Types Are Calling Conventions

Maximilian C. Bolingbroke
University of Cambridge
mb566@cam.ac.uk

Simon L. Peyton Jones
Microsoft Research
simonpj@microsoft.com

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.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory – Semantics; D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages; D.3.4 [*Programming Languages*]: Processors – Optimization

General Terms Languages, Performance

1. Introduction

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

$$f :: Int \to Bool \to (Int, Bool)$$

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 are as 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 optimisations 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

Haskell'09, September 3, 2009, Edinburgh, Scotland, UK.

the imprecise, conservative calling convention implied by the type system as it stands.

2.1 Strict arguments

Consider the following function:

$$f :: Bool \to Int$$
$$f\ x = \mathbf{case}\ x\ \mathbf{of}\ True \to \ldots ; False \to \ldots$$

This function is certainly strict in its argument x. GHC uses this information to generate more efficient code for *calls* to f, using call-by-value to avoid allocating a thunk for the argument. However, when generating the code for the *definition* of f, can we 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

$$map\ f\ [fibonacci\ 10, 1234]$$

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:

$$map\ (\lambda x.\ \mathbf{case}\ x\ \mathbf{of}\ y \to f\ y)\ [\ldots]$$

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 $\mathbf{case}\ x\ \mathbf{of}\ y \to e$ 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 generated evaluated-ness checks for these, without the use of any ad-hocery?

2.2 Multiple arguments

Consider these two functions:

$$f\ x\ y = x + y$$
$$g\ x = \mathbf{let}\ z = factorial\ 10\ \mathbf{in}\ \ \lambda y \to x + y + z$$

They have the same type ($Int \to Int \to Int$), 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 *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 *ad-hoc* arity information; and the latter imposes a performance penalty [2].

[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.

Shorthand		Expansion	
$\overline{x}^n$	$\triangleq$	$\langle x_1, \ldots, x_n \rangle$	$(n \geqslant 0)$
$\overline{x}$	$\triangleq$	$\langle x_1, \ldots, x_n \rangle$	$(n \geqslant 0)$
x	$\triangleq$	$\langle x \rangle$	Singleton
$\overline{x}, \overline{y}$	$\triangleq$	$\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$	Concatenation

Figure 1: Notation for sequences

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

$$\begin{array}{l} zipWith = \lambda f :: (a \to b \to c).\ \lambda xs :: List\ a.\ \lambda ys :: List\ b. \\ \quad \mathbf{case}\ xs\ \mathbf{of} \\ \quad\quad Nil \to Nil \\ \quad\quad (Cons\ x\ xs') \to \\ \quad \mathbf{case}\ ys\ \mathbf{of} \\ \quad\quad Nil \to Nil \\ \quad\quad (Cons\ y\ ys') \to Cons\ (f\ x\ y)\ (zipWith\ f\ xs'\ ys') \end{array}$$

The functional argument f is always applied to two arguments, and it seems a shame that we cannot somehow communicate that information to the functions that are actually given to $zipWith$ so that they might be compiled with a less pessimistic calling convention.

2.3 Optionally-strict source languages

Leaving the issue of compilation aside, Haskell's source-level type system is not expressive enough to encode an important class of invariants about how far an expression has been evaluated. For example, you might like to write a function that produces a list of certainly-evaluated Ints, which we might write as $[!Int]$. We do not attempt to solve the issues of how to expose this functionality to the user in *this* paper, but we make a first step along this road by describing an intermediate language which is able to express such types.

2.4 Multiple results

In a purely functional language like Haskell, there is no direct analogue of a reference parameter, such as you would have in an imperative language like C++. This means that if a function wishes to return multiple results it has to encapsulate them in a data structure of some kind, such as a tuple:

$$splitList :: [Int] \to (Int, [Int])$$
$$splitList\ xs = \mathbf{case}\ xs\ \mathbf{of}\ (y : ys) \to (y, ys)$$

Unfortunately, creating a tuple means that you need to allocate a blob of memory on the heap – and this can be a real performance drag, especially when functions returning multiple results occur in tight loops.

How can we compile functions which – like this one – return multiple results, efficiently?

3. Strict Core

We are now in a position to discuss the details of our proposed compiler intermediate language, which we call Strict Core$_{\text{ANF}}$[2].

Strict Core$_{\text{ANF}}$ makes extensive use of sequences of variables, types, values, and terms, so we pause to establish our notation for sequences. We use angle brackets $\langle x_1, x_2, \ldots, x_n \rangle$ to denote

[2] ANF stands for A-normal form, which will be explained further in Section 3.6

a possibly-empty sequence of n elements. We often abbreviate such a sequence as $\overline{x}^n$ or, where n is unimportant, as $\overline{x}$. When no ambiguity arises we abbreviate the singleton sequence $\langle x \rangle$ to just x. All this notation is summarised in Figure 1.

We also adopt the "variable convention" (that all names are unique) throughout this paper, and assume that whenever the environment is extended, the name added must not already occur in the environment – α-conversion can be used as usual to get around this restriction where necessary.

3.1 Syntax of Strict Core$_{\text{ANF}}$

Strict Core$_{\text{ANF}}$ 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 τ, shown in Figure 2, embodies this idea: a function type takes the form $\overline{b} \to \overline{\tau}$, 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:

$$
\begin{array}{lcl}
\mathit{f1} & : & \mathit{Int} \to \mathit{Int} \\
 & : & \langle \_ : \mathit{Int} \rangle \to \langle \mathit{Int} \rangle \\
\mathit{f2} & : & \langle \alpha : \star, \alpha \rangle \to \alpha \\
 & : & \langle \alpha : \star, \_ : \alpha \rangle \to \langle \alpha \rangle \\
\mathit{f3} & : & \langle \alpha : \star, \mathit{Int}, \alpha \rangle \to \langle \alpha, \mathit{Int} \rangle \\
 & : & \langle \alpha : \star, \_ : \mathit{Int}, \_ : \alpha \rangle \to \langle \alpha, \mathit{Int} \rangle \\
\mathit{f4} & : & \alpha : \star \to \mathit{Int} \to \alpha \to \langle \alpha, \mathit{Int} \rangle \\
 & : & \langle \alpha : \star \rangle \to \langle \langle \_ : \mathit{Int} \rangle \to \langle \langle \_ : \alpha \rangle \to \langle \alpha, \mathit{Int} \rangle \rangle \rangle
\end{array}
$$

In each case, the first line uses simple syntactic abbreviations, which are expanded in the subsequent line. The first, *f1*, takes one argument and returns one result[3]. The second, *f2*, shows a polymorphic function: Strict Core uses the notation of dependent products, in which a single construct (here $\overline{b} \to \overline{\tau}$) subsumes both $\forall$ and function arrow. However Strict Core is not dependently typed, so that types cannot depend on terms: for example, in the type $\langle x : \mathit{Int} \rangle \to \langle \tau \rangle$, the result type τ cannot mention x. For this reason, we always write value binders in types as underscores "_", and usually omit them altogether, writing $\langle \mathit{Int} \rangle \to \langle \tau \rangle$ instead.

