Calling Variadic Functions from a Strongly-typed Language

Matthias Blume

Toyota Technological Institute at Chicago

blume@tti-c.org

Mike Rainey

University of Chicago

mrainey@cs.uchicago.edu

John Reppy

University of Chicago

jhr@cs.uchicago.edu

Abstract

The importance of providing a mechanism to call C functions from high-level languages has been understood for many years and, these days, almost all statically-typed high-level-language implementations provide such a mechanism. One glaring omission, however, has been support for calling variadic C functions, such as `printf`. Variadic functions have been ignored because it is not obvious how to give static types to them and because it is not clear how to generate calling sequence when the arguments to the function may not be known until runtime. In this paper, we address this longstanding omission with an extension to the NLFFI foreign-interface framework used by Standard ML of New Jersey (SML/NJ) and the MLton SML compiler. We describe two different ways of typing variadic functions in NLFFI and an implementation technique based on the idea of using state machines to describe calling conventions. Our implementation is easily retargeted to new architectures and ABIs, and can also be easily added to any HOT language implementation that supports calling C functions.

Categories and Subject Descriptors   D.3.4 [*Programming Languages*]: Processors

General Terms   Languages

Keywords   foreign-function interfaces, compilers, interpreters

1. Introduction

Most statically-typed high-level languages provide a mechanism for calling C functions. Such a mechanism is required to allow access to system services and the wide selection of C libraries. Because C is an unsafe language with a weak type system, there has been significant focus on the problem of assigning high-level types to C functions [FLMP99, Blu01, CFH+03]. None of this existing work, however, addresses the problem of typing variadic C functions, such as

```
int printf (const char *, ...);
```

The ellipses in this function prototype specifies that calls to this function may take zero or more arguments in addition to the character pointer. Since this prototype does not fix the number or types of arguments to `printf`, it is hard to give it a strong static type. Furthermore, in a functional language, such as ML, a variadic function should be a first-class value that can be used at many, possibly

unknown, call sites, which means that we cannot have a fixed marshaling policy for its arguments.

In this paper, we describe an approach to supporting variadic C functions as first-class citizens in strongly-typed languages, such as ML or Haskell. We have implemented this mechanism in the SML/NJ compiler as an extension to Blume's NLFFI mechanism [Blu01], but the techniques are not specific to SML/NJ.

Our approach requires a minor effort on behalf of the programmer. To make a variadic call, the programmer supplies just two pieces of information: the C header file containing the function prototype and the call site in ML. The foreign-function interface handles everthing else.

There are two main obstacles to supporting variadic functions in ML: the first is providing a typing scheme that is both statically checkable while supporting variable arity functions. One approach is to define a tagged union of argument types and to represent the arguments as a list of tagged values. We reject this approach because it does not fit well with the NLFFI embedding of the C type system. Instead, we use a technique suggested by Danvy for typing unparsing functions [Dan98]. The second, and more difficult, obstacle is implementing a call to a foreign variadic function, when we do not have static knowledge of either the number or types of arguments. To address this problem we use a state-machine based model of calling conventions [BD95, OLR06] that is interpreted at call time. Since this interpreter must be implemented in machine code and is specific to the host operating system and architecture, we generate it from a declarative description.

1.1 Motivation

Since most C functions are not variadic, this restriction on foreign-interface mechanisms has not been viewed as severe, but there are a number of compelling examples where support for calling variadic functions is required. Formatted I/O is the most common example, which include not only the C standard I/O library, but I/O operations for many other common C libraries, such as curses, SQLite, and MPI. Another example are libraries for constructing data structures, such as RPC marshaling and the ATerm library [ATe].

The Cocoa libraries on Mac OS X are written in Objective-C, which means that to access the native GUI components one has to be able to interoperate with Objectice-C objects. The Objective-C runtime has a C interface; specifically, the function

```
id objc_msgSend (id, SEL, ...);
```

can be used to send a message specified by `SEL` to the object specified by `id`. By providing support for variadic functions, we make it possible to interoperate with Objective-C libraries, such as as Cocoa.

1.2 Contributions

This paper makes the following contributions:

- Our design enables a concise implementation, and as a consequence provides new justification for fully supporting variadic functions in a foreign-function mechanism.

- We describe how to handle the static typing of variadic functions in the NLFFI type framework.

- We describe an interpreter for generating the calling sequences for variadic functions.

- Our implementation isolates machine dependencies in a declarative specification of the calling convention and uses the ML-Risc code-generation framework to generate the interpreter from a single portable source code.

- The generated interpreter is an assembly program with C calling conventions that can be compiled and linked into other language implementations.

1.3 Roadmap

The rest of the paper is organized as follows. In the next section, we review the way in which NLFFI embeds the C type system into the ML type system. Then in Section 3, we describe two approaches to typing variadic functions in NLFFI. Sections 4 through 7 describe the implementation of variadic calls. We then discuss related work in Section 8 and conclude in Section 9.

2. NLFFI

NLFFI [Blu01] is a foreign function interface for Standard ML. It was originally developed for the SML/NJ compiler and has subsequently been adapted to work with MLton as well. The main design principle behind NLFFI is to follow the idea of *data-level interoperability* [FPR00, FPR01]. This principle means that all C types and interfaces are made directly available to the high-level language by presenting them as values of various *abstract types*. These abstract types come equipped with operations that closely correspond to syntactic constructs of the C language. As a result, access to all C data can be done directly from the high-level language, and there never is any need for writing low-level "glue" code that deals with data bit-level data representation, marshaling of function arguments and results, *etc.*

The ML-side type structure of these C-level data and operations have been carefully designed in such a way that the ML type system effectively enforces the same rules that would normally be enforced by a C compiler. (Thus the choice of the acronym "NLFFI," which stands for *no longer foreign function interface*, a pun which is meant to express that the rules of the "foreign" language are "natively" expressed and enforced by the type system of the high-level language.)

Notice that NLFFI does not attempt to unify ML-side representatives of C types with commonly available ML types, even where naively this would seem to make sense. For example, the C type `double` is not represented by the ML type `real` (or, even better, `Real64.real`). Instead, there is a new abstract type `C.double` for which NLFFI provides conversion routines to and from ML's `real`. In short, the philosophy is to keep the two type worlds separate, but to also provide all necessary tools that make it possible to write the code that mediates between these worlds directly in the high-level language itself. One main advantage of this design is that it fully separates the foreign interface from implementation details of the high-level language, in particular the problem of data representation.

NLFFI consists of two main parts.

- The first part is a library implementing the "fixed" part of the interface, *i.e.*, the part that corresponds to predefined types, type constructors, and operations of the C language. A prime example of such a fixed type constructor is C's star (`*`), a unary type constructor taking a type `T` to the type of pointers to `T`. Examples of fixed operations are C's *address-of* operator (`&`) and C's pointer-dereferencing operator (also written `*`).

- The second part is a tool (called `ml-nlffigen`) for converting C interfaces containing type definitions as well as function- and variable declarations into a corresponding ML interface. This tool parses an actual C source or header file and uses the front-end of a C compiler (implemented using SML/NJ's CKit Library[1]) to extract the necessary information for constructing the ML equivalent of the interface. Among other administrative issues, the tool performs two important sub-tasks.

  - It creates new ML-side abstract types representing freshly declared C-side `struct` and `union`-types.

