Type-Based Uncurrying for Unknown Function Calls

(Extended abstract)

Morten Rhiger
Roskilde University
Roskilde, Denmark
mir@ruc.dk

ABSTRACT

In strict functional languages, a left-to-right evaluation of an application of a curried function to n arguments generally requires creating $n-1$ intermediate closures and making n function calls. A type-based first-order uncurrying transformation eliminates these closures and requires just one call, but it only applies to calls to known (let-bound) functions. A type-based higher-order uncurrying transformation of calls to unknown (lambda-bound) functions is generally unsound. We propose to extend a higher-order uncurrying transformation with a local syntactic check in order to guarantee soundness and we investigate the extent to which existing code pass this check.

ACM Reference Format:
Morten Rhiger. 2023. Type-Based Uncurrying for Unknown Function Calls: (Extended abstract). In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In strict functional languages with a left-to-right evaluation order, evaluating the n-argument *curried* function call

```
f e1 e2 ... en
```

(where, for simplicity, we assume that `f` is a *trivial* expression, i.e, a value or a variable that require no further evaluation) generally requires generating $n-1$ intermediate closures and performing n calls, as expressed by the following explicit translation of the expression into A-Normal Form [3] (where, for simplicity, we assume that each e_i is a *serious* expressions, i.e., a call that does require further evaluation):

```
let v1 = e1
    c1 = f v1
    v2 = e2
    c2 = c1 v2
       ⋮
    vn = en
in  cn−1 vn
```

Thus, passing n arguments to the body of the curried function `f` is expensive in terms of both time and space: Each of the n calls require manipulation of registers or the stack and involve jumps to and from the code associated with the called closures. Similarly, generating each of the $n-1$ closures require allocating heap memory for the closure and initializing this memory.

On the other hand, evaluating the n-argument *uncurried* function call

```
f (e1, e2, ..., en)
```

does not generate any intermediate closures and performs just one call, as expressed by the following translation into A-Normal Form:

```
let v1 = e1
    v2 = e2
       ⋮
    vn = en
in  f (v1, v2, ..., vn)
```

For this reason, strict functional languages with a left-to-right evaluation order often prefer uncurried function calls. For example, in Standard ML (which is evaluated left-to-right [7]) many library routines are "unnecessarily" uncurried in the sense that they take tuples of input for apparently no other reason than to avoid the extra cost associated with the analogous curried call.

In order to support efficient evaluation of curried function calls, compilers for functional languages often attempt to optimize the calls by using in internal uncurried representation of functions, and by implicitly coercing these internal representations to a curried form whenever needed by the surface-level program [2, 4]. While this is straightforward for calls to known (let-bound) functions, it is generally not sound and complete for unknown (lambda-bound) functions in the presence or side effects.

The outline of this extended abstract is as follows. In Section 2 we present a source lambda calculus with strict, left-to-right evaluation order, a target lambda calculus with explicit multi-argument functions, and a type-directed translation from the source to the target language that produce efficient uncurried calls to known (i.e. let-bound) functions. In Section 3 we extend this to also handle calls to some unknown (i.e. lambda-bound) functions and we characterize exactly which. This is our main result. In Section 4 we examine which of a set of existing (library) functions fall within this characterization and can thus be compiled efficiently.

As a side remark, note that under a *right-to-left* evaluation of the n-argument curried function call

```
f e1 e2 ... en
```

control is not transferred to the body of `f` until all arguments are evaluated. This can be exploited by a right-to-left *push/enter* evaluation model [5, 6] that first pushes the values of each of the e_is on the execution stack and only then jumps to the body of `f`. This avoids the generation of intermediate closures and calling

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

these closures, but requires bookkeeping in the runtime system to keep track of the arities of functions. In this extended abstract we consider a only a left-to-right evaluation order and we stick to compile-time translations that eliminate curried functions and calls.

2 FIRST-ORDER UNCURRYING

In this section we present a type-directed translation from an ordinary lambda calculus with single-argument functions to a lambda calculus with explicit multi-argument function abstractions and applications. This is similar to the first-order fragments of Hannan and Hicks [4] and of Dargaye and Leroy [2] (although we stick to a simply-typed lambda calculus).