The next example, *f3*, illustrates a polymorphic function that takes a type argument and two value arguments, and returns two results. Finally, *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 imperatives. For example, Strict Core$_{\text{ANF}}$ has n-ary application $a\ \overline{g}$; and a function may return multiple results $\overline{a}$. A possibly-recursive collection of *heap values* may be allocated with **valrec**, where a heap value is just a lambda or constructor application. Finally, evaluation is performed by **let**; since the term on the right-hand side may return multiple values, the **let** may bind multiple values. Here, for example, is a possible definition of *f3* above:

$$\mathit{f3} = \lambda \langle \alpha : \star, x : \mathit{Int}, y : \alpha \rangle .\ \langle y, x \rangle$$

In support of the multi-value idea, terms are segregated into three syntactically distinct classes: atoms a, heap values v, and

[3] Recall Figure 1, which abbreviates a singleton sequence $\langle \mathit{Int} \rangle$ to Int

Variables x, y, z

Type Variables α, β

Kinds

κ	::=	$\star$	Kind of constructed types
	$\mid$	$\kappa \to \kappa$	Kind of type constructors

Binders

b	::=	$x : \tau$	Value binding
	$\mid$	$\alpha : \kappa$	Type binding

Types

τ, υ, σ	::=	$\mathbf{T}$	Type constructors
	$\mid$	α	Type variable references
	$\mid$	$\overline{b} \to \overline{\tau}$	Function types
	$\mid$	$\tau\ \upsilon$	Type application

Atoms

a	::=	x	Term variable references
	$\mid$	ℓ	Literals

Atoms In Arguments

g	::=	a	Value arguments
	$\mid$	τ	Type arguments

Multi-value Terms

e	::=	$\overline{a}$	Return multiple values
	$\mid$	**let** $\overline{x : \tau\ =\ e}$ **in** e	Evaluation
	$\mid$	**valrec** $\overline{x : \tau\ =\ v}$ **in** e	Allocation
	$\mid$	$a\ \overline{g}$	Application
	$\mid$	**case** a **of** $\overline{p \to e}$	Branch on values

Heap Allocated Values

v	::=	$\lambda \overline{b}.\ e$	Closures
	$\mid$	$\mathbf{C}\ \overline{\tau}, \overline{a}$	Constructed data

Patterns

p	::=	_	Default case
	$\mid$	ℓ	Matches exact literal value
	$\mid$	$\mathbf{C}\ \overline{x : \tau}$	Matches data constructor

Data Types

d	::=	$\mathbf{data}\ \mathbf{T}\ \overline{\alpha : \kappa} = c \mid \ldots \mid c$	Data declarations
c	::=	$\mathbf{C}\ \overline{\tau}$	Data constructors

Programs $\overline{d}, e$

Typing Environments

Γ	::=	ϵ	Empty environment
	$\mid$	$\Gamma, x : \tau$	Value binding
	$\mid$	$\Gamma, \alpha : \kappa$	Type binding
	$\mid$	$\Gamma, \mathbf{C} : \overline{b} \to \langle \mathbf{T}\ \overline{\alpha} \rangle$	Data constructor binding
	$\mid$	$\Gamma, \mathbf{T} : \kappa$	Type constructor binding

Syntactic sugar

	Shorthand		Expansion
Value binders	τ	$\triangleq$	$\_ : \tau$
Thunk types	$\{\tau_1, \ldots, \tau_n\}$	$\triangleq$	$\langle \rangle \to \langle \tau_1, \ldots, \tau_n \rangle$
Thunk terms	$\{e\}$	$\triangleq$	$\lambda \langle \rangle .\ e$

Figure 2: Syntax of Strict Core$_{\text{ANF}}$

$$\boxed{\Gamma \vdash^{\kappa} \tau \ : \ \kappa}$$

$$\frac{\mathbf{T}{:}\kappa \in \Gamma}{\Gamma \vdash^{\kappa} \mathbf{T} \ : \ \kappa} \ \textsc{TyConData} \qquad \frac{\mathbf{B}(\mathbf{T}) = \kappa}{\Gamma \vdash^{\kappa} \mathbf{T} \ : \ \kappa} \ \textsc{TyConPrim}$$

$$\frac{\alpha{:}\kappa \in \Gamma}{\Gamma \vdash^{\kappa} \alpha \ : \ \kappa} \ \textsc{TyVar} \qquad \frac{\Gamma \vdash \overline{b} \ : \ \Gamma' \quad \forall i.\Gamma' \vdash^{\kappa} \tau_i \ : \ \star}{\Gamma \vdash^{\kappa} \overline{b} \to \overline{\tau} \ : \ \star} \ \textsc{TyFun}$$

$$\frac{\Gamma \vdash^{\kappa} \tau \ : \ \kappa_1 \to \kappa_2 \quad \Gamma \vdash^{\kappa} \upsilon \ : \ \kappa_1}{\Gamma \vdash^{\kappa} \tau \ \upsilon \ : \ \kappa_2} \ \textsc{TyConApp}$$

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

multi-value terms e. An *atom* a is a trivial term – a literal, variable reference, or (in an argument position) a type. A *heap value* υ 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 *multi-value term* (e) is a term that either diverges, or evaluates to several (zero, one, or more) values simultaneously.

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 $\Gamma \vdash e : \overline{\tau}$ types a multi-valued term e, giving it a multi-type $\overline{\tau}$. There are similar judgements for atoms a, and values υ, except that they possess types (not multi-types). An important invariant of Strict Core$_{\text{ANF}}$ is this: variables and values have types τ, not multi-types $\overline{\tau}$. In particular, the environment Γ maps each variable to a *type* τ (not a multi-type).

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

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

- $\mathbf{L}$ Maps literals to their built-in types
- $\mathbf{B}$ Maps built-in type constructors to their kinds – the domain must contain at least all of the type constructors returned by $\mathbf{L}$

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 *configuration* to another. A configuration is given by $\langle H; \ e; \ \Sigma \rangle$, where H represents the heap, e is the term under evaluation, and Σ represents the stack – the syntax of stacks and heaps is given in Figure 6.

We denote the fact that a heap H contains a mapping from x to a heap value υ by $H[x \mapsto \upsilon]$. This stands in contrast to a pattern such as $H, x \mapsto \upsilon$, where we intend that 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 EVAL begins evaluation of a multi-valued term e_1, pushing onto the stack the frame **let** $\overline{x{:}\tau} = \bullet$ **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 RET returns a multiple value to the **let** frame, binding the $\overline{x}$ to the (atomic) returned values $\overline{a}$. In this latter rule, the simultaneous substitution models the idea that e_1 returns multiple values *in registers* to its caller. The static semantics (Section 3.2) guarantees that the number of returned values exactly matches the number of binders.

$$\boxed{\Gamma \vdash_a a \ : \ \tau}$$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash_a x \ : \ \tau} \ \textsc{Var} \qquad \frac{\mathbf{L}(\ell) = \tau}{\Gamma \vdash_a \ell \ : \ \tau} \ \textsc{Lit}$$

$$\boxed{\Gamma \vdash e \ : \ \overline{\tau}}$$

$$\frac{\forall i.\Gamma \vdash_a a_i \ : \ \tau_i}{\Gamma \vdash \overline{a} \ : \ \overline{\tau}} \ \textsc{Multi}$$

$$\frac{\Gamma \vdash e_1 \ : \ \overline{\tau} \quad \Gamma, \overline{x{:}\tau} \vdash e_2 \ : \ \overline{\sigma}}{\Gamma \vdash \textbf{let} \ \overline{x{:}\tau} \ = \ e_1 \ \textbf{in} \ e_2 \ : \ \overline{\sigma}} \ \textsc{Let}$$

$$\frac{\forall j.\Gamma, \overline{x{:}\tau} \vdash_v \upsilon_j \ : \ \tau_j \quad \Gamma, \overline{x{:}\tau} \vdash e_2 \ : \ \overline{\sigma}}{\Gamma \vdash \textbf{valrec} \ \overline{x{:}\tau \ = \ \upsilon} \ \textbf{in} \ e_2 \ : \ \overline{\sigma}} \ \textsc{ValRec}$$

$$\frac{\Gamma \vdash_a a \ : \ \overline{b} \to \overline{\tau} \quad \Gamma \vdash_{\text{app}} \overline{b} \to \overline{\tau} \ @ \ \overline{g} \ : \ \overline{\upsilon}}{\Gamma \vdash a \ \overline{g} \ : \ \overline{\upsilon}} \ \textsc{App}$$

$$\frac{\Gamma \vdash_a a \ : \ \tau_{\text{scrut}} \quad \forall i.\Gamma \vdash_{\text{alt}} p_i \ \to \ e_i \ : \ \tau_{\text{scrut}} \Rightarrow \overline{\tau}}{\Gamma \vdash \textbf{case} \ a \ \textbf{of} \ \overline{p \ \to \ e} \ : \ \overline{\tau}} \ \textsc{Case}$$

$$\boxed{\Gamma \vdash_v \upsilon \ : \ \tau}$$

$$\frac{\Gamma \vdash \overline{b} \ : \ \Gamma' \quad \Gamma' \vdash e \ : \ \overline{\tau}}{\Gamma \vdash_v \lambda \overline{b}.e \ : \ \overline{b} \to \overline{\tau}} \ \textsc{Lam}$$

