Efficient data representation in polymorphic languages

Xavier Leroy

Efficient data representation in polymorphic languages

Xavier Leroy
INRIA Rocquencourt, France

Abstract

Languages with polymorphic types (e.g. ML) have traditionally been implemented using Lisp-like data representations—everything has to fit in one word, if necessary by being heap-allocated and handled through a pointer. The reason is that, in contrast with conventional statically-typed languages such as Pascal, it is not possible to assign one unique type to each expression at compile-time, an absolute requirement for using more efficient representations (e.g. unallocated multi-word values). In this paper, we show how to take advantage of the static polymorphic typing to mix correctly two styles of data representation in the implementation of a polymorphic language: specialized, efficient representations are used when types are fully known at compile-time; uniform, Lisp-like representations are used otherwise.

1 Introduction

Most programming languages include some kind of type system. Among the numerous motivations for using a type system, I shall focus on two main goals: 1- to make programs safer, and 2- to allow for better compilation.

The first concern is to ensure data integrity in programs. Many operations are meaningless when they are performed on values of the wrong kind, such as, for instance, applying a boolean as if it was a function. In these cases, the results are either meaningless, unpredictable values, or a hardware fault. One of the aims of a type system is to prevent such run-time type errors. From this standpoint, typing can be either static (performed at compile-time), or dynamic (performed at run-time, just before type-constrained operations). But in any case, typing must be strong: there should be no way to claim that a value has a given type when in fact it does not.

Another aim of a type system is to support efficient compilation. Most hardware architectures are somehow typed, in the sense that some resources are dedicated to operate on certain kinds of data. For instance, many processors have two sets of registers, one set to hold integers and pointers, and the other to hold floating-point values. On integer registers, no instructions are provided to perform floating-point computations, and vice-versa. In addition, floating-point registers are usually wider than integer registers, therefore a floating-point number cannot fit in an integer register. When mapping a programming

language on these architectures, it is important to know which values are floating-point numbers and which ones are not, otherwise the correctness of the compilation would be compromised. For instance, to compile an assignment `a := b`, it is crucial to know the types of variables `a` and `b`, in order to determine their exact size, and copy the right number of bytes. Of course, this type information must be available at compile-time, hence the emphasis is on static typing, here. From this standpoint, strong and weak typing are equally acceptable, provided that typing violations are explicitly mentioned in the source code.

Given a strong type system with static checking, both approaches can be combined in a single language, as in Algol-68, Pascal or Modula-2 for instance. This gives the best of both worlds: type safety in a well-compiled language. However, it seems difficult to achieve the same results with more powerful type systems than the one of Pascal. Using types to determine the size (and other relevant information) of a value works fine when every value has exactly one type. This condition is too restrictive in practice: one cannot write "generic" functions once for all and then apply them to data of several types, provided that it makes sense. For instance, a sorting function should be able to operate on many kinds of arrays (e.g. arrays of integers and arrays of strings), assuming a suitable comparison function is provided.

Therefore, many advanced type systems lift the restriction that every value has a unique, statically-known type, through the introduction of concepts such as type abstraction, subtyping, inheritance, and polymorphism. See Cardelli and Wegner's [3] for a uniform presentation of these features. In the following, I concentrate on polymorphism, as in ML [9]. Polymorphic type systems allow type expressions to contain universally quantified type variables. For instance, a sorting function should, for all types T, take an array of elements of type T and return another array of elements of type T, given in addition an ordering predicate over T, that is a function taking a pair of elements of type T and returning a boolean. Its polymorphic type is therefore $\forall T.\, (T \times T \to \texttt{Bool}) \to \texttt{Array}(T) \to \texttt{Array}(T)$. As the quantification implies, the type variable T can be substituted by any actual type, such as `Int` or `String`. In a sense, the type formula given above summarizes all the possible types for the sorting function.

When a value can safely belong to several types, we cannot always determine statically all its characteristics. In the previous example, the sorting function is likely to copy elements of the array given as parameter, but the type of these elements can be any instance of the type variable T, that is any type. Therefore, we do not know at compile-time how many bytes to move to perform the copy. Consider two solutions to this problem.

One is to defer the compilation of polymorphic function until they are actually applied to values of known types. Then, we can deduce the type which the polymorphic function is used with, and therefore get all the information (e.g. sizes) we need to compile. Of course, we will have to compile several specialized versions of a polymorphic functions, if it is used with several different types. This technique is often used for Ada's "generics". Its strength is that it allows the use of efficient representations, and the production of good code (as efficient as if the function was monomorphic). However, it results in code duplication and loss of separate compilation. In addition, it is hard to maintain the illusion that polymorphic functions are still first-class objects. For instance, compilation of polymorphic functions built on top of other polymorphic functions, or functions taking

polymorphic objects as arguments must be deferred in turn. This may lead to an explosion in the number of specializations of a polymorphic function we have to compile—this number could even be infinite, as in the following example, written in ML syntax:

```
datatype 'a chain = Empty | Cell of 'a * 'a list chain;
fun length_of_chain Empty = 0
 |  length_of_chain (Cell(x,rest)) = 1 + length_of_chain rest;
```

The other approach to polymorphic compilation is to revert to *uniform* data representations. That is, data representation for all types share a common format, allowing "generic" operations such as parameter passing, function return, assignment, ... to be compiled without knowing the type of the values they manipulate. In particular, all representations must have the same size, usually one machine word, and any data which does not fit in one word has to be heap-allocated and represented by a pointer (assuming pointers do fit in one word). In addition, the calling conventions must be exactly the same for all functions. With these representation constraints, type information is no longer needed to generate correct code, and compiling polymorphic functions is no longer a problem. This approach is used in most implementations of ML to date.

The main drawback of the second approach is that uniform representations are not as efficient as the kind of *specialized* representations used in monomorphic languages. First, all data which does not fit naturally in one word must be heap-allocated. This is much more expensive than carrying around multi-word representations in several registers, resulting in high heap consumption and frequent garbage collection. Second, communications across function calls are not very efficient: a function returning a floating-point number, just computed in a floating-point register, has to copy it in the heap and return a pointer to it, only to have the callee dereference that pointer, and reload the number in a floating-point register before using it.

This use of uniform data representations is not the main reason why current ML compilers do not produce as efficient code as, say, Modula compilers. However, as modern compiling techniques are applied to the ML language, I think this representation problem will show up as a serious bottleneck for efficient compilation of ML, and similarly for other polymorphic languages.

In this paper, I set out to reconcile the safety offered by strong static typing, the convenience and conceptual cleanliness of having polymorphic functions compiled only once, just like regular functions, and the efficiency of specialized data representations. Since the first two requirements imply polymorphic functions must work on uniform representations anyway, the third requirement has to be relaxed slightly. In this paper, I attempt to mix uniform and specialized representations in the implementation of a polymorphic language. It is intended that monomorphic functions work only on specialized representations, the uniform representations being used only to communicate with polymorphic functions. The main question is, then, How can one infer where to insert the necessary conversions between representations?

The remainder of the paper is organized as follows: section 2 presents the problem of data representation in the case of a monomorphic calculus. Uniform and specialized representations are contrasted by giving two compilation schemes for a simple stack-based

abstract machine. This is mostly implementor's wisdom, but presented in a uniform setting. Using the same approach, section 3 tackles the representation problem in the case of a polymorphic calculus. The main novelty of this paper—combining polymorphism and specialized representations—is informally presented here, then formalized using an intermediate calculus, with a restricted notion of polymorphism. Section 4 aims at showing that specialized representations can be profitably used in the ML language. To this end, the main features of the type system of ML are recalled, and their compatibility with specialized representations checked. Finally, we give a few concluding remarks in section 5

2 The monomorphic case

In this section, we present the problem of data representation in the simple case where every term has exactly one type, and this type is known at compile-time.