The source language is the simply-typed lambda calculus:

$$\text{(Source expressions)} \quad E ::= x \mid \lambda x.\, E \mid E\, E'$$

We assume a strict, left-to-right evaluation order of this calculus (but leave out the definition of the semantics from this extended abstract).

The target language is the simply-typed lambda calculus with explicit multi-argument functions. Expressions in the source language are given types in the target language. (In practice, these *representation types* are only used to guide the analysis and are not exposed to the programmer.) We let b range over a set of (non-function) base types. Square brackets encapsulate the multiple (formal or actual) parameters to function abstractions and applications.

$$\begin{aligned} \text{(Types)} \quad & t, u ::= b \mid [\overline{t}] \to u \\ \text{(Target expressions)} \quad & e ::= x \mid \lambda[\overline{x{:}t}].\, e \mid e\, [\overline{e'}] \\ \text{(Sequences of } \chi\text{s)} \quad & \overline{\chi} ::= \chi_1, \ldots, \chi_n \end{aligned}$$

A sequence $\overline{\chi} = \chi_1, \ldots, \chi_n$ may be empty, in which case $n = 0$. We let $|\chi_1, \ldots, \chi_n|$ = n. Again, we assume a strict, left-to-right evaluation order of this calculus.

In both the source and target languages, we allow monomorphic let bindings with the syntax `let` $x = e_1$ `in` e_2. These expressions can either be given types and translations by a straightforward extension of the type system below, or they can be seen as syntactic sugar for $(\lambda x.\, e_2)\, e_1$. We shall also allow $e_1; e_2$ as a shorthand for `let` $x = e_1$ `in` e_2 (for some variable x not in e_2) and we assume that there are predefined functions that produce side effects (such as `print`) in both the source and the target language.

We define a typing relation $\Gamma \vdash E : t/e$ that states that in type environment Γ, source expression E has type t and translates to target expression e.

2.1 Core translation

We can give a trivial non-optimizing type-directed translation that only produces single-argument functions and calls by stating the following typing rules for lambda abstractions and applications:

$$\frac{\Gamma, x{:}t \vdash E : u/e}{\Gamma \vdash \lambda x.\, E : [t] \to u/\lambda[x{:}t].\, e} \quad \text{(abs1)}$$

$$\frac{\Gamma \vdash E_1 : [t_2] \to t/e_1 \quad \Gamma \vdash E_2 : t_2/e_2}{\Gamma \vdash E_1\, E_2 : t/e_1[e_2]} \quad \text{(app1)}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t/x} \quad \text{(var)}$$

$$\frac{\Gamma, x_1{:}t_1, \ldots, x_n{:}t_n \vdash E : u/e}{\Gamma \vdash \lambda x_1.\, \cdots \lambda x_n.\, E : [t_1, \ldots, t_n] \to u/\lambda[x_1{:}t_1, \ldots, x_n{:}t_n].\, e} \quad \text{(abs*)}$$

$$\frac{\Gamma \vdash E : [t_1, \ldots, t_n] \to u/e \quad \left(\Gamma \vdash E_i : t_i/e_i\right)_{i \in 1..n}}{\Gamma \vdash E\, E_1\ \ldots\ E_n : u/e\, [e_1, \ldots, e_n]} \quad \text{(app*)}$$

Figure 1: First-order type-directed uncurrying, core

Such a type system would accept precisely any simply-typed lambda expression, but with "trivial translations." The goal is to characterize when curried lambda abstractions and applications can safely be translated into multi-arguments uncurried abstractions and applications. A common, simple, and effective approach characterizes syntactically a function defined by nested lambda abstractions

$$\lambda x_1.\, \lambda x_2.\, \cdots \lambda x_n.\, E$$

as eligible for uncurrying by giving it type

$$[t_1, t_2, \ldots, t_n] \to u$$

and by translating it into

$$\lambda[x_1{:}t_1, x_2{:}t_2, \cdots, x_n{:}t_n].\, e$$

(where e is the translation of E). And dually, when an expression E has type

$$[t_1, t_2, \ldots, t_n] \to u$$

(with translation e) then the nested applications

$$E\, E_1\, E_2\ \ldots\ E_n$$

is translated into

$$e\, [e_1, e_2, \ldots, e_n]$$

(where each E_i has type t_i with translation e_i). (In the examples that follow, we assume that a lowercase target expressions e is the translation of a capitalized source expresions E if they have the same subscripts or primes.)

