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

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

Variadic functions in C

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

printf (“%d”, 10);
```

Variadic functions in 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

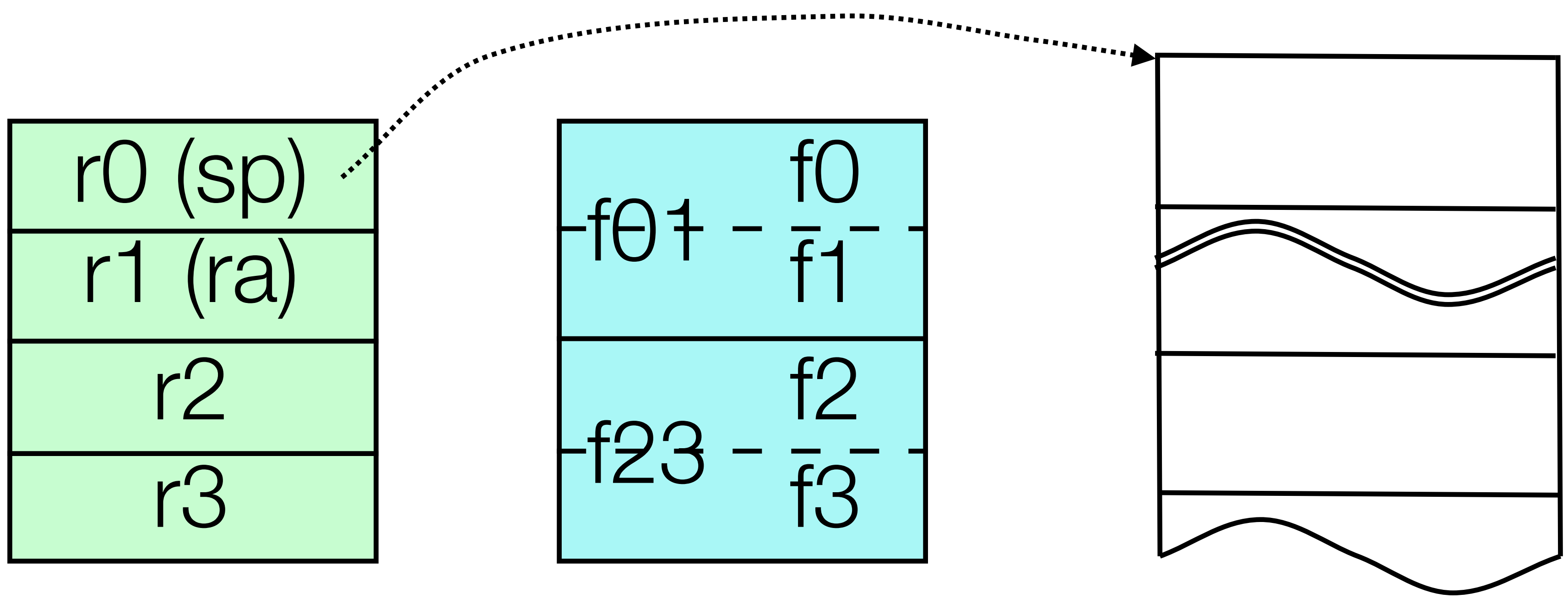
Why?

- 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

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

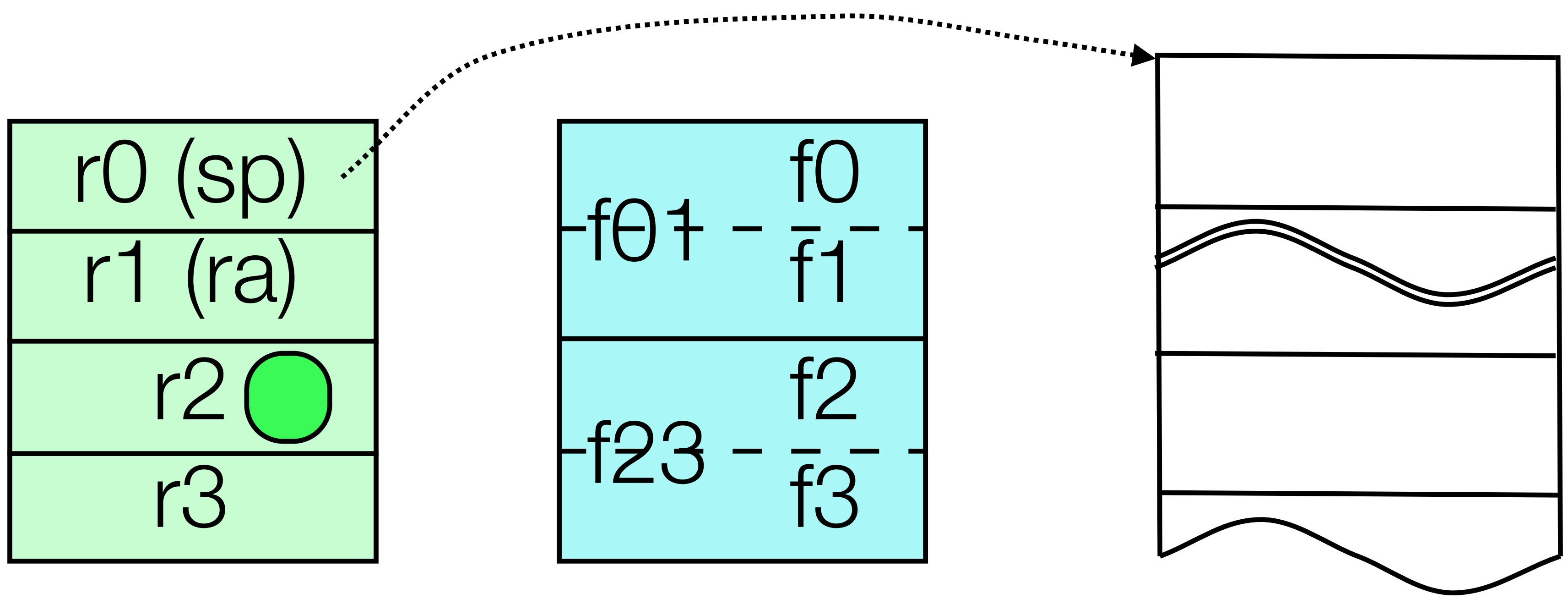