  - It also issues special (non-standard) ML code that the ML compiler later uses to generate calling sequences for every function prototype encountered within the C interface.

Example: Suppose a C header file contains the following two declarations:

```
extern double sin (double);
extern double atan2 (double, double);
```

For these two function declarations, `ml-nlffigen` produces two ML structures, named `F_sin` and `F_atan2`, respectively. Among other things, each of the structures contains a value component `fptr` representing the C function pointer to the respective C function.[2] In particular, we have the following specifications:

```
val F_sin.fptr : unit ->
      (C.double -> C.double) C.fptr
val F_atan2.fptr : unit ->
      (C.double * C.double -> C.double) C.fptr
```

For each `fptr`, NLFFI provides a wrapper (named `f`) as a convenience, since the by far most common use is that of actually invoking the underlying function. The wrapper includes built-in conversions from and to ML's native types, so that we would see the following specifications:

```
val F_sin.f : Real64.real -> Real64.real
val F_atan2.f : Real64.real * Real64.real
      -> Real64.real
```

3. NLFFI and variadic functions

The creation of calling sequences specific to function prototypes in the C interface works only for "fixed" prototypes, *i.e.*, those not making use of C's *ellipsis* notation. In the case of variadic calls, it is not the prototype but the call site that determines the calling sequence. Thus, NLFFI's approach, which is based on using C-side type information, breaks down at this point.

As a result, NLFFI — like most of its foreign-function interface brethren — has punted on the issue of variadic C functions. In this paper, we explain how we have rectified this situation.

3.1 Call site specific type information

The type of a variadic C function does not fully specify how it can be used. The details are left to each individual call site and must be harvested there in order to generate the correct calling sequence that adheres to the calling conventions. In a sense, we can think of variadic C functions as being *ad-hoc polymorphic*, since the code to be generated is determined by a process similar to overloading resolution.

---

[1] The CKit Library is available from `smlnj.org`.

[2] The `fptr` values are actually suspensions, since access to the underlying C values goes through a dynamic linking layer.

The SML language does not provide support for programmer-declared overloading, which makes it difficult to mirror the behavior of C directly, but one might imagine that it would be possible to (ab)use SML's *parametric* polymorphism. For example, `printf` could be considered to have the type

```
val printf : string -> 'a -> int
```

The type `string` represents the fixed (*i.e.*, statically known) portion of the argument list, while the universally quantified type variable `'a` represents the variable portion, corresponding to `printf`'s C prototype.

```
int printf (const char *, ...);
```

Each call site would instantiate `'a` to a particular ground type, which would then serve as the basis for determining the correct calling sequence to be used at that site. For example, a call with two arguments, a `real` and a `int` could simply be.

```
printf "x=%d y=%f" (3, Math.pi)
```

Here `'a` is instantiated to `real * int`, and the SML compiler could make use of this information to generate the same calling sequence that a C compiler would generate for

```
printf ("x=%d y=%f", 3, M_PI);
```

Unfortunately, this technique would not work robustly, because in some sense the SML type system is too flexible. For example, it lets us write code of the form.

```
fun myprintf fmt args =
        printf ("** " ^ fmt) args
```

Here, the calling sequence for the foreign call of `printf` would not be determined by *its own* call site, but by that of the surrounding η-wrapper `myprintf`, which itself is not a foreign function. Allowing such things to happen seems to be in the spirit of the rest of the language, but going down this path makes the treatment of variadic foreign calls pervasive throughout large parts of the implementation or else requires whole-program analysis. These solutions seem far too complicated to be worth the effort of designing and implementing. Another solution is to force each call site of a variadic foreign function to fully instantiate the type of its argument. We reject this approach as well, since it would require a significant change of the SML type system, as the polymorphism in the type of `printf` would have to be distinguished from the polymorphism in the type of native SML values.

3.2   Tagged argument lists

Since it seems difficult to harvest enough static information at the call site of a foreign function, we adopt a more dynamic approach: arguments of a variadic call are given as a sequence, for example a list of SML values. Since the SML type system insists on homogeneous sequences, this requires that each argument be injected into some sort of universal type. The standard technique is to define a datatype that has one variant for each possible argument type.

Unfortunately, NLFFI provides an infinitely large family of possible argument types (just like the C type system does). There are `int`, `double`, `int*`, `double*`, `int**`, `double**`, and so forth. Of course, as far as calling conventions are concerned, it is not really necessary to distinguish between different pointer types as they all get represented identically as machine addresses. NLFFI itself represents all pointers as such addresses, it merely "dresses them up" as different abstract types using the technique of phantom types. Thus, instead of using an explicit datatype, we can define an abstract type `arg` and provide a set of injection functions. Some of the injection functions, for example that for pointers, are polymorphic and work on entire families of types.

```
type arg
val sint_arg : C.sint -> arg
...
val double_arg : C.double -> arg
val ptr_arg : 'o ptr -> arg
```

Variadic calls then take concrete lists of abstract `args`. For example, the above call of `printf` becomes.

```
printf "x=%d y=%f"
       [C.sint_arg 3, C.double_arg Math.pi]
```

3.3   Danvy-style typing

Instead of using a concrete list of `args` which requires explicit injection of every value into the `arg` type, it is also possible to employ a design that syntactically separates a specification of the type sequence from the list of values. The trick is to use a variant of Danvy's technique for statically typing `printf` in SML [Dan98].[3]

Suppose a particular call site of a variadic function that returns a value of type t_r provides arguments of types $t_1, \ldots, t_n$. We capture this fact using an abstract type constructor `va_sig` which is instantiated to type $(t_1 \rightarrow \cdots \rightarrow t_n \rightarrow t_r)$ `va_sig`. Let $t_1, \ldots, t_i$ with $i \leq n$ be a segment of the type list in such a variadic signature. Borrowing the technique of *difference lists* from logic programming, this sequence can be expressed as a pair $(t_1 \rightarrow \cdots \rightarrow t_i \rightarrow$ `'a`, `'a`$)$, where `'a` is a type variable. For this purpose the NLFFI interface provides the two-argument type constructor `vargs`. We want to be able to combine a partial argument specification of type

$$(t_1 \rightarrow \cdots \rightarrow t_i \rightarrow \text{'a, 'a) vargs}$$

with a specification of type

$$(t_{i+1} \rightarrow \cdots \rightarrow t_n \rightarrow \text{'b, 'b) vargs}$$

to form a specification of type

$$(t_1 \rightarrow \cdots \rightarrow t_n \rightarrow \text{'b, 'b) vargs.}$$

If (`'a`, `'b`) `vargs` is defined to be the type of functions from `'b va_sig` to `'a va_sig`, then this is conveniently achieved by SML's predefined function composition combinator **o**. The identity function `va_none` serves as the neutral element and represents an empty sequence of arguments. Ultimately, the values of type `vargs` are constructed from primitive single-argument specifications of type (`'e`, `'r`) `varg`. Since this is defined to be synonymous with (`'e -> 'a`, `'a`) `vargs`, each primitive specification adds one curried argument to the overall signature.

```
type 'a va_sig
type ('a, 'b) vargs = 'b va_sig -> 'a va_sig
type ('e, 'a) varg = ('e -> 'a, 'a) vargs
val va_none : ('a, 'a) vargs
```

For each NLFFI type or type family there is one value of type `varg`:

```
val va_sint : (sint, 'a) varg
...
val va_double : (double, 'a) varg
val va_ptr : ('o ptr, 'a) varg
```

Each of these values corresponds to one of the above-mentioned injection functions into the `arg` type. For convenience, our interface also provides argument combinators with built-in conversions from native ML types.

```
val va_ml_int : (Int32.int, 'a) varg
...
val va_ml_double : (Real64.real, 'a) varg
```

---

[3] Unlike OCaml's `format` mechanism, Danvy's solution relies just on the basic principles of the ML type- and module system and does not require any special compiler support. As a result, it is less ad-hoc and easily generalizes to arbitrary situations that deal with variadic calls.