Figure 1 presents the core of such a type-based translation. The rules abs1 and app1 are admissible in this system, and are not considered part of the type system.

The type system is sound and complete: It types and translates precisely the expressions of the simply-typed lambda calculus. But this comes at the "cost" of being nondeterministic: the number of lambda abstraction to group together (the n in rule abs*) must be chosen to match all call sites of the function (the least n in all instances of rule app* where this function is applied). This prevents a separate compilation of function definitions. For example, without knowledge of its call sites, the function defined by

$$\texttt{let}\ f = \lambda x_1.\, \lambda x_2.\, E$$

must conservatively be given type $[t_1] \to [t_2] \to u$ and trivial translation

$$\texttt{let}\ f = \lambda[x_1{:}t_1].\, \lambda[x_2{:}t_2].\, e$$

$$\frac{\Gamma \vdash E : t/e \quad t \le u/c}{\Gamma \vdash E : u/c(e)} \tag{sub}$$

$$\frac{(u_i \le t_i/c_i)_{i\in 1..n} \quad t \le u/c}{[t_1,\ldots,t_m,t_{m+1},\ldots,t_n] \to t \le [u_1,\ldots,u_m] \to [u_{m+1},\ldots,u_n] \to u/C} \tag{$\le$-curry}$$

where $C(e) = \texttt{let}\ f = e\ \texttt{in}$
$\lambda[x_1{:}u_1,\ldots,x_m{:}u_m].$
$\lambda[x_{m+1}{:}u_{m+1},\ldots,x_n{:}u_n].$
$c(f\ [c_1(x_1),\ldots,c_m(x_m),c_{m+1}(x_{m+1}),\ldots,c_n(x_n)])$

Figure 2: First-order type-directed uncurrying, subtyping

After all, the function may, somewhere, be applied to just one argument.

An important goal of type-based uncurrying is to separate the compilation (i.e., the typing and translation) of function *definitions* (abstractions) from their *uses* (applications). The extension in Section 2.2 fully uncurry known function definitions and introduce coercions when a use of a a function requires less currying. (This is known.) The extension in Section 3 fully uncurry (some) unknown function uses and introduce coercions when a definition of a a function requires less currying. (This is our main contribution.)

2.2 Coercion-based subtyping

To address the weakness of the core type system in Figure 1, we extend it with a coercion-based subtype mechanism. This allows multi-argument *curried* function to be translated efficiently to multi-argument *uncurried* functions since a function can now be coerced to the different types expected at its call sites.

Figure 2 presents the part of the type system that handles coercion-based subtyping. It introduces one additional typing rule sub and a subtype relation $t \le u/c$. This subtype relation states that type t is a subtype of type u, witnessed by the mapping c: The subtype relation is populated (currently by the single rule $\le$-curry) such that c maps expressions of type t to expressions of type u.

Note that since C in rule $\le$-curry is a transformation on expressions (rather that a function on values) it must in general generate a let-binding of the expression e to a variable outside the outermost λ and then apply that variable inside the innermost λ account for potential side effects in e. Note also that rule $\le$-curry allow for standard co- and contravariance among the result- and argument types.

In the special case where e is a trivial expresion and where no co- or contravariance is needed, rule $\le$-curry specialize to

$$\frac{}{[\overline{t},\overline{t'}] \to u \le [\overline{t}] \to [\overline{t'}] \to u/curry}$$

which insert an application of the familiar currying transformation

$$curry(e) = \lambda[\overline{x{:}t}].\,\lambda[\overline{x'{:}t'}].\,e\ [\overline{x},\overline{x'}]$$

that map (target) expressions of type $[\overline{t},\overline{t'}] \to u$ to (target) expressions of type $[\overline{t}] \to [\overline{t'}] \to u$.

