 9).

There is a simple syntactic criterion which tells if an expression of unboxed type is in head normal form (HNF) or not: an expression of unboxed type is in HNF if it is:

- a literal constant,
- an application of an unboxed constructor, or
- a variable.

(An unboxed constructor is a constructor of an unboxed algebraic data type — see Section 6.2.) The only surprise here is that a variable is an HNF; this follows from the fact that unboxed variables are bound to the bit-pattern corresponding to the value, so then cannot be bottom. Since functions are all boxed, no partial application is an HNF of unboxed type.

Now we can formalise our restriction:

> **Restriction 2.** *An expression of unboxed type which appears as the argument of an application, or as the right-hand side of a binding, must be in HNF.*

As an example of this restriction, the expression

```
f (x +# y)
```

is illegal because `(x +# y)` is not an HNF. It must instead be written

```
case (x +# y) of t# -> f t#
```

The evaluation has thereby been made explicit.

Restriction 2 permits recursive definitions of unboxed values, which at first look unreasonable. They do not seem to be very useful, however, so we ignore them until Section 12.4.

7.3 Restriction 3: no recursive unboxed data types

For the reasons discussed above in Section 6.2, and to ensure the domain equations of Section 9 have a least solution, we prohibit recursive definitions of unboxed data types:

Polytype	σ	$::=$	$\forall\alpha.\ \sigma \mid \tau$	
Monotype	τ	$::=$	$\pi \mid \nu$	
Boxed type	π	$::=$	α	Type variable
		$\mid$	$\tau_1 \rightarrow \tau_2$	Function type
		$\mid$	$\chi\ \pi_1 \ldots \pi_n$	Parameterised boxed data type
Unboxed type	ν	$::=$	`Int#`	
		$\mid$	`Float#`	
		$\mid$	$\chi\#\ \pi_1 \ldots \pi_n$	Parameterised unboxed data type

Figure 4: Syntax of types

Restriction 3. *There must be at least one boxed type involved in any recursive loop of types.*

This is rather similar to the usual rule that type synonyms must not be recursive.

8 The static semantics of unboxed types

Since all three restrictions are expressed in terms of types, it follows that the type system must be modified to embody them. Restriction 3 is a simple syntactic check, but Restrictions 1 and 2 are more interesting.

8.1 Types

The syntax for types is given in Figure 4. A monotype, τ, can be boxed or unboxed.

A boxed type, π, is a type variable, a function type, or a type constructor χ parameterised only over further boxed types. Each type constructor χ corresponds to a (boxed) algebraic data type declaration.

Functions are interesting. The argument and result of a function type can be any type, boxed or otherwise; but *the function itself is regarded as boxed* because it is represented by a pointer to a closure.

An unboxed type, ν, is either one of the built-in unboxed types such as `Int#` or `Float#`, or an unboxed type constructor $\chi\#$ parameterised over boxed types. Each such type constructor corresponds to an `unboxed` algebraic data type declaration.

The only surprise here is that data types cannot be parameterised over unboxed types. For example, does not the type `List Int#` (which would be written `[Int#]` in HASKELL) make perfect sense? The difficulty is that the garbage collector cannot tell that this particular list contains unboxed objects, which should not be treated as pointers; and furthermore, standard list cells might well be unable to accomodate, say, a double-precision floating pointer number. Another way to think of it is this: the list constructor ":" is polymorphic, and hence should not be applied to unboxed values (Restriction 1).

The only constructors which can have unboxed components are those which have been explicitly declared as such (such as `UCpx`). Notice that the type `UCpx` is a boxed type, despite having unboxed components. The restriction is that a polymorphic type cannot be *parameterised* with unboxed components.

