Abstract

It is common for compilers to derive the calling convention of a function from its type. Doing so is simple and modular but misses many optimisation opportunities, particularly in lazy, higher-order functional languages with extensive use of currying. We restore the lost opportunities by defining Strict Core, a new intermediate language whose type system makes the missing distinctions: laziness is explicit, and functions take multiple arguments and return multiple results.

1. Introduction

In the implementation of a lazy functional programming language, imagine that you are given the following function:

\[ f :: Int \rightarrow Bool \rightarrow (Int, Bool) \]

How would you go about actually executing an application of \( f \) to two arguments? There are many factors to consider:

- How many arguments are given to the function at once? One at a time, as currying would suggest? As many as are available at the application site? Some other answer?
- How does the function receive its arguments? In registers? On the stack? Bundled up on the heap somewhere?
- Since this is a lazy language, the arguments should be evaluated lazily. How is this achieved? If \( f \) is strict in its first argument, can we do something a bit more efficient by adjusting \( f \) and its callers?
- How are the results returned to the caller? As a pointer to a heap-allocated pair? Or in some other way?

The answers to these questions (and others) are collectively called the calling convention of the function \( f \). The calling convention of a function is typically determined by the function’s type signature. This suffices for a largely-first-order language like C, but it imposes unacceptable performance penalties for a language like Haskell, because of the pervasive use of higher-order functions, currying, polymorphism, and laziness. Fast function calls are particularly important in a functional programming language, so compilers for these languages – such as the Glasgow Haskell Compiler (GHC) – typically use a mixture of ad hoc strategies to make function calls efficient.

In this paper we take a more systematic approach. We outline a new intermediate language for a compiler for a purely functional programming language, that is designed to encode the most important aspects of a function’s calling convention directly in the type system of a concise lambda calculus with a simple operational semantics.

- We present Strict Core, a typed intermediate language whose types are rich enough to describe all the calling conventions that our experience with GHC has convinced us are valuable (Section 3). For example, Strict Core supports uncurried functions symmetrically, with both multiple arguments and multiple results.
- We show how to translate a lazy functional language like Haskell into Strict Core (Section 4). The source language, which we call FH, contains all the features that we are interested in compiling well – laziness, parametric polymorphism, higher-order functions and so on.
- We show that the properties captured by the intermediate language expose a wealth of opportunities for program optimization by discussing four of them – definition-site and use-site arity raising (Section 6.1 and Section 6.2), thunk speculation (Section 5.5) and deep unboxing (Section 5.6). These optimizations were awkward or simply inaccessible in GHC’s earlier Core intermediate language.

Although our initial context is that of lazy functional programming languages, Strict Core is a call-by-value language and should also be suitable for use in compiling a strict, pure, language such as Timber [1], or a hybrid language which makes use of both evaluation strategies.

No single part of our design is new, and we discuss related work in Section 7. However, the pieces fit together very nicely. For example: the symmetry between arguments and results (Section 3.1); the use of n-ary functions to get thunks “for free”, including so-called “multi-thunks” (Section 3.4); and the natural expression of algorithms and data structures with mixed strict/lazy behaviour (Section 3.5).

2. The challenge we address

In GHC today, type information alone is not enough to get a definitive specification of a function’s calling convention. The next few sections discuss some examples of what we lose by working with the imprecise, conservative calling convention implied by the type system as it stands.

2.1 Strict arguments

Consider the following function:

\[ f :: Bool \rightarrow Int \]
\[ f x = \text{case } x \text{ of } True \rightarrow \ldots ; \text{False} \rightarrow \ldots \]
This function is certainly strict in its argument \( x \). GHC uses this information to generate more efficient code for \( f \) calls, using call-by-value to avoid allocating a thunk for the argument. However, when generating the code for the definition of \( f \), we can really assume that the argument has already been evaluated, and hence omit instructions that checks for evaluated-ness? Well, no. For example, consider the call

\[
\text{map } f \ [\text{fibonacci} \ 10, 1234]
\]

Since map is used with both strict and lazy functions, map will not use call-by-value when calling \( f \). So in GHC today, \( f \) is conservative, and always tests its argument for evaluated-ness even though in most calls the answer is ‘yes’.

An obvious alternative would be to treat first-order calls (where the call site can “see” the definition of \( f \), and you can statically see that your use-site has as at least as many arguments as the definition site demands) specially, and generate a wrapper for higher-order calls that does the argument evaluation. That would work, but it is fragile. For example, the wrapper approach to a map call might do something like this:

\[
\text{map } \lambda x. \text{case } x \text{ of } y \to f \ y \ [\ldots]
\]

Here, the case expression evaluates \( x \) before passing it to \( f \), to satisfy \( f \)’s invariant that its argument is always evaluated\(^1\).

But, alas, one of GHC’s optimising transformations is to rewrite case \( x \) of \( y \to e \) to \( e[x/y] \), if \( e \) is strict in \( x \). This transformation would break \( f \)’s invariant, resulting in utterly wrong behaviour or even a segmentation fault – for example, if it lead to erroneously treating part of an unevaluated value as a pointer. GHC has a strongly-typed intermediate language that is supposed to be immune to segmentation faults, so this fragility is unacceptable. That is why GHC always makes a conservative assumption about evaluated-ness.

The generation of spurious evaluated-ness checks represents an obvious lost opportunity for the so-called “dictionary” arguments that arise from desugaring the type-class constraints in Haskell. These are constructed by the compiler so as to be non-bottoming, and hence may always be passed by value regardless of how a function uses them. Can we avoid generating evaluated-ness checks for these, without the use of any ad-hocery?

### 2.2 Multiple arguments

Consider these two functions:

\[
f \ x \ y = x + y
\]

\[
g \ x = \text{let } z = \text{factorial} \ 10 \ \text{in } \lambda y. x + y + z
\]

They have the same type \( \text{Int} \to \text{Int} \to \text{Int} \), but we evaluate applications of them quite differently – \( g \) can only deal with being applied to one argument, after which it returns a function closure, whereas \( f \) can and should be applied to two arguments if possible. GHC currently discovers this \textit{arity} difference between the two functions statically (for first-order calls) or dynamically (for higher-order calls). However, the former requires an apparently-modest but insidiously-pervasive propagation of \textit{ad-hoc} arity information; and the latter imposes a performance penalty \([2]\).