A system that combines Figures 1 and 2 can translate the function definition

$$\texttt{let}\ f = \lambda x_1.\,\lambda x_2.\,E$$

into

$$\texttt{let}\ f = \lambda[x_1{:}t_1, x_2{:}t_2].\,e$$

independently of calls to the function. Calls that receives as many arguments as expected are compiled efficiently. Calls to fewer arguments than expected results in target programs that involves a coercion. For example, the calls $f\,E_1\,E_2$ and $f\,E_3$ are translated to

$$f\ [e_1, e_2]$$

and

$$(\lambda[x_1{:}t_1].\,\lambda[x_2{:}t_2].\,f\ [x_1,x_2])\ [e_3]$$

respectively.

The type-based translation described here is well known [2, 5] and may capture as many as 80% of all calls [2, 6]. However, it only captures calls to known (let-bound) functions. Curried calls to unknown (lambda-bound) functions must, in general, remain curried. For example, in

$$\texttt{let}\ g = \lambda f.\,f\,E_1\ \ldots\ E_n$$

f must be given type $[t_1] \to \cdots \to [t_n] \to u$ and the expression translates to

$$\texttt{let}\ g = \lambda[f].\,f\ [e_1]\ \ldots\ [e_n]$$

involving an "inefficient" curried call to f.

3 HIGHER-ORDER UNCURRYING

What causes the generation of the inefficient curried call to unknown functions? Given the source program at the end of the previous section, can't we just instead generate

$$\texttt{let}\ g = \lambda[f].\,f\ [e_1,\ \ldots,\ e_n]$$

involving an "efficient" uncurried call and give f type $[t_1,\cdots,t_n] \to u$? The answer is (generally) no: Without a global flow analysis we cannot guarantee that no curried function will be passed as an argument to the function g. Thus, such a system would be incomplete with respect to the simply typed lambda calculus: Some programs typable in simply typed lambda calculus would not have a type in this system.

Ok, but can't we just *uncurry* such curried functions when passed as an argument to g, dually to how we *curry* in Section 2.2? Again

the answer is (generally) no: uncurrying is unsound if both (the function passed as) f and some of the e_is have side effects. For example, the source program

```
let g = λf. f 1 (print 3; 4)
in  g (λx.
         print 2;
         λy. y)
```

may (incorrectly) translate to the program

```
let g = λ[f]. f [1, (print 3; 4)]
in  g [λ[x, y].
         (λ[x].
             print 2;
             λy. y) [x] [y]]
```

that involves an uncurrying coercion. But where the source program first prints 2 and then prints 3, the target program first prints 3 and then prints 2.

3.1 Sound uncurrying coercions, idea

The translation above yields an incorrect result because both some arguments to f and the body of f itself have side effects. We propose a simple characterization of when calls to an unknown (lambda-bound) multi-argument curried function can be translated as "efficient" uncurried calls, namely this:

> If a lambda-bound variable f is applied to n *syntactic values* or *variables* at *all* of its occurrences then these calls can be translated into n-argument uncurried calls and the type of f (and hence the type of the lambda that binds it) must reflect this representation.

This proposal, and the formalization below, is the main result of this extended abstract.

As a motivating example, consider the following slightly modified definition of g from above:

$$\texttt{let}\ g = \lambda f.\, f\ 1\ 4$$

In the body of g, at all of its occurrences (of which there is only one), f is passed two arguments and both of these arguments are syntactic values. Thus, f can be given type $[\texttt{int}, \texttt{int}] \to u$, g can be given type $[[\texttt{int}, \texttt{int}] \to u] \to u$, and the let-binding is translated to

$$\texttt{let}\ g = \lambda[f].\, f\ [1, 4]$$

The compilation (typing and translation) of the definition of g is separated from the compilation of its uses: g can be directly applied to a function that can be uncurried according to type rule abs*. And if the argument to g cannot be uncurried by rule abs*, then a coercion uncurries that argument. For example, the application

```
g (λx.
     print 2;
     λy. y)
```

is translated to

```
g [λ[x, y].
     (λ[x].
          print 2;
          λy. y) [x] [y]]
```

Both the source and the target program prints 2 (and returns 4), as required.