$$\frac{\mathbf{C}{:}\overline{b} \to \langle \mathbf{T} \ \overline{\alpha} \rangle \in \Gamma \quad \Gamma \vdash_{\text{app}} \overline{b} \to \langle \mathbf{T} \ \overline{\alpha} \rangle \ @ \ \overline{\tau}, \overline{a} \ : \ \langle \upsilon \rangle}{\Gamma \vdash_v \mathbf{C} \ \overline{\tau}, \overline{a} \ : \ \upsilon} \ \textsc{Data}$$

$$\boxed{\Gamma \vdash_{\text{alt}} p \ \to \ e \ : \ \tau_{\text{scrut}} \Rightarrow \overline{\tau}}$$

$$\frac{\Gamma \vdash e \ : \ \overline{\tau}}{\Gamma \vdash_{\text{alt}} \_ \ \to \ e \ : \ \tau_{\text{scrut}} \Rightarrow \overline{\tau}} \ \textsc{DefAlt}$$

$$\frac{\mathbf{L}(\ell) = \tau_{\text{scrut}} \quad \Gamma \vdash e \ : \ \overline{\tau}}{\Gamma \vdash_{\text{alt}} \ell \ \to \ e \ : \ \tau_{\text{scrut}} \Rightarrow \overline{\tau}} \ \textsc{LitAlt}$$

$$\frac{\Gamma, \overline{x{:}\tau} \vdash_v \mathbf{C} \ \overline{\sigma}, \overline{x} \ : \ \langle \mathbf{T} \ \overline{\sigma} \rangle \quad \Gamma, \overline{x{:}\tau} \vdash e \ : \ \overline{\tau}}{\Gamma \vdash_{\text{alt}} \mathbf{C} \ \overline{x{:}\tau} \ \to \ e \ : \ \mathbf{T} \ \overline{\sigma} \Rightarrow \overline{\tau}} \ \textsc{ConAlt}$$

$$\boxed{\Gamma \vdash d \ : \ \Gamma}$$

$$\frac{\Gamma_0 = \Gamma, \mathbf{T}{:}\kappa_1 \to \ldots \to \kappa_m \to \star \quad \forall i.\Gamma_{i-1} \vdash c_i \ : \ \mathbf{T} \ \overline{\alpha{:}\kappa}^m \ \textbf{in} \ \Gamma_i}{\Gamma \vdash \textbf{data} \ \mathbf{T} \ \overline{\alpha{:}\kappa}^m = c_1 \mid \ldots \mid c_n \ : \ \Gamma_n} \ \textsc{DataDecl}$$

$$\boxed{\Gamma \vdash c \ : \ \mathbf{T} \ \overline{\alpha{:}\kappa} \ \textbf{in} \ \Gamma}$$

$$\frac{\forall i.\Gamma \vdash^{\kappa} \tau_i \ : \ \star}{\Gamma \vdash \mathbf{C} \ \overline{\tau} \ : \ \mathbf{T} \ \overline{\alpha{:}\kappa} \ \textbf{in} \ (\Gamma, \mathbf{C}{:}\overline{\alpha{:}\kappa}, \overline{\tau} \to \langle \mathbf{T} \ \overline{\alpha} \rangle)} \ \textsc{DataCon}$$

$$\boxed{\vdash \overline{d}, e \ : \ \tau}$$

$$\frac{\Gamma_0 = \epsilon \quad \forall i.\Gamma_{i-1} \vdash d_i \ : \ \Gamma_i \quad \Gamma_n \vdash e \ : \ \tau}{\vdash \overline{d}^n, e \ : \ \tau} \ \textsc{Program}$$

Figure 4: Typing rules for Strict Core$_{\text{ANF}}$

$$
\begin{array}{lrcll}
\textsc{Eval} & \langle H;\ \textbf{let}\ \overline{x{:}\tau} = e_1\ \textbf{in}\ e_2;\ \Sigma\rangle & \rightsquigarrow & \langle H;\ e_1;\ \textbf{let}\ \overline{x{:}\tau} = \bullet\ \textbf{in}\ e_2.\ \Sigma\rangle & \\
\textsc{Ret} & \langle H;\ \overline{a};\ \textbf{let}\ \overline{x{:}\tau} = \bullet\ \textbf{in}\ e_2.\ \Sigma\rangle & \rightsquigarrow & \left\langle H;\ e_2[\overline{a/x}];\ \Sigma\right\rangle & \\
\textsc{Alloc} & \langle H;\ \textbf{valrec}\ \overline{x{:}\tau = v}\ \textbf{in}\ e;\ \Sigma\rangle & \rightsquigarrow & \left\langle H, \overline{y \mapsto v[y/x]};\ e[\overline{y/x}];\ \Sigma\right\rangle & \overline{y} \notin dom(H) \\
\textsc{Beta} & \left\langle H[x \mapsto \lambda\overline{b}^n.\, e];\ x\ \overline{a}^n;\ \Sigma\right\rangle & \rightsquigarrow & \left\langle H;\ e[\overline{a/b}^n];\ \Sigma\right\rangle & (n > 0) \\
\textsc{Enter} & \langle H, x \mapsto \lambda\langle\rangle\,.\, e;\ x\ \langle\rangle;\ \Sigma\rangle & \rightsquigarrow & \langle H, x \mapsto \lightning;\ e;\ \textbf{update}\ x.\ \Sigma\rangle & \\
\textsc{Update} & \langle H;\ \overline{a};\ \textbf{update}\ x.\ \Sigma\rangle & \rightsquigarrow & \langle H[x \mapsto \mathit{IND}\ \overline{a}];\ \overline{a};\ \Sigma\rangle & \\
\textsc{Ind} & \langle H[x \mapsto \mathit{IND}\ \overline{a}];\ x\ \langle\rangle;\ \Sigma\rangle & \rightsquigarrow & \langle H;\ \overline{a};\ \Sigma\rangle & \\
\textsc{Case-Lit} & \left\langle H;\ \textbf{case}\ \ell\ \textbf{of}\ \overline{\ldots, \ell\ \rightarrow\ e, \ldots};\ \Sigma\right\rangle & \rightsquigarrow & \langle H;\ e;\ \Sigma\rangle & \\
\textsc{Case-Con} & \left\langle H[x \mapsto \mathbf{C}\ \overline{\tau}, \overline{a}^n];\ \textbf{case}\ x\ \textbf{of}\ \ldots, \mathbf{C}\ \overline{b}^n\ \rightarrow\ e, \ldots;\ \Sigma\right\rangle & \rightsquigarrow & \left\langle H;\ e[\overline{a/b}^n];\ \Sigma\right\rangle & \\
\textsc{Case-Def} & \langle H;\ \textbf{case}\ a\ \textbf{of}\ \overline{\ldots, \_\ \rightarrow\ e, \ldots};\ \Sigma\rangle & \rightsquigarrow & \langle H;\ e;\ \Sigma\rangle & \text{If no other match}
\end{array}
$$

Figure 7: Operational semantics of Strict Core$_{\text{ANF}}$

$$\boxed{\Gamma \vdash \overline{b}\ :\ \Gamma}$$

$$\frac{}{\Gamma \vdash \langle\rangle\ :\ \Gamma}\ \textsc{BndrsEmpty} \qquad \frac{\Gamma, \alpha{:}\kappa \vdash \overline{b}\ :\ \Gamma'}{\Gamma \vdash \alpha{:}\kappa, \overline{b}\ :\ \Gamma'}\ \textsc{BndrsTy}$$

$$\frac{\Gamma \vdash^{\kappa} \tau\ :\ \star \quad \Gamma, x{:}\tau \vdash \overline{b}\ :\ \Gamma'}{\Gamma \vdash x{:}\tau, \overline{b}\ :\ \Gamma'}\ \textsc{BndrsVal}$$

$$\boxed{\Gamma \vdash_{\text{app}} \overline{b} \rightarrow \overline{\tau}\ @\ \overline{g}\ :\ \overline{v}}$$

$$\frac{}{\Gamma \vdash_{\text{app}} \langle\rangle \rightarrow \overline{\tau}\ @\ \langle\rangle\ :\ \overline{\tau}}\ \textsc{AppEmpty}$$

$$\frac{\Gamma \vdash_a a\ :\ \sigma \quad \Gamma \vdash_{\text{app}} \overline{b} \rightarrow \overline{\tau}\ @\ \overline{g}\ :\ \overline{v}}{\Gamma \vdash_{\text{app}} (\_{:}\sigma, \overline{b}) \rightarrow \overline{\tau}\ @\ a, \overline{g}\ :\ \overline{v}}\ \textsc{AppVal}$$

$$\frac{\Gamma \vdash^{\kappa} \sigma\ :\ \kappa \quad \Gamma \vdash_{\text{app}} \left(\overline{b} \rightarrow \overline{\tau}\right)[\sigma/\alpha]\ @\ \overline{g}\ :\ \overline{v}}{\Gamma \vdash_{\text{app}} (\alpha{:}\kappa, \overline{b}) \rightarrow \overline{\tau}\ @\ \sigma, \overline{g}\ :\ \overline{v}}\ \textsc{AppTy}$$