For the higher-order case, consider the well-known list-combining combinator \textit{zipWith}, which we might write like this:

\[
\text{zipWith} = \lambda f :: (a \to b \to c). \lambda xs :: \text{List } a. \lambda ys :: \text{List } b.
\]

\[
\text{case } \text{xs of } \text{Nil } \to \text{Nil}
\]

\(^1\) In Haskell, a case expression with a variable pattern is lazy, but in GHC’s current compiler intermediate language it is strict, and that is the semantics we assume here.

\[^2\] ANF stands for A-normal form, which will be explained further in Section 3.6.
the environment – α-conversion can be used as usual to get around this restriction where necessary.

3.1 Syntax of Strict Core\textsubscript{ANF}

Strict Core\textsubscript{ANF} is a higher-order, explicitly-typed, purely-functional, call-by-value language. In spirit it is similar to System F, but it is slightly more elaborate so that its types can express a richer variety of calling conventions. The key difference from an ordinary typed lambda calculus, is this:

A function may take multiple arguments simultaneously, and (symmetrically) return multiple results.

The syntax of types \( \tau \), shown in Figure 2, embodies this idea: a function type takes the form \( \overline{b} \rightarrow \overline{\tau} \), where \( \overline{b} \) is a sequence of binders (describing the arguments of the function), and \( \overline{\tau} \) is a sequence of types (describing its results). Here are three example function types:

\[
\begin{align*}
\text{f1} & : \text{Int} \rightarrow \text{Int} \quad \langle \text{Int} \rangle \rightarrow \langle \text{Int} \rangle \\
\text{id} & : (\alpha \times \alpha, \alpha) \rightarrow \alpha \\
& \quad (\alpha \times \alpha, \alpha) \rightarrow (\alpha) \\
\text{f3} & : (\alpha \times, \text{Int}, \alpha) \rightarrow (\alpha, \text{Int}) \\
& \quad (\alpha \times, \text{Int}, \alpha) \rightarrow (\alpha, \text{Int}) \\
\text{f4} & : \alpha \times \rightarrow \text{Int} \rightarrow \alpha \rightarrow \langle \text{Bool}, \text{Int} \rangle \\
& \quad (\alpha \times \ast \rightarrow \text{Int} \rightarrow \alpha \rightarrow \langle \text{Bool}, \text{Int} \rangle) \\
\end{align*}
\]

In each case, the first line uses simple syntactic abbreviations, which are expanded in the subsequent line. The first, \( \text{f1} \), takes one argument and returns one result\(^3\). Strict Core\textsubscript{ANF} expresses functions using the notation of dependent products. For example, in System F the identity function \( \text{id} \) has type \( \forall \alpha \cdot (\alpha \times \alpha, \alpha) \rightarrow \alpha \), but in Strict Core\textsubscript{ANF} it has the type \( (\alpha \times, \alpha) \rightarrow (\alpha) \) (although another possibility would be \( (\alpha \times) \rightarrow \langle \alpha, \alpha \rangle \rightarrow (\alpha) \)), reflecting the fact that there is a choice of calling convention. As is conventional, the term binder “\( \times \)” may be omitted since it cannot be mentioned. The next example, \( \text{f3} \), illustrates a polymorphic function that takes a type argument and two value arguments, and returns two results. Finally, \( \text{f4} \) gives a curried version of the same function.

Admittedly, this uncurried notation is more complicated than the unary notation of conventional System F, in which all functions are curried. The extra complexity is crucial because, as we will see in Section 3.3, it allows us to express directly that a function takes several arguments simultaneously, and returns multiple results.

The syntax of terms (also shown in Figure 2) is driven by the same imperative. For example, Strict Core\textsubscript{ANF} has n-ary application \( \overline{c} \overline{\ell} \); and a function may return multiple results \( \overline{\pi} \). A possibly-recursive collection of \textit{heap values} may be allocated with \texttt{valrec}, where a heap value is just a lambda or constructor application. Finally, evaluation is performed by \texttt{let}; since the term on the right-hand side may return multiple values, the \texttt{let} may bind multiple values. Here, for example, is a possible definition of \( \text{f3} \) above:

\[
f3 = \lambda (\alpha \times, x : \text{Int}, y : \alpha) \cdot (y, x)
\]

In support of the multi-value idea, terms are segregated into three syntactically distinct classes: atoms \( \alpha \), heap values \( v \), and multi-value terms \( e \). An \textit{atom} \( \alpha \) is a trivial term – a literal, variable reference, or (in an argument position) a type. A \textit{heap value} \( v \) is a heap-allocated constructor application or lambda term. Neither atoms nor heap values require evaluation. The third class of terms is much more interesting: a \textit{multi-value term} \( e \) is a term that either diverges, or evaluates to several (zero, one, or more) values simultaneously.

\(^3\)Recall Figure 1, which abbreviates a singleton sequence \( \langle \text{Int} \rangle \) to \text{Int}.
\[
\begin{align*}
\Gamma \vdash \alpha : \kappa & \quad \forall \alpha : \kappa \vdash \beta : \gamma & \quad \forall \gamma \vdash \pi : \tau & \quad \forall \tau \vdash \nu : \kappa_1 & \quad \forall \kappa_1 \rightarrow \kappa_2 \vdash \nu : \kappa_1 \\
T : \kappa \in \Gamma & \quad \forall \kappa \vdash T : \kappa & \quad B(T) = \kappa & \quad \forall \kappa \vdash T : \kappa & \quad \forall \kappa \vdash \text{TyConData} & \quad \forall \kappa \vdash \text{TyConPrim}
\end{align*}
\]

**Figure 3:** Kinding rules for Strict Core\text{ANF}

### 3.2 Static semantics of Strict Core\text{ANF}

The static semantics of Strict Core\text{ANF} is given in Figure 3, Figure 4 and Figure 5. Despite its ineluctable volume, it should present few surprises. The term judgement $$\Gamma \vdash e : \tau$$ types a multi-valued term $$e$$, giving it a multi-type $$\tau$$. There are similar judgements for atoms $$a$$, and values $$v$$, except that they possess types (not multi-types). An important invariant of Strict Core\text{ANF} is this: variables and values have types $$\tau$$, not multi-types $$\tau$$. In particular, the environment $$\Gamma$$ maps each variable to a type $$\tau$$ (not a multi-type).