2.1 A simply-typed language

We consider a small language based on the simply-typed λ-calculus with constants. The only data structures are pairs. The constants include integer and floating-point numbers, of base types `Int` and `Float`, as well as primitive operations such as `succ_int` and `add_float`.

The syntax of this calculus is as follows. We write i for an integer, f for a floating-point number, c for a constant, x or y for a variable, a or b for terms, and A, B for types.

$$\begin{array}{rcl} c & ::= & i \mid f \mid \texttt{succ\_int} \mid \texttt{add\_float} \mid \ldots \\ a & ::= & c \mid x \mid \lambda x : A.\, b \mid b(a) \mid (a, b) \mid a.\texttt{fst} \mid a.\texttt{snd} \\ A & ::= & \texttt{Int} \mid \texttt{Float} \mid A \to B \mid A \times B \end{array}$$

Typing rules are classical. They are written in structural operational semantics style [11], as a set of axioms and inference rules defining the judgement "under assumptions E, term a has type A", written $E \vdash a : A$. The typing environment E consists in a sequence of assumptions of the form $x : A$, meaning that variable x is assumed to have type A. For each constant c, we write $T(c)$ for its associated type; in particular, $T(i) = \texttt{Int}$, $T(f) = \texttt{Float}$, $T(\texttt{add\_float}) = \texttt{Float} \times \texttt{Float} \to \texttt{Float}$, and so on.

$$E \vdash c : T(c) \qquad E_1, x : A, E_2 \vdash x : A$$

$$\frac{x : A, E \vdash b : B}{E \vdash \lambda x : A.\, b : A \to B} \qquad \frac{E \vdash b : A \to B \quad E \vdash a : A}{E \vdash b(a) : B}$$

$$\frac{E \vdash a : A \quad E \vdash b : B}{E \vdash (a, b) : A \times B} \qquad \frac{E \vdash a : A \times B}{E \vdash a.\texttt{fst} : A \quad E \vdash a.\texttt{snd} : B}$$

$$
\begin{array}{rcl}
\mathcal{U}_E(i) & = & \texttt{Const}(i) \\
\mathcal{U}_E(f) & = & \texttt{Const}(\langle f_{low}, f_{high} \rangle) \\
\mathcal{U}_E(x) & = & \texttt{Access}(pos_x(E)) \\
\mathcal{U}_E(\lambda x : A.\, b) & = & \texttt{Abstr}(\mathcal{U}_{x:A,E}(b), \texttt{Return}) \\
\mathcal{U}_E(b(a)) & = & \mathcal{U}_E(b), \mathcal{U}_E(a), \texttt{Apply} \\
\mathcal{U}_E(a, b) & = & \mathcal{U}_E(a), \mathcal{U}_E(b), \texttt{Pair} \\
\mathcal{U}_E(a.\texttt{fst}) & = & \mathcal{U}_E(a), \texttt{First} \\
\mathcal{U}_E(a.\texttt{snd}) & = & \mathcal{U}_E(a), \texttt{Second} \\
\mathcal{U}_E(\texttt{add\_float}(a, b)) & = & \mathcal{U}_E(a), \mathcal{U}_E(b), \texttt{AddFloat} \\
\mathcal{U}_E(\texttt{add\_float}) & = & \texttt{Abstr}(\ \texttt{Access}(0), \texttt{First}, \texttt{Access}(0), \texttt{Second},\ ) \\
 & & \qquad \texttt{AddFloat}, \texttt{Return}
\end{array}
$$

Figure 1: Compilation scheme with uniform representations

2.2 Evaluation using uniform representations

We now give a compilation scheme for this small language. The target machine is a stack-based machine with environments, where functional values are represented by closures. Call-by-value is assumed, to be consistent with the strict semantics of ML. This machine is very close to Landin's SECD [7] and Cardelli's FAM [1]. The compilation scheme is given in figure 1. It is straightforward, except maybe for the treatment of variables. The value of a variable is to be found at run-time in the environment (a tuple of values). To access this variable, we need to know its position inside this tuple. That's the *raison d'être* of the compilation environment E, similar in structure to typing environments: it records the name of all free variables, in the order they will appear in the run-time environment. Then, the position of a variable x in an environment described by E is simply:

$$
\begin{array}{rcl}
pos_x(\emptyset) & \text{is} & \text{undefined} \\
pos_x(x : A, E) & = & 0 \\
pos_x(y : A, E) & = & 1 + pos_x(E)
\end{array}
$$

This simple compilation scheme does not make use of typing information. This fact has deep consequences on the way data are represented in the machine, namely that *all data must fit in one word.* Indeed, the functions $\lambda x{:}A.\, x$, for all types A, have exactly the same code, and are applied in exactly the same way. This means that the instructions to apply a function to an argument, fetch a value from the environment, and returning a value, must operate uniformly on data of any type. This implies that all data representations have the same size, and in case of register machines with several register classes, that they all use the same register class.

As a consequence, data which do not fit in one word are allocated in the heap, and handled through a pointer. (We assume that any pointer fits in one word). We write

	Code	Stack	Environment
Before	$\texttt{Const}(v), C$	S	γ
After	C	v, S	γ
Before	$\texttt{Access}(n), C$	S	$\gamma = \langle v_1, \ldots, v_n, \ldots \rangle$
After	C	v_n, S	γ
Before	$\texttt{Abstr}(C_1), C$	S	γ
After	C	$\langle \langle C_1 \rangle, \gamma \rangle, S$	γ
Before	$\texttt{Apply}, C$	$v, \langle \langle C_1 \rangle, \langle \Gamma_1 \rangle \rangle, S$	γ
After	C_1	$\langle C \rangle, \gamma, S$	$\langle v, \Gamma_1 \rangle$
Before	$\texttt{Return}, C$	$v, \langle C_0 \rangle, \gamma_0, S$	γ
After	C_0	v, S	γ_0
Before	$\texttt{Pair}, C$	v_2, v_1, S	γ
After	C	$\langle v_1, v_2 \rangle, S$	γ
Before	$\texttt{First}, C$	$\langle v_1, v_2 \rangle, S$	γ
After	C	v_1, S	γ
Before	$\texttt{Second}, C$	$\langle v_1, v_2 \rangle, S$	γ
After	C	v_2, S	γ
Before	$\texttt{AddFloat}, C$	$\langle f'_{low}, f'_{high} \rangle, \langle f_{low}, f_{high} \rangle, S$	γ
After	C	$\langle (f + f')_{low}, (f + f')_{high} \rangle, S$	γ

Figure 2: An abstract machine with uniform representations

$\langle S \rangle$ for a pointer to the sequence of words S, located in the heap. For instance, the pair of values v_1 and v_2 is represented by $\langle v_1, v_2 \rangle$, and similarly for closures. Regarding constants, we assume that integers may fit in one word, but that high-precision floating-point numbers f require two words, written f_{low} and f_{high}, therefore f is represented by $\langle f_{low}, f_{high} \rangle$.

These representations lead to the transition function given in figure 2.