Figure 5: Typing rules dealing with multiple abstraction and application

Heap values	h	::=	$\lambda\overline{b}.\ e$	Abstraction
		|	$\mathbf{C}\ \overline{\tau}, \overline{a}$	Constructor
		|	$\mathit{IND}\ \overline{a}$	Indirection
		|	$\lightning$	Black hole
Heaps	H	::=	$\epsilon \mid H, x \mapsto h$	
Stacks	Σ	::=	ϵ	
		|	**update** $x.\ \Sigma$	
		|	**let** $\overline{x{:}\tau} = \bullet$ **in** $e.\ \Sigma$	

Figure 6: Syntax for operational semantics of Strict Core$_{\text{ANF}}$

- Rule 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 is not already used in the heap, and freshen both the $\overline{v}$ and e to reflect this renaming.
- Rule BETA performs β-reduction, by *simultaneously* substituting for all the binders in one step. This simultaneous substitution models the idea of calling a function passing several arguments *in registers*. The static semantics guarantees that the number of arguments at the call site exactly matches what the function is expecting.

Rules CASE-LIT, CASE-CON, and CASE-DEF deal with pattern matching (see Section 3.5); while ENTER, UPDATE, and IND deal with thunks (Section 3.4)

3.4 Thunks

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

If we only cared about call-by-*name*, we could model a thunk as a nullary function (a function binding 0 arguments) with type $\langle\rangle \rightarrow \mathit{Int}$. Then we could thunk a term e by wrapping it in a nullary lambda $\lambda\langle\rangle\,.\,e$, and force a thunk by applying it to $\langle\rangle$. This call-by-name approach would unacceptably lose sharing, but we can readily turn it into call-by-need by treating nullary functions (henceforth called thunks) specially in the operational semantics (Figure 7), which is what we do:

- In rule ENTER, an application of a thunk to $\langle\rangle$ pushes onto the stack a *thunk update frame* mentioning the thunk name. It also overwrites the thunk in the heap with a *black hole* ($\lightning$), to express the fact that entering a thunk twice with no intervening update is always an error [3]. We call all this *entering*, or *forcing*, a thunk.
- When the machine evaluates to a result (a vector of atoms $\overline{a}$), UPDATE overwrites the black hole with an indirection $\mathit{IND}\ \overline{a}$, pops the update frame, and continues as if it had never been there.
- Finally, the IND rule ensures that, should the original thunk be entered to again, the value saved in the indirection is returned directly (remember – the indirection overwrote the pointer to the thunk definition that was in the heap), so that the body of the thunk is evaluated at most once.

We use *thunking* to describe the process of wrapping a term e in a nullary function $\lambda\langle\rangle\,.\,e$. 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 Core$_{\text{ANF}}$ supports *multi-valued* thunks, with a type such as $\langle\rangle \rightarrow \langle \mathit{Int}, \mathit{Bool}\rangle$, or (using our syntactic sugar) $\{\mathit{Int}, \mathit{Bool}\}$. 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 $\langle Int, Bool\rangle \rightarrow Int$ are absent (not used in the body) then the function can be safely transformed to one of type $\langle\rangle \rightarrow Int$, but not one of type 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).

One potential side effect of this, for example, we may detect that the unit is absent in a function of type $\langle()\rangle \rightarrow Int$ and turn it into one of type $\langle\rangle \rightarrow Int$. This might increase memory usage, as the resulting function has its result memoized! Although this is a bit surprising, it is at least not a property peculiar to our intermediate language - this is actually the behaviour of GHC today, and the same issue crops up in other places too – such as when "floating" lets out of lambdas [4].

3.5 Data types

We treat Int and $Char$ as built-in types, with a suitable family of (call-by-value) operations. A value of type $Char$ is an *evaluated* character, not a thunk (ie. like ML, not like Haskell), and similarly Int. To allow a polymorphic function to manipulate values of these built-in types, they must be *boxed* (ie. represented by a heap pointer like every other value). A real implementation, however, might have additional *unboxed* (not heap allocated) types, $Char\#$, $Int\#$, which do not support polymorphism [5], but we ignore these issues here.

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

$$\mathbf{data}\ List\ a{:}* = Nil \mid Cons\ \langle\{a\}, \{List\ a\}\rangle$$

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 **valrec** expression.

The operational semantics of **case** expressions are given in rules CASE-LIT, CASE-CON, and CASE-DEF, which are quite conventional (Figure 7). Notice that, unlike Haskell, **case** does not perform evaluation – that is done by **let** in EVAL. The only subtlety (present in all such calculi) is in rule CASE-CON: the constructor $\mathbf{C}$ must be *applied* to both its type and value arguments, whereas a *pattern match* for $\mathbf{C}$ 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 [6].

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 program:

  $$(\lambda\langle\alpha{:}*, x{:}\alpha\rangle.\ x)\ \langle Int, 1\rangle$$

  The compiler may manifestly see, using purely local information, that it can perform β-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 β-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 Core$_{\text{ANF}}$ which we simply call Strict Core, by making use of the following simple extension to the grammar and type rules:

$$a ::= \ldots \mid e \mid v \qquad \frac{\Gamma \vdash e\ :\ \langle\tau\rangle}{\Gamma \vdash_a e\ :\ \tau}\ \textsc{Sing} \qquad \frac{\Gamma \vdash_v v\ :\ \tau}{\Gamma \vdash_a v\ :\ \tau}\ \textsc{Val}$$

The semantics of the new form of atom are given by a standard ANFing transformation into Strict Core$_{\text{ANF}}$. 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:

$$\begin{array}{lcl} f1 & : & Int \rightarrow Bool \rightarrow (Int, Bool) \\ f2 & : & \langle Int, Bool\rangle \rightarrow (Int, Bool) \\ f3 & : & (Int, Bool) \rightarrow \langle Int, Bool\rangle \\ f4 & : & \langle\{Int\}, Bool\rangle \rightarrow (Int, Bool) \end{array}$$

Here *f1* is a curried function, taking its arguments one at a time; *f2* takes two arguments at once, but returns a heap-allocated pair; *f3* takes a heap-allocated pair and returns two results (presumably in registers); while *f4* takes two arguments at once, but the first is a thunk. In this way, Strict Core$_{\text{ANF}}$ 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 Core$_{\text{ANF}}$ has a simple type-erased counterpart, where type binders in λs, type arguments and heaps values have been dropped. A natural consequence of this erasure is that functions such as $\langle a : *\rangle \rightarrow \langle Int\rangle$ will be converted into thunks (like $\langle\rangle \rightarrow \langle Int\rangle$), so their results will be shared.