The only other unusual feature is the tiresome auxiliary judgement $$\Gamma \vdash \text{app} \ b \rightarrow \pi \ @ \ g : \varnothing$$, shown in Figure 5, which computes the result type $$\pi$$ that results from applying a function of type $$\mathbf{b} \rightarrow \tau$$ to arguments $$\mathbf{g}$$.

The last two pieces of notation used in the type rules are for introducing primitives and are as follows:

- \text{L} \quad Maps built-in type constructors to their kinds – the domain must contain at least all of the type constructors returned by \text{L}

### 3.3 Operational semantics of Strict Core\text{ANF}

Strict Core\text{ANF} is designed to have a direct operational interpretation, which is manifested in its small-step operational semantics, given in Figure 7. Each small step moves from one \text{configuration} to another. A configuration is given by \langle \mathcal{H} ; e ; \Sigma \rangle, where \mathcal{H} represents the heap, $$e$$ is the term under evaluation, and $$\Sigma$$ represents the stack – the syntax of stacks and heaps is given in Figure 6.

We denote the fact that a heap $$\mathcal{H}$$ contains a mapping from $$x$$ to a heap value $$v$$ by $$\mathcal{H}[x \mapsto v]$$. This stands in contrast to a pattern such as $$\mathcal{H} , x \mapsto v$$, where we intend that $$\mathcal{H}$$ does not include the mapping for $$x$$.

The syntax of Strict Core is carefully designed so that there is a 1–1 correspondence between syntactic forms and operational rules:

- Rule \text{EVAL} begins evaluation of a multi-valued term $$e_1$$, pushing onto the stack the frame \text{let} $$\pi \tau \varepsilon = \bullet \text{ in } e_2$$. Although it is a pure language, Strict Core\text{ANF} uses call-by-value and hence evaluates $$e_1$$ before $$e_2$$. If you want to delay evaluation of $$e_1$$, use a thunk (Section 3.4).

- Dually, rule \text{RET} returns a multi-value to the \text{let} frame, binding the $$\pi$$ to the (atomic) returned values $$\mathbf{v}$$. In this latter rule, the simultaneous substitution models the idea that $$e_1$$ returns multiple values in \text{registers} to its caller. The static semantics (Section 3.2) guarantees that the number of returned values exactly matches the number of binders.

- Rule \text{ALLOC} performs heap allocation, by allocating one or more heap values, each of which may point to the others. We model the heap address of each value by a fresh variable $$y$$ that

**Figure 4:** Typing rules for Strict Core\text{ANF}
3.4 Thunks

Because Strict Core\textsubscript{ANF} is a call-by-value language, if we need to
delay evaluation of an expression we must explicitly \textit{thunk} it in the
program text, and correspondingly \textit{force} it when we want to
actually access the value.

We use \textit{thunking} to describe the process of wrapping a term \(e\) in
a nullary function \(\lambda \cdot e\). Because thunking is so common, we
use syntactic sugar for the thunking operation on both types and
expressions – if something is enclosed in \{braces\} then it is a
thunk. See Figure 2 for details.
An unusual feature is that Strict CoreANF supports multi-valued thunks, with a type such as \( \langle \rangle \rightarrow \langle \text{Int, Bool} \rangle \), or (using our syntactic sugar) \{\text{Int, Bool}\}. Multi-thunks arose naturally from treating thunks as a special kind of function, but this additional expressiveness turns out to allow us to do at least one new optimisation: deep unboxing (Section 5.6).

Arguably, we should not conflate the notions of functions and thunks, especially since we have special cases in our operational semantics for nullary functions. However, the similarity of thunks and nullary functions does mean that some parts of the compiler can be cleaner if we adopt this conflation. For example, if the compiler detects that all of the arguments to a function of type \( \langle \text{Int, Bool} \rangle \rightarrow \text{Int} \) are absent (not used in the body) then the function can be safely transformed to one of type \( \langle \rangle \rightarrow \text{Int} \), but not one of type \( \text{Int} \) – as that would imply that the body is always evaluated immediately. Because we conflate thunks and nullary functions, this restriction just falls out naturally as part of the normal code for discarding absent arguments rather than being a special case (as it is in GHC today).

### 3.5 Data types

We treat \text{Int} and \text{Char} as built-in types, with a suitable family of (call-by-value) operations. A value of type \text{Char} is an evaluated character, not a thunk (i.e. like ML, not like Haskell), and similarly \text{Int}. To allow a polymorphic function to manipulate values of these built-in types, they must be boxed (i.e. represented by a heap pointer like every other value). A real implementation, however, might have additional unboxed (not heap allocated) types, \text{Char}\#, \text{Int}\#, which do not support polymorphism [4], but we ignore these issues here.

All other data types are built by declaring a new algebraic data type, using a declaration \text{d}, each of which has a number of constructors (\text{c}). For example, we represent the (lazy) list data type with a top-level definition like so:

\[
\text{data List } a::=\text{Nil} \mid \text{Cons}\{\text{c}\}, \text{\{List } a\}\}
\]

Applications of data constructors cause heap allocation, and hence (as we noted in Section 3.3), values drawn from these types can only be allocated by a \text{valrec} expression.