2.3 Inefficiencies of uniform representations

The evaluation mechanism presented above is simple, but not very efficient. As an example, let us consider the function $f(x, y) = x + 2y$, where x and y are reals, represented in floating-point. Since our calculus does not directly support functions with several arguments, some transformation is required. We can make it into a function taking a pair:

$$f_{pair} = \lambda z : \texttt{Float} \times \texttt{Float}.\, \texttt{add\_float}(z.\texttt{fst}, \texttt{add\_float}(z.\texttt{snd}, z.\texttt{snd}))$$

or into a function taking x and returning another function (this technique is known as *currying*):

$$f_{cur} = \lambda x : \texttt{Float}.\, \lambda y : \texttt{Float}.\, \texttt{add\_float}(x, \texttt{add\_float}(y, y)).$$

Both versions are inefficient in terms of heap allocation and memory accesses. First, each floating-point addition must allocate two words in the heap to store its result, and perform

$$
\begin{array}{rcl}
\mathcal{S}_E(i^{\mathtt{Int}}) & = & \mathtt{Const}(i) \\
\mathcal{S}_E(f^{\mathtt{Float}}) & = & \mathtt{Const}(f_{high}), \mathtt{Const}(f_{low}) \\
\mathcal{S}_E(x^A) & = & \mathtt{Access}(p+s-1), \ldots, \mathtt{Access}(p) \\
 & & \text{where } p = pos_x(E) \text{ and } s = size(A) \\
\mathcal{S}_E(\lambda x : A.\, b^B) & = & \mathtt{Abstr}(\mathcal{S}_{x:A,E}(b^B), \mathtt{Return}_{size(B)}) \\
\mathcal{S}_E(b(a^A)) & = & \mathcal{S}_E(b), \mathcal{S}_E(a), \mathtt{Apply}_{size(A)} \\
\mathcal{S}_E(a, b) & = & \mathcal{S}_E(a), \mathcal{S}_E(b) \\
\mathcal{S}_E(a^{A\times B}.\mathtt{fst}) & = & \mathcal{S}_E(a), \mathtt{First}_{size(A),size(B)} \\
\mathcal{S}_E(a^{A\times B}.\mathtt{snd}) & = & \mathcal{S}_E(a), \mathtt{Second}_{size(A),size(B)} \\
\mathcal{S}_E(\mathtt{add\_float}(a, b)) & = & \mathcal{S}_E(a), \mathcal{S}_E(b), \mathtt{AddFloat} \\
\mathcal{S}_E(\mathtt{add\_float}) & = & \mathtt{Abstr}(\ \mathtt{Access}(0), \mathtt{Access}(1), \mathtt{Access}(2), \mathtt{Access}(3), \\
 & & \qquad \mathtt{AddFloat}, \mathtt{Return}_2)
\end{array}
$$

Figure 3: Compilation scheme with specialized representations

three memory accesses. This is especially absurd in the case of the innermost addition, whose result is used only once, by the next instruction. Admittedly, a simple analysis of the code could detect that, and avoid allocating the intermediate result. But the final result must be allocated anyway, as required by the calling convention.

The passing of the two parameters is also inefficient. In the case of the uncurried form, the caller has to build a pair of the two arguments, which means allocating two words in the heap and performing two memory writes, only to have the callee discard the pair and solely use its components, at the cost of one memory access for each use of a parameter. In the case of the curried form, the main flaw is the building of an intermediate closure between the passing of the first and the second argument. (This closure corresponds to the partial application of the function to its first argument.)

To be more efficient, it is clear now that we have to lift the restriction that any value must either fit in one word or be allocated, and be able to handle unallocated multi-word values. To do so, we need to statically keep track of the size of all values and results. (In case of a register machine with several classes of registers, we would have to record the suitable register class for each value). Obviously, all this information is already contained in the typing of the program; what we shall present now is a compilation scheme taking advantage of the types.

2.4 Evaluation using specialized representations

The new compilation function is given in figure 3. It corresponds to the case where data representations are as "flat" as possible: floating-point numbers are not allocated, pairs are simple concatenations of the sequences of words representing their components, and for closures, only the environment tuple is allocated, but the pair of the code pointer and

	Code	Stack	Environment
Before	$\texttt{Const}(v); C$	S	γ
After	C	v, S	γ
Before	$\texttt{Access}(n), C$	S	$\gamma = \langle v_1, \ldots, v_n, \ldots \rangle$
After	C	v_n, S	γ
Before	$\texttt{Abstr}(C_1), C$	S	γ
After	C	$\langle C_1 \rangle, \gamma, S$	γ
Before	$\texttt{Apply}_i, C$	$v_i, \ldots, v_1, \langle C_1 \rangle, \langle E_1 \rangle, S$	γ
After	C_1	$\langle C \rangle, \gamma, S$	$\langle v_1, \ldots, v_i, E_1 \rangle$
Before	$\texttt{Return}_i, C$	$v_i, \ldots, v_1, \langle C_0 \rangle, \gamma_0, S$	γ
After	C_0	$v_i, \ldots, v_1, S$	γ_0
Before	$\texttt{First}_{i,j}, C$	$w_j, \ldots, w_1, v_i, \ldots, v_1, S$	γ
After	C	$v_i, \ldots, v_1, S$	γ
Before	$\texttt{Second}_{i,j}, C$	$w_j, \ldots, w_1, v_i, \ldots, v_1, S$	γ
After	C	$w_j, \ldots, w_1, S$	γ
Before	$\texttt{AddFloat}, C$	$f'_{low}, f'_{high}, f_{low}, f_{high}, S$	γ
After	C	$(f + f')_{low}, (f + f')_{high}, S$	γ

Figure 4: An abstract machine with specialized representations

the environment pointer is unallocated.

We use the convention that all terms are subexpressions of a given closed term a_0, the whole program. We write a^A to indicate that a was given the type A in the (unique) typing derivation of a_0, in the empty environment. This annotation is used to determine the size (number of words used in the representation) of data of type A, which determines in turn the position of the first word of a variable x in the run-time environment:

$$\begin{array}{rcl@{\qquad}rcl} size(\texttt{Int}) & = & 1 & pos_x(\emptyset) & \text{is} & \text{undefined} \\ size(\texttt{Float}) & = & 2 & pos_x(x : A, E) & = & 0 \\ size(A \to B) & = & 2 & pos_x(y : A, E) & = & size(A) + pos_x(E) \\ size(A \times B) & = & size(A) + size(B) & & & \end{array}$$

The transition function for the corresponding machine is given in figure 4.

With this new evaluation mechanism, the previous example function ($f(x, y) = x+2y$) executes much more efficiently. The uncurried version takes as argument an unallocated pair of unallocated floating-point numbers, that is, four words on the stack. Intermediate results are held in the stack, without any heap allocation or heap accesses. The final result, an unallocated float, is returned to the caller as two words on top of the stack. The curried version benefits similarly from unallocated floats. In addition, the intermediate closure returned to the caller between the passing of the first and second arguments is not allocated either, but left as two words on the stack (a code pointer, an environment pointer), ready to be applied to the second argument.

2.5 Performance comparisons

Specialized representations lead to less heap allocation and less memory accesses than uniform representations. In the case of uniform representations, multi-word values are always allocated in the heap when created, and reloaded when used, while this is not true in the case of specialized representations.

On the other hand, specialized representations generate more stack or register moves. The savings in heap accesses far outweights them, except in extreme cases where some data are discarded. For instance, applying a function taking a 10-word argument and returning a constant requires ten stack moves using specialized representations, and only one using uniform representations. This does not happen frequently in actual programming, however.

3 The polymorphic case

3.1 A polymorphic language

We now consider a polymorphic language based on the second-order λ-calculus, as introduced by Girard [5] and independently by Reynolds [12]. At the level of types, we introduce universal quantification, with the intent that a term of type $\forall X.\, A[X]$ can be used with types $A[B]$ for all types B. At the level of terms, the corresponding elimination construct is application of a term a to a type B, written $a(B)$. The introduction construct is abstraction over a type variable X, written $\Lambda X.\, A$.

$$\begin{array}{rcl} a & ::= & c \mid x \mid \Lambda X.\, a \mid a(B) \mid \lambda x : A.\, b \mid b(a) \mid (a, b) \mid a.\texttt{fst} \mid a.\texttt{snd} \\ A & ::= & X \mid \forall X.\, A \mid \texttt{Int} \mid \texttt{Float} \mid A \to B \mid A \times B \end{array}$$

Second-order λ-calculus is one of the purest and most general approaches to polymorphism, but very few programming languages implement it in its full generality (Poly [8], Quest [2]). The ML language proposes a restricted version of it: it requires that universal quantifiers be in prenex position: that all type expressions are of the form $\forall X_1 \ldots \forall X_n.\, A$, where A does not contain quantifiers. This makes type inference possible, using the well-known Damas-Milner algorithm [4], while type inference for second-order λ-calculus is still an open problem. In the following, we do not need the prenex quantification hypothesis, and therefore consider arbitrary quantification.