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

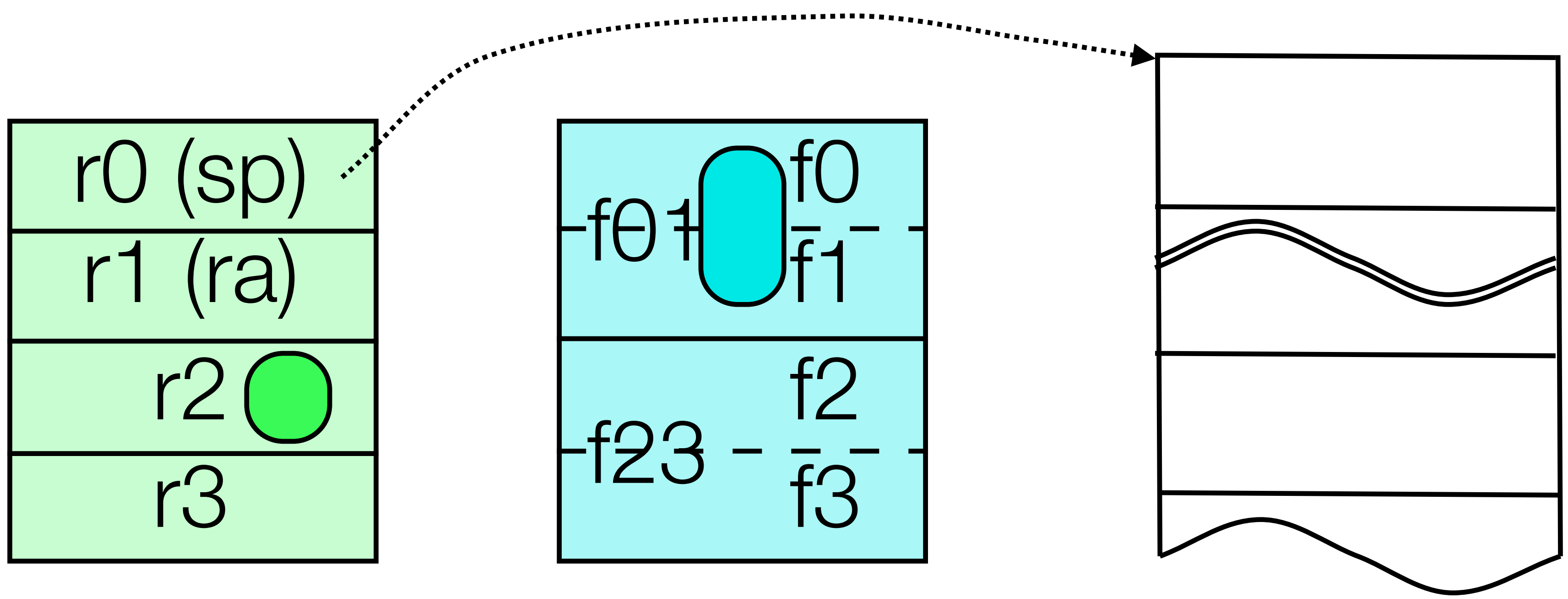
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

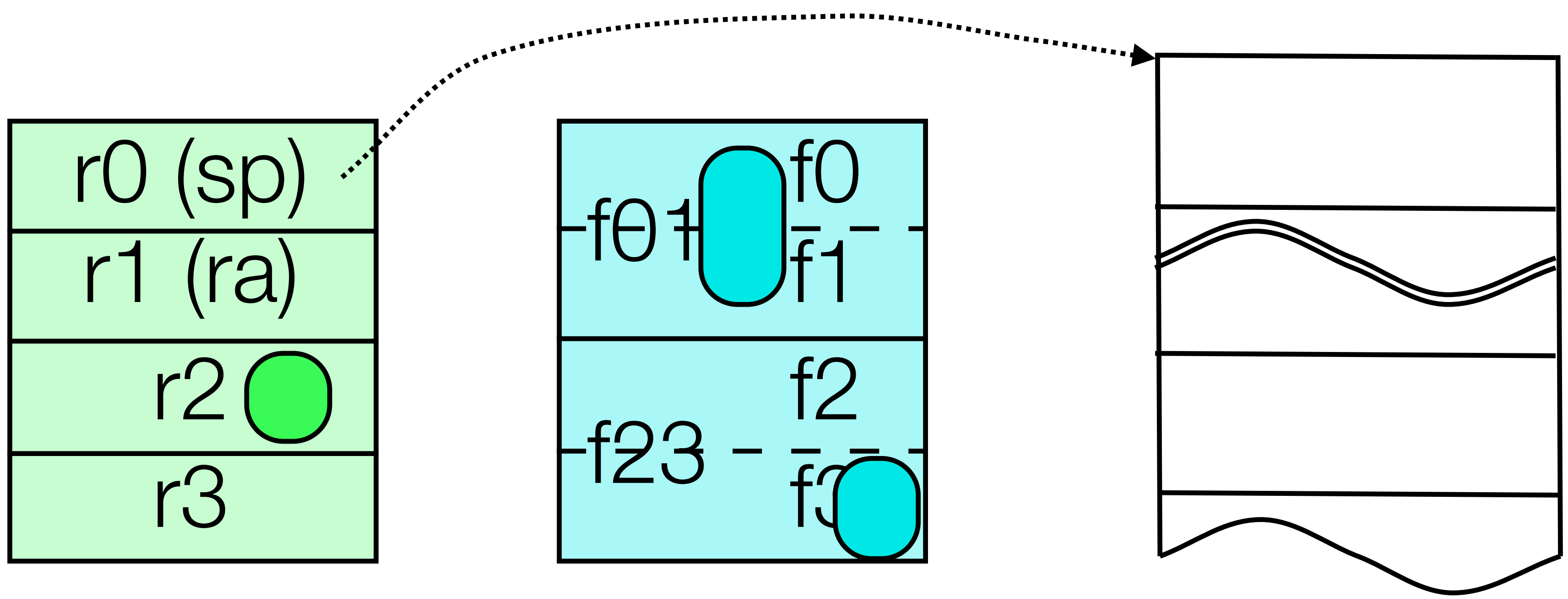
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

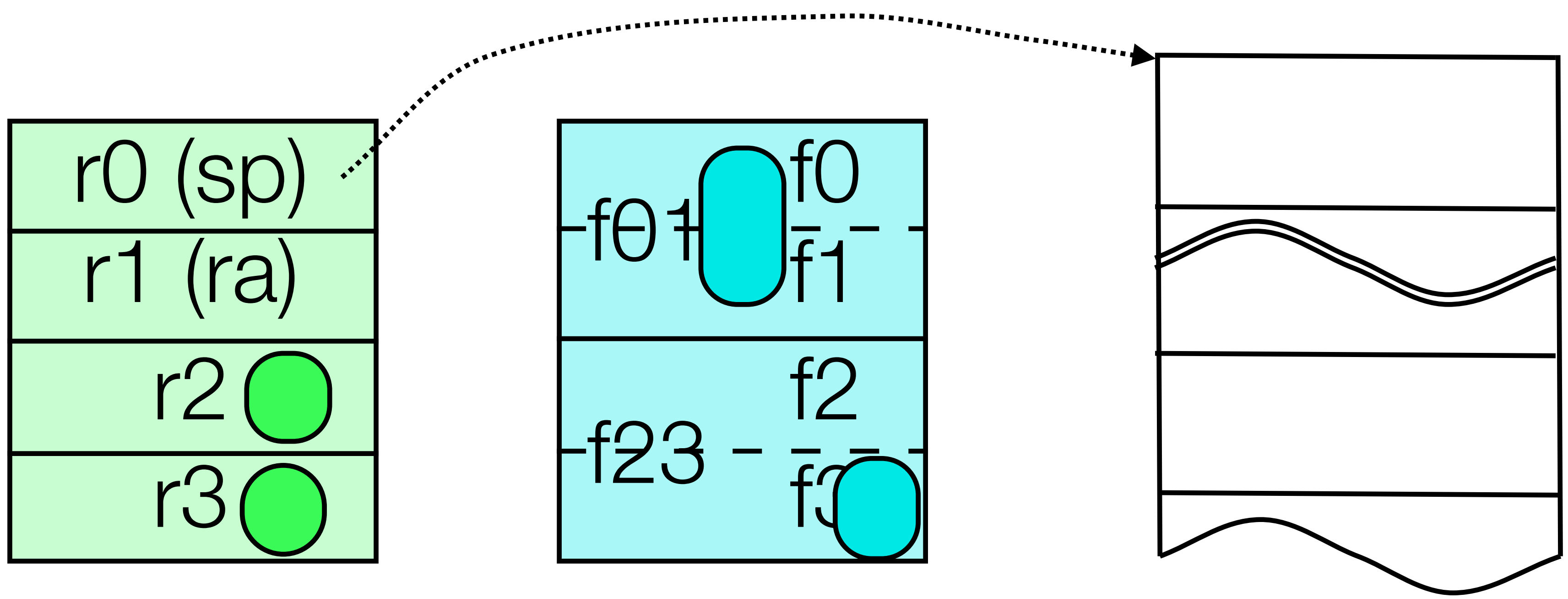
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

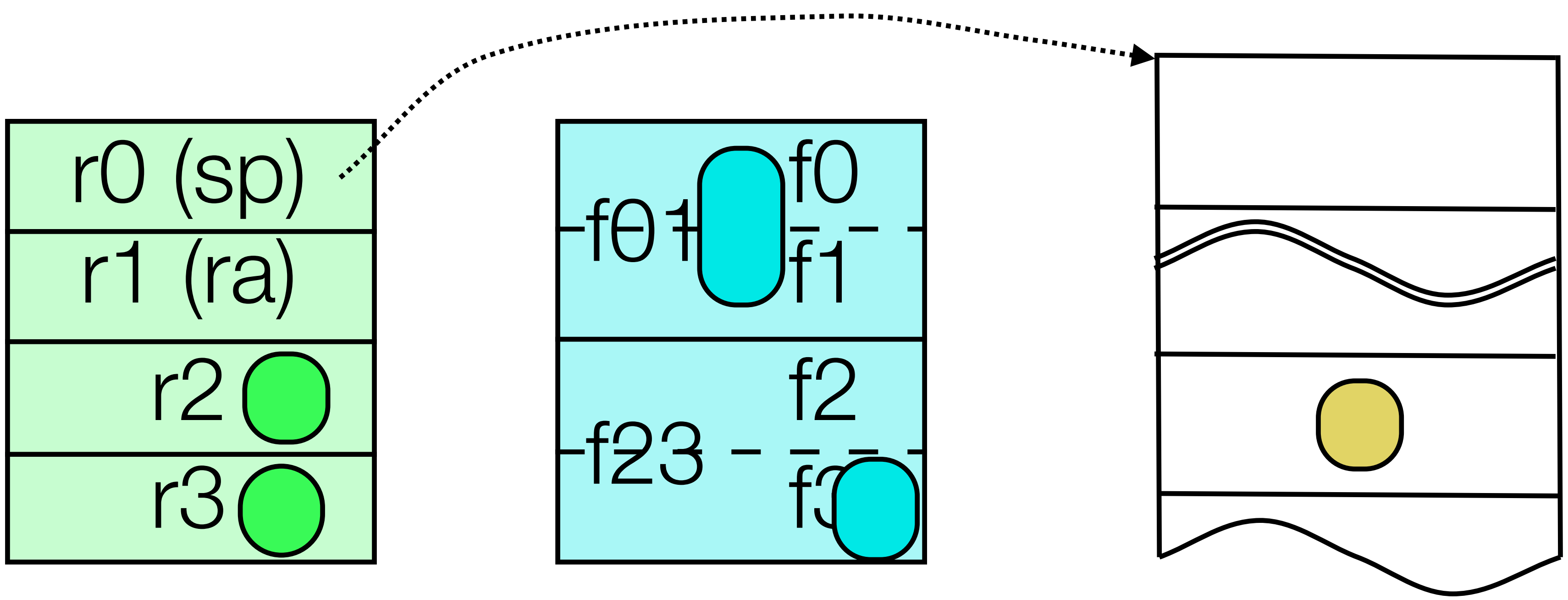
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

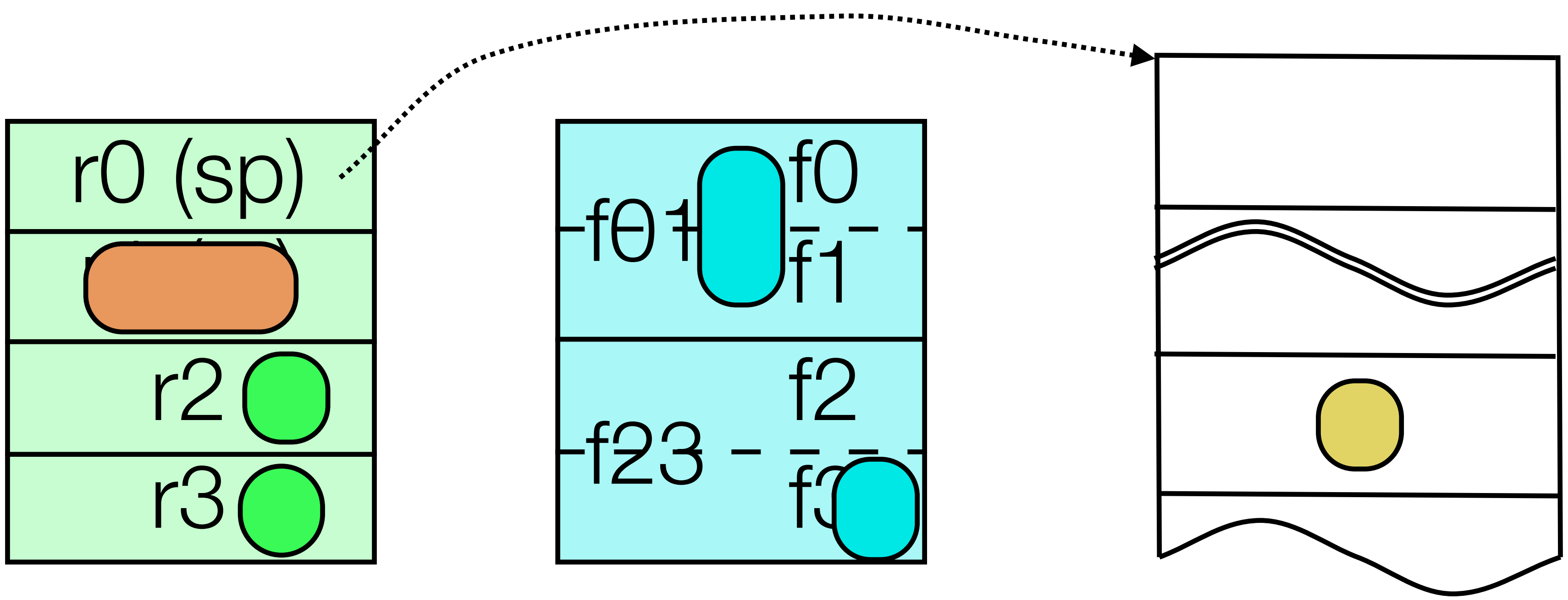
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