Notice that since type `va_sig` is abstract, all `vargs` values that can ever be constructed have types of the form $(t_1 \to \cdots \to t_n \to$ `'a`, `'a`) `vargs`. As explained above, the type variable `'a` can be further instantiated to extend the list of arguments on the right. In a complete argument specification (*i.e.*, one that is being presented to `va_call`, see below), `'a` represents the return type.

A variadic function takes its arguments in two parts: a fixed one (corresponding to the portion before the ellipsis in the C prototype) and a variable one (corresponding to the ellipsis itself). To express this, NLFFI defines an abstract type constructor and a function for dispatching calls.

```
type ('f, 'r) va_fptr
val va_call : ('f, 'r) va_fptr ->
                    ('a, 'r) vargs -> 'f -> 'a
```

A value of type (`'f`, `'r`) `va_fptr` represents a variadic function with fixed arguments of type `'f` and a result of type `'r`. The variable part of the argument list is not mentioned here, since it is chosen on a per-call basis.

The first argument to `va_call` is a pointer to the function to be called. The corresponding specification of the variable part of the arguments is given as the second curried argument. Recall that its type has the form $(t_1 \to \cdots t_n \to$ `'r`, `'r`) `vargs` where $t_1, \ldots, t_n$ are the types of the individual values within the variable portion of the argument list.

The signature of `va_call` forces `'r` to match the overall result type. Therefore, if `f` has type (t_f, t_r) `va_fptr` and `spec` has type $(t_1 \to \cdots \to t_n \to$ `'r`, `'r`) `vargs`, then the type of the call `va_call f spec` has the type $t_f \to t_1 \to \cdots \to t_n \to t_r$, which is exactly what we desire.

In our running example, the `va_fptr` value representing `printf` would have the following type:[4]

$$\textbf{val } printf : (string, sint) \text{ va\_fptr}$$

A use of `printf` with an `int` and a `double` argument then looks like this.

```
va_call printf (va_ml_sint o va_ml_double)
        "x=%d y=%f" 3 Math.pi
```

In other words, the instantiated `printf` (without its arguments) has the following type.

```
va_call printf (va_ml_sint o va_ml_double) :
       string -> Int32.int -> Real64.real -> sint
```

3.4 Implementation of the Danvy-style abstract interface

The abstract interface described above does not commit to a particular representation of its abstract type constructors. One simple strategy is to fall back to a representation of argument lists as list of tagged unions — at least internally.[5]

Danvy's approach is to make use of *continuation-passing* style in the implementation of the interface. In particular, type `'a` `va_sig` is the type of continuations that expect a list of arguments (in NLFFI's internal representation) and map it to a result of type `'a`.

---

[4] This is a slight simplification, as the actual type of the fixed argument would be `(schar, ro) obj ptr`. This corresponds to C's `const char *`.

[5] A more involved approach is to use the interface to drive the *Staged Allocation* machinery (see Section 5) directly, which would make it possible to avoid some intermediate data structures by immediately going to *located arguments* (see Section 6).

```
datatype arg
   = SINT_ARG of internal_sint
     ...
   | DOUBLE_ARG of internal_double
   | PTR_ARG of internal_addr

type 'a va_sig = arg list -> 'a
type ('a, 'b) vargs = 'b va_sig -> 'a va_sig
type ('e, 'a) varg = ('e -> 'a, 'a) vargs
fun va_none s = s
```

Each predefined value of type `varg` adds one argument and arranges for it to be pushed onto the list of tagged values collected so far.

```
fun va_sint k l i = k (SINT_ARG i :: l)
...
fun va_double k l d = k (DOUBLE_ARG d :: l)
fun va_ptr k l p = k (PTR_ARG p :: l)
```

Variadic function values are represented by closures that take two arguments: the fixed portion of the arguments and the variable portion represented as a list of `arg` values. When such a closure is invoked, it combines fixed and variable portions into a single `arg` list that is suitable for being used by the low-level mechanism for dispatching a C call. Function `va_call` receives such a closure together with its `varargs` specification as well as its fixed arguments. It then "runs" the specification, causing it to pick up the variable portion as additional curried arguments. The initial continuation given to the specification receives these variable arguments as a list of `arg` values. (Notice that the list contains the arguments in reverse order.) At this point `va_call` has obtained all the necessary ingredients, so it is now able to invoke the variadic function closure.

```
type ('f, 'a) va_fptr = 'f * arg list -> 'a
fun va_call f spec fixed =
    spec (fn args => f (fixed, rev args)) []
```

The code for the function closure varies only in the part that marshals the fixed arguments. Therefore, it is generated by the `ml-nlffigen` tool based on the function's C prototype. For the example of `printf`, here is a sketch of what the generated code looks like

```
type c_funptr = internal_addr
val printf_funptr : c_funptr = ...
fun printf_fptr (fixed, vargs) =
    dispatch_lowlevel_call
       (printf_funptr, PTR_ARG fixed :: vargs)
```

The only fixed argument of `printf` is a C string, which means it is a pointer value. Therefore, it gets added to the joint argument list using the `PTR_ARG` tag.

We give a detailed description of the implementation of `dispatch_lowlevel_call`, *i.e.*, the technically most difficult part, later in the paper — beginning with Section 4.

3.5 Specialized protocols

In C there is no mechanism that reliably lets a variadic function detect the end of its argument list. To cope with this problem, different commonly used functions use different protocols for communicating this information from the caller to the callee.

For example, `printf` relies on the format string given as the first and only fixed argument. Format specifications (substrings starting with a `%`-character) within the format string indicate the number and types of subsequent arguments. In other cases where the type (or at least the representation) within the argument list does not vary, one sometimes relies on passing a special "end marker," *i.e.*, a value that is distinguishable from all possible arguments. In C, such a value is often taken to be `NULL`. One example for such a function is the Unix `execl` variant of the `exec` system call. Another option is to precede the variable part with an integer

argument that specifies the number of values that follow. Other conventions might require arguments to be passed in pairs or other similar patterns.

These protocols are inherently unsafe, since they are designed and implemented in an *ad hoc* fashion, so the compiler does not know about them. More seriously, even if they were known to the compiler, it would often not be possible to statically determine whether the caller has provided correct information.

Although on the ML side we face the same inherent limitations, the programmer can capture some of these protocols by hiding their fragile parts behind abstractions. Implementing such abstractions is quite easy under the list-of-tagged-values design of the interface. We now illustrate how it can be done in the setting of Danvy-style typing. For this we examine three of the cases mentioned above, excluding the `printf` case since its solution was essentially already given in Danvy's original work [Dan98].