Informally, typechecking rules are those of the simply-typed language extended by the following two rules:

$$\frac{E \vdash a : A}{E \vdash \Lambda X.\, a : \forall X.\, A} \qquad \frac{E \vdash a : \forall X.\, A}{E \vdash a(B) : A\{X \leftarrow B\}}$$

The actual rules are slightly more complex, since we must take care of the scope of type variables. This means that not all well-formed type expressions are valid types in a given context, and similarly for environments. We use two auxiliary predicates, $E \vdash A$ type, meaning that A is a valid type in environment E, and $\vdash E$ env, meaning that E is a valid environment.

$$\frac{\vdash E \text{ env}}{E \vdash c : T(c)} \qquad \frac{\vdash E_1, x : A, E_2 \text{ env}}{E_1, x : A, E_2 \vdash x : A}$$

$$\frac{E \vdash A \text{ type} \quad E, x : A \vdash b : B}{E \vdash \lambda x : A.\, b : A \to B} \qquad \frac{E \vdash b : A \to B \quad E \vdash a : A}{E \vdash b(a) : B}$$

$$\frac{E, X \text{ type} \vdash a : A}{E \vdash \Lambda X.\, a : \forall X.\, A} \qquad \frac{E \vdash a : \forall X.\, A \quad E \vdash B \text{ type}}{E \vdash a(B) : A\{X \leftarrow B\}}$$

$$\frac{E \vdash a : A \quad E \vdash b : B}{E \vdash (a, b) : A \times B} \qquad \frac{E \vdash a : A \times B}{E \vdash a.\texttt{fst} : A \quad E \vdash a.\texttt{snd} : B}$$

The auxiliary predicates are defined as follows:

$$\vdash \emptyset \text{ env}$$

$$\frac{\vdash E \text{ env} \quad X \notin \mathrm{Dom}(E)}{\vdash E, X \text{ type env}} \qquad \frac{\vdash E \text{ env} \quad x \notin \mathrm{Dom}(E) \quad E \vdash A \text{ type}}{\vdash E, x : A \text{ env}}$$

$$\frac{\vdash E \text{ env}}{E \vdash \texttt{Int} \text{ type} \quad E \vdash \texttt{Float} \text{ type}} \qquad \frac{\vdash E_1, X \text{ type}, E_2 \text{ env}}{E_1, X \text{ type}, E_2 \vdash X \text{ type}}$$

$$\frac{E, X \text{ type} \vdash A \text{ type}}{E \vdash \forall X.\, A \text{ type}} \qquad \frac{E \vdash A \text{ type} \quad E \vdash B \text{ type}}{E \vdash A \to B \text{ type} \quad E \vdash A \times B \text{ type}}$$

3.2 Evaluation using uniform representations

In the case where all data have uniform, single-word representations, the abstract machine needs no special provision to accommodate polymorphic programs. The evaluation mechanism of section 2.2 already implements polymorphism in some sense, since, for instance, the term $\lambda x : \texttt{Int}.\, x$ is compiled in such a way that it can be applied to data of any type, not just `Int`, and return it unmodified. Therefore, we can use the abstract machine of figure 2 as is, along with the compilation scheme of figure 1. We just have to add the two following cases, stating that abstraction on a type variable and application to a type have no computational contents (in this case, they are mere typechecking annotations):

$$\begin{aligned} \mathcal{U}_E(\Lambda X.\, a) &= \mathcal{U}_E(a) \\ \mathcal{U}_E(a(B)) &= \mathcal{U}_E(a) \end{aligned}$$

3.3 Evaluation using specialized representations

Let us try now to implement a polymorphic language with non-uniform data representations, such as multi-word values. Things are not as easy as in the monomorphic case. We already dismissed the approach of compiling several specializations of a polymorphic term on demand. Therefore, when compiling a polymorphic term, we are left with no alternative but to assume that all values of unknown types (that is, whose type is a type variable) are represented in a uniform way, as in the previous section. However, when we have more information on the type of a value, we would like to use specialized representations, for the sake of efficiency. In particular, we hope to compile fully monomorphic terms as efficiently as in the case of the monomorphic calculus.

This requirement implies that every value has two representations, a uniform one, to be used for communication with polymorphic functions, and a specialized, efficient one, possibly spanning several words and taking advantage of special hardware, to be used the rest of the time. We refer to the former as "the wrapped representation", and to the latter as "the unwrapped representation." When we apply a polymorphic function to a value of known type, it is likely that the value will be unwrapped, while the function expects it wrapped. Therefore, the compiler will sometimes have to insert explicit coercions between the two representations; we write $\texttt{wrap}_A$ and $\texttt{unwrap}_A$ for the coercions operating on values of type A. (We mention the type A explicitly to emphasize that those coercions are not polymorphic functions operating uniformly on all data, but rather functions defined in an *ad-hoc* way for each type A.)

Consider the example of the `reverse_pair` function:

$$\texttt{reverse\_pair} = \Lambda X.\, \Lambda Y.\, \lambda z : X \times Y.\, (z.\texttt{snd}, z.\texttt{fst}).$$

Since the types of $z.\texttt{fst}$ and $z.\texttt{snd}$ are unknown, these values must be wrapped. However, z itself is known to be a pair, so there is no need to wrap it. Therefore, the calling conventions of `reverse_pair` is as follows: it expects two words on the stack, which are wrapped representations of $z.\texttt{fst}$ and $z.\texttt{snd}$, and returns two words on the stack. Now, let us consider the following application:

$$\texttt{reverse\_pair}\ (\texttt{Float})\ (\texttt{Int} \times \texttt{Float})\ (3.14, (7, 2.718)).$$

We assume that the two floating-point constants are allocated "flat", as well as the two pairs. The argument is therefore represented by the five words:

$$(3.14)_{low}, (3.14)_{high}, 7, (2.718)_{low}, (2.718)_{high}.$$

Before passing it to `reverse_pair`, some transformations are required: wrap the first component of the pair (the first two words,) resulting in the single word $\langle (3.14)_{low}, (3.14)_{high} \rangle$; similarly for the second component, leading to $\langle 7, (2.718)_{low}, (2.718)_{high} \rangle$. The resulting two words are a suitable argument for `reverse_pair`. On return, the stack holds the two words:

$$\langle 7, (2.718)_{low}, (2.718)_{high} \rangle, \langle (3.14)_{low}, (3.14)_{high} \rangle.$$

Two steps of unwrapping lead to the 5-tuple $7, (2.718)_{low}, (2.718)_{high}, (3.14)_{low}, (3.14)_{high}$, which is the unwrapped representation of $((7, 2.718), 3.14)$, as expected.