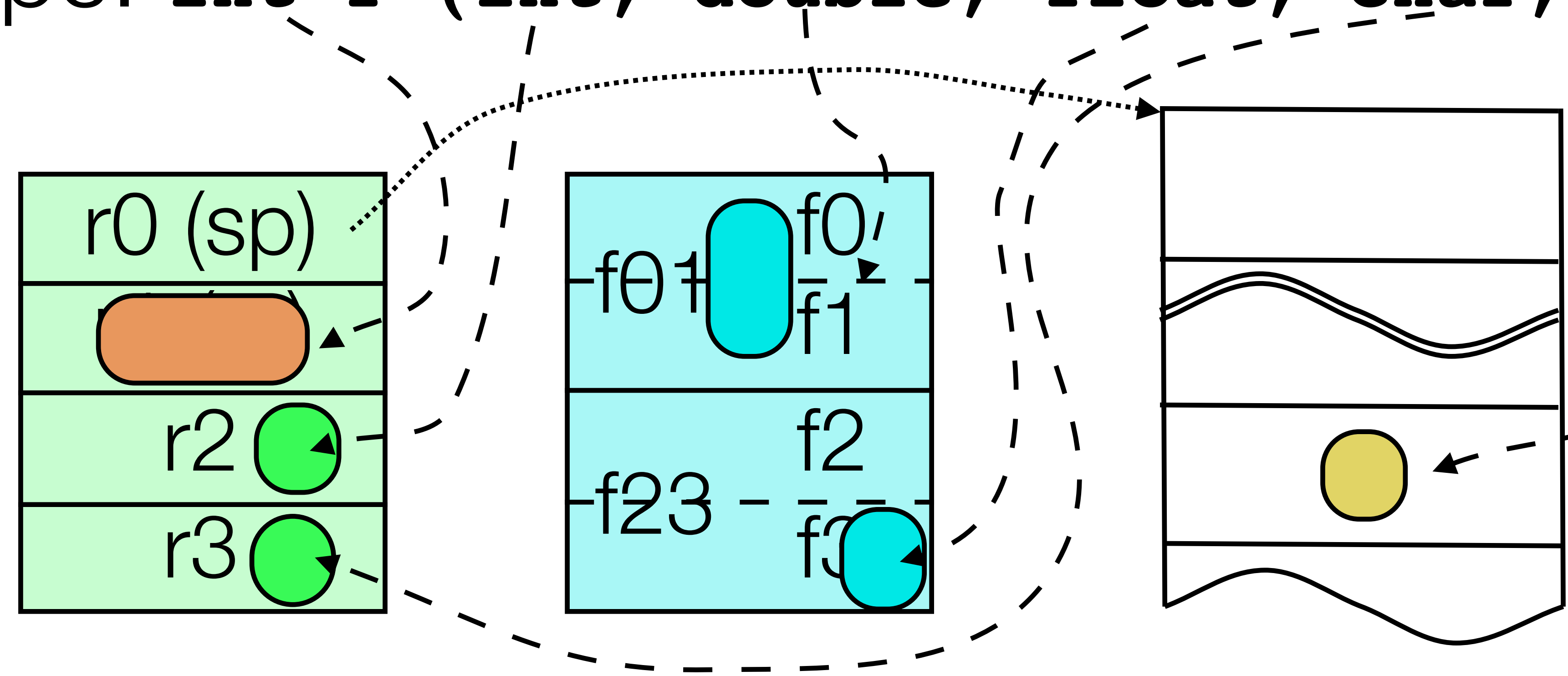
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

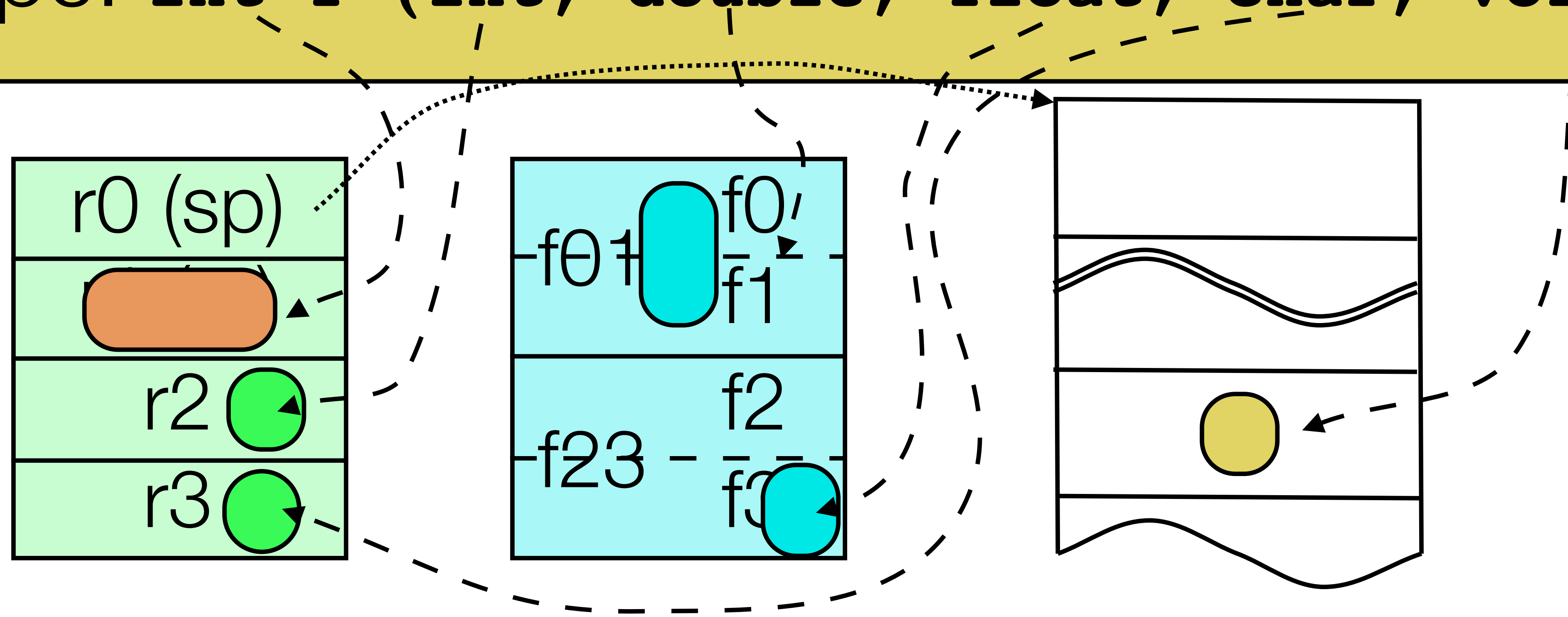
Prototype: `int f (int, double, float, char, void *);`



Calling a fixed-arity C function

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

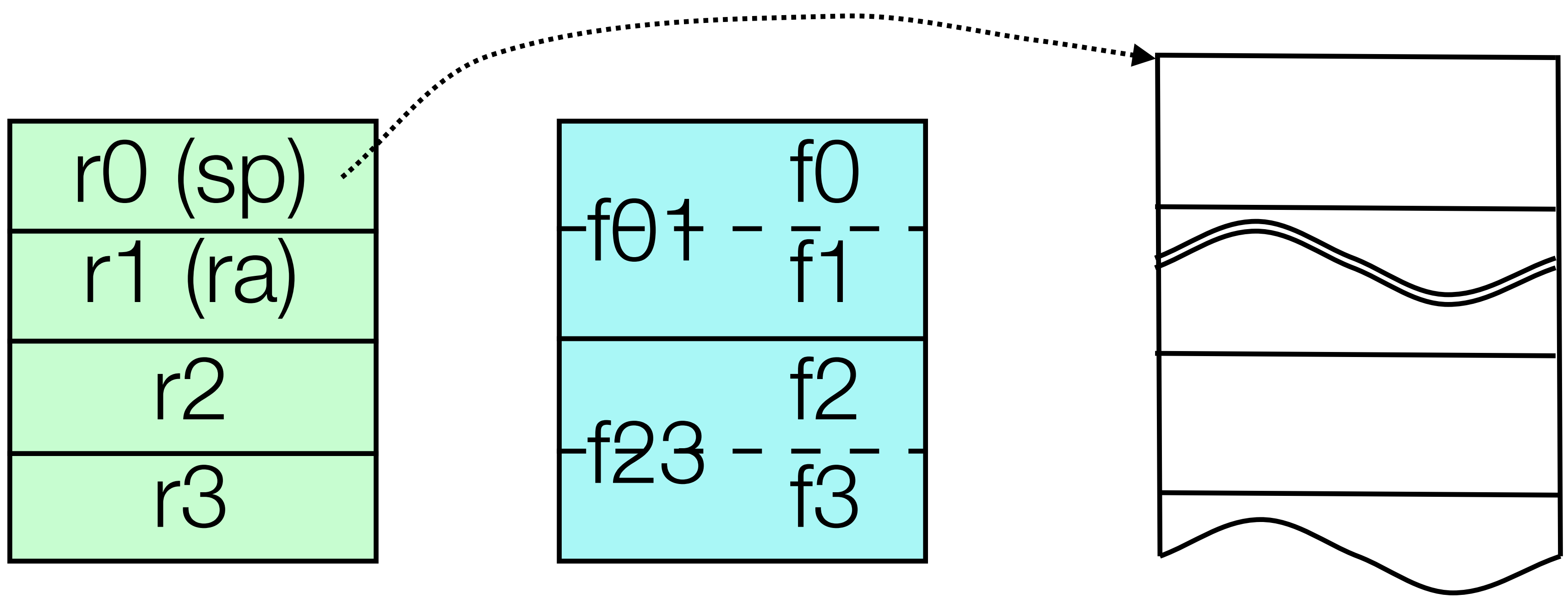
Prototype: `int f (int, double, float, char, void *);`



Calling a variadic function *from* C

Call:

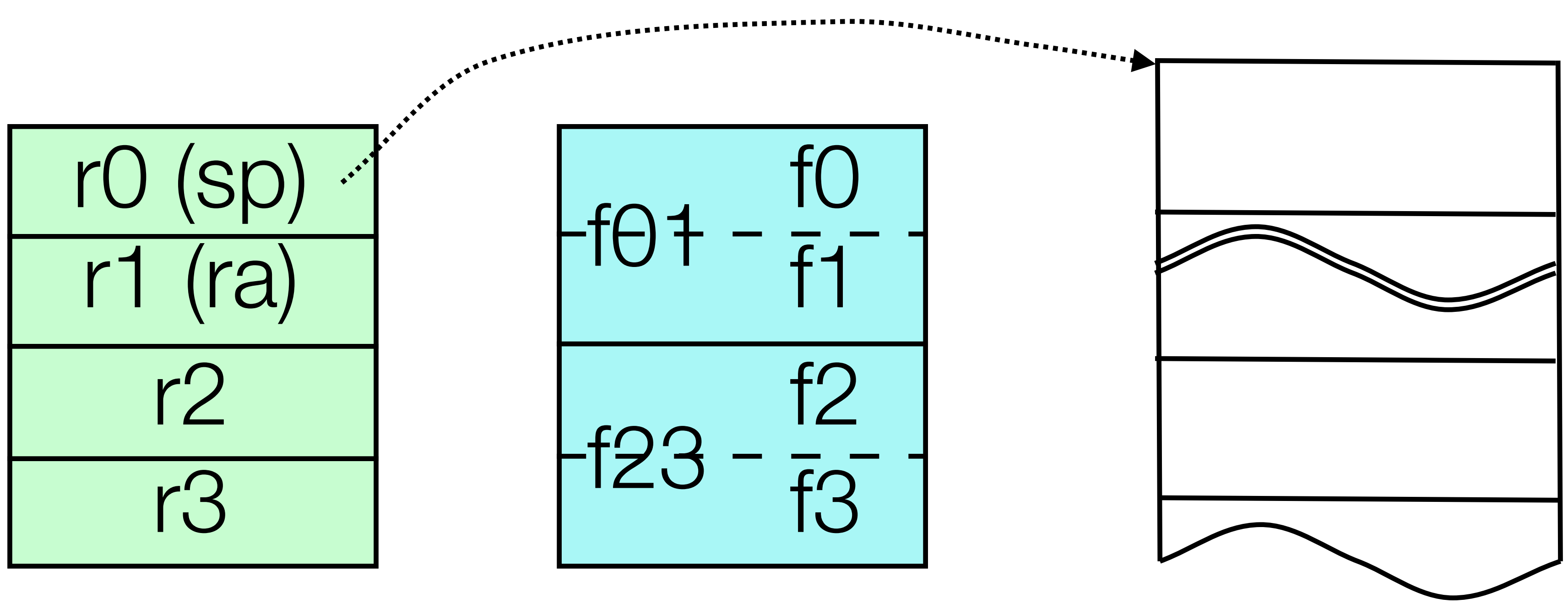
```