8.2 Typing

We move on to consider how the type rules can be adjusted to accomodate the new constraints. Fortunately it is rather easy. Most type systems have a rule looking like this (for typical examples of complete typing rules see Damas & Milner [1982] or Hancock's chapter in Peyton Jones [1987]):

$$\text{SPEC}\quad \frac{A \vdash e : \forall\alpha.\ \sigma}{A \vdash e : \sigma[\tau/\alpha]}$$

This rule is used to instantiate polymorphic types, by substituting an arbitrary monotype τ for the type variable α in σ. All that we need to do to implement Restriction 1 is to ensure that polymorphic types are never instantiated with unboxed types, thus:

$$\text{SPEC'}\quad \frac{A \vdash e : \forall\alpha.\ \sigma}{A \vdash e : \sigma[\pi/\alpha]}$$

That is, only boxed types π should be substituted for α. Restriction 2 is also easy to embody. We need two rules for application instead of one:

$$\text{AP}\quad \frac{\begin{array}{l} A \vdash e_1 : \pi \rightarrow \tau \\ A \vdash e_2 : \pi \end{array}}{A \vdash e_1\ e_2 : \tau} \qquad\qquad \text{AP\#}\quad \frac{\begin{array}{l} A \vdash e_1 : \nu \rightarrow \tau \\ A \vdash h : \nu \end{array}}{A \vdash e_1\ h : \tau}$$

The *AP* rule applies to functions taking boxed arguments, while the *AP#* rule insists that an unboxed argument must be an HNF, h. The rule for `let` and `letrec` need to altered in a similar way.

And that is all! No further changes are required to the type system.

8.3 Type inference

Next, we consider what changes need to be made to a type inference system to accomodate the new rules and the accompanying two kinds of types.

SPEC is usually implemented in a type inference engine by substituting a fresh type variable for the universally quantified variable α. This is subsequently specialised as much as necessary by unification. To implement the new SPEC rule we therefore need to modify the unification process: *the unifier should fail to unify an unboxed type with a polymorphic type variable.* For example, consider the expression

```
id 4#
```

where `id` is the polymorphic identity function with type $\forall\alpha.\ \alpha \rightarrow \alpha$, and `4#` is an unboxed integer of type `Int#`. The type of `id` is instantiated to $\beta \rightarrow \beta$, where β is a fresh type variable. Then an attempt will be made to unify β with `Int#`, and at this point a type error will be reported.

This is not quite all we have to do. Consider the function definition

```
uInc x = x +# 1#
```

Function definitions are usually typed by adding the assumption `x` : α to the type environment, where α is a fresh type variable, and then typing the body. This yields a substitution which can be applied to α to give the argument type for `f`. In this case we do not want the type checker to complain at the unification of α with `Int#`; rather we want type inference to succeed in attributing to `uInc` the type `Int#` $\rightarrow$ `Int#`.

What is required is two forms of type variables: uncommitted ones, which can unify with anything; and boxed ones, which cannot unify with unboxed types. Function arguments are given uncommitted type variables, while boxed type variables are used to instantiate polymorphic types.

The same trick can be used to implement the two forms of AP rule. If the argument of the function is not simple variable, we unify the type of the argument with a boxed type variable. In effect, this tags the argument type (which might at this stage be only an uncommitted type variable), so that it can only subsequently unify with a boxed type, as required.

This completes our sketch of the modest changes required to a type inference system to accomodate unboxed types. It is a little surprising that the type *system* does not need two kinds of type variables; only the type *inferencer* does. It would be possible to introduce two kinds of type variables into the type system as well, and it might make it easier to prove soundness and completeness results if this were done.

9 The dynamic semantics of unboxed types

Next we consider what changes need to be made to the dynamic semantics of the Core language to accomodate unboxed types. As before, we need to give a domain model for each type, and we need to give the semantic equations.

9.1 The model

The first question is: what is the domain $\mathcal{D}[\![\texttt{Int\#}]\!]$, corresponding to the type of unboxed integers? Our first attempts to answer this question suggested that it should be the usual domain of integers, including $\perp$; but that meant that $\mathcal{D}[\![\texttt{Int}]\!]$, the domain of boxed integers, had to have an extra bottom. This double lifting gives rise to all sorts of complications, over which we draw a kindly veil.