The operational semantics of case expressions are given in rules \text{CASE-LIT, CASE-CON}, and \text{CASE-DEF}, which are quite conventional (Figure 7). Notice that, unlike Haskell, \text{case} does not perform evaluation – that is done by \text{let} in \text{eval}. The only subtlety (present in all such calculi) is in rule \text{CASE-CON}: the a constructor \text{C} must be applied to both its type and value arguments, whereas a pattern \text{match} for \text{C} binds only its value arguments. For the sake of simplicity we restrict ourselves to vanilla Haskell 98 data types, but there is no difficulty with extending Strict Core to include existentials, GADTs, and equality constraints [5].

### 3.6 A-normal form and syntactic sugar

The language as presented is in so-called A-normal form (ANF), where intermediate results must all be bound to a name before they can be used in any other context. This leads to a very clear operational semantics, but there are at least two good reasons to avoid the use of ANF in practice:

- In the implementation of a compiler, avoiding the use of ANF allows a syntactic encoding of the fact that an expression occurs exactly once in a program. For example, consider the following code:

\[
(\lambda (a:\ast, x:a). \ x) \ (\text{Int}, 1)
\]

The compiler may manifestly see, using purely local information, that it can perform \(\beta\)-reduction on this term, without the worry that it might increase code size. The same is not true in a compiler using ANF, because the ability to do \(\beta\)-reduction without code bloat depends on your application site being the sole user of the function – a distinctly non-local property!

- Non-ANFed terms are often much more concise, and tend to be more understandable to the human reader.

In the remainder of the paper we will adopt a non-ANFed variant of Strict CoreANF which we simply call Strict Core, by making use of the following simple extension to the grammar and type rules:

\[
\begin{align*}
a ::=& \ldots \mid e \mid v \\
\Gamma \vdash e : \langle \tau \rangle & \quad \text{SING} \quad \Gamma \vdash_\alpha v : \tau \\
\Gamma \vdash_\alpha v : \tau & \quad \text{VAL}
\end{align*}
\]

The semantics of the new form of atom are given by a standard ANFing transformation into Strict CoreANF. Note that there are actually several different choices of ANF transformation, corresponding to a choice about whether to evaluate arguments or functions first, and whether arguments are evaluated right-to-left or vice-versa. The specific choice made is not relevant to the semantics of a pure language like Strict Core.

### 3.7 Types are calling conventions

Consider again the example with which we began this paper. Here are several different Strict Core types that express different calling conventions:

\[
\begin{align*}
f_1 &: \text{Int} \rightarrow \text{Bool} \rightarrow \langle \text{Int, Bool} \rangle \\
f_2 &: \langle \text{Int, Bool} \rangle \rightarrow \langle \text{Int, Bool} \rangle \\
f_3 &: \langle \text{Int, Bool} \rangle \rightarrow \langle \text{Int, Bool} \rangle \\
f_4 &: \langle \langle \text{Int} \rangle, \text{Bool} \rangle \rightarrow \langle \text{Int} \rangle
\end{align*}
\]

Here \(f_1\) is a curried function, taking its arguments one at a time; \(f_2\) takes two arguments at once, but returns a heap-allocated pair; \(f_3\) takes a heap-allocated pair and returns two results (presumably in registers); while \(f_4\) takes two arguments at once, but the first is a thunk. In this way, Strict CoreANF directly expresses the answers to the questions posed in the Introduction.

By expressing all of these operational properties explicitly in our intermediate language we expose them to the wrath of the optimiser. Section 5 will show how we can use this new information about calling convention to cleanly solve the problems considered in the introduction.

### 3.8 Type erasure

Although we do not explore it further in this paper, Strict CoreANF has a simple type-erased counterpart, where type binders in \text{As}, type arguments and heaps values have been dropped. A natural consequence of this erasure is that functions such as \(\langle a : \ast \rangle \rightarrow \langle \text{Int} \rangle\) will be converted into thunks (like \(\langle \rangle \rightarrow \langle \text{Int} \rangle\)), so their results will be shared.

### 4. Translating laziness

We have defined a useful-looking target language, but we haven’t yet shown how we can produce terms it in from those of a more traditional lazy language. In this section, we present a simple source language that captures the essential features of Haskell, and show how we can translate it into Strict Core.

Figure 8 presents a simple, lazy, explicitly-typed source language, a kind of featherweight Haskell, or FH. It is designed to be a suitable target language for the desugaring of programs written in Haskell, and is deliberately similar to GHCs current intermediate language (which we call Core). Due to space constraints, we omit the type rules and dynamic semantics for this language – suffice to
say that they are perfectly standard for a typed lambda calculus like
System $\lambda\omega$ [6].

4.1 Type translation

The translation from FH to Strict Core types is given by Figure 9. The
principal interesting feature of the translation is the way it deals
with function types. Function arguments are thunked, reflecting the
call-by-need semantics of application in FH, but result types are
left unthunked. This means that after being fully applied, functions
eagerly evaluate to get their result. If a use-site of that function
wants to delay the evaluation of the application it must explicitly
create a thunk.

Furthermore, both $\forall$ and function types translate to 1-ary func-
tions returning a 1-ary result in Strict Core.

4.2 Term translation

The translation from FH terms to those in Strict Core becomes
almost inevitable given our choice for the type translation, and is
given by Figure 10. It satisfies the invariant:

$$\forall \sigma, \tau. \forall e : \tau \vdash e : \sigma \Rightarrow \forall \sigma, \tau. \forall [\tau] \vdash [e] : ([\sigma])$$

The translation makes extensive use of our syntaxic sugar and
ability to write non-ANFed terms, because the translation to Strict

Core$_{\text{AF}}$ is highly verbose. For example, the translation for applica-
tions into Strict Core$_{\text{AF}}$ would look like this:

$$[e_1 \ e_2] = \text{let } \ell = [e_1] \text{ in } \text{valrec } x = \lambda \ell . [e_2] \text{ in } f (x)$$

The job of the term translation is to add explicit thunks to the
Strict Core output wherever we had implicit laziness in the FH
input program. To this end, we add thunks around the result of the
translation in “lazy” positions – namely, arguments to applications
and in the right hand side of let bindings. Dually, when we need to
access a variable, it must have been the case that the binding
site for the variable caused it to be thunked, and hence we need to
explicitly force variable accesses by applying them to $\ell$.