Example: end marker

To support the end marker `NULL`, our interface provides a special combinator `va_null`.

```
val va_null : ('a, 'a) vargs
```

Like `va_none` it does not alter the type of the curried function. However, it causes a constant value to be passed at the specified position. The low-level implementation of `va_null` is straightforward.

```
fun va_null k l = k (PTR_ARG NULL_const :: l)
```

This idea can be generalized to provide arbitrary "constant" arguments, *i.e.*, values that are given within the specification itself and which do not have a corresponding curried argument. The combinator `va_const` converts a value of type `('e, 'a) varg` into a function of type `'e -> ('a, 'a) vargs`:

```
val va_const : ('e, 'a) varg ->
               'e -> ('a, 'a) vargs
```

The following one-liner implements `va_const`; we will see a use of it in our last example:

```
fun va_const c x k l = c k l x
```

Now let us use `va_null` to guarantee that calls to `execl` receive the trailing `NULL` argument.[6] Recall that the C prototype is

```
int execl (const char *path, const char *arg, ...);
```

This results in the following "raw" NLFFI representative:[7]

```
val execl : (string * string, sint) va_fptr
```

The "safe" version of `execl`, which always passes the required trailing `NULL` without having to specify it, can be coded up simply as

```
fun safe_execl spec path arg =
    va_call execl (spec o va_null) (path, arg)
```

Example: argument count

Suppose function `sum` sums its n arguments of type `double` but requires n to be passed as a fixed `int` argument first

```
double sum (int n, ...);
```

The raw NLFFI representative is then

```
val sum : (sint, double) va_fptr
```

---

[6] For the purpose of this example we do not enforce the other invariant, namely that all arguments before the end marker must be strings. A technique for constraining arguments to particular types is shown later.

[7] Again, we simplify and write `string` for `(schar, ro) obj ptr`.

Behind an abstraction, we can count the number of arguments within the specification — and at the same time restrict all arguments to type `double`. The idea is to mimic the roles of types `va_sig` and `vargs` using new abstract types `ssig` and `svargs`. Here a value of type `'a ssig` is a partially constructed specification paired with the number of elements in it

```
structure Sum :> sig
    type 'a ssig
    type ('a, 'b) svargs = 'b ssig -> 'a ssig
    val sdouble : (double -> 'a, 'a) svargs
    val safe_sum : ('a, double) svargs -> 'a
  end = struct
    type 'a ssig = ('a, double) vargs * int
    type ('a, 'b) svargs = 'b ssig -> 'a ssig
    fun sdouble (s, n) = (va_double o s, n+1)
    fun safe_sum spec = let
        val (spec', n) = spec (va_none, 0)
        in va_call sum spec' n end
  end
```

In the body of function `safe_sum` we first run the "outer" specification, resulting in the argument count `n` together with the full "inner" specification to be handed off to `va_call`. Since the only constructor for `svargs` is `sdouble`, it is guaranteed that all arguments to `safe_sum` will be of type `double`.

This approach guarantees that the first argument to `sum` will specify the correct number of arguments. A similar approach can be used to check for consistency in the use of functions such as `printf`, where the first argument implies not only the number of additional arguments but also their types. Of course, such a consistency check would occur at run time.[8]

Example: pairing arguments

Suppose we have a function `plot` that takes one `string` argument and then a list of `int`-`double` pairs. Moreover, let the value `0` mark the end of the list. Like the C prototype

```
void plot (const char *, ...);
```

the raw NLFFI representative does not capture these invariants

```
val plot : (string, unit) va_fptr
```

Again, the solution is to hide the raw `plot` function behind an abstract interface. Its overall structure is similar to that of the `sum` example, except `psig` is literally an abstract copy of `va_sig`.

```
structure Plot :> sig
    type 'a psig
    type ('a, 'b) pvargs = 'b psig -> 'a psig
    val sint_double :
        (sint -> double -> 'a, 'a) pvargs
    val plot : ('a, unit) pvargs -> string -> 'a
  end = struct
    type 'a psig = 'a va_sig
    type ('a, 'b) pvargs = 'b psig -> 'a psig
    fun sint_double s = va_sint (va_double s)
    fun plot spec =
        va_call plot (spec o va_const va_sint 0)
  end
```

Notice the use of `va_const` applied to `va_sint` and the value `0`, which causes the unconditional addition of that value to the end of the argument list.
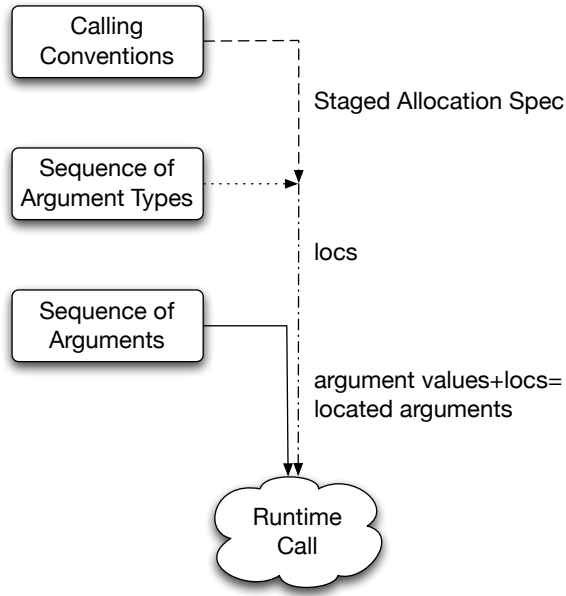
Figure 1. Three pieces of information go into the generation of a run-time call: the calling conventions, the sequence of argument types, and the sequence of corresponding argument values. The time at which the sequence of types is known varies depending on whether or not the function in question is variadic and on whether the call is performed from C or as a foreign call from ML.

4. Variadic calls at the machine level

Despite being conceptually straightforward, the remaining piece of a foreign-function interface is a challenge to implement. To make a foreign call, we must execute a particular sequence of machine instructions that first place arguments in specific machine registers and stack locations, then call the foreign function, and finally obtain the return result. We call this sequence of instructions the *calling sequence*. On most architectures, calling sequences for variadic functions are either identical to or are slightly more complex than fixed-arity functions, but, as we will see, variadic calling sequences are significantly trickier to support in ML.

Example: variadic calling sequence Here we consider the x86-64 calling sequence of our running example.

```
printf("x=%d y=%f", 3, 3.14);
```

To place arguments, we copy the first two to general-purpose registers using 64- and 32-bit moves respectively and copy the third to a floating-point register using a 64-bit move. Because this function is variadic, we must store the number of floating-point arguments in the `%rdx` register. This step is only necessary for variadic functions, and is the only variation from a fixed-arity function on the x86-64. After we perform these steps, we jump to the code address of `printf`. If the arguments had outnumbered the available registers, we would need to spill some to the stack.

The blueprint for generating calling sequences is the *calling convention*. Calling conventions are a surprisingly difficult part of implementing a compiler, as many researchers have noted [BD95,

---

[8] Some C compilers perform this particular test (*i.e.*, consistency of arguments for `printf` and its brethren) statically. However, this is a fairly brittle ad-hoc solution, since it does not generalize to user-defined functions and argument-passing protocols or even just the situation where the format string is not statically known.