A much better solution is to use *unpointed domains*, that is domains which lack a bottom element. (A good introduction is given by Schmidt [1986].) So the domain of unboxed integers is defined thus:

$$\mathcal{D}[\![\texttt{Int\#}]\!] = \{\text{The fixed-precision integers}\}$$

The domain has no bottom element, and the ordering relation is just the identity. The lack of a bottom element for unboxed domains corresponds nicely to our intuition: a variable of unboxed type can never be bound to bottom, because the whole computation will diverge before the binding takes place.

The difference between the domains corresponding to `Int` and `Int#` is that the latter is not lifted. Now it is possible to see why we made the outermost lifting explicit in our model for algebraic data types in Section 3.3: it makes it easy to generalise the model to unboxed data types, merely by omitting the lifting. Specifically, for an unboxed algebraic data type declaration, of form

$$\texttt{data unboxed}\ \chi\#\ \alpha_1 \ldots \alpha_t \;=\; c_1\ \tau_{11} \ldots \tau_{1a_1} \mid \ldots \mid c_n\ \tau_{n1} \ldots \tau_{na_n}$$

we derive the following functor definition:

$$\begin{array}{rcl} \chi\#\ d_1 \ldots d_t & = & s_1 + \ldots + s_n \\ & & \text{where}\ \rho \;=\; [\alpha_1 \mapsto d_1,\ \ldots, \alpha_t \mapsto d_t] \\ & & \phantom{\text{where}\ }s_i \;=\; \mathcal{D}[\![\tau_{i1}]\!]\ \rho \times \ldots \times \mathcal{D}[\![\tau_{ia_i}]\!]\ \rho \qquad (1 \leq i \leq n) \end{array}$$

The only difference from the boxed case is the absence of the outermost lifting.

The inductive definition for $\mathcal{D}[\![]\!]$ from Section 3.3 goes through largely unchanged:

$$\begin{array}{lcl} \mathcal{D}[\![\alpha]\!]\ \rho & = & \rho\ \alpha \\ \mathcal{D}[\![\texttt{Int\#}]\!]\ \rho & = & \{\text{The set of fixed-precision integers}\} \\ \mathcal{D}[\![\texttt{Float\#}]\!]\ \rho & = & \{\text{The set of fixed-precision floating-point numbers}\} \\ \mathcal{D}[\![\tau \to \pi]\!]\ \rho & = & (\mathcal{D}[\![\tau]\!]\ \rho) \to (\mathcal{D}[\![\pi]\!]\ \rho) \\ \mathcal{D}[\![\tau \to \nu]\!]\ \rho & = & (\mathcal{D}[\![\tau]\!]\ \rho) \to (\mathcal{D}[\![\nu]\!]\ \rho)_\perp \\ \mathcal{D}[\![\chi_n\ \tau_1 \ldots \tau_n]\!]\ \rho & = & \chi_n\ (\mathcal{D}[\![\tau_1]\!]\ \rho) \ldots (\mathcal{D}[\![\tau_n]\!]\ \rho) \\ \mathcal{D}[\![\chi\#_n\ \tau_1 \ldots \tau_n]\!]\ \rho & = & \chi\#_n\ (\mathcal{D}[\![\tau_1]\!]\ \rho) \ldots (\mathcal{D}[\![\tau_n]\!]\ \rho) \end{array}$$

The main difference comes in the function space construction. A function returning an unboxed value may fail to terminate, so its result type must be lifted[2].

The resulting domain equations are well-founded provided Restriction 3 is observed, namely that every recursive loop of type declarations includes at least one boxed type.

[2] This decision is not as *ad hoc* as it may appear. The restricted form of function application permitted by Restriction 2 directly implements Kleisli composition over the lifting monad, where an arrow from A to B is a function from A to $B_\perp$ (Moggi [1989]).

