Calling Variadic Functions from a Strongly Typed Language

Matthias Blume  Toyota Technological Institute at Chicago
Mike Rainey  University of Chicago
John Reppy  University of Chicago
Variadic functions in C

```c
int printf (const char *, ...);
```
Variadic functions in C

\[
\text{int } \text{printf} \ (\text{const char } *, \ldots); \]

\[
\text{printf} \ (\text{“%d"}, 10); \]
Variadic functions in C

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

printf ("%d", 10);

printf ("%g: %d(%f)\n", 10.0, 3, 0.25);
```
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions

• High-level interface via “Danvy-style” typing
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions

• High-level interface via “Danvy-style” typing
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions

• High-level interface via “Danvy-style” typing
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions

• High-level interface via “Danvy-style” typing
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions

• High-level interface via “Danvy-style” typing
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions

• High-level interface via “Danvy-style” typing
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions

• High-level interface via “Danvy-style” typing
Outline

• Why are we doing this?

• How does calling variadic functions work in C?

• Why doesn’t the same approach work in ML?

• Located Arguments via Staged Allocation: Our solution to the (low-level part of the) problem

• Conclusions
Why?
Why?

- Being able to call `printf`?
  - no
Why?

• Being able to call `printf`?
  • no

• Utility:
  • Some APIs rely heavily on variadic functions
Why?

- Being able to call `printf`?
  - no

- Utility:
  - Some APIs rely heavily on variadic functions

- Completeness:
  - NLFFI models the entire C type system - but (until now) with the single exception of variadic functions
Calling a fixed-arity C function

Prototype: \( \texttt{int f (int, double, float, char, void *)} \);

Call: \( j = f(\, i, x, w, c, p) \);
Calling a fixed-arity C function

Prototype: `int f (int, double, float, char, void *)`;

Call: `j = f (i, x, w, c, p);`
Calling a fixed-arity C function

Prototype: `int f (int, double, float, char, void *);`

Call: `j = f (i, x, w, c, p);`
Calling a fixed-arity C function

Prototype: `int f (int, double, float, char, void *);`

Call: `j = f (i, x, w, c, p);`
Calling a fixed-arity C function

Prototype: \( \text{int } f (\text{int}, \text{double}, \text{float}, \text{char}, \text{void } *) \);
Calling a fixed-arity C function

Prototype: `int f (int, double, float, char, void *);`

Call: `j = f (i, x, w, c, p);`
Calling a fixed-arity C function

Call: \( j = f (i, x, w, c, p); \)

Prototype: \( \text{int } f (\text{int, double, float, char, void *}); \)
Calling a fixed-arity C function

Call:

\[ j = f (i, x, w, c, p); \]

Prototype:

\[ \text{int } f (\text{int, double, float, char, void *}); \]
Calling a fixed-arity C function

Call:

\[ j = f (i, x, w, c, p); \]

Prototype: \texttt{int f (int, double, float, char, void *)};
Calling a variadic function \textbf{from} C

Call: \[ j = f (i, x, w, c, p); \]
Calling a variadic function from C

Types: \texttt{int i; double x; float w; char c; void \*p;}
Call: \hspace{1cm} j = f(i, x, w, c, p);
Calling a variadic function from C

Types: int i; double x; float w; char c; void *p;
Call: \( j = f(i, x, w, c, p); \)

**implied**

Prototype: int f (int, double, double, int, void *);
Calling a variadic function from C

Types: int i; double x; float w; char c; void *p;
Call: j = f (i, x, w, c, p);

implied
Prototype: int f (int, double, double, int, void *);
Calling a variadic function from C

Types: int i; double x; float w; char c; void *p;
Call: j = f (i, x, w, c, p);

implied
Prototype: int f (int, double, double, int, void *);
Calling from ML: What is the problem?

- The pieces we need:
  - Calling conventions
  - The sequence of types
  - The argument values
Calling from ML: What is the problem?

- The pieces we need:
  - Calling conventions
  - The sequence of types
  - The argument values
Calling from ML: What is the problem?

- The pieces we need:
  - Calling conventions
  - The sequence of types
  - The argument values
Calling a variadic function from ML

\texttt{val} \ j = f(i, x, w, c, p)
Calling a variadic function from ML

Val \( j = f \ (i, \ x, \ w, \ c, \ p) \)

**implied (inferred?)**

Prototype: \(? \ f \ (\ ?, \ ?, \ ?, \ ?, \ ?, \ ?);\)
Calling a variadic function from ML