OLR06]. Conventions tend to vary significantly across different architectures and operating systems, and C conventions are no exception. The official definitions for C conventions consist of elaborate and subtle rules that are often defined in prose. There have been many documented cases where the conventions themselves were buggy or imprecise or where compilers had incorrect encodings of the conventions.

Language-based approaches have proven to be an elegant solution for implementing calling conventions. Two successful examples of this approach are CCL [BD95] and staged allocation [OLR06]. Their basic technique is as follows. They define a domain-specific language for specifying calling conventions, and an allocator machine. The allocator machine takes a calling convention specification and a function prototype, and assigns the argument and return parameters to machine locations. For instance, suppose we fed our `printf` example to the allocator machine. We would get back the locations we need for passing the parameters, which are the two general-purpose registers and the floating-point register. With these locations in hand, the compiler can easily generate the full calling sequence.

The final step of the language-based approach, statically generating the calling sequence, does not work for our variadic calls. For reasons we describe below, we must instead rely on dynamic techniques. As illustrated in Figure 1, there are three major pieces of information that must be available by the time a foreign function call is generated.

1. The calling conventions for the current combination of machine architecture, operating system, and compiler.

2. The sequence of types corresponding to the arguments of a particular call.

3. The sequence of argument values.

These three pieces become available at different times. Calling conventions are fixed for a given compiler on a given platform. At the other extreme, argument values generally do not become available until the call is actually to be performed at runtime. The time at which the types of the arguments become known differs depending on the kind of function that is being called and also depending on whether the call is native or foreign.

- For ordinary (fixed-arity) functions, the types are given by the prototype. They do not vary with call sites.

- For variadic functions to be called from C, the types become known locally on a per call site basis.

- For variadic functions to be called from ML, the types become available along with the corresponding argument values at runtime.[9]

To generate the calling sequence for a particular call, one only needs to know the first two pieces of information: the calling conventions and the sequence of types. As we have explained, in ML we do not have a good handle on the argument types until run time. Thus, the caller must be able to perform *any* calling sequence on the fly, which makes supporting variadic calls a thorny issue.

Because of these challenges, existing foreign-interface mechanisms for strongly-typed languages omit support for variadic functions. It turns out, however, that with the right choice of techniques and the reuse of existing machinery for generating calling sequences, we can implement variadic function calls as a retargetable and portable library. To do so, we must overcome a major technical challenge: how can we perform any calling sequence on the fly,

---

[9] In principle, at least for Standard ML, the types could be obtained from a whole-program analysis. But this relies on certain fragile properties of the type system, e.g., the absence of polymorphic recursion.
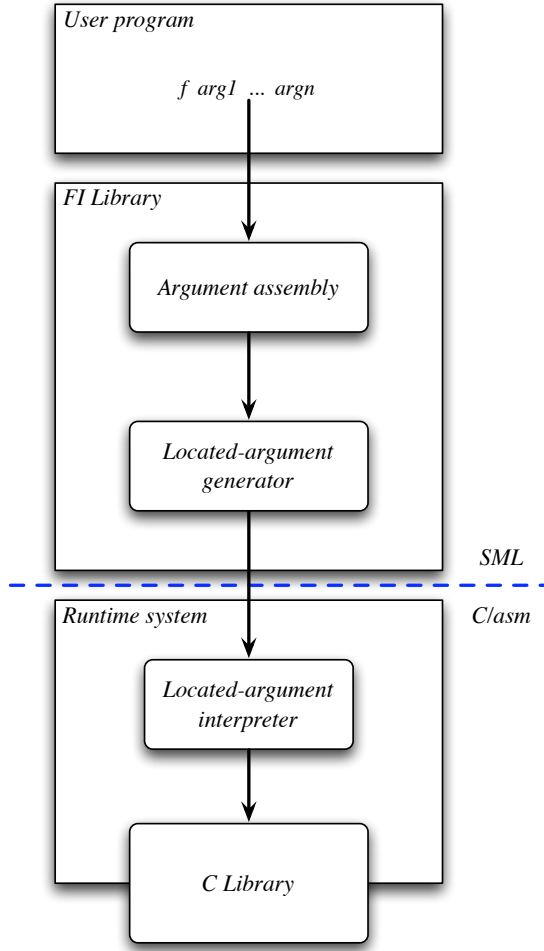
Figure 2. Calling $f\,arg_1\,\dots\,arg_n$ from SML

but at the same time maintain a reasonable complexity budget? Our solution rests on the following two insights:

First insight We can re-use the language-based approaches, *e.g.* staged allocation, to allocate machine locations just before we need them. Although this step gets us most of the way to a solution, we are seemingly stuck: somehow we need to turn those locations into a calling sequence. One could imagine using a just-in-time compiler to generate calling sequences on the fly, but SML/NJ does not provide such a feature.

Second insight Instead of generating code, we can program an *interpreter* to perform the calling sequence on the fly. For each step of the calling sequence, the interpreter processes instructions such as, "move the 32-bit integer argument x into register r", or "move the 64-bit floating-point argument y into stack location l". We store these instructions in a data structure that we have dubbed a *located argument*.

Figure 2 gives an overview of the way that a variadic-function call is implemented in our system. As shown in this figure, making the call to a variadic function f consists of the following steps:

1. A sequence of tagged arguments $arg_1, \dots, arg_n$ to a sequence of argument locations $loc_1, \dots, loc_n$.

```
datatype loc_kind
    = GPR
    | GSTK
    | FPR
    | FSTK
type req = (width * loc_kind * int)
```

Figure 3. Requests in staged allocation.

2. Convert each arg_i/loc_i pair to the located argument $larg_i$ and then these located arguments are passed to the call-sequence interpreter.

3. Interpret each $larg_i$ in order, and finally make the call to f.

The first two steps are implemented in SML, while the interpreter is a machine-generated assembly routine.

5. Argument assembly

For this step of our implementation, we draw upon the technique of staged allocation [OLR06]. We made this design choice for several reasons. The implementation effort is minimal: the staged-allocation machine, which is the bulk of the code, is defined by a succinct operational semantics with 17 state transitions. Many calling conventions are already supported, so getting the details right is dead simple. And we have already implemented staged allocation as a stand-alone library that is included with the MLRisc code-generation framework.

The main idea of staged allocation is to view the placement of parameters as an allocation problem. Each parameter is assigned a *request*, which consists of an integer bit width; a kind, which indicates the kind of location in which a parameter might be passed; and an integer alignment that constrains the address of whatever memory location holds the parameter. Figure 3 shows our encoding of requests. For generality, the original staged-allocation work does not specify all of the kinds that are necessary, but we can be explicit and use only those necessary for C: integer registers and stack locations and floating-point registers and stack locations.

We convert an argument to a request by obtaining the width, kind and alignment. These details are architecture specific and uninteresting, so for our discussion we assume that the following function is available.

```
fun argToRequest (a : arg) =
        (widthOfArg a,
         kindOfArg a,
         alignOfArg a)
```

Recall our example of `printf` from Section 4, We represent its arguments as follows,

```
[PTR_ARG 0x102200, SINT_ARG 3, DOUBLE_ARG 3.14]
```

where the address `0x102200` is a pointer to the string `"x=%d y=%f"`. On the x86-64 these arguments would yield the following requests.

```
[(64, GPR, 8), (32, GPR, 8), (64, FPR, 8)]
```

5.1 Calling conventions

In staged allocation, we specify calling conventions in a small formal language. Instructions in this language are called *stages*. A stage receives a request and either modifies that request and passes it on to a future stage or the stage attempts to allocate a location. Calling conventions usually consist of two sequences of stages, one that specifies how to pass parameters and one that specifies how to return results.