9.2 The semantic equations

The major difference in the semantics of programs involving unboxed values is that unboxed arguments are evaluated before calling the function to which they are passed. At first it looks as though this will require a significant adjustment to the semantics, but this is the point at which Restriction 2 comes into its own. Because Restriction 2 has already made all evaluation explicit, *no changes whatsoever are required to the semantic equations of Figure 3.*

Whilst the equations themselves do not change, the definition of the domains **Val** and **Env** need to be adjusted in a somewhat subtle way.

First, the valuation of an expression of unboxed type may fail to terminate, and hence, as with function types, we need to lift the result domain:

$$\mathbf{Val} = \left(\bigcup_{\pi} \mathcal{D}[\![\pi]\!]\right) \cup \left(\bigcup_{\nu} \mathcal{D}[\![\nu]\!]_{\perp}\right)$$

Does the same need to be done for **Env**? No it does not, because we claim that *a variable of unboxed type can never be bound to bottom.* To justify this claim, consider all the places where a variable of unboxed type can be bound:

- The application of a lambda abstraction. Here the argument can only be a variable (Restriction 2), so if the claim is true of the argument it will also be true of the variable bound by the abstraction.
- The evaluation of a `case` alternative. Here variables are bound to the components of a constructor. But if any of these components is of unboxed type, then the corresponding argument at the call of the constructor will be a variable, and hence cannot be bottom.
- The evaluation of a `case` alternative. Here a variable is bound to the value which the `case` evaluates. But if this value is bottom then the `case` diverges, and so the binding never takes place.

This all corresponds directly to our intuition. The run-time environment will contain pointers to (perhaps as yet unevaluated) boxed values, and some unboxed values. The latter cannot be bottom! So we retain the same definition for **Env**:

$$\mathbf{Env} = \bigcup_{\tau} (var_{\tau} \rightarrow \mathcal{D}[\![\tau]\!])$$

9.3 Transforming programs involving unboxed types

The whole thrust of this paper has been to expose programs involving unboxed values to optimising transformations (cf Section 4). It is legitimate to ask whether all the transformations we know so well still apply in the new setting. This is not a trivial question. For example, suppose that we did not impose Restriction 2, so that unboxed arguments could be non-trivial expressions. Then β-reduction is no longer valid! For example, consider the expression:

```
(\x.3) (f 3)
```

where `f` has type `Int -> Int#`, and `f` happens to diverge on argument `3`. Then, because unboxed arguments must be evaluated before a call, this expression has value $\perp$. But a simple β reduction transforms this expression to `3`, and hence changes the meaning.

The most important reason for Restriction 2 is to prevent this sort of problem. All the usual program transformations remain valid in the new setting. We can justify this claim by observing that *all the existing semantic equations remain unchanged, so if a transformation previously preserved correctness then it will do so in the extended language as well.* This sounds obvious, but many of our earlier attempts at a dynamic semantics required different semantic rules, and so the correctness of transformations was a matter for speculation.

There is a small caveat. We need to check that the result of a program transformation still obeys Restrictions 1 and 2; that is, they are still well-typed in the sense of Section 8.

10 Language design issues

The sole reason for introducing unboxed values in the first place is to improve efficiency. There are no programs which one can write using unboxed values which cannot be written equally easily without. It is therefore far from obvious whether unboxed values should be exposed to the programmer at all; it would be quite possible to use them solely as a convenient notation inside the compiler, helping to support the compilation-by-transformation paradigm as discussed in Sections 4 and 5.

Even so, there is a strong case for making unboxed values directly available to the programmer, including the generalisations proposed in Section 6, as we now discuss. (A possible compromise would be to make such language extensions available only to the *systems* programmer; for example, the person who writes the complex-number arithmetic package.)