```
val j = f i, x, w, c, p
```

**implied (inferred?)**

**Prototype:**

```
int f (int, double, double, int, ptr);
```
Calling a variadic function from ML

\texttt{val \ j = f (i, x, w, c, p)}

\textit{implied} (inferred?)

Prototype: \texttt{int f (int, double, double, int, ptr);}
Calling a variadic function from ML

\texttt{val j = f (i, x, w, c, p)}

\textit{implied\ (inferred?)}

Prototype: \texttt{int f (double, double, double, int, ptr)};
Calling a variadic function from ML

val \( j = f \ (i, x, w, c, p) \)

**implied** (inferred?)

**Prototype:** \( \text{int } f \ (\text{double, double, double, double, int, ptr}); \)
Calling a variadic function from ML

\[
\text{val } j = f (i, x, w, c, p)
\]

*implied* (inferred?)

**Prototype:** \texttt{int } f (\texttt{double, double, double, double, int, ptr});

\[\begin{align*}
\text{r0 (sp)} & \\
\text{r1 (ra)} & \\
\text{r2} & \\
\text{r3} & \\
\end{align*}\]
Calling a variadic function from ML

val \( \text{j} = f \ (i, \ x, \ w, \ c, \ p) \)

*implied* (inferred?)

Prototype: \( \text{int } f \ (\text{double, double, double, int, ptr}); \)
Calling a variadic function from ML

```
val j = f (i, x, w, c, p)
```

*implied* (inferred?)

Prototype: `int f (double, double, double, int, ptr);`

---

M. Blume, M. Rainey, J. Reppy

*Calling Variadic Functions from a Strongly-Typed Language*

ML'08, Victoria BC, Sep 21, 2008
Calling a variadic function from ML

```ml
val j = f (i, x, w, c, p)
```

*implied* (inferred?)

Prototype: `int f (double, double, double, int, ptr);`

---

M. Blume, M. Rainey, J. Reppy

*Calling Variadic Functions from a Strongly-Typed Language*

ML'08, Victoria BC, Sep 21, 2008
Calling a variadic function from ML

\[
\text{val } j = f(i, x, w, c, p)
\]

implied (inferred?)
Prototype: int f (double, double, double, int, ptr);
Calling a variadic function from a polymorphic ML function

\[
\text{fun } g \ i = f(i, x, w, c, p) + 1
\]

inferred

Prototype: \( \text{int } f(\alpha, \text{double, double, int, ptr}); \)
If we want to use static ML type information...
If we want to use static ML type information...

- Two routes:
If we want to use static ML type information...

• Two routes:

1. Monomorphize

• ... but that requires whole-program analysis (e.g., as in MLton)
If we want to use static ML type information...

**Two routes:**

1. Monomorphize
   - ... but that requires whole-program analysis (e.g., as in MLton)

2. Use intensional type information
   - ... complicated
   - ... not static, i.e., is a runtime technique
So ultimately...
So ultimately...

```ocaml
datatype arg = INT of xint
           | CHAR of xchar
           | FLOAT of xfloat
           | DOUBLE of xdouble
           | PTR of xaddr

val dispatch : xaddr * arg list -> unit
```
So ultimately...