Bearing all this in mind, here is the translation for a simple
application of a polymorphic identity function to 1:

$$[\langle \Lambda x : \alpha. \ x \ : \alpha \rangle \ 1] \ = \ \langle \Lambda x : \alpha. \ x \ : \alpha \rangle \ 1$$

4.3 Data type translation

In any translation from FH to Strict Core we must account for
(a) the translation of data type declarations themselves, (b) the
translation of constructor applications, and (c) the translation of
pattern matching. We begin with (a), using the following FH data
type declaration for lists:

$$\text{List} ::= \text{Nil} | \text{Cons} \alpha (\text{List} \alpha)$$

The translation $D$, shown in Figure 11 yields this Strict Core de-
claration:

$$\text{data} \ \text{List} \ 
\alpha ::= \ 
\text{Nil} | \text{Cons} \ 
\langle \alpha \rangle, \ 
\langle \text{List} \alpha \rangle$$
\[
D[d]\]

\[
D \{\text{data } T \alpha_1 \ldots \alpha_n = C_1 \tau_1 \ldots \tau_n\} = \text{data } T \pi_{\alpha_1 \ldots \alpha_n} = C_1 \{\tau_1\} \ldots \{\tau_n\}
\]

\[
W[d]\]

\[
W \{\text{data } T \pi_{\alpha_1 \ldots \alpha_n} = C_1 \tau_1^{\alpha_1} \ldots \tau_n^{\alpha_n}\}
\]

\[
\{\tau, e\} = D[d], \text{valrec } W[d] \text{ in } [e]
\]

Figure 11: Translation from FH to Strict Core programs

The arguments are thunked, as you would expect, but the constructor is given an uncurried type of (value) arity 2. So the types of the data constructor Cons before and after translation are:

FH Cons : \forall \alpha. \alpha \rightarrow \text{List } \alpha \rightarrow \text{List } \alpha

Strict Core Cons : \{\alpha : \tau, \{\alpha\} \rightarrow \{\text{List } \alpha\}\} \rightarrow \{\text{List } \alpha\}

We give Strict Core data constructors an uncurried type to reflect their status as expressing the built-in notions of allocation and pattern matching (Figure 7). However, since the type of Strict-Core Cons is not simply the translation of the type of the FH Cons, we define a top-level wrapper function Cons\textsuperscript{wrap} which does have the right type:

\[
\text{Cons}\textsuperscript{wrap} = \lambda \alpha. : \tau. \lambda x : \{\alpha\}. \lambda xs : \{\text{List } \alpha\}. \text{Cons } \{\alpha, x, xs\}
\]

Now, as Figure 10 shows, we translate a call of a data constructor C to a call of C\textsuperscript{wrap}. (As an optimisation, we refrain from thunking the definition of the wrapper and forcing its uses, which accounts for the different treatment of C and x in Figure 10.) We expect that the wrappers will be inlined into the program by an optimisation pass, exposing the more efficient calling convention at the original data constructor use site.

The final part of the story is the translation of pattern matching. This is also given in Figure 10 and is fairly straightforward once you remember that the types of the bound variables must be thunked to reflect the change to the type of the data constructor functions.

Finally, the translation for programs, also given in Figure 11, ties everything together by using both the data types and expression translations.

4.4 The seq function

A nice feature of Strict Core\textsubscript{ANF} is that it is possible to give a straightforward definition of the primitive seq function of Haskell:

\[
\text{seq} : \{\alpha : *, \beta : \to \{\alpha\} \to \{\beta\} \to \beta\} = \lambda \alpha : *, \lambda \beta : *, \lambda x : \{\alpha\}. \lambda y : \{\beta\}. \text{let } \gamma : \alpha \to \gamma \text{ in } y \gamma
\]

5. Putting Strict Core to work

In this section we concentrate on how the features of Strict Core can be of aid to an optimising compiler that uses it as an intermediate language. These optimisations all exploit the additional operational information available from the types-as-calling-conventions correspondence in order to improve the efficiency of generated code.

5.1 Routine optimisations

Strict Core has a number of equational laws that have applications to program optimisation. We present a few of them in Figure 12.

The examples we present in this section will usually already have had these equational laws applied to them, if the rewrite represents an improvement in their efficiency or readability. For an example of how they can improve programs, notice that in the translation we give from FH, variable access in a lazy context (such as the argument of an application) results in a redundant thunking and forcing operation. We can remove that by applying the \(\eta\) law:

\[
[f y] = [f] (\lambda () \cdot [y]) = f () (\lambda () \cdot y ()) = f () (y)
\]

5.2 Expressing the calling convention for strict arguments

Let’s go back to the first example of a strict function from Section 1:

\[
f :: \text{Bool} \to \text{Int}
f x = \text{case } x \text{ of } True \to \ldots ; \text{False } \to \ldots
\]

We claimed that we could not, while generating the code for \(f\), assume that the \(x\) argument was already evaluated, because that is a fragile property that would be tricky to guarantee for all call-sites. In Strict Core, the evaluated/non-evaluated distinction is apparent in the type system, so the property becomes robust. Specifically, we can use the standard worker/wrapper transformation [7, 8] to \(f\) as follows:

\[
f\text{weak} : \text{Bool} \to \text{Int}
f\text{weak} = \lambda x : \text{Bool}. \text{case } x \text{ of } True \to \ldots ; \text{False } \to \ldots
\]

\[
f : \{\text{Bool}\} \to \text{Int}
f = \lambda x : \{\text{Bool}\}. f\text{weak } x ()
\]

Here the worker \(f\text{weak}\) takes a definitely-evaluated argument of type \(\text{Bool}\), while the wrapper \(f\) takes a lazy argument and forces it before calling \(f\). By inlining the \(f\) wrapper selectively, we will often be able to avoid the forcing operation altogether, by cancelling it with explicit thunk creation. Because every lifted (i.e. lazy) type in Strict Core has an unlifted (i.e. strict) equivalent, we are able to express all of the strictness information resulting from strictness analysis by a program transformation in this style. This is unlike the situation in GHC today, where we can only do this for product types; in particular, strict arguments with sum types such as \(\text{Bool}\) have their strictness information applied in a much more ad-hoc manner.