j = f (i, x, w, c, p);
```



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);`**



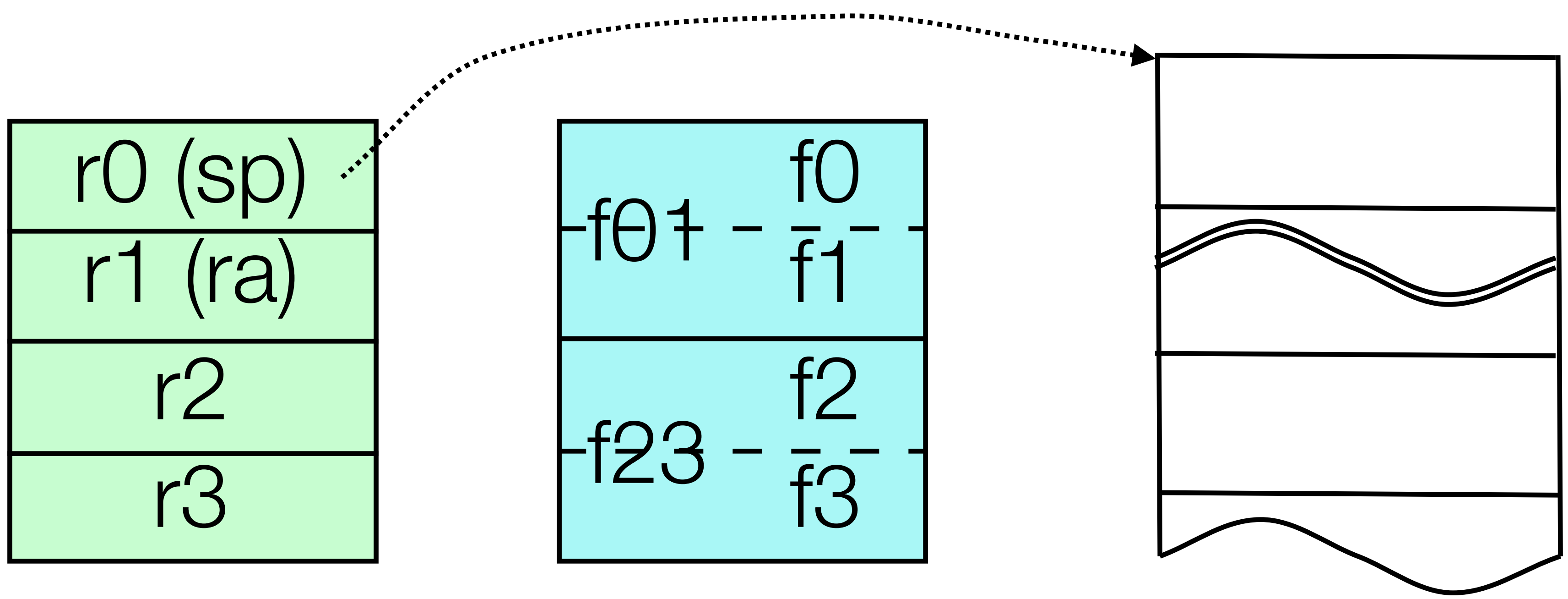
Calling a variadic function *from* C

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 *);`



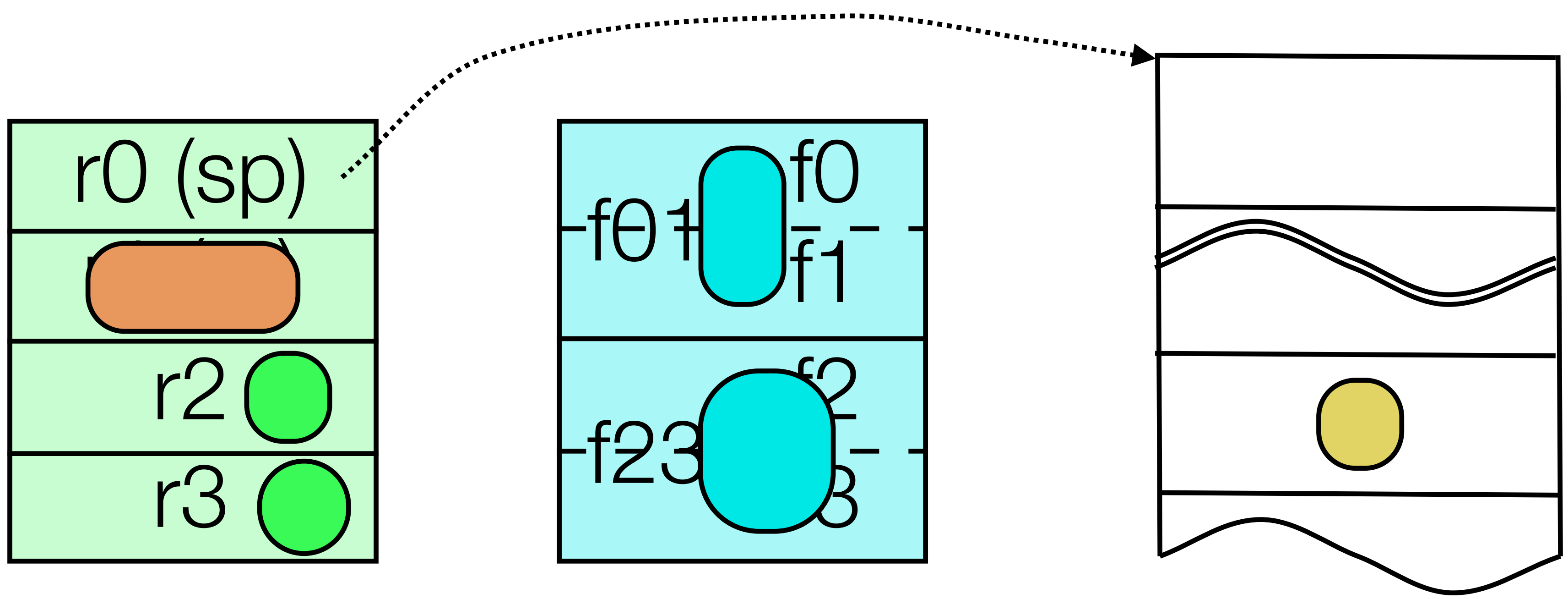
Calling a variadic function *from* C

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 *);`

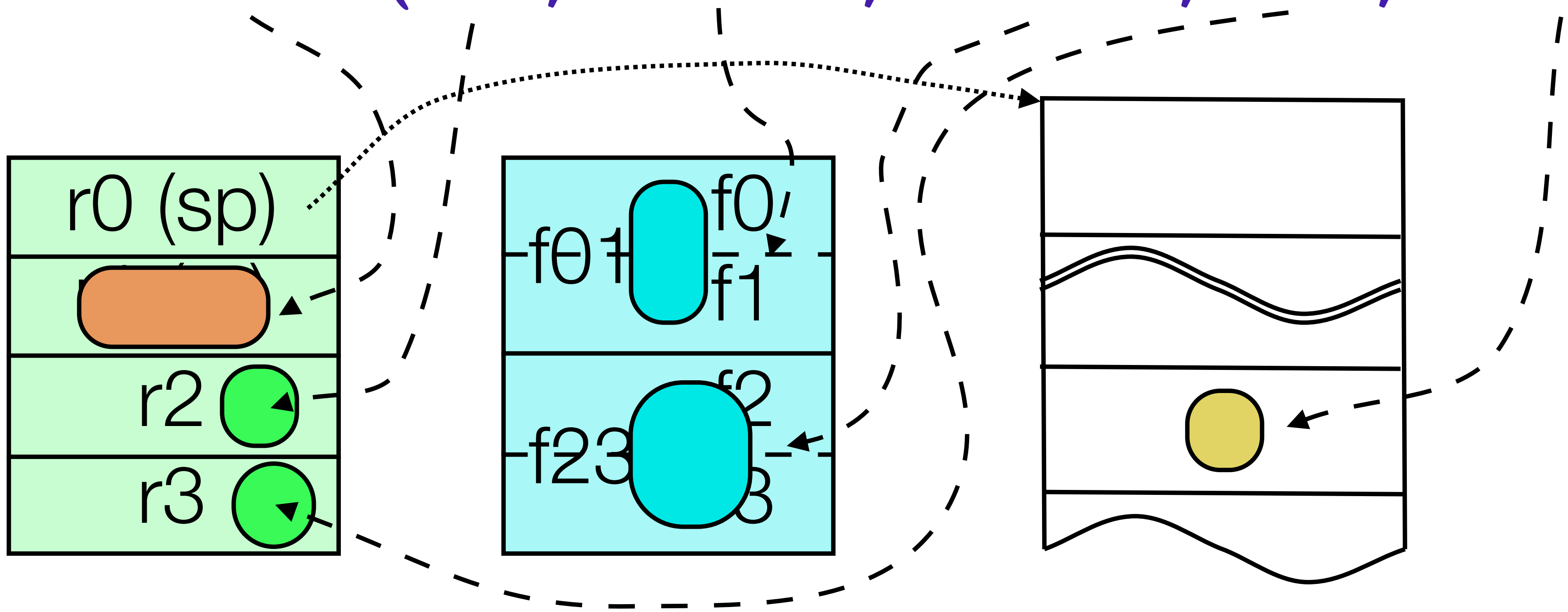


Calling a variadic function *from* C

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?