```ml
datatype arg = INT of xint 
            | CHAR of xchar 
            | FLOAT of xfloat 
            | DOUBLE of xdouble 
            | PTR of xaddr 

val dispatch : xaddr * arg list -> unit 
```

```
dispatch (f, [INT i, DOUBLE x, FLOAT w, CHAR c, PTR p])
```
So ultimately...
So ultimately...

- ... we don’t make use of the ML compiler’s type information
So ultimately...

- ... we don’t make use of the ML compiler’s type information
  
  ‣ Solution is not ML-specific.
So ultimately...

- ... we don’t make use of the ML compiler’s type information
  - Solution is not ML-specific.
  - It can be adapted even to dynamically typed settings.
So ultimately...

- ... we don’t make use of the ML compiler’s type information

  - Solution is not ML-specific.

  - It can be adapted even to dynamically typed settings.

  - Use “universal” type of C values in statically typed setting.
Calling a variadic function

\[ \text{dispatch} \ (f, \ [\text{INT}\ i, \ \text{DOUBLE}\ x, \ \text{FLOAT}\ w, \ \text{CHAR}\ c, \ \text{PTR}\ p]) \]
Calling a variadic function

```
proto f (int, double, double, int, void *);
```

```
dispatch (f, [int i, double x, float w, char c, ptr p])
```

implied

**Prototype:** `int f (int, double, double, int, void *)`;

```
  r0 (sp)  f0
  r1 (ra)  f1
  r2
  r3

  f2
  f3
```
Calling a variadic function

\[ \text{dispatch}(f, \{\text{INT } i, \text{ DOUBLE } x, \text{ FLOAT } w, \text{ CHAR } c, \text{ PTR } p\}) \]

*implied*

*Prototype:* \( \text{int } f \text{ (int, double, double, int, void *)}; \)
Calling a variadic function

\[
\text{dispatch} \ (f, [\text{INT } i, \text{ DOUBLE } x, \text{ FLOAT } w, \text{ CHAR } c, \text{ PTR } p])
\]

**Prototype:** `int f (int, double, double, int, void *)`
Calling Variadic Functions from a Strongly-Typed Language

M. Blume, M. Rainey, J. Reppy

RUNTIME SYSTEM

FI LIBRARY

User program

f arg1 ... argn

C LIBRARY

Located-argument generator

Located-argument interpreter

SML

C/asm
“Staged Allocation”  
*(Olinsky, Lindig, Ramsey; POPL’06)*

Reuses existing specs;  
< 600 lines of (new) ML code
“Staged Allocation”  
(Olinsky, Lindig, Ramsey; POPL’06)

Reuses existing specs;  
< 600 lines of (new) ML code

Generated for different architectures  
from a single MLRISC template

< 400 lines of ML code;  
result is not implementation-  
or language-specific
Located arguments

\[
\text{dispatch} \ (f, \ [\text{INT} \ i, \ \text{DOUBLE} \ x, \ \text{FLOAT} \ w, \ \text{CHAR} \ c, \ \text{PTR} \ p])
\]
Staged Allocation
Staged Allocation

\[
\text{arg} \quad \text{constructor (e.g., } \text{DOUBLE}, \ldots) \quad \text{value}
\]
Staged Allocation