Let us now consider an example involving higher-order functions:

$$\texttt{map\_pair} = \Lambda X.\, \Lambda Y.\, \lambda f : X \to Y.\, \lambda z : X \times X.\, (f\ z.\texttt{fst}, f\ z.\texttt{snd})$$
$$\texttt{int\_of\_float} : \texttt{Float} \to \texttt{Int}$$
$$\texttt{map\_pair}\ (\texttt{Float})(\texttt{Int})\ (\texttt{int\_of\_float})\ (3.14, 2.718)$$

According to our principles, the functional `map_pair` expects its parameter f to be a function taking one word (a wrapped representation) as argument, and returning one word (another wrapped representation). In addition, the parameter z should be an unwrapped pair of two wrapped values, as previously. However, the `int_of_float` primitive function expects an unwrapped floating-point argument (two words), and produces an unwrapped integer. Therefore, `map_pair` cannot be applied directly to `int_of_float`; it must be given a version of `int_of_float` which takes a wrapped floating-point number as argument, and returns a wrapped integer as result, that is, with obvious notations:

$$\lambda x : \texttt{Wrapped}(\texttt{Float}).\, \texttt{wrap}_{\texttt{Float}}(\texttt{int\_of\_float}(\texttt{unwrap}_{\texttt{Float}}(x))).$$

The rest of this example proceeds as above. The important point is that higher-order functions may require their functional arguments to be transformed in order to accommodate wrapped arguments or results instead of unwrapped ones, and vice-versa. This transformation does not require recompilation of the function. It merely puts some "stub code" around it, performing the right `wrap` and `unwrap` operations.

The rest of this section formalizes a compilation scheme based on the ideas above. This is a two-step process: first, a translation into another polymorphic calculus, where the duality of wrapped/unwrapped representations is taken into account; then, a code generation phase, combining cases from section 2.4 for the unwrapped values and section 2.2 for the wrapped, uniformly represented values.

3.3.1 A restricted polymorphic calculus

First, the distinction between wrapped and unwrapped representations is made explicit in the types, through the introduction of a new type operator, `Wrapped`. Informally, for all types A, the type $\texttt{Wrapped}(A)$ contains all wrapped representations of values of type A. At the level of terms, we add the operators $\texttt{wrap}_A$ and $\texttt{unwrap}_A$, which map A to $\texttt{Wrapped}(A)$ and conversely. Then, we restrict polymorphism by requesting that type variables range over the class of wrapped types, that is all $\texttt{Wrapped}(A)$ where A is a type, instead of the full class of types. By analogy with bounded quantification [3], we use the notation $\forall X \leq \texttt{Wrapped}.\, A$ for this restricted universal quantification. Conversely, for type application $a(B)$, we require that B is a wrapped type, that is either $\texttt{Wrapped}(B')$, or a type variable Y. The syntax of the restricted calculus is therefore as follows:

$$\begin{aligned} a \ &::= \ c \mid x \mid \texttt{wrap}_A(a) \mid \texttt{unwrap}_A(a) \mid \Lambda X \leq \texttt{Wrapped}.\, a \\ &\quad \mid a(B) \mid \lambda x : A.\, b \mid b(a) \mid (a, b) \mid a.\texttt{fst} \mid a.\texttt{snd} \\ A \ &::= \ X \mid \texttt{Wrapped}(A) \mid \forall X \leq \texttt{Wrapped}.\, A \mid \texttt{Int} \mid \texttt{Float} \mid A \to B \mid A \times B \end{aligned}$$

The typing rules are almost the same as those of section 3.1, with additional rules for the `Wrapped`, `wrap` and `unwrap` operators, and a different treatment of type application.

To help distinguishing both calculi, we use $\vdash^r$ instead of $\vdash$ for the typing judgements of this calculus. Here are the rules that differ from the one of the general calculus:

$$\frac{E \vdash^r a : A}{E \vdash^r \mathtt{wrap}_A(a) : \mathtt{Wrapped}(A)} \qquad \frac{E \vdash^r a : \mathtt{Wrapped}(A)}{E \vdash^r \mathtt{unwrap}_A(a) : A}$$

$$\frac{E, X \text{ type} \vdash^r a : A}{E \vdash^r \Lambda X \leq \mathtt{Wrapped}.\, a : \forall X \leq \mathtt{Wrapped}.\, A}$$

$$\frac{E \vdash^r a : \forall X \leq \mathtt{Wrapped}.\, A \quad E \vdash^r Y \text{ type}}{E \vdash^r a(Y) : A\{X \leftarrow Y\}} \qquad \frac{E \vdash^r a : \forall X \leq \mathtt{Wrapped}.\, A \quad E \vdash^r B \text{ type}}{E \vdash^r a(\mathtt{Wrapped}(B)) : A\{X \leftarrow B\}}$$

$$\frac{E, X \text{ type} \vdash^r A \text{ type}}{E \vdash^r \forall X \leq \mathtt{Wrapped}.\, A \text{ type}} \qquad \frac{E \vdash^r A \text{ type}}{E \vdash^r \mathtt{Wrapped}(A) \text{ type}}$$

3.3.2 Translation into the restricted calculus

Here, we provide translations for terms and types of the original, polymorphic calculus into the restricted calculus given above. The translation function is written $[\,]$. On types, it simply consists in restricting all quantifications, as follows:

$$[\forall X.\, A] = \forall X \leq \mathtt{Wrapped}.\, [A]$$

and then it is extended as a congruence over all types. On terms, the translation transforms application to a type, so as to generate the kind of "stub" code needed in the example above. We have:

$$\begin{aligned} [\Lambda X.\, a] &= \Lambda X \leq \mathtt{Wrapped}.\, [a] \\ [a^{\forall X.\, A}(B)] &= \mathcal{T}_{X \leftarrow B}([a](\mathtt{Wrapped}[B])^A) \end{aligned}$$

Again, it is extended as a congruence over all terms. The hard work is done by the twin auxiliary functions $\mathcal{T}_{X \leftarrow B}(a^A)$ and $\overline{\mathcal{T}}_{X \leftarrow B}(a^A)$, defined inductively on A in figure 5. They are responsible for specializing X to B in any term a of type A. The function $\mathcal{T}$ deals with positive occurences of the coercion, and $\overline{\mathcal{T}}$ with negative occurences. When $A = X$, we simply unwrap or wrap a. When X is not free in A, and especially when A is a base type, nothing needs to be done. When A is a product type, we specialize recursively its two components, and pair them together. When A is a function type, we build a function that takes an argument x, specializes it recursively, applies a to x, and specializes the result. To specialize the argument x, we switch to the other transformation ($\overline{\mathcal{T}}$ instead of $\mathcal{T}$ and conversely), because of the contravariance of the arrow. The case of a universal type is similar, but easier.

An example of translation is given in figure 6. The translation usually creates many β-redexes, which we reduced on the fly for the sake of readability.

It remains to show that this translation is sensible, in particular, that it preserves semantics. First, we can map terms of the restricted calculus back to the full calculus

$$
\begin{array}{rcl}
\mathcal{T}(a^X) & = & \mathtt{unwrap}_{[B]}(a) \\
\mathcal{T}(a^Y) & = & a \text{ if } X \neq Y \\
\mathcal{T}(a^{\mathtt{Int}}) & = & a \\
\mathcal{T}(a^{\mathtt{Float}}) & = & a \\
\mathcal{T}(a^{A_1 \to A_2}) & = & \lambda x : [A_1\{X \leftarrow B\}].\, \mathcal{T}(a(\overline{\mathcal{T}}(x^{A_1}))^{A_2}) \text{ where } x \text{ not free in } a \\
\mathcal{T}(a^{A_1 \times A_2}) & = & (\mathcal{T}(a.\mathtt{fst}^{A_1}), \mathcal{T}(a.\mathtt{snd}^{A_2})) \\
\mathcal{T}(a^{\forall X.\, A}) & = & a \\
\mathcal{T}(a^{\forall Y.\, A}) & = & \Lambda Y \leq \mathtt{Wrapped}.\, \mathcal{T}(a(Y)^A) \\
 & & \\
\overline{\mathcal{T}}(a^X) & = & \mathtt{wrap}_{[B]}(a) \\
\overline{\mathcal{T}}(a^Y) & = & a \text{ if } X \neq Y \\
\overline{\mathcal{T}}(a^{\mathtt{Int}}) & = & a \\
\overline{\mathcal{T}}(a^{\mathtt{Float}}) & = & a \\
\overline{\mathcal{T}}(a^{A_1 \to A_2}) & = & \lambda x : [A_1]\{X \leftarrow \mathtt{Wrapped}[B]\}.\, \overline{\mathcal{T}}(a(\mathcal{T}(x^{A_1}))^{A_2}) \text{ where } x \text{ not free in } a \\
\overline{\mathcal{T}}(a^{A_1 \times A_2}) & = & (\overline{\mathcal{T}}(a.\mathtt{fst}^{A_1}), \overline{\mathcal{T}}(a.\mathtt{snd}^{A_2})) \\
\overline{\mathcal{T}}(a^{\forall X.\, A}) & = & a \\
\overline{\mathcal{T}}(a^{\forall Y.\, A}) & = & \Lambda Y \leq \mathtt{Wrapped}.\, \overline{\mathcal{T}}(a(Y)^A)
\end{array}
$$