Consider the `UCpx` data type introduced in Section 6.1. In principle, it is possible that a very clever program analyser could look at a program written in terms of `Cpx` and `addCpx` and figure out that it would not change the meaning of the program to rewrite it in terms of `UCpx` and `addUCpx`. In practice, this is far beyond what current compiler technology can do, because data can be placed in a data structure in one part of the program, and used somewhere else entirely. In any case, such an analysis would be impossible in the presence of separate compilation.

In short, the efficiency improvements arising from generalised unboxed algebraic data types cannot realistically be obtained without involving the programmer; yet these performance improvements can be substantial.

The other trouble with making unboxed values part of the source language is that Restrictions 1 and 2 are very tiresome. The remaining parts of this section consider how they may be alleviated.

10.1 Automatic evaluation

It is a simple matter to lift Restriction 2 in the source language, by transforming the program to obey the restriction after type inference is complete. For example, whenever a function is applied to a non-HNF argument of unboxed type, the application is enclosed in a `case` expression which evaluates the argument and binds it to a variable, which is then passed to the function.

Similarly, whenever a `let`-expression binds a non-HNF value of unboxed type, it is replaced with the corresponding `case` expression.

This transformation cannot be applied to `letrec` expressions, because there *is* no corresponding `case` expression. For the same reason, it cannot be applied to the top-level bindings of a program.

10.2 Automatic coercion

It is also possible to lift Restriction 1, which prohibits unboxed values from being passed to polymorphic functions, at least where there is a boxed form of the same data type, by coercing to and from the boxed form.

For example, a value of type `Int#` may be passed to a polymorphic function, by first being coerced type `Int` by applying the `Int` constructor to it. Similarly, a value of type `Int` can be coerced into one of type `Int#` by evaluating it and extracting its value. This is exactly the approach taken by Leroy [1991].

For example, the expression:

```
foldr (+#) 0# [4#, 5#]
```

would, after the coercions have been introduced, look like this:

```
case (foldr (\x. \y. case x of Int x# ->
                     case y of Int y# ->
                     case (+# x# y#) of t# -> Int t#)
      (Int 0#)
      [Int 4#, Int 5#])
of
Int t1# -> t1#
```

The unboxed constants have been boxed to make them compatible with the polymorphic functions `foldr` and the list constructor; the corresponding coercions have been done to the function passed to `foldr`; and the result of the `foldr` is then coerced back to `Int#`.

It is debatable whether all of this extra stuff should get introduced by the compiler, perhaps without the programmer realising that it is taking place. After all, the whole purpose of the exercise is to improve performance, so hidden performance losses are bad news.

The other problem is that, in general, it is possible to insert coercions in more than one place, and still obtain a correct program. Satish Thatte gives a good presentation of the issues (Thatte [1990]).

11 Related work

There is a long tradition of compilation by transformation, starting with Steele's Rabbit compiler (Steele [1978]), which pioneered continuation-passing style (CPS). CPS allows many representation decisions to be exposed, and expressed in the language being transformed, rather than being hidden in a black-box code generator. The Orbit compiler (Kranz [1988]) for Scheme is built on these ideas to make a production-quality optimising compiler.

Appel and MacQueen's Standard ML compiler is a further development in this line, with even greater modularity especially near the back end (Appel & Jim [1989]). Kelsey's thesis, entitled "Compilation by program transformation", is also CPS-based, but he handles imperative languages as well (Kelsey [1989]).

Fradet and LeMetayer describe another transformation-based compiler based on CPS (Fradet & Metayer [1990]; Fradet & Metayer [1988]). Their approach is unusual in that their target code is a continuation-based combinator language which has a direct reading either as a functional program or as a sequence of abstract machine instructions. The trouble is that they are thereby forced to make an early commitment to some low-level representation decisions, such as stack layout.

All of this work relates to compilation-by-transformation of languages with strict semantics. CPS is wonderful for such languages, because it makes explicit the order of evaluation which is implied by the semantics. For non-strict language, where the order of evaluation is demand-driven, CPS is not nearly so useful. A `case` expression is the nearest we get to it ("evaluate this expression and then continue in one of these ways, depending on the result you get").