4. Translating laziness

We have defined a useful-looking target language, but we haven not yet shown how we can produce terms in it 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

Types

τ, υ, σ	$::=$	$\mathbf{T}$	Type constructors
	$\mid$	α	Type variables
	$\mid$	$\tau \to \tau$	Function types
	$\mid$	$\forall \alpha{:}\kappa.\tau$	Quantification
	$\mid$	$\tau\ \tau$	Type application

Expressions

e	$::=$	ℓ	Unlifted literals
	$\mid$	$\mathbf{C}$	Built-in data constructors
	$\mid$	x	Variables
	$\mid$	$e\ e$	Value application
	$\mid$	$e\ \tau$	Type application
	$\mid$	$\lambda x{:}\tau.\ e$	Functions binding values
	$\mid$	$\Lambda \alpha{:}\kappa.\ e$	Functions binding types
	$\mid$	$\mathbf{let}\ \overline{x{:}\tau\ =\ e}\ \mathbf{in}\ e$	Recursive name binding
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{p\ \to\ e}$	Evaluation and branching

Patterns

p	$::=$	$\_$	Default case / ignores eval. result
	$\mid$	ℓ	Matches exact literal value
	$\mid$	$\mathbf{C}\ \overline{x{:}\tau}$	Matches data constructor

Data Types

d	$::=$	$\mathbf{data}\ \mathbf{T}\ \overline{\alpha{:}\kappa} = c \mid \ldots \mid c$	Data declarations
c	$::=$	$\mathbf{C}\ \overline{\tau}$	Data constructors

Programs $\overline{d}, e$

Figure 8: The FH language

$$\boxed{[\![\tau\ :\ \kappa]\!]\ :\ \kappa}$$

$$\begin{array}{rcl}
[\![\mathbf{T}]\!] & = & \mathbf{T} \\
[\![\alpha]\!] & = & \alpha \\
[\![\tau_1 \to \tau_2]\!] & = & \{[\![\tau_1]\!]\} \to [\![\tau_2]\!] \\
[\![\forall \alpha{:}\kappa.\tau]\!] & = & \alpha{:}\kappa \to [\![\tau]\!] \\
[\![\tau_1\ \tau_2]\!] & = & [\![\tau_1]\!]\ \ [\![\tau_2]\!]
\end{array}$$

Figure 9: Translation from FH to Strict Core types

$$\boxed{[\![e\ :\ \tau]\!]\ :\ \langle [\![\tau]\!] \rangle}$$

$$\begin{array}{rcl}
[\![\ell]\!] & = & \ell \\
[\![\mathbf{C}]\!] & = & \mathbf{C}^{\text{wrap}} \\
[\![x]\!] & = & x\ \langle\rangle \\
[\![e\ \tau]\!] & = & [\![e]\!]\ \ [\![\tau]\!] \\
[\![\Lambda \alpha{:}\kappa.\ e]\!] & = & \lambda \alpha{:}\kappa.\ [\![e]\!] \\
[\![e_1\ e_2]\!] & = & [\![e_1]\!]\ \ \{[\![e_2]\!]\} \\
[\![\lambda x{:}\tau.\ e]\!] & = & \lambda x{:}\{[\![\tau]\!]\}.\ [\![e]\!] \\
[\![\mathbf{let}\ \overline{x{:}\tau = e}\ \mathbf{in}\ e_b]\!] & = & \mathbf{valrec}\ \overline{x{:}\{[\![\tau]\!]\} = \{[\![e]\!]\}}\ \mathbf{in}\ [\![e_b]\!] \\
[\![\mathbf{case}\ e_s\ \mathbf{of}\ \overline{p\ \to\ e}]\!] & = & \mathbf{case}\ [\![e_s]\!]\ \mathbf{of}\ \overline{[\![p]\!]\ \to\ [\![e]\!]}
\end{array}$$

$$\boxed{[\![p]\!]}$$

$$\begin{array}{rcl}
[\![\ell]\!] & = & \ell \\
[\![\mathbf{C}\ \overline{x{:}\tau}]\!] & = & \mathbf{C}\ \overline{x{:}\{[\![\tau]\!]\}} \\
[\![_]\!] & = & _
\end{array}$$

Figure 10: Translation from FH to Strict Core expressions

$$\boxed{\mathcal{D}\ [\![d]\!]}$$

$$\begin{array}{l}
\mathcal{D}\ [\![\mathbf{data}\ \mathbf{T}\ \overline{\alpha{:}\kappa} = \mathbf{C}_1\ \overline{\tau}_1 \mid \ldots \mid \mathbf{C}_n\ \overline{\tau}_n]\!] \\
\quad = \mathbf{data}\ \mathbf{T}\ \overline{\alpha{:}\kappa} = \mathbf{C}_1\ \overline{\{[\![\tau]\!]\}}_1 \mid \ldots \mid \mathbf{C}_n\ \overline{\{[\![\tau]\!]\}}_n
\end{array}$$

$$\boxed{\mathcal{W}\ [\![d]\!]}$$

$$\mathcal{W}\ [\![\mathbf{data}\ \mathbf{T}\ \overline{\alpha{:}\kappa}^r = \mathbf{C}_1\ \overline{\tau}_1^{m_1} \mid \ldots \mid \mathbf{C}_n\ \overline{\tau}_n^{m_n}]\!]$$

$$= \begin{cases}
\ldots \\
\mathbf{C}_k^{\text{wrap}} = \lambda \alpha_1{:}\kappa_1 \ldots \lambda \alpha_r{:}\kappa_r. \\
\qquad \lambda x_1{:}\{[\![\tau_{1,k}]\!]\} \ldots \lambda x_{m_k}{:}\{[\![\tau_{m_k,k}]\!]\}. \\
\qquad \mathbf{C}_k\ (\overline{\alpha}^r, \overline{x}^{m_k}) \\
\ldots
\end{cases}$$

$$\boxed{[\![\overline{d}, e]\!]}$$

$$[\![\overline{d}, e]\!] \quad = \quad \overline{\mathcal{D}\ [\![d]\!]}, \mathbf{valrec}\ \overline{\mathcal{W}\ [\![d]\!]}\ \mathbf{in}\ [\![e]\!]$$

Figure 11: Translation from FH to Strict Core programs

the type rules and dynamic semantics for this language – suffice to say that they are perfectly standard for a typed lambda calculus like System Fω [7].

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. First, the translation makes no use of n-ary types at all: both $\forall$ and function types translate to 1-ary functions returning a 1-ary result.

Second, 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.

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:

$$\overline{x{:}\tau} \vdash_{\text{FH}} e : \upsilon \quad \Longrightarrow \quad \overline{x{:}\{[\![\tau]\!]\}} \vdash [\![e]\!] : \langle [\![\upsilon]\!] \rangle$$

The translation makes extensive use of our syntactic sugar and ability to write non-ANFed terms, because the translation to Strict Core$_{\text{ANF}}$ is highly verbose. For example, the translation for applications into Strict Core$_{\text{ANF}}$ would look like this:

$$\begin{array}{rcl}
[\![e_1\ e_2]\!] & = & \mathbf{let}\ \langle f \rangle = [\![e_1]\!]\ \mathbf{in} \\
& & \mathbf{valrec}\ x = \lambda\, \langle\rangle\, .\ [\![e_2]\!]\ \mathbf{in}\ f\ \langle x \rangle
\end{array}$$

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 $\langle\rangle$.

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

$$[\![(\Lambda \alpha{:}\star.\ \lambda x{:}\alpha.\ x)\ \mathit{Int}\ 1]\!] \quad = \quad (\lambda \alpha{:}\star.\ \lambda x{:}\{\alpha\}.\ x\ \langle\rangle)\ \mathit{Int}\ \{1\}$$

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:

$$\mathbf{data}\ \mathit{List}\ \alpha{:}* = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

The translation $\mathcal{D}$, shown in Figure 11 yields this Strict Core declaration:

$$\mathbf{data}\ \mathit{List}\ \alpha{:}* = \mathit{Nil} \mid \mathit{Cons}\ \langle\{\alpha\}, \{\mathit{List}\ \alpha\}\rangle$$

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:

$$\begin{array}{rl} \text{FH} & \mathit{Cons} : \forall\alpha.\alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha \\ \text{Strict Core} & \mathit{Cons} : \langle \alpha : \star, \{\alpha\}, \{\mathit{List}\ \alpha\}\rangle \to \langle \mathit{List}\ \alpha\rangle \end{array}$$

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 $\mathit{Cons}^{\text{wrap}}$ which *does* have the right type:

$$\mathit{Cons}^{\text{wrap}} = \lambda\alpha{:}*.\ \lambda x{:}\{\alpha\}.\ \lambda xs{:}\{\mathit{List}\ \alpha\}.\ \mathit{Cons}\ \langle\alpha, x, xs\rangle$$