We suggested in Section 2 that this notion could be used to improve the desugaring of dictionary arguments. At this point, the approach should be clear: during desugaring of Haskell into Strict Core, dictionary arguments should not be wrapped in explicit thunks, ever. This entirely avoids the overhead of evaluatedness checking for such arguments.

5.3 Exploiting the multiple-result calling convention

Our function types have first-class support for multiple arguments and results, so we can express the optimisation enabled by a constructed product result (CPR) analysis [9] directly. For example, translating splitList from Section 2.4 into Strict Core yields the following program:

\[
\text{splitList} = \{\lambda xs : \{\text{List } \text{Int}\}. \text{case } xs \to (\ldots) \{\text{Int, List } \text{Int, } y, ys\}\}
\]

Here we assume that we have translated the FH pair type in the standard way to the following Strict Core definition:

\[
\text{data } (_, \alpha : *, \beta : *) = (\ldots) \{\alpha\}, \{\beta\}\}
\[ \beta \]
\[ \eta \]
\[ \text{let} \]
\[ \text{let} \]
\[ \text{valrec} \]
\[ \text{case-constructor-elim} \]
\[ \text{case-literal-elim} \]

The translation of this program into Strict Core will introduce a wholly unnecessary thunk around \( xs \), thus

\[ \text{valrec} \{ \text{List Int} \} = \{ \text{Cons \{ Int, y, ys \} } \] 

It is obviously stupid to build a thunk for something that is already a value, so we would prefer to see

\[ \text{valrec} \{ \text{List Int} = \text{Cons \{ Int, y, ys \} } \] 

but now references to \( xs \) in the body of the \( \text{valrec} \) will be badly-typed! As usual, we can solve the impendence mismatch by adding an auxiliary definition:

\[ \text{valrec} \{ \text{List Int} = \text{Cons \{ Int, y, ys \} } \]
\[ \text{valrec} \{ \text{List Int} \} = \{ \text{xs'} } \]

Indeed, if you think of what this transformation would look like in \( \text{Strict Core}_{\text{ANF}} \), it amounts to floating a \( \text{valrec} \) for \( xs' \) out of a thunk, a transformation that is widely useful [10]. Now, several optimisations suggest themselves:

- We can inline \( xs \) freely at sites where it is forced, thus \( (xs \, \emptyset) \), which then simplifies to just \( xs' \).

- Operationally, the thunk \( \lambda \emptyset \cdot xs' \) behaves just like \( \text{IND} \, xs' \), except that the former requires an update (Figure 7). So it would be natural for the code generator to allocate an \( \text{IND} \) directly for a nullary lambda that returns immediately.

- GHC’s existing runtime representation goes even further: since every heap object needs a header word to guide the garbage collector, it costs nothing to allow an evaluated \( \text{Int} \) to be enterable. In effect, a heap object of type \( \text{Int} \) can also be used to represent a value of type \( \{ \text{Int} \} \), an idea we call \( \text{auto-lifting} \). That in turn means that the binding for \( xs \) generates literally no code at all – we simply use \( xs' \) where \( xs \) is mentioned.

One complication is that \( \text{thunks cannot be auto-lifted} \). Consider this program:

\[ \text{valrec} \{ \text{Int} \} = \{ \perp \}
\[ \text{valrec} \{ \{ \text{Int} \} \} = \{ f \}
\[ g \emptyset \]

Clearly, the program should terminate. However if we adopt-auto lifting for thunks then at runtime \( g \) and \( f \) will alias and hence we will cause the evaluation of \( \perp ! \) So we must restrict auto-lifting to thunks of non-polymorphic, non-thunk types. (Another alternative would be to restrict the kind system so that thunks of thunks and instantiation of type variables with thunk types is disallowed, which might be an acceptable tradeoff.)

5.6 Deep unboxing

Another interesting possibility for optimisation in Strict Core is the exploitation of “deep” strictness information by using \( n \)-ary thunks to remove some heap allocated values (a process known as \( \text{unboxing} \)). What we mean by this is best understood by example:

<table>
<thead>
<tr>
<th>( \beta )</th>
<th>( \eta )</th>
<th>( \text{let} )</th>
<th>( \text{let} )</th>
<th>( \text{valrec} )</th>
<th>( \text{valrec} )</th>
<th>( \text{case-constructor-elim} )</th>
<th>( \text{case-literal-elim} )</th>
</tr>
</thead>
</table>
| \( \lambda x. \, \text{case} \, x \, \emptyset \) of | \( R \rightarrow \ldots \) | \( \ldots (\text{case} \, x \, \emptyset \) of | \( G \rightarrow \ldots \) | \( B \rightarrow \ldots \) \ldots \)
| \( \lambda x. \, \text{let} \, x' = x \emptyset \) in | \( \text{case} \, x' \) of | \( R \rightarrow \ldots \) | \( \ldots (\text{case} \, x' \) of | \( G \rightarrow \ldots \) | \( B \rightarrow \ldots \) \ldots \)

This stands in contrast to GHC today, where an ad-hoc mechanism tries to discover opportunities for exactly this optimisation.
6. Arity raising

Finally, we move on to two optimisations that are designed to improve function arity – one that improves arity at a function by examining how the function is defined, and one that realises an improvement by considering how it is used. These optimisations are critical to ameliorating the argument-at-a-time worst case for applications that occurs in the output of the naïve translation from FH. GHC does some of these arity-related optimisations in an ad-hoc way already; the contribution here is to make them more systematic and robust.