```
arg constructor (e.g., DOUBLE, ...)  value
```

```
allocation request (e.g., (64, FPR, 8), ...)
```
### Staged Allocation

arg constructor (e.g., `DOUBLE`, ...)  
value  

allocation request (e.g., `(64, FPR, 8)`, ...)  

machine location (e.g., `REG "f4"`, ...)
Staged Allocation

\[ \text{arg} \quad \text{constructor} \quad (\text{e.g., DOUBLE, ...}) \quad \text{value} \]

\[ \text{allocation request} \quad (\text{e.g., (64, FPR, 8), ...}) \]

\[ \text{allocation state} \quad \text{cc} \]

\[ \text{machine location} \quad (\text{e.g., REG "f4", ...}) \]
Staged Allocation

\[ \text{arg} \quad \text{constructor (e.g., \texttt{DOUBLE}, \ldots)} \quad \text{value} \]
\[ \downarrow \]
\[ \text{allocation request (e.g., (64, \texttt{FPR}, 8), \ldots)} \]
\[ \downarrow \]
\[ \text{allocation state} \quad \xrightarrow{\text{cc}} \quad \text{allocation state’} \]
\[ \downarrow \]
\[ \text{machine location (e.g., \texttt{REG “f4”}, \ldots)} \]
Staged Allocation

\[ \text{arg} \quad \text{constructor (e.g., DOUBLE, ...)} \]

\[ \text{allocation request (e.g., (64, FPR, 8), ...)} \]

\[ \text{allocation state} \rightarrow \text{cc} \rightarrow \text{allocation state'} \]

\[ \text{machine location (e.g., REG "f4", ...)} \]

\[ \text{value} \]
Staged Allocation

arg constructor (e.g., \texttt{DOUBLE}, ...)

allocation request (e.g., \texttt{(64, FPR, 8)}, ...)

machine location (e.g., \texttt{REG "f4"}, ...)

"located" argument

allocation state

\texttt{cc}

allocation state'

value

value
Located arguments

\[
\text{dispatch} \ (f, \ [\text{INT} \ i, \ \text{DOUBLE} \ x, \ \text{FLOAT} \ w, \ \text{CHAR} \ c, \ \text{PTR} \ p])
\]
Located arguments

\[
\text{dispatch} \ (f, \ [\text{\textbf{INT}} \ i, \ \text{\textbf{DOUBLE}} \ x, \ \text{\textbf{FLOAT}} \ w, \ \text{\textbf{CHAR}} \ c, \ \text{\textbf{PTR}} \ p])
\]

<table>
<thead>
<tr>
<th>“r2”</th>
<th>“f01”</th>
<th>“f23”</th>
<th>“r3”</th>
<th>“(sp+24)”</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{r0 (sp)} & \quad \text{r1 (ra)} \\
\text{r2} & \quad \text{r3}
\end{align*}
\]

\[
\begin{align*}
\text{f01} & \quad \text{f0} \\
\text{f1} & \quad \text{f2} \\
\text{f23} & \quad \text{f3}
\end{align*}
\]
Located arguments

dispatch (f, [\textsc{int} \ i, \ \textsc{double} \ x, \ \textsc{float} \ w, \ \textsc{char} \ c, \ \textsc{ptr} \ p])
Located arguments

\[
\text{dispatch} \ (f, \ [\text{INT} \ i, \ \text{DOUBLE} \ x, \ \text{FLOAT} \ w, \ \text{CHAR} \ c, \ \text{PTR} \ p])
\]

<table>
<thead>
<tr>
<th>&quot;r2&quot;</th>
<th>&quot;f01&quot;</th>
<th>&quot;f23&quot;</th>
<th>&quot;r3&quot;</th>
<th>&quot;(sp+24)&quot;</th>
</tr>
</thead>
</table>

\[\begin{array}{|c|c|c|c|c|}
\hline
r0 \ (sp) & r1 \ (ra) & f0 \ f1 & f2 \ f3 & \text{(sp+24)} \\
\hline
\end{array}\]
Conclusions
Conclusions

- Difficult to utilize static type information for generating the calling sequence for variadic FFI calls.

- Use runtime techniques instead.
Conclusions

- Difficult to utilize static type information for generating the calling sequence for variadic FFI calls.
  - Use runtime techniques instead.

- Separate the generation of located arguments from their actual placement into machine registers and stack locations.
  - First task can be done in the high-level language;
  - second task must be done at the assembly level
Conclusions

- Difficult to utilize static type information for generating the calling sequence for variadic FFI calls.
  - Use runtime techniques instead.
- Separate the generation of located arguments from their actual placement into machine registers and stack locations.
  - First task can be done in the high-level language;
  - second task must be done at the assembly level
- Reuse existing technology (Staged Allocation, MLRISC).
  - Modular implementation.
- Overall implementation effort is very manageable.
Thank you!