Now, as Figure 10 shows, we translate a call of a data constructor $\mathbf{C}$ to a call of $\mathbf{C}^{\text{wrap}}$. (As an optimisation, we refrain from thunking the definition of the wrapper and forcing its uses, which accounts for the different treatment of $\mathbf{C}$ 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.[4]

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$_{\text{ANF}}$ is that it is possible to give a straightforward definition of the primitive seq function of Haskell:

$$\begin{array}{l} \mathit{seq} : \{\alpha{:}* \to \beta{:}* \to \{\alpha\} \to \{\beta\} \to \beta\} \\ \quad = \{\lambda\alpha{:}*.\ \lambda\beta{:}*.\ \lambda x{:}\{\alpha\}.\ \lambda y{:}\{\beta\}.\ \mathbf{let}\ \_ : \alpha = x\ \langle\rangle\ \mathbf{in}\ y\ \langle\rangle\} \end{array}$$

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 η law:

$$[\![f\ y]\!] = [\![f]\!]\ \ \langle\lambda\langle\rangle\,.\ [\![y]\!]\rangle = f\ \langle\rangle\ \ \langle\lambda\langle\rangle\,.\ y\ \langle\rangle\rangle = f\ \langle\rangle\ \ \langle y\rangle$$

5.2 Expressing the calling convention for strict arguments

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

$$\begin{array}{l} f :: \mathit{Bool} \to \mathit{Int} \\ f\ x = \mathbf{case}\ x\ \mathbf{of}\ \mathit{True} \to \ldots ; \mathit{False} \to \ldots \end{array}$$

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. Specficically, we can use the standard worker/wrapper transformation [8, 9] to f as follows:

$$\begin{array}{l} f_{\text{work}} : \mathit{Bool} \to \mathit{Int} \\ f_{\text{work}} = \lambda x{:}\mathit{Bool}.\ \mathbf{case}\ x\ \mathbf{of}\ \mathit{True}\ \langle\rangle \to \ldots ; \mathit{False}\ \langle\rangle \to \ldots \\ f : \{\mathit{Bool}\} \to \mathit{Int} \\ f = \lambda x{:}\{\mathit{Bool}\}.\ f_{\text{work}}\ \langle x\ \langle\rangle\rangle \end{array}$$

Here the worker f_{work} takes a definitely-evaluated argument of type 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 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 [10] directly. For example, translating $\mathit{splitList}$ from Section 2.4 into Strict Core yields the following program:

$$\begin{array}{l} \mathit{splitList} = \{\lambda xs{:}\{\mathit{List}\ \mathit{Int}\}.\ \mathbf{case}\ xs\ \langle\rangle\ \mathbf{of} \\ \quad \mathit{Cons}\ \langle y{:}\{\mathit{Int}\}, ys{:}\{\mathit{List}\ \mathit{Int}\}\rangle \to (,)\ \langle \mathit{Int}, \mathit{List}\ \mathit{Int}, y, ys\rangle\} \end{array}$$

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

$$\mathbf{data}\ (,)\ \alpha{:}*\ \beta{:}* = (,)\ \langle\{\alpha\}, \{\beta\}\rangle$$

After a worker/wrapper transformation informed by CPR analysis we obtain a version of the function that uses multiple results, like so:

$$\begin{array}{l} \mathit{splitList}_{\text{work}} = \lambda xs{:}\{\mathit{List}\ \mathit{Int}\}.\ \mathbf{case}\ xs\ \langle\rangle\ \mathbf{of} \\ \quad \mathit{Cons}\ \langle y{:}\{\mathit{Int}\}, ys{:}\{\mathit{List}\ \mathit{Int}\}\rangle \to \langle y, ys\rangle \\ \mathit{splitList} = \{\lambda xs{:}\{\mathit{List}\ \mathit{Int}\}. \\ \quad \mathbf{let}\ \langle y{:}\{\mathit{Int}\}, ys{:}\{\mathit{List}\ \mathit{Int}\}\rangle = \mathit{splitList}_{\text{work}}\ xs \\ \quad \mathbf{in}\ (,)\ \langle \mathit{Int}, \mathit{List}\ \mathit{Int}, y, ys\rangle\} \end{array}$$

[4] It is straightforward (albeit it fiddly) to extend this scheme with support for strict fields in data types, which is necessary for full Haskell 98 support.

β	$\textbf{valrec}\ x{:}\tau\ =\ \lambda\overline{b}^n.\ e\ \textbf{in}\ x\ \overline{a}^n$	$=$	$e[\overline{a/b}^n]$
η	$\textbf{valrec}\ x{:}\tau\ =\ \lambda\overline{b}^n.\ y\ \overline{b}^n\ \textbf{in}\ e$	$=$	$\textbf{let}\ \langle x{:}\tau\rangle = \langle y\rangle\ \textbf{in}\ e$
let	$\textbf{let}\ \overline{x{:}\tau}\ =\ \overline{a}\ \textbf{in}\ e$	$=$	$e[\overline{a/x}]$
let-float	$\textbf{let}\ \overline{x{:}\tau}_1 = (\textbf{let}\ \overline{y{:}\sigma}_2 = e_1\ \textbf{in}\ e_2)\ \textbf{in}\ e_3$	$=$	$\textbf{let}\ \overline{y{:}\sigma}_2 = e_1\ \textbf{in let}\ \overline{x{:}\tau}_1\ =\ e_2\ \textbf{in}\ e_3$
valrec-float	$\textbf{let}\ \overline{x{:}\tau} = (\textbf{valrec}\ \overline{y{:}\sigma = e}\ \textbf{in}\ e_2)\ \textbf{in}\ e_3$	$=$	$\textbf{valrec}\ \overline{y{:}\sigma = e}\ \textbf{in let}\ \overline{x{:}\tau}\ =\ e_2\ \textbf{in}\ e_3$
valrec-join	$\textbf{valrec}\ \overline{x{:}\tau = e}\ \textbf{in valrec}\ \overline{y{:}\sigma = e}\ \textbf{in}\ e$	$=$	$\textbf{valrec}\ \overline{x{:}\tau = e}, \overline{y{:}\sigma = e}\ \textbf{in}\ e$
case-constructor-elim	$\textbf{valrec}\ x{:}\tau = \mathbf{C}\ \overline{\tau}, \overline{a}^n\ \textbf{in case}\ x\ \textbf{of}\ \ldots\ \mathbf{C}\ \overline{b}^n\ \rightarrow\ e\ \ldots$	$=$	$\textbf{valrec}\ x{:}\tau = \mathbf{C}\ \overline{\tau}, \overline{a}^n\ \textbf{in}\ e[\overline{a/b}^n]$
case-literal-elim	$\textbf{case}\ \ell\ \textbf{of}\ \ldots\ \ell\ \rightarrow\ e\ \ldots$	$=$	e

Figure 12: Sample equational laws for Strict Core$_{\text{ANF}}$

Once again, inlining the wrapper *splitList* at its call sites can often avoid the heap allocation of the pair $((,))$.

Notice that the worker is a multi-valued function that returns two results. GHC as it stands today has a notion of an "unboxed tuple" type supports multiple return values, but this extension has never fitted neatly into the type system of the intermediate language. Strict Core gives a much more principled treatment of the same concept.