Counters
```
structure SA = StagedAllocation
val [cParamStk, cParamGpr, cParamFpr] =
      [SA.freshCounter (), SA.freshCounter (),
         SA.freshCounter ()]
val [cRetGpr, cRetFpr] =
      [SA.freshCounter (), SA.freshCounter ()]
```
Store
```
val store0 =
     SA.init [cParamStk, cParamGpr, cParamFpr,
               cRetFpr, cRetGpr]
```
Registers
```
val paramGprs = [rdi, rsi, rdx, rcx, r8, r9]
val paramFprs = [xmm0, xmm1, xmm2, xmm3, xmm4,
                     xmm5, xmm6, xmm7]
val retGprs = [rax, rdx]
val retFprs = [xmm0, xmm1]
```
Argument-passing specification
```
val params : SA.stage list = [
    (* stage 1 *)
    SA.WIDEN (fn w => Int.max (64, w)),
    (* stage 2 *)
    SA.CHOICE [   (* stage 2.1 *)
       (fn (w, k, store) => k = GPR,
        SA.SEQ [
            SA.BITCOUNTER cParamGpr,
            SA.REGS_BY_BITS(cParamGpr, paramGprs)
       ]),        (* stage 2.2 *)
       (fn (w, k, store) => k = FPR,
        SA.SEQ [
            SA.BITCOUNTER cParamFpr,
            SA.REGS_BY_BITS(cParamFpr, paramFprs)
       ]),        (* stage 2.3 *)
       (fn (w, k, store) =>
              k = GSTK orelse k = FSTK,
        SA.OVERFLOW {counter=cParamStk,
                      blockDirection=SA.UP,
                      maxAlign=16})
      ],
    (* stage 3 *)
    SA.OVERFLOW {counter=cParamStk,
                  blockDirection=SA.UP,
                  maxAlign=16}
   ]
```
Returning specification
```
val rets : SA.stage list = [
    SA.WIDEN (fn w => Int.max (64, w)),
    SA.CHOICE [
       (fn (w, k, store) => k = GPR,
        SA.SEQ [
            SA.BITCOUNTER cRetGpr,
            SA.REGS_BY_BITS (cRetGpr, retGprs)]),
       (fn (w, k, store) => k = FPR,
        SA.SEQ [
            SA.BITCOUNTER cRetFpr,
            SA.REGS_BY_BITS (cRetGpr, retFprs)])]
   ]
```

Figure 4. The x86-64 calling convention for staged allocation.

Example: x86-64 calling convention   Figure 4 shows our encoding of the C calling convention for the x86-64. We need several pieces to make the full specification.

Counters These variables track the state of the allocation sequence. For example, the counter cPassStk holds the stack offset and the counter cPassGpr holds the bits allocated to the parameter-passing integer registers.

Store This container maps counters to their integer values.

Registers These lists include all the possible registers for passing parameters and returning values.

Argument-passing specification This sequence of stages encodes how to pass arguments. In the first stage, we widen the bit width of the request up to 64 bits. We then enter a *choice* stage, where we evaluate a predicate that receives the triple (w, k, store) and makes a decision based on the kind k. If the kind of request is an integer (Stage 2.1), we try to allocate a general-purpose register. This process entails incrementing the bit counter with the BITCOUNTER instruction and then picking out the next available register with the REGS_BY_BITS instruction. The process is similar for floating-point requests (Stage 2.2). In the case that we must pass the request in memory (Stage 2.3), we allocate bytes from the *overflow block*, which in our case is just the C stack. The direction UP causes us to allocate from higher to lower addresses. The alignment can be at most 16-bytes. If none of the previous choices are taken (Stage 3), we allocate bytes for the request on the stack.

Returning specification This sequences of stages encodes how to pass return parameters. The rules are similar to how we pass arguments, except for a couple differences. Return parameters use a separate set of registers, and there are no rules for passing on the stack. This omission appears wrong, since the x86-64 conventions specify that structs are returned on the stack. But because the *caller* is responsible for allocating stack space, we do not need to worry about it here.

5.2   The staged allocation machine

The heart of staged allocation is an allocator machine that is defined by an operational semantics. This machine takes a calling-convention specification, a store, and sequence of requests, and it returns a machine location. We represent machine locations by the loc datatype.

```
datatype loc
   = REG of reg
   | BLOCK_OFFSET of int
   | COMBINE of (loc * loc)
   | NARROW of (loc * width * loc_kind)
```

Registers and stack offsets have the expected meaning. There are also composite locations, which include combinations and narrowing. Narrowed locations specify a necessary run-time type coercion. As an example, to pass a char parameter on the x86-64, we need to coerce it to a long.

The allocate function implements a single step of the machine; each step consumes a calling convention, request, and store, and returns a location and a modified store.

```
val allocate : stage list -> (req * store)
                     -> (loc * store)
```

Using standard techniques, we lift our machine to operate on sequences of requests.

```
val allocateSeq : stage list -> (req list * store)
                    -> (loc list * store)
```

5.3   Dynamically invoking staged allocation

The code in Figure 5 shows our run-time technique for invoking staged allocation. To select the appropriate calling conventions, we dynamically check the architecture and the operating system if necessary.

Example: x86-64 locations   If hand staged allocation the requests from our printf example, we get back x86-64 registers for passing our parameters.

```
fun dynStagedAlloc reqs = let
    val (params, store0) =
            if (Compiler.architecture = "x86_64")
                then (X8664CConv.params,
                        X8664CConv.store0)
                else (* handle other machines *)
                        ...

    val (seq, _) =
            allocateSeq params (reqs, store0)
    in
        seq
    end
```

Figure 5. Calling staged allocation for different machines.

```
[NARROW(REG rdi,  64, GPR),
 NARROW(REG rsi,  64, GPR),
 NARROW(REG xmm0, 64, FPR)]
```

Example: Sparc locations For variety, we consider our `printf`
example on the Sparc. We pass the first two parameters in general-
purpose registers as before, but pass the floating-point parameter in
two general purpose registers.

```
[NARROW(REG r8,   32, GPR),
 NARROW(REG r9,   32, GPR),
 COMBINE (
   NARROW(REG r10, 32, GPR),
   NARROW(REG r11, 32, GPR))]
```

6. Located arguments

Conceptually, the runtime interpreter takes the list of locations
(type `loc`) and the corresponding list of argument values. Based
on the location information it places each argument in the correct
register or stack location and then dispatches the call by jumping to
the entry point of the function.

To minimize the complexity of the interpreter, it is useful to
pre-process its arguments by combining each argument value with
its location information. During this process we also flatten the
arguments. The resulting combination of a flattened argument and
its location is what we call a located argument. It contains just
enough information to encode a single step of the calling sequence.