3.2 Sound uncurrying coercions, formalization

To formalize the characterization just outlined, a type system need to keep track of which (unknown) functions are always applied to n syntactic values or variables (i.e, trivial expressions). To this end we extend the target language with a type $\{\overline{t}\} \to u$ of functions that must only be applied to syntactic values and we single out trivial expressions from the syntax of expressions. The result is as follows.

$$\begin{array}{rl} \text{(Types)} & t, u ::= b \mid [\overline{t}] \to u \mid \{\overline{t}\} \to u \\ \text{(Expressions)} & e ::= v \mid e\ [\overline{e'}] \\ \text{(Trivial Expressions)} & v ::= x \mid \lambda[\overline{x{:}t}].\, e \end{array}$$

We extend the type system with one new typing rule for the application of unknown functions that are always applied to n syntactic values (rule appv*) and we add an uncurrying coercion for the $\{\overline{t}\} \to u$ function type (rule ≤-uncurry). There is no explicit introduction rule associated with (or dedicated expression syntax for) the type $\{\overline{t}\} \to u$. Instead expressions of type $\{\overline{t}\} \to u$ are generated by lifting expressions of type $[\overline{t}] \to u$ (rule ≤-lift). The result is shown in Figure 3. In rule appv*, each v_i is a trivial expression.

Again, in the special case where e is a trivial expresion and where no co- or contravariance is needed, rule ≤-uncurry specialize to

$$\frac{}{\{\overline{t}\} \to \{\overline{t'}\} \to t \le \{\overline{t}, \overline{t'}\} \to u / \mathit{uncurry}}$$

which insert an application of the familiar uncurrying transformation

$$\mathit{uncurry}(e) = \lambda[\overline{x{:}t}, \overline{x'{:}t'}].\, e\ [\overline{x}]\ [\overline{x'}]$$

that map (target) expressions of type $\{\overline{t}\} \to \{\overline{t'}\} \to u$ to (target) expressions of type $\{\overline{t}, \overline{t'}\} \to u$.

Let us revisit the previous example. In the following definition

$$\texttt{let}\ g = \lambda f.\, f\ 1\ 4$$

the unknown f can be given type $\{\texttt{int}, \texttt{int}\} \to u$, since at all of its occurrences (of which there is only one), f is applied to two trivial expressions. Therefore, g has type

$$[\{\texttt{int}, \texttt{int}\} \to u] \to u$$

and its definition is translated into

$$\texttt{let}\ g = \lambda[f].\, f\ [1, 4]$$

If the source program applies g to a function that can be uncurried according to type rule abs*, then that function gets type $[\texttt{int}, \texttt{int}] \to u$, which, by rule (≤-lift) can be trivially coerced to type $\{\texttt{int}, \texttt{int}\} \to u$.

If the source program applies g to a function that cannot be uncurried by rule abs*, that function gets type $[\texttt{int}] \to [\texttt{int}] \to u$.

$$\frac{\Gamma(f) = \{t_1, \ldots, t_n\} \to b \quad \left(\Gamma \vdash E_i : t_i/v_i\right)_{i \in 1..n}}{\Gamma \vdash f\ E_1\ \ldots\ E_n : b/f\ [v_1, \ldots, v_n]} \tag{appv*}$$

$$\frac{\left(u_i \le t_i/c_i\right)_{i \in 1..n} \quad t \le u/c}{\{t_1, \ldots, t_m\} \to \{t_{m+1}, \ldots, t_n\} \to t \le \{u_1, \ldots, u_m, u_{m+1}, \ldots, u_n\} \to u/U} \tag{≤-uncurry}$$

where $U(e) = \texttt{let}\ f = e\ \texttt{in}$
$\lambda[x_1{:}u_1, \ldots, x_m{:}u_m, x_{m+1}{:}u_{m+1}, \ldots, x_n{:}u_n].$
$c(f\ [c_1(x_1), \ldots, c_m(x_m)]\ [c_{m+1}(x_{m+1}), \ldots, c_n(x_n)])$

$$\frac{}{[\bar{t}] \to u \le \{\bar{t}\} \to u/id} \tag{≤-lift}$$

where $id(e) = e$

Figure 3: Higher-order type-directed uncurrying

But such a function can be coerced first to type $\{\texttt{int}\} \to \{\texttt{int}\} \to u$ by rule (≤-lift) and then to type $\{\texttt{int}, \texttt{int}\} \to u$ by rule (≤-uncurry). And as required, a function of type $\{\texttt{int}, \texttt{int}\} \to u$ can precisely be passed to g.