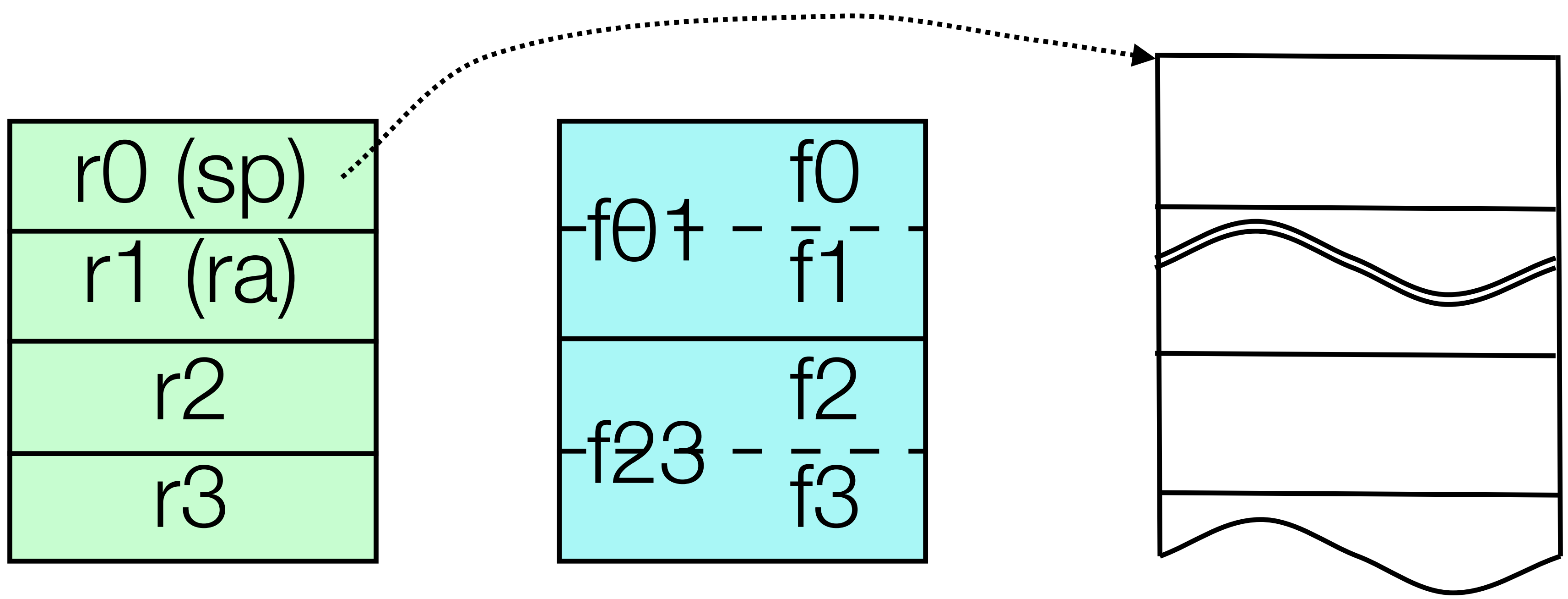
- The pieces we need:
  - Calling conventions
  - The sequence of types
  - The argument values

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

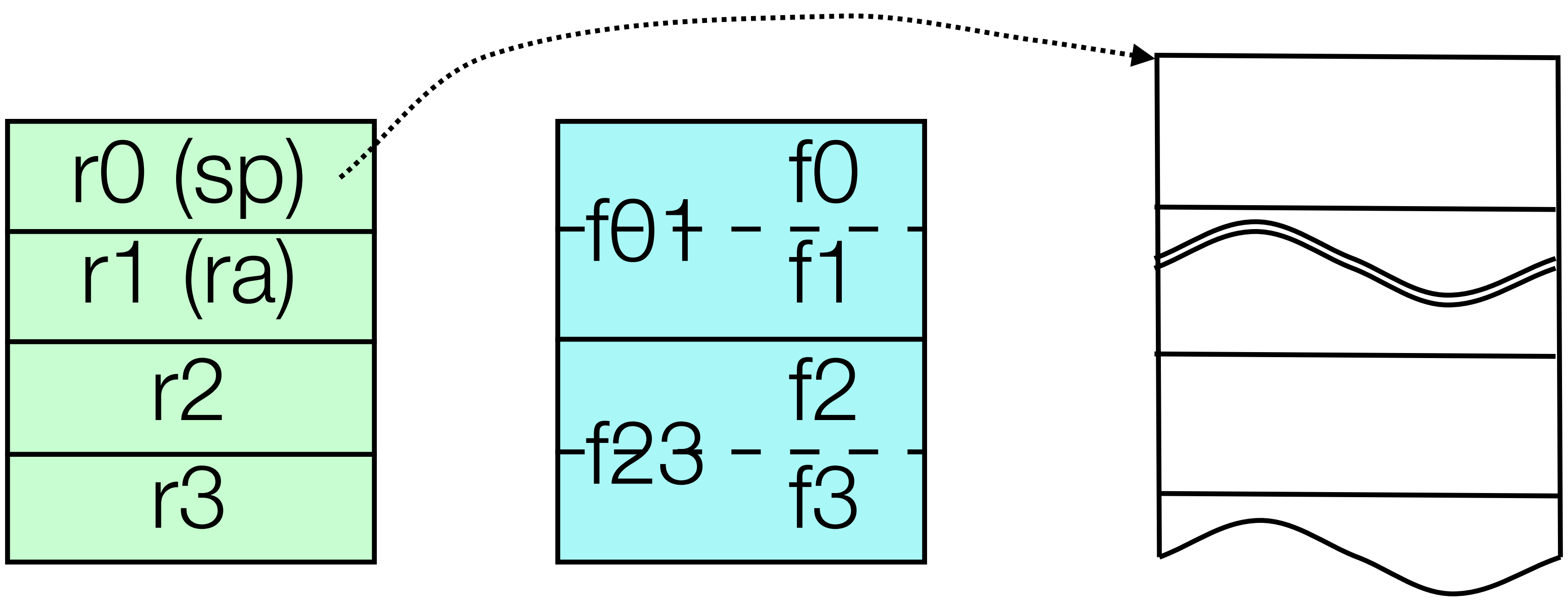
```
val j = f (i, x, w, c, p)
```



Calling a variadic function from ML

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

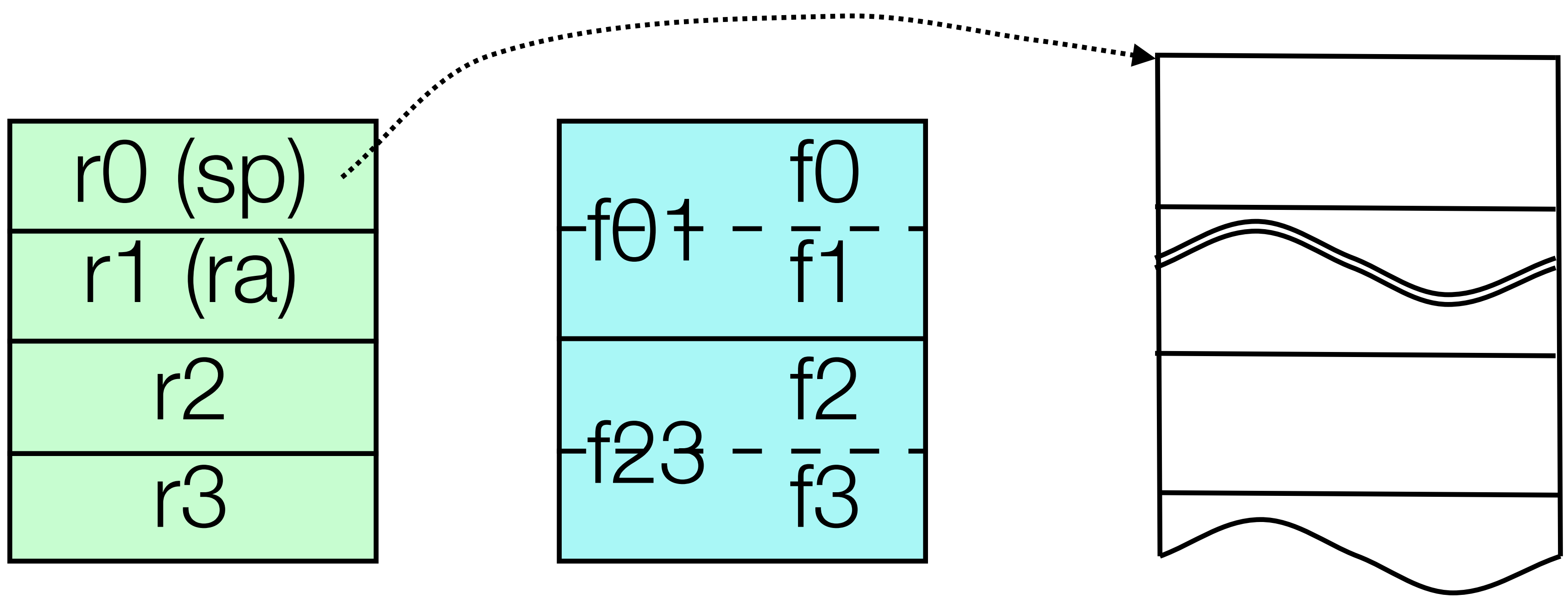
implied (inferred?) Prototype: `? f (?, ?, ?, ?, ?);`



Calling a variadic function from ML

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

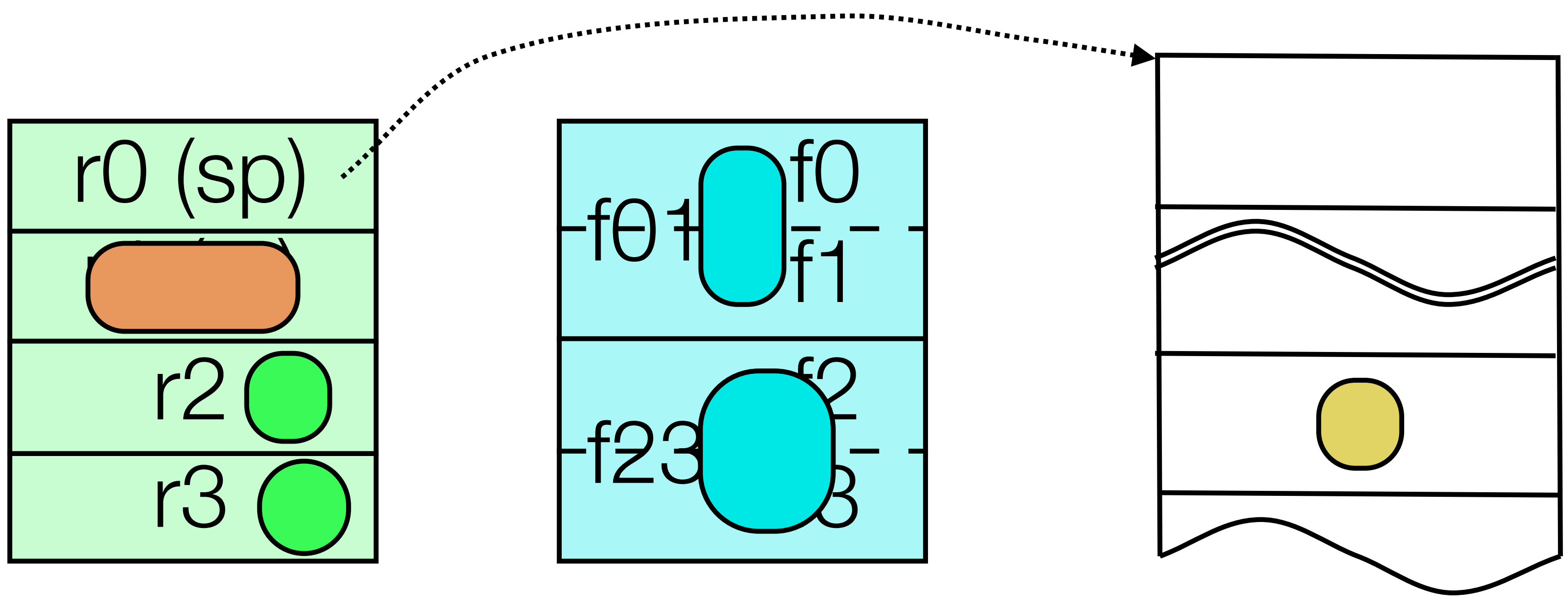
implied (inferred?)