5.4 Redundant evaluation

Consider this program:

$$\begin{array}{l}\textbf{data}\ \mathit{Colour} = R \mid G \mid B\\ f\ x = \textbf{case}\ x\ \textbf{of}\\ \qquad\qquad R \rightarrow \ldots\\ \qquad\qquad \_ \rightarrow\ \ldots(\textbf{case}\ x\ \textbf{of}\ G \rightarrow \ldots; B \rightarrow \ldots)\ldots\end{array}$$

In the innermost **case** expression, we can be certain that x has already been evaluated – and we might like to use this information to generate better code for that inner **case** split, by omitting evaluatedness checks. However, notice that it translates into Strict Core like so:

$$\begin{array}{l}f = \{\lambda x.\ \textbf{case}\ x\ \langle\rangle\ \textbf{of}\\ \qquad\qquad R\ \langle\rangle \rightarrow \ldots\\ \qquad\qquad \_ \rightarrow \ldots(\textbf{case}\ x\ \langle\rangle\ \textbf{of}\ G\ \langle\rangle \rightarrow \ldots\\ \qquad\qquad\qquad\qquad\qquad\qquad B\ \langle\rangle \rightarrow \ldots)\ldots\}\end{array}$$

It is clear that to avoid redundant evaluation of x we can simply apply common-subexpression elimination (CSE) to the program:

$$\begin{array}{l}f = \{\lambda x.\ \textbf{let}\ x' = x\ \langle\rangle\ \textbf{in}\\ \qquad\qquad \textbf{case}\ x'\ \textbf{of}\ R\ \langle\rangle \rightarrow \ldots\\ \qquad\qquad\qquad\qquad \_ \rightarrow \ldots(\textbf{case}\ x'\ \textbf{of}\ G\ \langle\rangle \rightarrow \ldots\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad B\ \langle\rangle \rightarrow \ldots)\ldots\}\end{array}$$

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

5.5 Thunk elimination

There are some situations where delaying evaluation by inserting a thunk just does not seem worth the effort. For example, consider this FH source program:

$$\textbf{let}\ xs{:}\mathit{List}\ \mathit{Int} = \mathit{Cons}\ \mathit{Int}\ y\ ys$$

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

$$\textbf{valrec}\ xs{:}\{\mathit{List}\ \mathit{Int}\} = \{\mathit{Cons}\ \langle \mathit{Int}, y, ys\rangle\}$$

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

$$\textbf{valrec}\ xs{:}\mathit{List}\ \mathit{Int} = \mathit{Cons}\ \langle \mathit{Int}, y, ys\rangle$$

but now references to xs in the body of the **valrec** will be badly-typed! As usual, we can solve the impedence mis-match by adding an auxiliary definition:

$$\begin{array}{l}\textbf{valrec}\ xs'{:}\mathit{List}\ \mathit{Int} = \mathit{Cons}\ \langle \mathit{Int}, y, ys\rangle\ \textbf{in}\\ \textbf{valrec}\ xs{:}\{\mathit{List}\ \mathit{Int}\} = \{xs'\}\end{array}$$

Indeed, if you think of what this transformation would look like in Strict Core$_{\text{ANF}}$, it amounts to floating a **valrec** (for xs') out of a thunk, a transformation that is widely useful [4]. Now, several optimisations suggest themselves:

- We can inline xs freely at sites where it is forced, thus $(xs\ \langle\rangle)$, which then simplifies to just xs'.
- Operationally, the thunk $\lambda\,\langle\rangle\,.\,xs'$ behaves just like $\mathit{IND}\ xs'$, except that the former requires an update (Figure 7). So it would be natural for the code generator to allocate an 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 Int to be enterable. In effect, a heap object of type Int can also be used to represent a value of type $\{\mathit{Int}\}$, an idea we call *auto-lifting*. That in turn means that the binding for xs generates literally no code at all – we simpy use xs' where xs is mentioned.

One complication is that *thunks cannot be auto-lifted*. Consider this program:

$$\textbf{valrec}\ f{:}\{\mathit{Int}\} = \{\bot\}\ \textbf{in valrec}\ g{:}\{\{\mathit{Int}\}\} = \{f\}\ \textbf{in}\ g\ \langle\rangle$$

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 $\bot$! 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 *unboxing*). What we mean by this is best understood by example:

$$\begin{array}{l}\textbf{valrec}\ f{:}\{(\{\mathit{Int}\},\{\mathit{Int}\})\} \rightarrow \mathit{Int}\\ \quad = \lambda\langle pt{:}\{(\{\mathit{Int}\},\{\mathit{Int}\})\}\rangle.\\ \qquad \textbf{valrec}\ c{:}\mathit{Bool} = \ldots\ \textbf{in}\\ \qquad \textbf{case}\ c\ \textbf{of}\ \mathit{True}\ \langle\rangle \rightarrow 1\\ \qquad\qquad\qquad \mathit{False}\ \langle\rangle \rightarrow \textbf{case}\ pt\ \langle\rangle\ \textbf{of}\ (x,y) \rightarrow\\ \qquad\qquad\qquad\qquad\qquad (+)\ \langle x\ \langle\rangle, y\ \langle\rangle\rangle\end{array}$$

Typical strictness analyses will not be able to say definitively that f is strict in pt (even if c is manifestly False!). However, some strictness analysers might be able to tell us that *if* pt *is*

ever evaluated then both of its components certainly are. Taking advantage of this information in a language without explicit thunks would be fiddly at best, but in our intermediate language we can use the worker/wrapper transformation to potentially remove some thunks by adjusting the definition of f like so:

$$\begin{array}{l}\textbf{valrec } f_{\text{work}} : \{Int, Int\} \to Int = \\ \quad \lambda pt' : \{Int, Int\}. \\ \quad\quad \textbf{valrec } c : Bool = \ldots \textbf{ in} \\ \quad\quad\quad \textbf{case } c \textbf{ of } True\ \langle\rangle \to 1 \\ \quad\quad\quad\quad\quad\quad False\ \langle\rangle \to \textbf{let } \langle x' : Int, y' : Int\rangle = pt'\ \langle\rangle \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{in } (+)\ \langle x', y'\rangle, \\ f : \{(\{Int\}, \{Int\})\} \to Int = \\ \quad \lambda(pt : \{(\{Int\}, \{Int\})\}) \to \\ \quad\quad \textbf{valrec } pt' : \{Int, Int\} = \\ \quad\quad\quad \{\ \textbf{case } pt\ \langle\rangle \textbf{ of } (x, y) \to \langle x\ \langle\rangle, y\ \langle\rangle\rangle\} \\ \quad\quad \textbf{in } f_{\text{work}}\ pt'\end{array}$$

Once again, inlining the new wrapper function at the use sites has the potential to cancel with pair and thunk allocation by the callers, avoiding heap allocation and indirection.

Note that the ability to express this translation actually depended on the ability of our new intermediate language to express multi-thunks (Section 3.4) – i.e. thunks that when forced, evaluate to multiple results, without necessarily allocating anything on the heap.

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 naive 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:

$$\textbf{valrec } f : Int \to Int \to Int = \lambda x : Int.\ \lambda y : Int.\ e \textbf{ in } f\ 1\ 2$$

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_{work}, is then responsible for the rest of the calculation of the function[5]:

$$\begin{array}{l}\textbf{valrec } f_{\text{work}} : \langle Int, Int\rangle \to Int = \lambda\langle x : Int, y : Int\rangle.\ e \\ \quad\quad f : Int \to Int \to Int = \lambda x : Int.\ \lambda y : Int.\ f_{\text{work}}\ \langle x, y\rangle \\ \textbf{in } f\ 1\ 2\end{array}$$

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 vigourously 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:

$$\begin{array}{l}\textbf{valrec } f_{\text{work}} : \langle Int, Int\rangle \to Int = \lambda\langle x : Int, y : Int\rangle.\ e \\ \textbf{in } f_{\text{work}}\ \langle 1, 2\rangle\end{array}$$

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:

$$\begin{array}{l}\textbf{valrec } g : Int \to Int \to Int \\ \quad\quad = (\lambda x : Int.\ \textbf{let } s = \mathit{fibonacci}\ x\ \textbf{in} \lambda y : Int.\ \ldots) \textbf{ in} \\ \textbf{let } h : Int \to Int = g\ 5 \textbf{ in } h\ 10 + h\ 20\end{array}$$

Because we share the partial application of g (by naming it h), we will only compute the application $\mathit{fibonacci}\ 5$ once. However, if we were to "improve" the arity of g by turning it into a function of type $\langle Int, Int\rangle \to Int$, 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 λs 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 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 **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 **valrec** bound functions, but also to uses of *higher-order functional arguments*. After translation of the $\mathit{zipWith}$ 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:

$$\begin{array}{l}\textbf{valrec } \mathit{zipWith} : \langle a : *, b : *, c : *, \{\{a\} \to \{b\} \to c\}, \\ \quad\quad\quad\quad\quad\quad\quad\quad List\ a, List\ b\rangle \to List\ c \\ \quad = \lambda\langle a : *, b : *, c : *, f : \{\{a\} \to \{b\} \to c\}, \\ \quad\quad\quad xs : List\ a, ys : List\ b\rangle. \\ \quad\quad \textbf{case } xs \textbf{ of } Nil\ \langle\rangle \to Nil\ c \\ \quad\quad\quad\quad\quad\quad\quad Cons\ \langle x : \{a\}, xs' : \{List\ a\}\rangle \to \\ \quad\quad \textbf{case } ys \textbf{ of } Nil\ \langle\rangle \to Nil\ c \\ \quad\quad\quad\quad\quad\quad\quad Cons\ \langle y : \{b\}, ys' : \{List\ b\}\rangle \to \\ \quad\quad Cons\ \langle c, f\ \langle\rangle\ x\ y, \mathit{zipWith}\ \langle a, b, c, f, xs'\ \langle\rangle, ys'\ \langle\rangle\rangle\rangle.\end{array}$$

[5] Since e may mention f, the two definitions may be mutually recursive.