6.1 Definition-site arity raising

Consider the following Strict Core binding:

\[ \text{valrec } f : \{ (\{ \text{Int} \}, \{ \text{Int} \}) \} \rightarrow \text{Int} = \lambda (p t : \{ (\{ \text{Int} \}, \{ \text{Int} \}) \}) \].

This code is a perfect target for one of the optimisations that Strict Core lets us express cleanly: definition-site arity raising. Observe that currently callers of \( f \) are forced to apply it to its arguments one at a time. Why couldn’t we change the function so that it takes both of its arguments at the same time?

We can realise the arity improvement for \( f \) by using, once again, a worker/wrapper transformation. The wrapper, which we give this the original function name, \( f \), simply does the arity adaptation before calling into a worker. The worker, which we call \( f_{\text{work}} \), is then responsible for the rest of the calculation of the function\(^4\):

\[ \text{valrec } f : \text{Int} \rightarrow \text{Int} = \lambda x : \text{Int}. \lambda y : \text{Int}. e \text{ in } f 1 2 \]

At this point, no improvement has yet occurred – indeed, we will have made the program worse by adding a layer of indirection via the wrapper! However, once the wrapper is vigorously inlined at the call sites by the compiler, it will often be the case that the wrapper will cancel with work done at the call site, leading to a considerable efficiency improvement:

\[ \text{valrec } f_{\text{work}} : \langle \text{Int}, \text{Int} \rangle \rightarrow \text{Int} = \lambda (x : \text{Int}, y : \text{Int}). e \text{ in } f_{\text{work}} 1 2 \]

This is doubly true in the case of recursive functions, because by performing the worker/wrapper split and then inlining the wrapper into the recursive call position, we remove the need to heap-allocate a number of intermediate function closures representing partial applications in a loop.

Although this transformation can be a big win, we have to be a bit careful about where we apply it. The ability to apply arguments one at a time to a curried function really makes a difference to efficiency sometimes, because call-by-need (as opposed to call-by-name) semantics allows work to be shared between several invocations of the same partial application. To see how this works, consider this Strict Core program fragment:

\[ \text{valrec } g : \text{Int} \rightarrow \text{Int} = \langle (\lambda x : \text{Int}. \text{let } s = \text{fibonacci } x \text{ in } \lambda y : \text{Int}. \ldots ) \rangle \text{ in } h \]

Because we share the partial application of \( g \) (by naming it \( h \)), we will only compute the application \( \text{fibonacci } 5 \) once. However, if we were to “improve” the arity of \( g \) by turning it into a function of type \( \langle \text{Int}, \text{Int} \rangle \rightarrow \text{Int} \), then it would simply be impossible to express the desired sharing! Loss of sharing can easily outweigh the benefits of a more efficient calling convention.

Identifying some common cases where no significant sharing would be lost by increasing the arity is not hard, however. In particular, unlike \( g \), it is safe to increase the arity of \( f \) to 2, because \( f \) does no work (except allocate function closures) when applied to fewer than 2 arguments. Another interesting case where we might consider raising the arity is where the potentially-shared work done by a partial application is, in some sense, cheap – for example, if the sharable expressions between the \( \lambda s \) just consist of a bounded number of primitive operations. We do not attempt to present a suitable arity analysis in this paper; our point is only that Strict Core gives a sufficiently expressive medium to express its results.

6.2 Use-site arity raising

This is, however, not the end of the story as far as arity raising is concerned. If we can see all the call-sites for a function, and none of the call sites share partial applications of less than \( n \) arguments, then it is perfectly safe to increase the arity of that function to \( n \), regardless of whether or not the function does work that is worth sharing if you apply fewer than \( n \) arguments. For example, consider function \( g \) from the previous sub-section, and suppose the the body of its \( \text{valrec } \) was \( \ldots (g \, p \, q) \ldots (g \, r \, s) \ldots \); that is, every call to \( g \) has two arguments. Then no sharing is lost by performing arity raising on its definition, but considerable efficiency is gained.

This transformation not only applies to \( \text{valrec } \) bound functions, but also to uses of higher-order functional arguments. After translation of the \( \text{zipWith} \) function from Section 2.2 into Strict Core, followed by discovery of its strictness and definition-site arity properties, the worker portion of the function that remains might look like the following:

\[ \text{valrec } f_{\text{work}} : \langle \text{Int}, \text{Int} \rangle \rightarrow \text{Int} = \lambda (x : \text{Int}, y : \text{Int}). e \text{ in } f_{\text{work}} 1 2 \]
Notice that if only ever applied in the body to three arguments at a time – \( \langle \rangle \), \( x \) and \( y \) (or rather \( x \) and \( y \)). Based on this observation, we could re-factor \( \text{zipWith} \) so that it applied its function argument to all these arguments (namely \( \langle x, y \rangle \)) at once. The resulting wrapper would look like this (omitting a few types for clarity):

\[
\text{valrec} \ \text{zipWith}: \langle a:*, b:*, c:*, \{\{a\} \rightarrow \{b\} \rightarrow c\}, de L_2 \text{ language} [14] \text{ which also explored the possibility of an optimising compiler suitable for both strict and lazy languages. We share with } L_2 \text{ an explicit representation of thunking and forcing operations, but take this further by additionally representing the operational notions of unboxing (through multiple function results) and arity. The } L_2 \text{ language shares with the MIL the fact that it makes an attempt to support } \text{impure} \text{ strict languages, which we do not – though impure operations could potentially be desugared into our intermediate language using a state-token or continuation passing style to serialize execution.}

The IL language [16] represents thunks explicitly by way of continuations with a logical interpretation, and is to our knowledge the first time that auto-lifting is discussed in the literature. It seems likely that some way could be found to adapt the logic based approach of this paper to accommodate a treatment of arity and multiple-value expressions, as long as some way is adopted to distinguish between “boxed” and “unboxed” uses of the \( \land \) tuple type formation rule.