The case-of-case transformation and its cousins (case-of-constructor, case-of-variable in a scope where the variable is bound to a constructor) are not new. They were the key transformations in the deforestation algorithm given by Wadler [1990], whose purpose is to eliminate intermediate data structures in functional programs. Indeed, the work described here could be seen as an exploration of the effects of exposing unboxed types to deforestation, the intermediate data structures in this case being the boxes around values.

None of these works directly addresses the main theme of this paper, namely the treatment of unboxed values in a non-strict language. The Clean compiler from Nijmegen (Brus et al. [1987]) does support unboxed values at the program level, but the details of what it does and how it works are not published.

Peterson's paper, whose title "Untagged data in tagged environments" is superficially similar to this paper, addresses a different, though related, problem (Peterson [1989]). Suppose that one took the ideas of this paper but dropped Restriction 1 (which restricts polymorphism), using tagging to identify unboxed data. Peterson then proposes an analysis to discover regions of the program in which the tags need not be attached to the unboxed values, which is of course much more efficient than manipulating the tagged representation. It is not clear how well his analysis would work in a non-strict language where the order of evaluation is much harder to predict.

Leroy's work has already been mentioned (Leroy [1991]). It deals with a strict language, and concentrates mainly on the coercion issues discussed in Section 10. It nicely complements this paper.

12 Further work

12.1 Overloading the built-in operators

In Section 6.1 we introduced the following example:

```
data Cpx = Cpx Int Int
addCpx (Cpx r1 i1) (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
```

Suppose that one had written the former definition of `Cpx` and `addCpx`, and then decided to make the components of `Cpx` unboxed. It would be nice if one could just replace `Int` with `Int#` in the definition of the `Cpx` data type, but make no change to the `addCpx` function, thus:

```
data Cpx = Cpx Int# Int#
addCpx (Cpx r1 i1) (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
```

(We are assuming that Restriction 2 is lifted as discussed above.) As things stand, `addCpx` and every other function which manipulates the components of a `Cpx` constructor, needs to be changed to use use unboxed operations instead of boxed ones:

```
addCpx (Cpx r1 i1) (Cpx r2 i2) = Cpx (r1 +# r2) (i1 +# i2)
```

This is a nuisance, and it would be nice if the compiler could figure such things out for itself. At first it seems that HASKELL's overloading mechanism might solve the problem, but this is defeated by Restriction 1. Hence we cannot declare an instance of the class `Num` for unboxed integers `Int#`.

12.2 Foreign-language interfacing

One of the reasons it is quite tricky to call subroutines written in another language (eg C) from a non-strict functional program is because other language generally manipulate unboxed values. Once we can manipulate unboxed values in the functional language, it is likely to become easier to build a direct interface to other imperative languages. We have not investigated this yet.

12.3 Unboxed functions

Does it make any sense to talk of unboxed *functions*? For example, a C program can pass code addresses around, and then call them.

In general, a function is represented by a code pointer and an environment. An unboxed function could perhaps be a code pointer together with a pointer to an environment. But then not much has been gained compared with making the code pointer just one more field in the environment.

The time that there would be a substantial benefit would be for functions which had no free variables; that is, no environment. Such functions can indeed be represented by just a code address, just like C.

12.4 Recursive definitions of unboxed values

Recursive definitions of unboxed values are permitted by Restriction 2, provided their right-hand sides are HNFs, and they even make sense! For example, given the type definitions

```
data unboxed Two# = Two# One Int#
data One = One Two#
```

the following recursive definition makes sense, attributing to `x` the type `Two#`:

```
x = Two# (One x) 3#
```

The way to understand such a definition is by naming each subexpression, and then, in the definition of each boxed variable, replacing each unboxed variable by its definition. This leaves a set of recursive definitions of boxed values, and some non-recursive definitions of unboxed values. In this example, the result is:

```
x = Two# y 3#
y = One (Two# y 3#)
```

Only the definition of `y` is recursive. We can always do this operation because of Restriction 3, provided there is no loop of the form

```
p# = q#
q# = p#
```

This rather unsatisfactory caveat is one reason we left this section under "Further work". In the light of the transformation given above, another approach would be to rule out recursive definitions of unboxed values.