Thus, translating the calls to the unknown f in the body of g into uncurried calls still allows g to receive both curried and uncurried functions whether or not they have any side effects. Thus, soundness and completeness is reestablished.

4 REAL-WORLD EXAMPLES

To evaluate the usefulness of the type-directed translation presented in Sections 2 and 3, functions documented in the Standard ML Basis [8] or the SML/NJ Libraries [9] and functions from the actual SML/NJ implementation (which are undocumented but still exported from the structure) have been used as examples. Among these functions, we are interested in those that take multi-argument functions as input and check whether calls to these (unknown) function can be translated into uncurried calls. In Standard ML, calls to these functions are already uncurried so we actually check if they *could instead* have been curried and if the proposed translation could represent them internally as efficient uncurried calls.

Many library functions take one or more functions as input, but few take multi-argument functions as input. We have identified the 16 functions in Table 1. All of these functions take a multi-argument function as their first input. These unknown function are already uncurried, but nothing prevents them from being rewritten to a curried style. A manual check reveals that except for `foldri`, `foldMapl`, and `foldMapr` all calls to these unknown multi-argument function supply all argument (three for `foldli`, two for the rest) and that these argument are indeed syntactically trivial expressions (values or variables). Figure 4 shows two typical examples. In the figure, calls to the unknown is shown in red.

The implementations of `foldri`, `foldMapl`, and `foldMapr` are shown in Figure 5. These functions fail to match the requirements: one of the unknowns passed to these functions is applied to a serious expression. In the figure, calls to the unknown is shown in red and the offending serious argument is underlined. In other words, 13 out of the 16 library functions in Table 1 (which comprise all the candidates that we found in the libraries) can be rewritten such that calls to the unknown first argument are curried, but the calls to these unknown functions can still be translated to efficient uncurried calls.

"Effect-agnostic" functions that do not match the requirements from Section 3 can sometimes be rewritten (by the programmer) so that they do. For example, if e_1 and e_2 are serious expression,

$$\texttt{let}\ g = \lambda f.\ f\ e_1\ e_2$$

can be rewritten to

$$\begin{aligned}\texttt{let}\ g' = \lambda f.\ &\texttt{let}\ v_1 = e_1\ \texttt{in}\\ &\texttt{let}\ v_2 = e_2\ \texttt{in}\\ &\texttt{in}\ f\ v_1\ v_2\end{aligned}$$

These functions not equivalent: if e_2 has an effect and f has an effect after having received its first argument, g and g' produce these effects in different order. But higher-order functions like this contrived example or like the library functions from Table 1 seldom relies on a particular ordering of effects, so the rewritten function can often replace the original. And in the rewritten function, f *is* applied to only trivial expressions.

5 CONCLUSIONS AND RELATED WORK

We have extended a well-known approach to automatically uncurrying of calls to known (let-bound) functions to also handle calls to unknown (lambda-bound) functions. The translation is type-directed and provides a local characterization of when calls to an unknown function can be uncurried.

Benton et al. implement a compiler from ML to the Java Virtual Machine that replace curried functions by multi-argument uncurried functions via arity raising [1]. It is unclear exactly which functions and function calls are compiled efficiently. Hannan and Hicks validate the correctness of transformations involving uncurrying [4]. Dargaye and Leroy characterize and prove the correctness of possible higher-order uncurrying coercions [2]. Both Hannan and Hicks and Dargaye and Leroy study type (and translation) systems that are nondeterministic. In contrast, we propose a deterministic