Hannan and Hicks have previously introduced the arity use-site optimization under the name “higher-order uncurrying” [17] as a type-directed analysis on a source language. They also separately introduced an optimisation called “higher-order arity raising” [18] which attempts to unpack tuple arguments where possible – this is a generalisation of the existing worker/wrapper transformations GHCl currently does for strict product parameters. However, their analyses only consider a strict language, and in the case of uncurrying does not try to distinguish between cheap and expensive computation in the manner we propose above. Leroy et al. [19] demonstrated a verified version of the framework which operates by coercion insertion, which is similar to our worker/wrapper approach.

7. Related work

Benton et al’s Monadic Intermediate Language (MIL) [11] is similar to our proposed intermediate language. The MIL included both \( n \)-ary lambdas and multiple returns from a function, but lacked a treatment of thunks due to aiming to compile a strict language. MIL also included a sophisticated type system that annotated the return type of functions with potential computational effects, including divergence. This information could be used to ensure the soundness of arity-changing transformations – i.e. uncurrying is only sound if a partial application has no computational effects.

Both MIL and the Bigloo Scheme compiler [12] (which could express \( n \)-ary \emph{lambdas}, included versions of what we have called arity definition-site analysis. However, the MIL paper does not seem to consider the work-duplication issues involved in the arity raising transformation, and the Bigloo analysis was fairly simple minded – it only coalesced manifestly adjacent \emph{lambdas}, without allowing (for example) potentially shareable work to be duplicated as long as it was cheap. We think that both of these issues deserve a more thorough investigation. A simple arity definition-site analysis is used by SML/NJ [13], though the introduction of \( n \)-ary arguments is done by a separate argument flattening pass later on in the compiler rather than being made immediately manifest.

In MIL, function application used purely static arity information. Bigloo used a hybrid static/dynamic arity dispatch scheme, but unfortunately do not appear to report on the cost (or otherwise) of operating purely using static arity information.

The intermediate language discussed here is in some ways an extension an the \( L_2 \) language [14] which also explored the possibility of an optimising compiler suitable for both strict and lazy languages. We share with \( L_2 \) an explicit representation of thunking and forcing operations, but take this further by additionally representing the operational notions of unboxing (through multiple function results) and arity. The \( L_2 \) language shares with the MIL the fact that it makes an attempt to support \emph{impure} strict languages, which we do not – though impure operations could potentially be desugared into our intermediate language using a state-token or continuation passing style to serialize execution.

The IL language [16] represents thunks explicitly by way of continuations with a logical interpretation, and is to our knowledge the first time that auto-lifting is discussed in the literature. It seems likely that some way could be found to adapt the logic based approach of this paper to accommodate a treatment of arity and multiple-value expressions, as long as some way is adopted to distinguish between “boxed” and “unboxed” uses of the \( \land \) tuple type formation rule.

Hannan and Hicks have previously introduced the arity use-site optimization under the name “higher-order uncurrying” [17] as a type-directed analysis on a source language. They also separately introduced an optimisation called “higher-order arity raising” [18] which attempts to unpack tuple arguments where possible – this is a generalisation of the existing worker/wrapper transformations GHCl currently does for strict product parameters. However, their analyses only consider a strict language, and in the case of uncurrying does not try to distinguish between cheap and expensive computation in the manner we propose above. Leroy et al. [19] demonstrated a verified version of the framework which operates by coercion insertion, which is similar to our worker/wrapper approach.
8. Conclusions and further work

In this paper we have described what we believe to be a interesting point in the design space of compiler intermediate languages. By making information about a function’s calling convention totally explicit in the intermediate language type system, we expose it to the optimiser – in particular we allow optimisation of decisions about function arity. A novel concept – n-ary thunks – arose naturally from the process of making calling convention explicit, and this in turn allows at least one novel and previously-inexpressible optimisation (deep unboxing) to be expressed.

This lazy $\lambda$-calculus FH we present is similar to System FC, GHC’s current intermediate language. For a long time, a lazy language was, to us at least, the obvious intermediate language for a lazy source language such as Haskell – so it was rather surprising to discover that an appropriately-chosen strict calculus seems to be in many ways better suited to the task!

However, it still remains to implement the language in GHC and gain practical experience with it. In particular, we would like to obtain some quantitative evidence as to whether purely static arity dispatch leads to improved runtimes compared to a dynamic consideration of the arity of a function such as GHC implements at the moment. A related issue is pinning down the exact details of how a hybrid dynamic/static dispatch scheme would work, and how to implement it without causing code bloat from the extra checks. We anticipate that we can reuse existing technology from our experience with the STG machine [20] to do this.

Although we have presented, by way of examples, a number of compiler optimisations that are enabled or put on a firmer footing by the use of the new intermediate language, we have not provided any details about how a compiler would algorithmically decide when and how to apply them. In particular, we plan to write a paper fully elucidating the details of the two arity optimisations (Section 6.2 and Section 6.1) in a lazy language and reporting on our practical experience of their effectiveness.

There are a number of interesting extensions to the intermediate language that would allow us to express even more optimisations. We are particularly interested in the possibility of using some features of the $\Pi\Sigma$ language [21] to allow us to express even more optimisations in a typed manner. In particular, adding unboxed $\Sigma$ types would address an asymmetry between function argument and result types in Strict Core – binders may not appear to the right of a function arrow currently. They would also allow us to express unboxed existential data types (including function closures, should we wish) and GADTs. Another $\Pi\Sigma$ feature – types that can depend on "tags" – would allow us to express unboxed sum types, but the implications of this feature for the garbage collector are not clear.

We would like to expose the ability to use "strict" types to the compiler user, so Haskell programs can, for example, manipulate lists of strict integers ($[!\text{Int}]$). Although it is easy to express such things in the Strict Core language, it is not obvious how to go about exposing this ability in the source language in a systematic way – work on this is ongoing.

9. Acknowledgements

This work was partly supported by a PhD studentship generously provided by Microsoft Research. We would like to thank Paul Blain Levy for the thought provoking talks and discussions he gave while visiting the University of Cambridge which inspired this work. Thanks are also due to Duncan Coutts, Simon Marlow, Alan Mycroft and Dominic Orchard for their helpful comments and suggestions.

References