Figure 5: Generation of stub code to accommodate restricted polymorphism. ($\mathcal{T}(a^A)$ and $\overline{\mathcal{T}}(a^A)$ abbreviate $\mathcal{T}_{X \leftarrow B}(a^A)$ and $\overline{\mathcal{T}}_{X \leftarrow B}(a^A)$, respectively.)

by erasing the `wrap` and `unwrap` nodes, and the bounded quantifications. It is easy to see that for all a, the translation $[a]$ is mapped back to a term which reduces to a. This means that if we identify the wrapped and unwrapped representations, a and $[a]$ produce the same results. It remains to show that in $[a]$ we do not use a wrapped value when an unwrapped one is expected and conversely. However, such an error is caught by the type system of the restricted calculus. Hence, we just have to prove that the translation of a well-typed term is well-typed. The following lemma expresses the correctness of the auxiliary translation functions with respect to types.

Lemma 1 *Let a be a term, E be an environment of the restricted calculus, and A, B be two types of the full polymorphic calculus.*

- *If $E \stackrel{r}{\vdash} a : [A]\{X \leftarrow \mathtt{Wrapped}[B]\}$, then $E \stackrel{r}{\vdash} \mathcal{T}_{X \leftarrow B}(a^A) : [A\{X \leftarrow B\}]$.*
- *If $E \stackrel{r}{\vdash} a : [A\{X \leftarrow B\}]$, then $E \stackrel{r}{\vdash} \overline{\mathcal{T}}_{X \leftarrow B}(a^A) : [A]\{X \leftarrow \mathtt{Wrapped}[B]\}$.*

Proof: Easy inductive argument on A. □

Proposition 1 *Let a, A, E be a term, a type, and an environment of the full calculus. If $E \vdash a : A$, then in the restricted calculus, $[E] \stackrel{r}{\vdash} [a] : [A]$.*

$$
\begin{array}{l}
\mathcal{T}_{X \leftarrow \texttt{Float}}(\texttt{double}^{(X \to X) \to (X \to X)}) \\
= \lambda f : \texttt{Float} \to \texttt{Float}.\, \mathcal{T}(\texttt{double}(\overline{\mathcal{T}}(f^{X \to X}))^{X \to X}) \\
= \lambda f : \texttt{Float} \to \texttt{Float}.\, \lambda x : \texttt{Float}. \\
\quad \texttt{unwrap}_{\texttt{Float}}(\texttt{double}\ (\lambda y : \texttt{Wrapped}(\texttt{Float}).\, \texttt{wrap}_{\texttt{Float}}(f(\texttt{unwrap}_{\texttt{Float}}(y)))) \\
\qquad\qquad\qquad\qquad (\texttt{wrap}_{\texttt{Float}}(x)) \\
\\
\text{Hence:} \\
[\texttt{double}^{\forall X.\,(X \to X) \to (X \to X)}(\texttt{Float})(\lambda x : \texttt{Float}.\, \texttt{add\_float}(x)(1))(3.14)] \\
\xrightarrow{*} \texttt{unwrap}_{\texttt{Float}}(\texttt{double}\ (\lambda y : \texttt{Wrapped}(\texttt{Float}).\, \texttt{wrap}_{\texttt{Float}}(\texttt{add\_float}(\texttt{unwrap}_{\texttt{Float}}(y))(1))) \\
\qquad\qquad\qquad\qquad (\texttt{wrap}_{\texttt{Float}}(3.14))
\end{array}
$$

Figure 6: An example of translation

Proof: By induction on the length of the proof of $E \vdash a : A$. □

3.3.3 Compiling the restricted calculus

To compile the restricted calculus, we use the same abstract machine with specialized representations as in section 2.4 (figure 4). The compilation scheme is almost the same. We simply state that the size of a value of type $\texttt{Wrapped}(A)$ is always one, and give two additional rules for the translation of `wrap` and `unwrap`. The only constraints are that $\texttt{unwrap}_A$ must be the inverse of $\texttt{wrap}_A$, and $\texttt{wrap}_A$ must produce one-word data. A simple solution for $\texttt{wrap}_A$ is to heap-allocate data occupying more than one word, and keep one-word data unchanged; symmetrically, $\texttt{unwrap}_A$ performs nothing if $size(A) = 1$, and dereferences the value of a otherwise. We add the two corresponding instructions `Box` and `Unbox` in the abstract machine (figure 7). The resulting compilation scheme is given in figure 8, with the *size* function defined as follows:

$$
\begin{array}{rcl}
size(\texttt{Int}) & = & 1 \\
size(\texttt{Float}) & = & 2 \\
size(A \to B) & = & 2 \\
size(A \times B) & = & size(A) + size(B) \\
size(\texttt{Wrapped}(A)) & = & 1
\end{array}
$$

3.4 Performance comparison

As in the case of the monomorphic calculus, specialized representations lead to less heap allocation and pointer dereferencing, at the expense of more stack or register moves. In the case of uniform representations, each primitive performs implicit `unwrap` operations on its arguments, and `wrap` operations on its results. Using specialized representations, we managed to remove some `wrap` and `unwrap` operations when we have enough type information.

	Code	Stack	Environment
Before	$\texttt{Box}_i, C$	$v_1, \ldots, v_i, S$	γ
After	C	$\langle v_1, \ldots, v_i \rangle, S$	γ
Before	$\texttt{Unbox}_i, C$	$\langle v_1, \ldots, v_i \rangle, S$	γ
After	C	$v_1, \ldots, v_i, S$	γ

Figure 7: Additional instructions for wrapping and unwrapping.

It is hard to give more precise comparisons, due to the following fact: in the case of uniform representation, wrapping and unwrapping are performed by the primitive operations, therefore unwrapping (on the arguments) is delayed as much as possible, while wrapping (of the results) is performed immediately. We say that with uniform representations, unwrapping is lazy, while wrapping is eager. In case of specialized representations, wrapping is delayed until a value is passed to a polymorphic function, while unwrapping the result of a polymorphic function is performed as soon as possible. Hence, in the case of specialized representations, unwrapping is eager and wrapping is lazy. This leads to a rather different behavior, presumably favorable, since wrapping is much more expensive than unwrapping.

On the negative side, it is true that the stub code inserted to change representations in the case of functions introduces additional function calls. Reductions at compile-time can often eliminate them, but for example if the function being transformed is a parameter of a functional, an additional call will remain.

4 Toward actual programming languages

The highly stylized calculi used in previous sections are not yet close to actual programming languages. They lack many important features: the only data structures are pairs, there are neither variants, nor general records; and recursive types are not allowed. In this section, we shall see how to integrate these features into the calculus of the previous section, and what additional constraints they put on the choice of data representations. Finally, we discuss the application to the ML language.