Acknowledgements

When we were enmeshed in various rather complicated attempts at a dynamic semantics for unboxed values, John Hughes had the insight that we should try using unpointed domains. Kei Davis, Cordelia Hall, Will Partain, John Peterson, Ryszard Kubiak and Phil Wadler all made lots of very useful comments which helped to improve the presentation. Our grateful thanks to them.

This work was done with the support of the SERC GRASP project, and the ESPRIT Semantique Basic Research Action.

Bibliography

AW Appel [1988], "Runtime tags aren't necessary," CS-TR-142-88, Department of Computer Science, Princeton University.

AW Appel & T Jim [Jan 1989], "Continuation-passing, closure-passing style," in *Proc ACM Conference on Principles of Programming Languages*, ACM, 293–302.

TH Brus, MCJD van Eckelen, MO van Leer & MJ Plasmeijer [Sept 1987], "Clean - a language for functional graph rewriting," in *Functional programming languages and computer architecture, Portland*, G Kahn, ed., LNCS 274, Springer Verlag, 364–384.

LMM Damas & R Milner [1982], "Principal type schemes for functional programs," in *POPL*, 207–212.

P Fradet & D Le Metayer [1990], "Compilation of functional languages by program transformation," IRISA, Campus de Beaulieu, Rennes.

P Fradet & D Le Metayer [June 1988], "Compilation of lambda-calculus into functional machine code," INRIA.

[1986], in *Abstract Interpretation of Declarative Languages*, C Hankin & S Abramsky, eds., Ellis Horwood, Chichester, 246–265.

P Hudak, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, SL Peyton Jones, M Reeve, D Wise & J Young [April 1990], "Report on the functional programming language Haskell," Department of Computing Science, Glasgow University.

R Kelsey [May 1989], "Compilation by program transformation," YALEU/DCS/RR-702, PhD thesis, Department of Computer Science, Yale University.

DA Kranz [May 1988], "ORBIT - an optimising compiler for Scheme," PhD thesis, Department of Computer Science, Yale University.

X Leroy [April 1991], "Efficient data representations in polymorphic languages," INRIA Research Report 1264, Rocquencourt.

E Moggi [June 1989], "Computational lambda calculus and monads," in *Logic in Computer Science, California*, IEEE.

J Peterson [Sept 1989], "Untagged data in tagged environments: choosing optimal representations at compile time," in *Functional Programming Languages and Computer Architecture, London*, Addison Wesley, 89–99.

SL Peyton Jones [1987], *The implementation of functional programming languages*, Prentice Hall.

SL Peyton Jones [Feb 1991], "The Spineless Tagless G-machine: a second attempt," Department of Computing Science, University of Glasgow.

SL Peyton Jones & Jon Salkild [Sept 1989], "The Spineless Tagless G-machine," in *Functional Programming Languages and Computer Architecture*, D MacQueen, ed., Addison Wesley.

DA Schmidt [1986], *Denotational semantics: a methodology for language development*, Allyn and Bacon.

MB Smyth & GD Plotkin [Nov 1982], "The category-theoretic solution of recursive domain equations," *SIAM Journal of Computing* 11, 761–783.

GL Steele [1978], "Rabbit: a compiler for Scheme," AI-TR-474, MIT Lab for Computer Science.

SR Thatte [1990], "Coercive type equivalence," Department of Maths and Computer Science, Clarkson University NY.

P Wadler [1990], "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science* 73, 231–248.