- `arg` contains the actual argument data;
- `k` specifies the location kind as in staged allocation;
- `width` is the actual bit width of the argument;
- `narrowing` specifies a possible narrowing of the argument;
- `loc` specifies either the register id or the stack offset for storing
  the argument;
- `offset` is the offset into the argument data. A nonzero offset
  indicates that the argument is split across multiple hardware
  locations. We scale the offset by the `width` field.

```
type located_argument = {
        arg : arg,
        k : loc_kind,
        width : int,
        narrowing : int option,
        loc : int,
        offset : int
    }
```

Given an argument and a staged-allocation location, it is trivial
to create the corresponding located arguments. We support only a
single narrowing operation per location, as is sufficient for most
machines. Combined locations, on the other hand, require multiple
located arguments, with each differentiated by its `offset`. Thus,

```
typedef void* Word_t;
enum loc_kind { GPR=0, FPR, GSTK, FSTK };
struct located_arg_s {
    union {
        Word_t* p; long l; int i;
        char* s; double d;
    } arg;
    loc_kind k;
    int width;
    int narrowing;
    int loc;
    int offset;
};
```

Figure 6. The C encoding for located arguments.

we define the function below, which takes an argument and a
location and returns one or more located arguments.

```
val mkLocatedArg : (arg * loc)
                        -> located_argument list
```

We lift this function to sequences of arguments by using a pairwise
map, where the lengths of the two input lists are the same.

```
val mkLocatedArgs =
        List.concat o ListPair.mapEq mkLocatedArg
```

Returning to our `printf` example , we have the following
located arguments (on the x86-64).

```
[ { arg=PTR_ARG 0x102200, k=GPR, width=64,
    narrowing=NONE, loc=rdi, offset=0 },
  { arg=SINT_ARG 3, k=GPR, width=32,
    narrowing=SOME 64, loc=rsi, offset=0 },
  { arg=DOUBLE_ARG 3.14, k=FPR, width=64,
    narrowing=NONE, loc=xmm0, offset=0 } ]
```

7. The interpreter

Unfortunately, we cannot simply pass the located argument list
directly to our interpreter, since we are unable to know which way
SML/NJ lays out this structure in memory. So, instead we marshal
the located arguments using a standard data layout.

```
val marshalLocdArgs : located_argument list
                        -> C.ptr
```

To keep the implementation portable, we require that the layout
conforms to that of the C struct in Figure 6.

In this process, we are also responsible for allocating space for
the located arguments. Our implementation currently uses heap
allocation through `malloc` for simplicity, but a more efficient
alternative is to allocate space on the stack.

7.1 Connecting ML and the interpreter

Although it is implemented as an assembly routine, we have chosen
to make our interpreter appear to outside code as an ordinary C
function. Specifically, the interpreter has a standard C prologue and
epilogue. This choice might be slightly less efficient than using
some specialized calling convention, but we believe that in this
case portability is more important. Because it has a fixed-arity C
prototype, any compiler that supports ordinary C calls can use our
interpreter.

We give our interpreter the *static* C interface below. It takes
a pointer to the vararg function we want to call and the located-
argument data structure, and carries out the vararg call.

```
extern void* VarArgInterp (
        void* varFun,
        struct located_arg_s* locdArgs,
        int nArgs);
```

With all these pieces in place, we can connect our ML pro-
gram to our interpreter. Figure 7 shows this implementation, which

```
fun dispatch_lowlevel_call (cFun, args) = let
        val reqs = List.map argToRequest args
        val locs = dynStagedAlloc reqs
        val locdArgs =
                mkLocatedArgs (args, locs)
        val nLocdArgs = List.length locdArgs
        val locdArgsForC =
                marshalLocdArgs locdArgs
        in
            primApplyCFun(VarargInterp,
                cFun,
                locdArgsForC,
                nLocdArgs)
        end
```

Figure 7. Connecting ML to our interpreter.

```
for i in 0 ... nArgs-1
  if locdArgs[i].kind = GPR
   then if locdArgs[i].width = 32
    then if locdArgs[i].narrowing = 0
      then if locdArgs[i].loc = r0
        then
         let j = locdArgs[i].offset * 4
         load.32 locdArgs[i].arg.s[j] into reg0
         ...
        ...
      ...
    ...
  end for
  call *varFn
```

Figure 8. Pseudocode for our vararg interpreter. The temporary `j` is the byte offset into the argument.

allocates locations, creates located arguments, marshals them for C, and finally passes them to our interpreter. We use the compiler's existing C-calling facility, which in our case is called `primApplyCFun`. As mentioned before, nothing special happens here, so any standard C-calling facility will do.

7.2 Implementing the interpreter

The pseudocode in Figure 8 is a sketch of our interpreter. For each located argument, we wind through a series of alternatives for where to place the argument. We represent each of these alternatives as a branch in the code. By doing so we free up as many registers as possible, since we are effectively encoding all intermediate steps in the program counter. The interpreter first resolves the kind, the width, the potentially necessary coercion, and the destination location. Next the interpreter places a chunk of the argument in the destination.

The nested branching structure of the interpreter leads to tedious and delicate implementations for each architecture. The x86-64, for example, requires cases for each of its 16 general-purpose and 16 floating-point registers. For each register it requires cases for the two possible sizes, an so on. In the end, the interpreter needs to cover more cases than we care to encode by hand.

7.3 Generating the interpreter

Luckily, we can use an off-the-shelf tool to generate our interpreter from a single source, and thereby keep all hand-written code in ML. To accomplish this task we use MLRisc, the retargetable code generator for SML/NJ [GGR94]. MLRisc models a generic RISC architecture with a simple register-transfer language called MLTree. We code our interpreter once as an MLTree program, and generate it for each supported architecture. Since we are able to

```
functor GenCoreInterpFn (
    val paramGprs : reg list
    val paramFprs : reg list
    val gprWidths : width list
    val fprWidths : width list
    val spReg : T.rexp
    val defaultWidth : width
    val callerSaveRegs : reg list
    val callerSaveFRegs : reg list
) :
  sig
    val genCoreInterp : {
        varFn : T.rexp,
        locdArgsPtr: reg,
        nArgs : T.rexp
      } -> T.stm list
  end
```

Figure 9. Signature of our interpreter generator.

generate all of the cases automatically, the generator code turns out to be fewer than 400 lines of SML code!

Our implementation generates MLTree programs, which are defined in the module below.

```
                structure T : MLTREE
```

The relevant parts of this language are tree expressions, which have the type `T.rexp`, and statements, which have the type `T.stm`. Tree expressions perform operations over registers. The tree below, for example, perfoms a 64-bit addition of a temporary register and the value at the top of the stack.

```
            T.ADD(64,
                T.REG(64, tmpReg),
                T.LOAD(64, T.REG(64, rsp)))
```

Statements are effectful operations that include stores, moves, and control transfers. The statement below copies a 64-bit integer literal into a temporary register.

```
            T.MV(64, tmpReg, T.LI(64, 128))
```

Our interpreter consists of a single *architecture-independent* core generator and several *machine-specific* wrappers. Figure 9 contains the signature for our core generator. We have implemented this generator as an ML functor that takes some machine-specific parameters and exports a generator function. More specifically, our functor takes the lists of parameter registers, the possible register widths, the expression containing the stack-pointer register, the default bit width, and the caller-save registers.

Our core generator `genCoreInterp` takes a pointer to the C function, a pointer to the located argument structure, and the number of located arguments. The statements that the generator returns are exactly those statements in Figure 8.