4.1 Sum types

While the previous calculi have a rudimentary notion of records, they have no notion of the symmetric data structure: variants (tagged unions). In typed calculi, variants are traditionally presented by sum types $A + B$. The elements of that type are either of type A, with tag `left`, or of type B, with tag `right`. The introduction constructs are the two constructors `inleft` and `inright`; the elimination construct is pattern-matching on the tag.

$$
\begin{array}{rcl}
\mathcal{P}_E(i^{\mathtt{Int}}) & = & \mathtt{Const}(i) \\
\mathcal{P}_E(f^{\mathtt{Float}}) & = & \mathtt{Const}(f_{high}), \mathtt{Const}(f_{low}) \\
\mathcal{P}_E(x^A) & = & \mathtt{Access}(p+s-1), \ldots, \mathtt{Access}(p) \\
 & & \text{where } p = pos_x(E) \text{ and } s = size(A) \\
\mathcal{P}_E(\mathtt{wrap}_A(a)) & = & \mathcal{P}_E(a) \text{ if } size(A) = 1 \\
\mathcal{P}_E(\mathtt{wrap}_A(a)) & = & \mathcal{P}_E(a), \mathtt{Box}_{size(A)} \text{ if } size(A) > 1 \\
\mathcal{P}_E(\mathtt{unwrap}_A(a)) & = & \mathcal{P}_E(a) \text{ if } size(A) = 1 \\
\mathcal{P}_E(\mathtt{unwrap}_A(a)) & = & \mathcal{P}_E(a), \mathtt{Unbox}_{size(A)} \text{ if } size(A) > 1 \\
\mathcal{P}_E(\lambda x : A.\, b^B) & = & \mathtt{Abstr}(\mathcal{P}_{x:A,E}(b^B), \mathtt{Return}_{size(B)}) \\
\mathcal{P}_E(b(a^A)) & = & \mathcal{P}_E(b), \mathcal{P}_E(a), \mathtt{Apply}_{size(A)} \\
\mathcal{P}_E(\Lambda X.\, a) & = & \mathcal{P}_E(a) \\
\mathcal{P}_E(a(B)) & = & \mathcal{P}_E(a) \\
\mathcal{P}_E(a, b) & = & \mathcal{P}_E(a), \mathcal{P}_E(b) \\
\mathcal{P}_E(a^{A\times B}.\mathtt{fst}) & = & \mathcal{P}_E(a), \mathtt{First}_{size(A),size(B)} \\
\mathcal{P}_E(a^{A\times B}.\mathtt{snd}) & = & \mathcal{P}_E(a), \mathtt{Second}_{size(A),size(B)} \\
\mathcal{P}_E(\mathtt{add\_float}(a, b)) & = & \mathcal{P}_E(a), \mathcal{P}_E(b), \mathtt{AddFloat} \\
\mathcal{P}_E(\mathtt{add\_float}) & = & \mathtt{Abstr}(\ \mathtt{Access}(0), \mathtt{Access}(1), \mathtt{Access}(2), \mathtt{Access}(3), \\
 & & \qquad \mathtt{AddFloat}, \mathtt{Return}_2)
\end{array}
$$

Figure 8: Compilation scheme for the restricted polymorphic calculus, mixing uniform and specialized representations.

$$
\frac{E \vdash a : A \quad E \vdash B \text{ type}}{E \vdash \mathtt{inleft}(a) : A + B}
\qquad\qquad
\frac{E \vdash A \text{ type} \quad E \vdash b : B}{E \vdash \mathtt{inright}(b) : A + B}
$$

$$
\frac{E \vdash a : A + B \quad E, x : A \vdash c : C \quad E, y : B \vdash d : C}{E \vdash \mathtt{case}\ a\ \mathtt{of}\ \mathtt{inleft}(x) \to c \mid \mathtt{inright}(y) \to d : C}
$$

Values of a sum type $A + B$ are usually represented as (tag, value) pairs. For instance, $\mathtt{inleft}(a)$ is represented as $(0, a)$, and $\mathtt{inright}(b)$ as $(1, b)$. Pattern-matching is then a simple test on the first component of the pair. Such pairs are not regular pairs, since the type of the second component depends on the value of the first one (A if it is 0, B if it is 1). This is no problem if uniform representations are used. In case of specialized representations, however, it may be the case that an element of type A and an element of type B have incompatible representations (e.g. different sizes); therefore, elements of type $A + B$ would have two incompatible representations, depending on the value of the tag, which is not known at compile-time.

A first solution is to take a representation compatible with both the one of A and the

one of B. For instance, regarding the size of $A + B$, we could take

$$size(A + B) = 1 + \max(size(A), size(B))$$

so that the second component of the dependent pair is always large enough to contain an element of A, or an element of B. This is the traditional implementation of variants in Pascal. The main drawback is some waste of space when $size(A)$ is not equal to $size(B)$. This becomes serious when the sums are heap-allocated, as parts of a large data structure.

Another solution is to systematically use uniform representations for the argument of a sum constructor. That way, a sum is always represented by one word for the tag, and one word for the argument, hence $size(A + B) = 2$. The necessary wrapping operations are performed by the constructors `inleft` and `inright`.

Both approaches are compatible with polymorphism. The compilation scheme given in section 3.3 easily extends to sums, with two additional cases for the generation of stub code:

$$\begin{aligned}\mathcal{T}(a^{A_1+A_2}) = \ & \texttt{case}\ a\ \texttt{of}\ \ \texttt{inleft}(x) \rightarrow \texttt{inleft}(\mathcal{T}(x^{A_1})) \\ & \quad\ \mid\ \texttt{inright}(y) \rightarrow \texttt{inright}(\mathcal{T}(y^{A_2}))\end{aligned}$$

and similarly for $\overline{\mathcal{T}}(a^{A_1+A_2})$.

4.2 Recursive types

Recursive types describe recursive data structures. In actual programming languages, they are usually introduced through the ability to name types and to refer to that name in the definition of the associated type. In a typed calculus, they arise when type expressions are not restricted to be finite trees any more, but allowed to be rational trees. We use the notation $\mu X.\, A$ for cycles; anywhere in a type expression, $\mu X.\, A$ can be replaced by $A\{X \leftarrow \mu X.\, A\}$, and conversely. For instance, the type of lists of integers is $\mu L.\, \texttt{Unit} + \texttt{Int} \times L$, where `Unit` is a special type containing exactly one value.

No other modifications of the type system are needed for recursive types. In particular, they introduce no additional typing rules. The main difficulty with recursive types is that the definitions we gave by induction on types are now possibly ill-founded: they should now read as a set of recursive equations, which may have no solution. For instance, with a naive *size* function, the size s of integer lists, defined as $\mu L.\, \texttt{Unit} + \texttt{Int} \times L$, must verify $s = 1 + \max(0, 1 + s)$, and this equation has no positive solution. This corresponds to the well-known fact that lists cannot be statically allocated "flat"; pointers must be used somewhere. Indeed, if we define integer lists as $\mu L.\, \texttt{Unit} + \texttt{Int} \times \texttt{Wrapped}(L)$, the size s' of this type is now $s' = 1 + \max(0, 1 + 1) = 3$, as expected.

To guarantee that all recursive types can be represented, it is necessary to introduce additional pointers in the representations. For instance, we may require that a value of a recursive type must be heap-allocated, and represented by a pointer, if it is a component of a structure (product or sum). This treatment can be integrated into the translation from general polymorphism to restricted polymorphism, as shown in figure 9.