Prototype: **`int f (int, double, double, int, ptr);`**



Calling a variadic function from ML

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

implied (inferred?)
Prototype: **`int f (int, double, double, int, ptr);`**

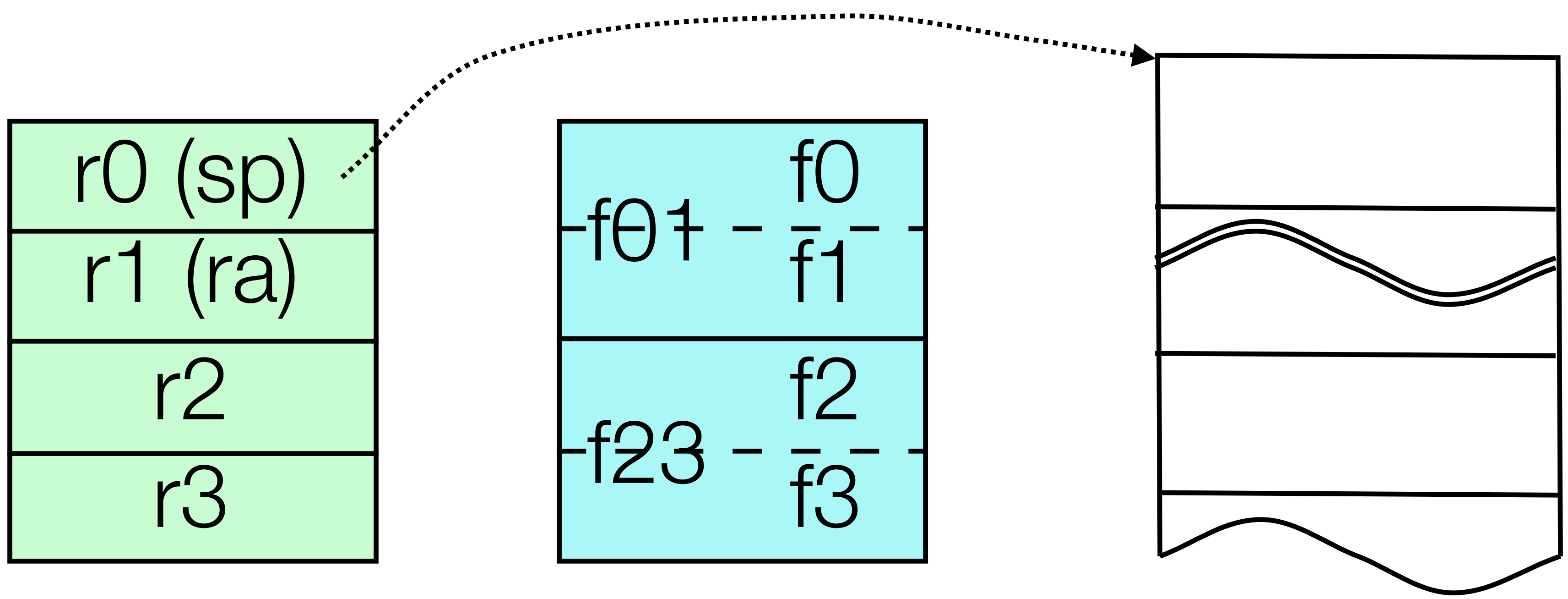


Calling a variadic function from ML

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

implied (inferred?)

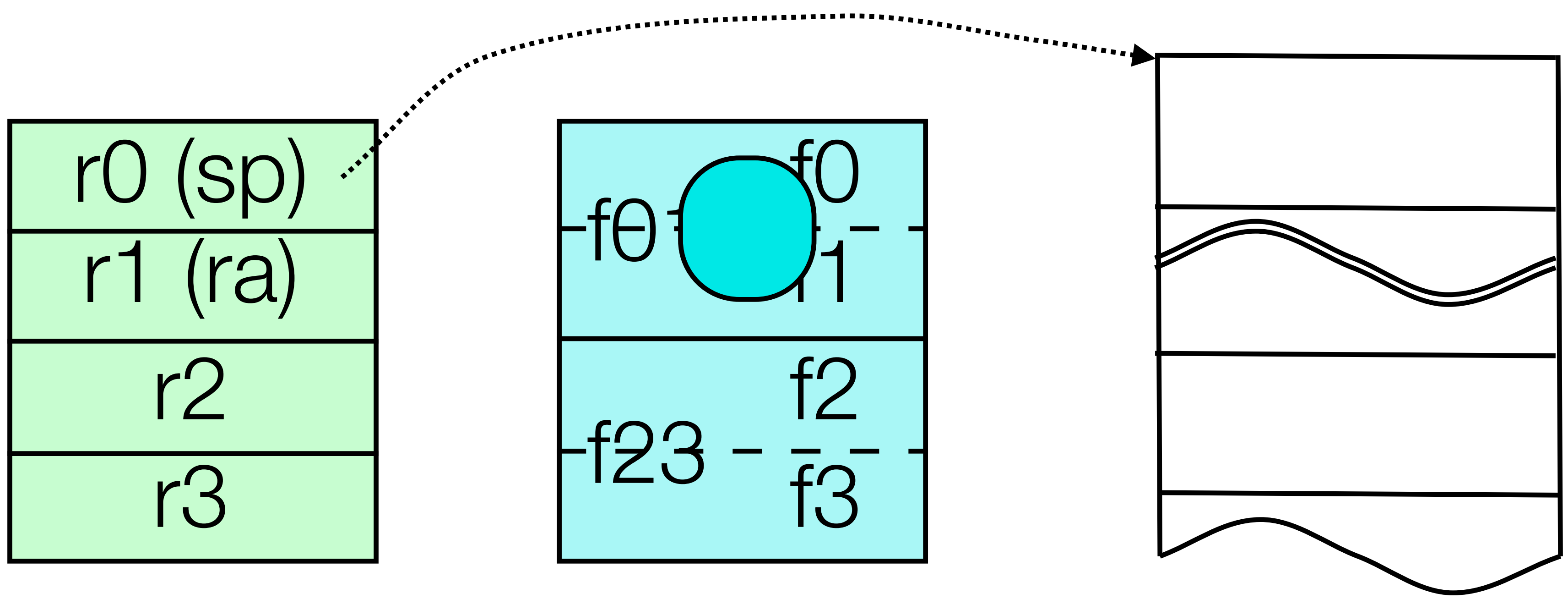
Prototype: **`int f (double, double, double, int, ptr);`**



Calling a variadic function from ML

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

implied (inferred?)