Function	Type	
`List.foldr`	$(\alpha \times \beta \to \beta) \to \beta \to \alpha\ \texttt{list} \to \beta$	
`List.foldl`	$(\alpha \times \beta \to \beta) \to \beta \to \alpha\ \texttt{list} \to \beta$	
`List.foldli`	$(\texttt{int} \times \alpha \times \beta \to \beta) \to \beta \to \alpha\ \texttt{list} \to \beta$	
`List.foldri`	$(\texttt{int} \times \alpha \times \beta \to \beta) \to \beta \to \alpha\ \texttt{list} \to \beta$	Fails
`List.reduce`	$(\alpha \times \alpha \to \alpha) \to \alpha \to \alpha\ \texttt{list} \to \alpha$	
`List.appi`	$(\texttt{int} \times \alpha \to \texttt{unit}) \to \alpha\ \texttt{list} \to \texttt{unit}$	
`List.mapi`	$(\texttt{int} \times \alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$	
`List.mapPartiali`	$(\texttt{int} \times \alpha \to \beta\ \texttt{option}) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$	
`List.findi`	$(\texttt{int} \times \alpha \to \texttt{bool}) \to \alpha\ \texttt{list} \to (\texttt{int} \times \alpha)\ \texttt{option}$	
`List.revMapi`	$(\texttt{int} \times \alpha \to \beta) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$	
`List.revMapPartiali`	$(\texttt{int} \times \alpha \to \beta\ \texttt{option}) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$	
`List.concatMapi`	$(\texttt{int} \times \alpha \to \beta\ \texttt{list}) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}$	
`List.foldMapl`	$(\alpha \times \gamma \to \gamma) \to (\beta \to \alpha) \to \gamma \to \beta\ \texttt{list} \to \gamma$	Fails
`List.foldMapr`	$(\alpha \times \gamma \to \gamma) \to (\beta \to \alpha) \to \gamma \to \beta\ \texttt{list} \to \gamma$	Fails
`List.collate`	$(\alpha \times \alpha \to \texttt{order}) \to \alpha\ \texttt{list} \times \alpha\ \texttt{list} \to \texttt{order}$	
`HashConsMap.insertWith`	$(\beta \times \beta \to \beta) \to (\alpha, \beta)\ \texttt{map} \times \alpha\ \texttt{obj} \times \beta \to (\alpha, \beta)\ \texttt{map}$	

Table 1: Higher-order ML library routines with multi-argument input functions

```
fun foldl f e []      = e
  | foldl f e (x::xr) = foldl f (f(x, e)) xr;

fun collate cmp (xs, ys) =
  let fun h []       []       = EQUAL
        | h []       (y1::yr) = LESS
        | h (x1::xr) []       = GREATER
        | h (x1::xr) (y1::yr) =
          case cmp(x1, y1) of
              EQUAL => h xr yr
            | res   => res
  in  h xs ys
  end
```

Figure 4: Example ML library functions that pass

```
fun foldri f init l =
  let fun lp (_, []) = init
        | lp (i, x::xs) = f (i, x, lp (i+1, xs))
  in  lp (0, l)
  end

fun foldMapl reduceFn mapFn init l =
  foldl (fn (x, acc) => reduceFn(mapFn x, acc)) init l

fun foldMapr reduceFn mapFn init l =
  foldr (fn (x, acc) => reduceFn(mapFn x, acc)) init l
```

Figure 5: ML library functions that fail

translation that uncurries all known function definitions and all unknown function calls whose arguments are trivial expressions.

We have also performed a preliminary manual search among ML library functions that revealed three examples of unknown calls that could not pass that local characterization. Since this result is independent of the *uses* of these library function, the translation presented allows both the flexibility provided by curried functions and an efficient implementation of definitions of and calls to these functions.

REFERENCES

[1] Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998*, pages 129–140. ACM, 1998.

[2] Zaynah Dargaye and Xavier Leroy. A verified framework for higher-order uncurrying optimizations. *High. Order Symb. Comput.*, 22(3):199–231, 2009.

[3] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[4] John Hannan and Patrick Hicks. Higher-order uncurrying. *High. Order Symb. Comput.*, 13(3):179–216, 2000.

[5] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[6] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, 2006.

[7] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, 1997.

[8] The Standard ML Basis Library. https://smlfamily.github.io/Basis/index.html. Accessed July 2023.

[9] The SML of NJ Library. https://www.smlnj.org/doc/smlnj-lib/index.html. Accessed July 2023.