Similarly, the stub code generated during the translation from general polymorphism to restricted polymorphism may now involve recursive functions. For instance, the function $f(a) = \mathcal{T}_{X \leftarrow \texttt{Float}}(a^{\mu Y.\, \texttt{Unit} + X \times Y})$, which specializes generic lists to lists of floats, is

	A recursive, B recursive,	A recursive, B not recursive,	A not recursive, B recursive,	A not recursive B not recursive
$[A \times B]$	$\texttt{Wrapped}[A] \times \texttt{Wrapped}[B]$	$\texttt{Wrapped}[A] \times [B]$	$[A] \times \texttt{Wrapped}[B]$	$[A] \times [B]$
$[a^A, b^B]$	$(\texttt{wrap}_A[a], \texttt{wrap}_B[b])$	$(\texttt{wrap}_A[a], [b])$	$([a], \texttt{wrap}_B[b])$	$([a], [b])$
$[a^{A\times B}.\texttt{fst}]$	$\texttt{unwrap}_A([a].\texttt{fst})$	$\texttt{unwrap}_A([a].\texttt{fst})$	$[a].\texttt{fst}$	$[a].\texttt{fst}$
$[a^{A\times B}.\texttt{snd}]$	$\texttt{unwrap}_B([a].\texttt{snd})$	$[a].\texttt{snd}$	$\texttt{unwrap}_B([a].\texttt{snd})$	$[a].\texttt{snd}$

Figure 9: Translation from general polymorphism to restricted polymorphism, in the presence of recursive types

defined as:

$$\begin{array}{rl} f(a) = & \texttt{case}\ a\ \texttt{of}\ \ \texttt{inleft}(n) \rightarrow \texttt{inleft}(n) \\ & \quad\ \mid\ \texttt{inright}(p) \rightarrow \texttt{inright}(\texttt{unwrap}_{\texttt{Float}}(p.\texttt{fst}), f(p.\texttt{snd})) \end{array}$$

That is, we should copy the list while applying the transformation $\texttt{unwrap}_{\texttt{Float}}$ to each of its elements. This is not practical, since this copying takes time and space proportional to the size of the original list, and is not correct when lists can be physically updated. Instead, we restrict further the representations of sums and products, and require that their components be systematically wrapped. This ensures that data structures never need to be recursively copied when converting between uniform and specialized representations. In particular, we may now take:

$$\begin{array}{rcccl} \mathcal{T}_{X\leftarrow B}(a^{A_1\times A_2}) & = & \overline{\mathcal{T}}_{X\leftarrow B}(a^{A_1\times A_2}) & = & a \\ \mathcal{T}_{X\leftarrow B}(a^{A_1+A_2}) & = & \overline{\mathcal{T}}_{X\leftarrow B}(a^{A_1+A_2}) & = & a \end{array}$$

We already came to that solution in the case of sums. For products, it may look overly restrictive. For instance, a pair of floating-point numbers cannot be represented "flat" any more, and requires heap-allocating both numbers. We shall see now how the introduction of general records instead of mere pairs alleviates this problem.

4.3 Records and variants

Actual programming languages, such as ML, provide general records and variants as data structures ("concrete types", in ML terminology), not only binary sums and binary products. Sums and products allow better naming of the components of a data structure, as well as more precise control over their types. For instance, coordinates in the plane could be described by the following record type:

```
type coord2 = {x: Float; y: Float}
```

which is more precise than `Float` × `Float`. In particular, no polymorphic function can destructure this record. Therefore, it can be represented efficiently by four words allocated "flat", without ever having to copy it with its components wrapped.

However, it is still possible to define polymorphic data structures by parameterizing a concrete type declaration, using type variables. For instance, binary polymorphic sum and product can be defined in the language as follows:

```
type X * Y = { fst: X; snd: Y };
type X + Y = inleft of X | inright of Y;
```

As we saw previously the components of such structures must use wrapped representations, in order to avoid copying. A simple, natural way to ensure this, while preserving efficient representations for types such as `coord2`, is to say that the representation of a concrete type is chosen once for all when it is defined. Components whose type is a type variable are, as usual, represented in the uniform way. In other terms, parameterized concrete types have one representation, whatever the parameters are: `List(Int)` and `List(Float)` and `List(`X`)` where X is a type variable all have compatible representations. This ensures that no recursive copying of data structures is ever needed.

The necessary wrapping and unwrapping are performed by the primitives for creation and access. To generate the right conversions, no special rules are needed: it suffices to treat these primitives as regular functions, possibly polymorphic. For instance, the projection `.x` for the type `coord2` can be given the type `coord2` $\rightarrow$ `Float`. Similarly, `.fst` has type $\forall X.\, \forall Y.\, X \times Y \rightarrow X$, and `inleft` has type $\forall X.\, \forall Y.\, X \rightarrow X + Y$. When these constructors are specialized to particular values of X and Y, the translation technique of section 3.3 automatically inserts the right coercions.

4.4 Application to ML

We have already covered the main features of ML. They can be integrated into compilation using specialized representations, at the cost of additional constraints on the representations. But it is sometimes necessary to revert to uniform representations. One may fear that these constraints are so restrictive that there is little benefit from specialized representations. This is not the case, however, as specialized representations address important weaknesses of ML implementations.

First of all, specialized representations allow efficient operations on base types which do not fit in a word, and in particular on floating-point numbers. This is an absolute requirement for any general-purpose programming language.

Second, though in ML all functions have exactly one argument (and one result), specialized representations makes it possible to pass a tuple of arguments to a function without heap-allocating the tuple, but simply by putting its components on the stack or in registers. Thus we get the efficiency of functions with several arguments (and several results as well), without having to modify the source language, nor restrict polymorphism in any way.

More generally, specialized representations allow monomorphic programs to be compiled just as efficiently as in traditional monomorphic languages, such as Modula for instance. The price to pay for polymorphism is paid only by those programs that actually use it. This contrasts with a tradition among high-level languages: that advanced features systematically hamper the performance of programs, even if they are not used.

5 Conclusions and related work

The techniques presented in this paper combine the cleanliness and expressiveness of unrestricted polymorphism with the efficiency of specialized representations. We developed one possible application, getting better implementation of an existing polymorphic language, ML. The dual application is to add full polymorphism to a conventional, Algol-like language while keeping efficient compilation of existing programs. One may hope that both approaches would converge toward languages with powerful type systems and efficient implementations, without putting strong constraints on the type system (for instance, abstract data types, or records with subtyping are easily accommodated), nor on the runtime system (in particular, interfacing with a garbage collector is still possible, using a combination of static type information for unwrapped data and run-time tagging for wrapped data).

Another interesting feature of our technique is to facilitate interfacing with existing code written in another language, such as libraries written in C: adopting unwrapped representations which are compatible with the ones of C (at least for the base types) alleviates the need for coercions between C and ML data formats.

There is comparatively little work on the problem of data representations in high-level languages. The fact that some typing is necessary to implement efficiently floating-point numbers and records is demonstrated in the design of many conventional imperative languages (such as C [6]), but is scarcely ever stated explicitly. In the area of Lisp compiling, tagging is known to be a performance bottleneck, and several attempts were made to avoid it, at least locally. A systematic approach to this problem can be found in a recent paper by Peterson [10]. He considers mixing tagged and untagged representations, in the setting of an untyped language, and focuses on finding an optimal mix of representations, one that minimizes total execution time. However, his analysis is expensive, and does not consider higher-order functions. Relying on type information, as presented herein, is certainly sub-optimal, but much more practical.

References

[1] Luca Cardelli. The Functional Abstract Machine. *Polymorphism*, 1(1), 1983.

[2] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[3] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing surveys*, 17(4), 1985.

[4] Luis Damas and Robin Milner. Principal type-schemas for functional programs. In *Proc. Symp. Principles of Programming Languages*, 1982.

[5] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'Etat, Université Paris VII, 1972.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Addison-Wesley, second edition 1988.

[7] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, pages 308–320, 1964.

[8] David C. J. Matthews. Poly manual. Technical Report 63, Computer Laboratory, University of Cambridge, 1985.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. The MIT Press, 1990.

[10] John Peterson. Untagged data in tagged environments: choosing optimal representations at compile-time. In *Functional Programming Languages and Computer Architecture*, 1989.

[11] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[12] J. C. Reynolds. Toward a theory of type structure. In *Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*. Springer-Verlag, 1974.