Prototype: **`int f (double, double, double, int, ptr);`**

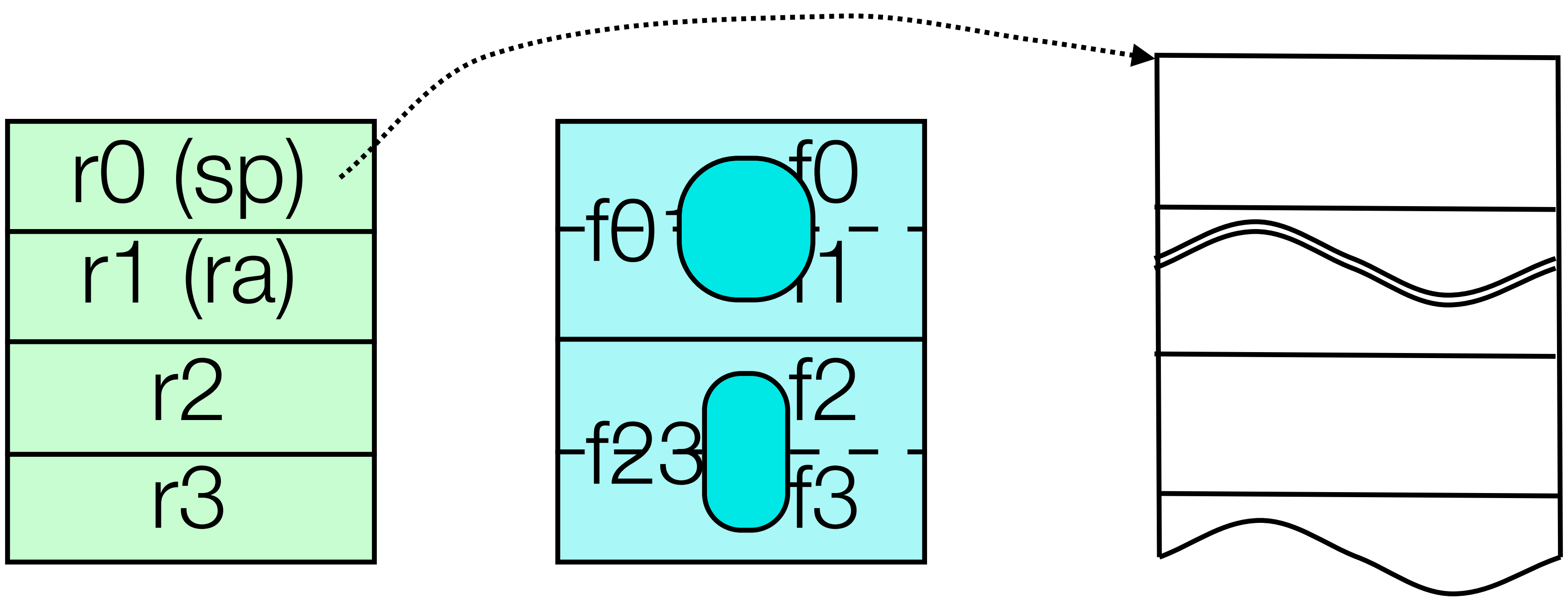


Calling a variadic function from ML

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

implied (inferred?)

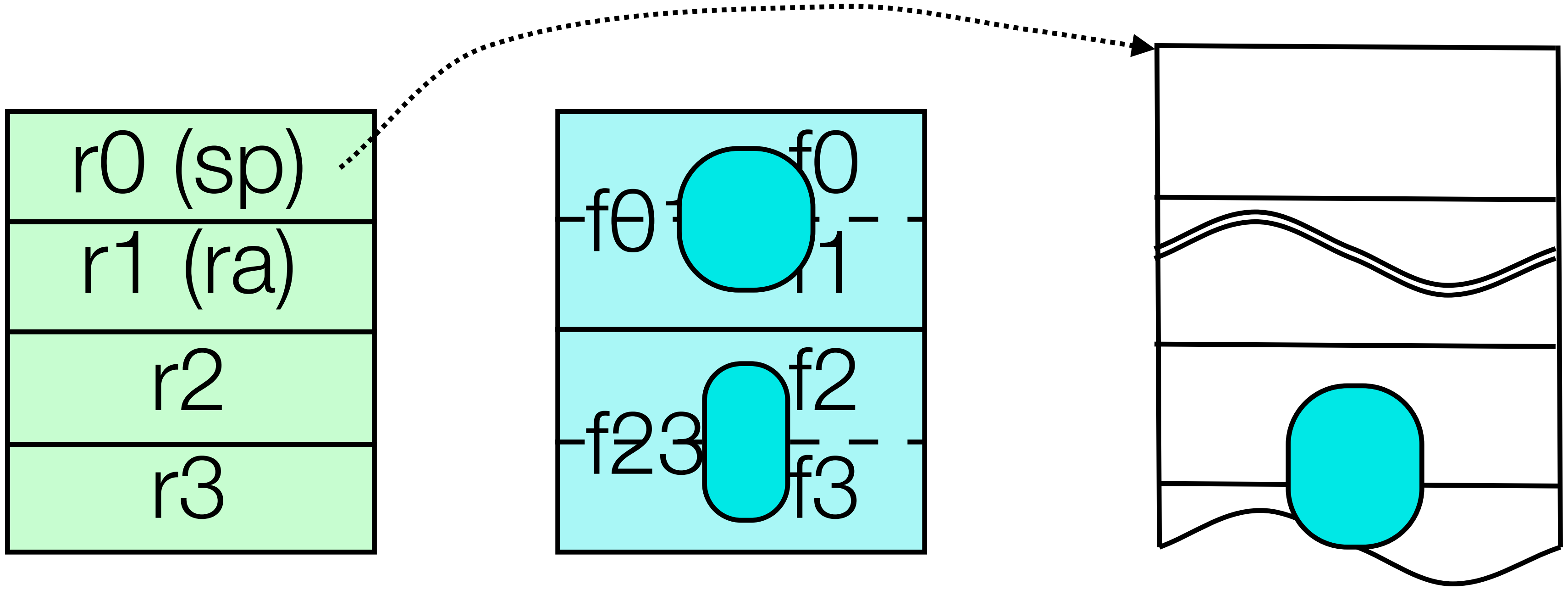
Prototype: **`int f (double, double, double, int, ptr);`**



Calling a variadic function from ML

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

implied (inferred?)
Prototype: **int f (double, double, double, int, ptr);**

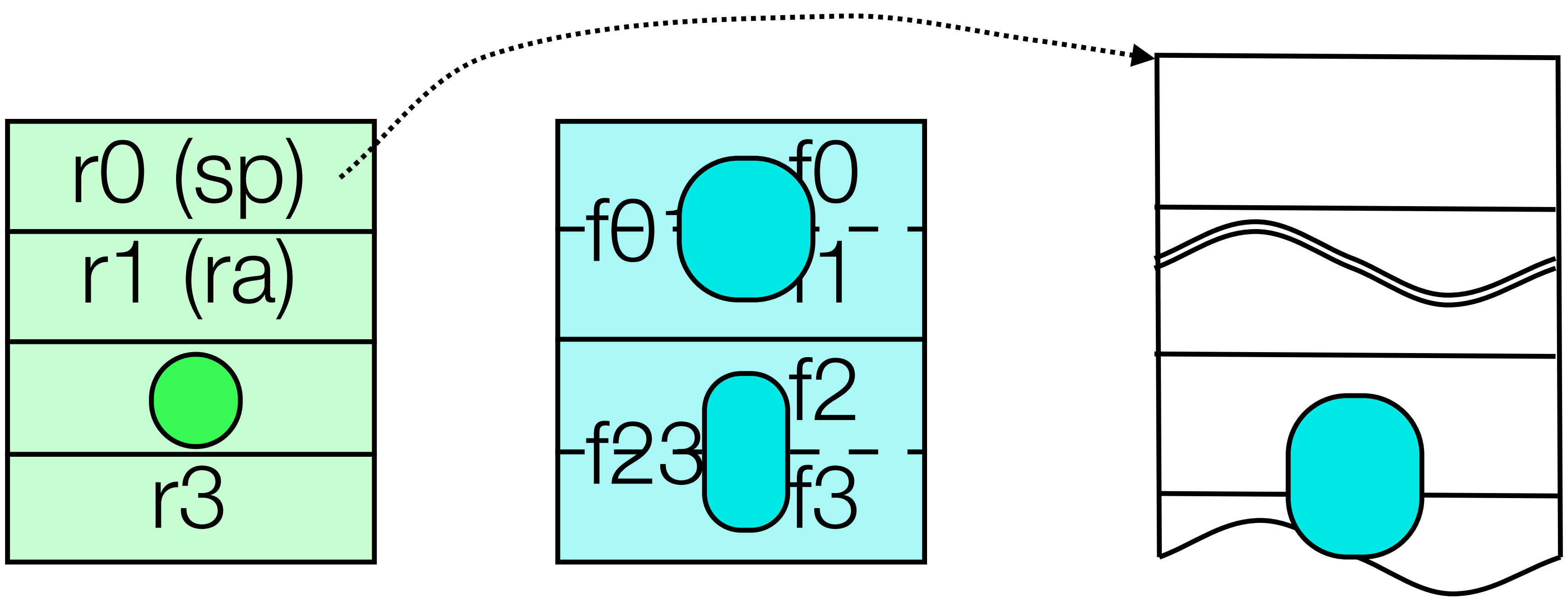


Calling a variadic function from ML

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

implied (inferred?)