Each machine-dependent wrapper has a different instantiation of the core generator functor, and has code for the C prologue and epilogue. We show these pieces of code for x86-64 in Figures 10 and 11 respectively. To generate the C prologue and epilogue, we place the architecture-independent code, `coreStms`, between the C prologue and epilogue. We also initialize the arguments register and the `%rax` register. As mentioned in Section 4, this register must contain an upper bound on the number of floating-point registers used in the call, so we initialize it to the number of possible floating-point parameter registers.

Once we have generated the statements for our interpreter, MLRisc takes over and produces an assembly file. From here, it is easy to generate a dynamically-linked library so that we can call the interpreter from ML.

```
structure GenCoreInterpX8664 =
    GenCoreInterpFn (
        val paramGprs = paramGprs
        val paramFprs = paramFprs
        val gprWidths = [8, 16, 32, 64]
        val fprWidths = [32, 64]
        val spReg = T.REG(64, rsp)
        val defaultWidth = 64
        val callerSaveRegs = callerSaveRegs
        val calleeSaveRegs = calleeSaveFRegs)
```

Figure 10. Instantiation of the core generator for x86-64.

```
fun genX8664Interp () = let
    val lab = Label.global "VarArgInterp"
    val argsReg = newReg()
    val coreStms =
            GenCoreInterp.genCoreInterp {
                cFun=param1,
                argsReg=argsReg,
                nArgs=param3
                }
    in
      (lab,
       List.concat [
         cPrologue
         [T.MV(64, argsReg, param2)],
         [T.MV(8, rax,
                T.LI (8, List.length paramFprs))],
         coreStms,
         cEpilogue
         ]
      ])
    end
```

Figure 11. The x86-64 interpreter-generator wrapper.

8. Related work

As discussed in the text, the typing scheme for variadic functions described in Section 3.3 was inspired by Danvy's implementation of unparsing functions in SML [Dan98]. The key difference between our work and his is that he was focused on providing a strongly-typed variadic signature to a specific class of SML functions, whereas we are providing a strongly-typed signature for variadic C functions.

Java is an example of a strongly-typed language that has builtin support for variadic functions. In Java, the arguments must all have the same type and the compiler automatically assembles them into a single array argument. In effect, Java uses the typing scheme described in Section 3.2.

Our implementation of variadic function calls is influenced by the work on state-machine-based descriptions of calling conventions [BD95, OLR06]. The difference between their work and ours is the difference between compilers and interpreters. They use a state machine to map a sequence of argument types to the calling-sequence machine code, whereas we map the argument types to a sequence of argument locations that are then interpreted to perform the call.

There are at least two C libraries for constructing function calls that have some similarities to our work. The GNU FFCall library provides a mechanism for constructing variadic function calls at runtime in C code.[10] It is designed to be used in embedded inter-

preters, but could possibly be used to generate calling conventions in a foreign-interface implementation. The main obstacle to such use is that the FFCall API is implemented using C-preprocessor macros. The implementation of variadic calls in FFCall consists of several thousand lines of hand-written C code that embodies the calling conventions of the specific architectures, whereas our generator is fewer than 400 lines of machine independent code.[11]

The libFFI library[12] is also designed to support calling C functions from interpreters, but, unlike FFCall, it does not directly target variadic functions.[13] To call a C function using libFFI, one first prepares the call by passing libFFI an array of type descriptors (the sequence of types in Figure 1), which yields a descriptor. The descriptor is then combined with an array of arguments to make calls to the function. This library is implemented using a combination of hand-written, machine and ABI-specific C and assembly code.

9. Conclusion

Existing foreign interface mechanisms for statically-typed languages do not support variadic functions. In this paper, we have described an extension to the NLFFI library to support variadic functions. We showed how to handle the static typing of variadic functions in the NLFFI framework. Next, we described an interpreter for performing variadic calls. We made two critical design decisions to reduce the implementation complexity. First, we reuse much of the compiler's calling-convention framework and, second, we generate our interpreter from a single source using the MLRisc framework.

We have kept our implementation down to a relatively small line count. Not counting the NLFFI component, we have about 600 lines of hand-written SML code, although we expect this number to increase slightly as we prepare a full release. This number contrasts with several thousand lines of handwritten C and assembly code to support these platforms in libFFI. The table below breaks down the different parts of our implementation by code size.

code base	lines of SML code
argument assembly (staged allocation)	< 400
located arguments and marshaling	< 200
interpreter generator	< 400

Note that the first code base, staged allocation, is part of a separate library that the compiler also uses to generate C calls. So, even though there are quite a few lines of code, we do not count them as part of our variadic-call implementation.

We have implemented the mechanism for variadic calls on the x86 and x86-64, although, since SML/NJ does not currently support the x86-64 ISA, we have only tested the interpreter on that platform. We plan to extend our implementation to other architectures, including the Sparc and PowerPC, in the near future.

Another benefit of our technique is *retargetability*. Because our interpreter is an assembly program with C calling conventions, we can port it to any compiler that supports C calls. We plan to port our implementation to the MLton compiler in the near future.

References

[ATe]   *The ATerm Programming Guide*. Available from `http://homepages.cwi.nl/~daybuild/daily-books/technology/aterm-guide/aterm-guide.html`.

---

[10] There is no published description of FFCall, but the manual pages are available at `http://www.haible.de/bruno/packages-ffcall.html`.

[11] Of course, we are relying on the MLRisc framework's knowledge of C calling conventions.

[12] The libFFI source and documentation is available from `http://sources.redhat.com/libffi/`.

[13] On many architectures it is possible to call variadic functions using libFFI to set up the call, but it depends on the ABI.

[BD95]   Bailey, M. W. and J. W. Davidson. A formal model of procedure calling conventions. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, 1995. pp. 298–310.

[Blu01]   Blume, M. No-longer-foreign: Teaching an ML compiler to speak C "natively". In N. Benton and A. Kennedy (eds.), *Proceedings of the First International Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, vol. 59 of *Electronic Notes in Theoretical Computer Science*, New York, NY, September 2001. Elsevier Science Publishers. Available from `http://www.elsevier.nl/locate/entcs/volume59.html`.

[CFH⁺03]   Chakravarty, M. M. T., S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. Peyton Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 Report. Available from `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2003.

[Dan98]   Danvy, O. Functional unparsing. *Journal of Functional Programming*, **8**(6), 1998, pp. 621–625.

[FLMP99]   Finne, S., D. Leijen, E. Meijer, and S. Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, September 1999, pp. 153–162.

[FPR00]   Fisher, K., R. Pucella, and J. Reppy. Data-level interoperability. *Technical report*, Bell Labs, Lucent Technologies, April 2000. Available from `http://moby.cs.uchicago.edu`.

[FPR01]   Fisher, K., R. Pucella, and J. Reppy. A framework for interoperability. In N. Benton and A. Kennedy (eds.), *Proceedings of the First International Workshop on Multi-Language Infrastructure and Interoperability (BABEL'01)*, vol. 59 of *Electronic Notes in Theoretical Computer Science*, New York, NY, September 2001. Elsevier Science Publishers. Available from `http://www.elsevier.nl/locate/entcs/volume59.html`.

[GGR94]   George, L., F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *Fifth International Conference on Compiler Construction*, April 1994, pp. 83–97.

[OLR06]   Olinsky, R., C. Lindig, and N. Ramsey. Staged allocation: a compositional technique for specifying and implementing procedure calling conventions. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 2006. ACM, pp. 409–421.