Notice that f is only ever applied in the body to three arguments at a time – $\langle\rangle$, x, and y (or rather $\langle x\rangle$ and $\langle y\rangle$). Based on this observation, we could re-factor $zipWith$ 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):

$$\begin{array}{l}\textbf{valrec}\ zipWith : \langle a:*, b:*, c:*, \{\{a\} \to \{b\} \to c\}, \\ \qquad\qquad List\ a, List\ b\rangle \to List\ c \\ \quad = \lambda\langle a:*, b:*, c:*, f, xs, ys\rangle. \\ \qquad \textbf{valrec}\ f' : \langle\{a\}, \{b\}\rangle \to c = \lambda\langle x, y\rangle.\ f\ \langle\rangle\ x\ y \\ \qquad \textbf{in}\ zipWith_{\text{work}}\ \langle a, b, c, f', xs, ys\rangle\end{array}$$

To see how this can lead to code improvement, consider a call $zipWith\ \langle Int, Int, Int, g, xs, ys\rangle$, where g is the function from Section 6.1. Then, after inlining the wrapper of $zipWith$ we can see locally that g is applied to all its arguments can can therefore be arity-raised. Now, the wrapper of g will cancel with definition of f', leaving the call we really want:

$$zipWith_{\text{work}}\ \langle Int, Int, Int, g_{\text{work}}, xs, ys\rangle$$

6.3 Reflections on arity-raising

Although the use-site analysis might, at first blush, seem to be more powerful than the definition-site one, it is actually the case that the two arity raising transformations are each able to improve the arities of some functions where the other cannot. In particular, for a compiler that works module-by-module like GHC, the use-site analysis will never be able to improve the arity of a top-level function as some of the call sites are unknown statically.

The key benefits of the new intermediate language with regard to the arity raising transformation are as follows:

- Arity in the intermediate language is more stable. It is almost impossible for a compiler transformation to accidentally reduce the arity of a function without causing a type error, whereas accidental reduction of arity is a possibility we must actively concern ourselves with avoiding in the GHC of today.
- Expressing arity in the type system allows optimisations to be applied to the arity of higher-order arguments, as we saw in Section 6.2.
- By expressing arity statically in the type information, it is possible that we could replace GHC's current *dynamic* arity discovery [2] with purely static arity dispatch. This requires that arity raising transformations like these two can remove enough of the argument-at-a-time worst cases such that we obtain satisfactory performance with no run-time tests at all.
- If purely static arity discovery turns out to be too pessimistic in practice (a particular danger for higher order arguments), it would still be straightforward to adapt the dynamic discovery process for this new core language, but we can avoid using it except in those cases where it could give a better result than static dispatch. Essentially, if we appear to be applying at least two groups of arguments to a function, then at that point we should generate code to dynamically check for a better arity before applying the first group.

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 functions), 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 $lambda$s, 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 extension of the $\mathcal{L}_2$ language [14] which also explored the possibility of an optimising compiler suitable for both strict and lazy languages. We share with $\mathcal{L}_2$ 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 $\mathcal{L}_2$ language shares with the MIL the fact that it makes an attempt to support *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.

GRIN [15] is another language that used an explicit representation of thunks and boxing properties. Furthermore, GRIN uses a first order program representation where the structure of closures is explicit – in particular, this means that unboxing of closures is expressible.

The FLEET language [16] takes yet another tack. Thunked and unthunked values have the same type, but can be distinguished by the compiler by inspecting flow labelling information attached to every type – if the flow information includes no label from a thunk creation site, then the value must be in WHNF. A variant of the language, CFleet, has n-ary abstraction but does not support n-ary result types.

The IL language [17] represents thunks explicitly by way of continuations with a logical interpretation, and is to our knowledge the first discussion of auto-lifting in the literature. Their logic based approach could perhaps be extended to accommodate a treatment of arity and multiple-value expressions if "boxed" and "unboxed" uses of the $\wedge$ tuple type former were distinguished.

Hannan and Hicks have previously introduced the arity use-site optimization under the name "higher-order uncurrying" [18] as a type-directed analysis on a source language. They also separately introduced an optimisation called "higher-order arity raising" [19] which attempts to unpack tuple arguments where possible – this is a generalisation of the existing worker/wrapper transformations GHC 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. [20] 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 an 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 λ-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 [21] 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 [22] to allow us to express even more optimisations in a typed manner. In particular, adding unboxed Σ 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 ($[!Int]$). Clean [23] has long supported strictness annotations at the top level of type declarations, (which have a straightforward transformation into Strict Core), but allowing strictness annotations to appear in arbitrary positions in types appears to require ad-hoc polymorphism, and it is not obvious how to go about exposing the extra generality in the source language in a systematic way.

Acknowledgments

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, Douglas McClean, Alan Mycroft, Dominic Orchard, Josef Svenningsson and the anonymous reviewers for their helpful comments and suggestions.

References

[1] A. P. Black, M. Carlsson, M. P. Jones, D. Kieburtz, and J. Nordlander. Timber: a programming language for real-time embedded systems. Technical Report CSE-02-002, Oregon Health & Science University, 2002.

[2] S. Marlow and S. Peyton Jones. How to make a fast curry: push/enter vs eval/apply. In *International Conference on Functional Programming*, pages 4–15, September 2004.

[3] J. Launchbury. A natural semantics for lazy evaluation. In *Principles of Programming Languages*, pages 144–154. ACM, January 1993.

[4] S. Peyton Jones, W. D Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *International Conference on Functional Programming*, 1996.

[5] S. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

[6] M. Sulzmann, M. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, 2007.

[7] J. Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[8] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, September 1998.

[9] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

[10] C. Baker-Finch, K. Glynn, and S. Peyton Jones. Constructed product result analysis for haskell. *Journal of Functional Programming*, 14(2):211–245, 2004.

[11] N. Benton, A. Kennedy, and G. Russell. Compiling standard ML to Java bytecodes. In *International Conference on Functional Programming*, pages 129–140, New York, NY, USA, 1998. ACM.

[12] M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *International Symposium on Static Analysis*, pages 366–381, London, UK, 1995. Springer-Verlag.

[13] A. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[14] S. Peyton Jones, M. Shields, J. Launchbury, and A. Tolmach. Bridging the gulf: a common intermediate language for ML and Haskell. In *Principles of Programming Languages*, pages 49–61, New York, NY, USA, 1998. ACM.

[15] U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, April 1999.

[16] K. Faxén. *Flow Inference, Code Generation, and Garbage Collection for Lazy Functional Languages*. PhD thesis, KTH Royal Institute Of Technology, June 1997.

[17] B. Rudiak-Gould, A. Mycroft, and S. Peyton Jones. Haskell is not not ML. In *European Symposium on Programming*, 2006.

[18] J. Hannan and P. Hicks. Higher-order uncurrying. *Higher Order Symbolic Computation*, 13(3):179–216, 2000.

[19] J. Hannan and P. Hicks. Higher-order arity raising. In *International Conference on Functional Programming*, pages 27–38, New York, NY, USA, 1998. ACM.

[20] Z. Dargaye and X. Leroy. A verified framework for higher-order uncurrying optimizations. March 2009.

[21] S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2:127–202, April 1992.

[22] T. Altenkirch and N. Oury. PiSigma: A core language for dependently typed programming. 2008.

[23] T. Brus, M. van Eekelen, M. van Leer, and M. Plasmeijer. Clean — a language for functional graph rewriting. *Functional Programming Languages and Computer Architecture*, pages 364–384, 1987.