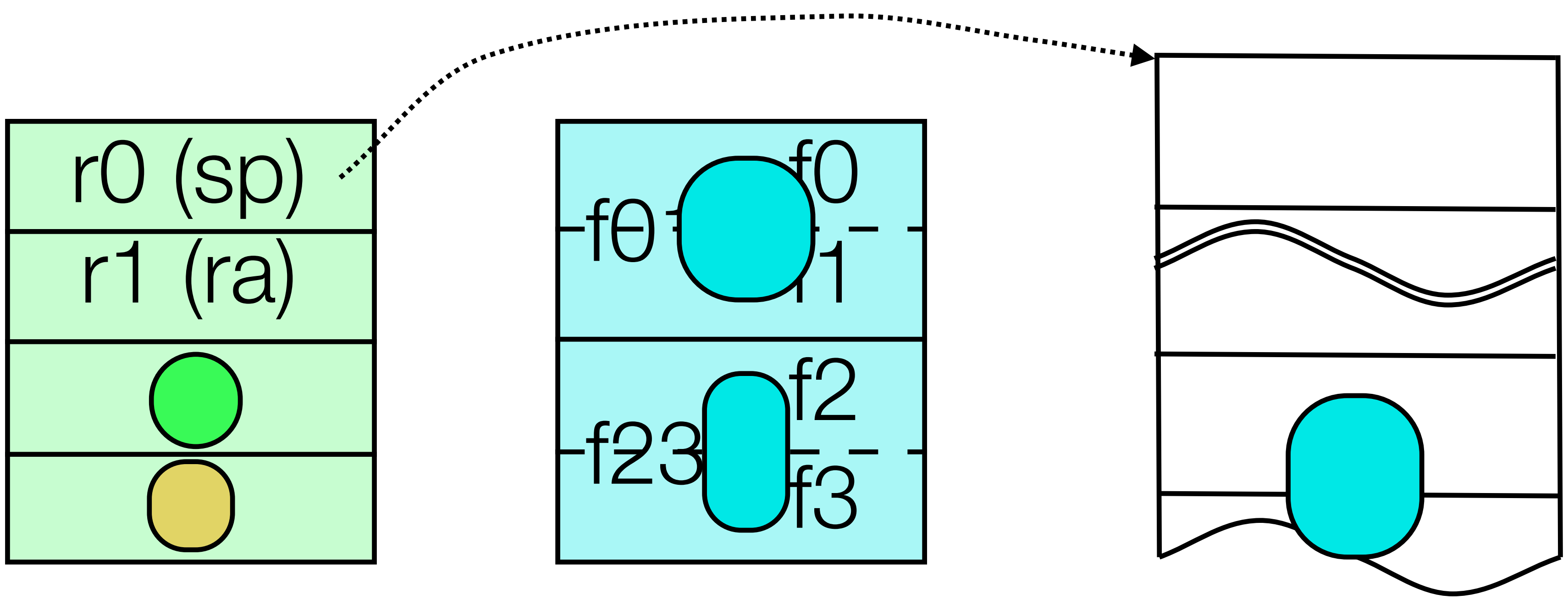
Prototype: **int f (double, double, double, int, ptr);**



Calling a variadic function from ML

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

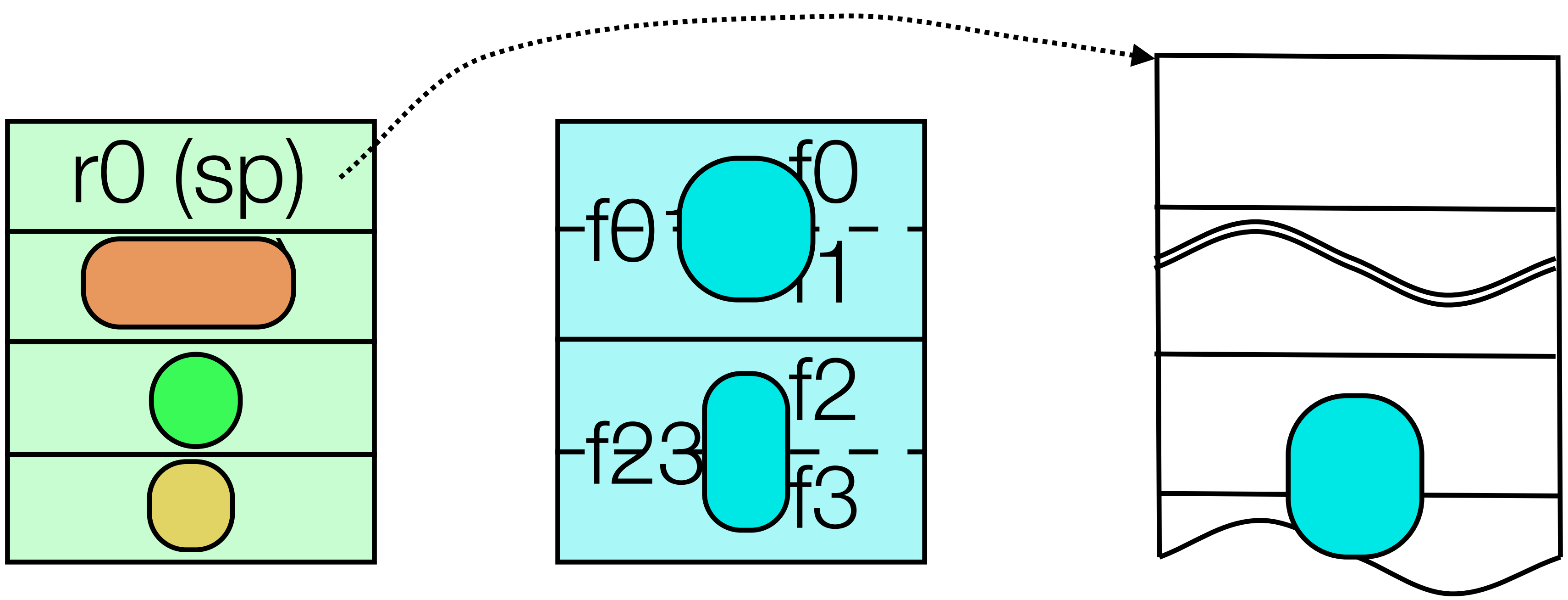
implied (inferred?)
Prototype: **int f (double, double, double, int, ptr);**



Calling a variadic function from ML

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

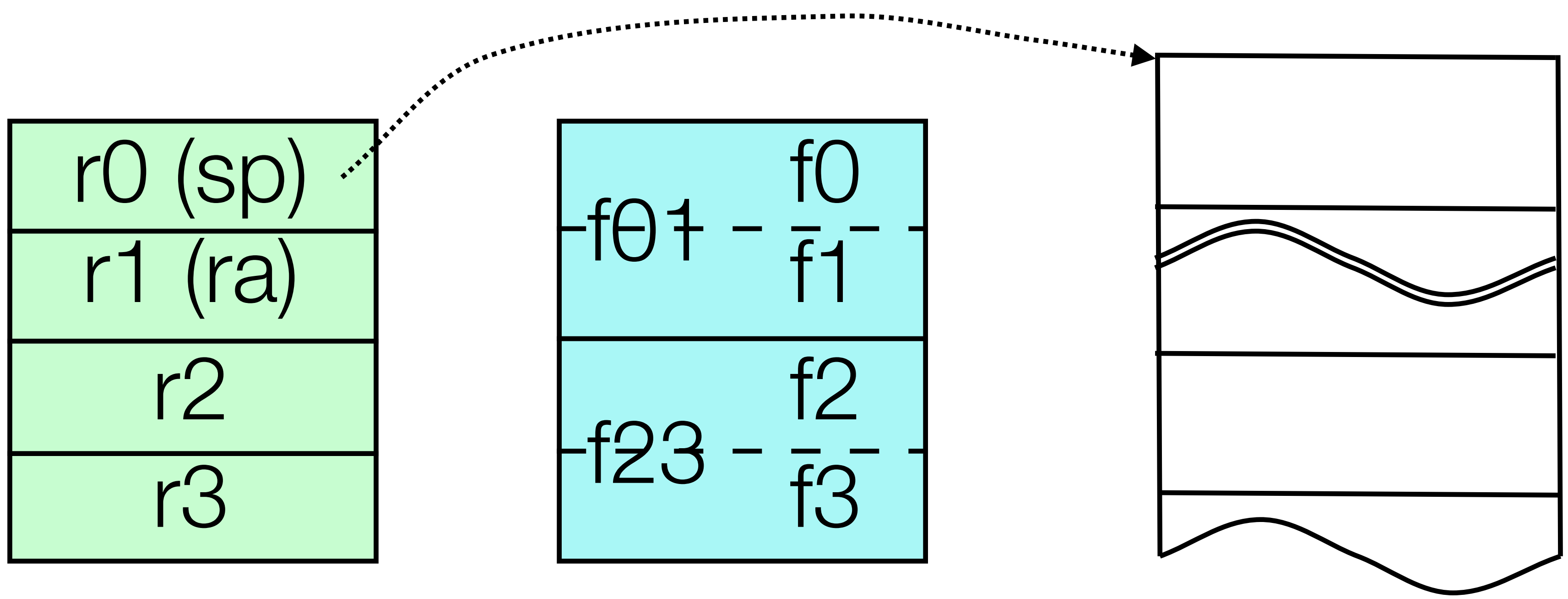
implied (inferred?)
Prototype: **int f (double, double, double, int, ptr);**



Calling a variadic function from a ***polymorphic*** ML function

```
fun g i =  f (i, x, w, c, p) + 1
```

inferred Prototype: `int f (α, 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...

```
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...

```
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...

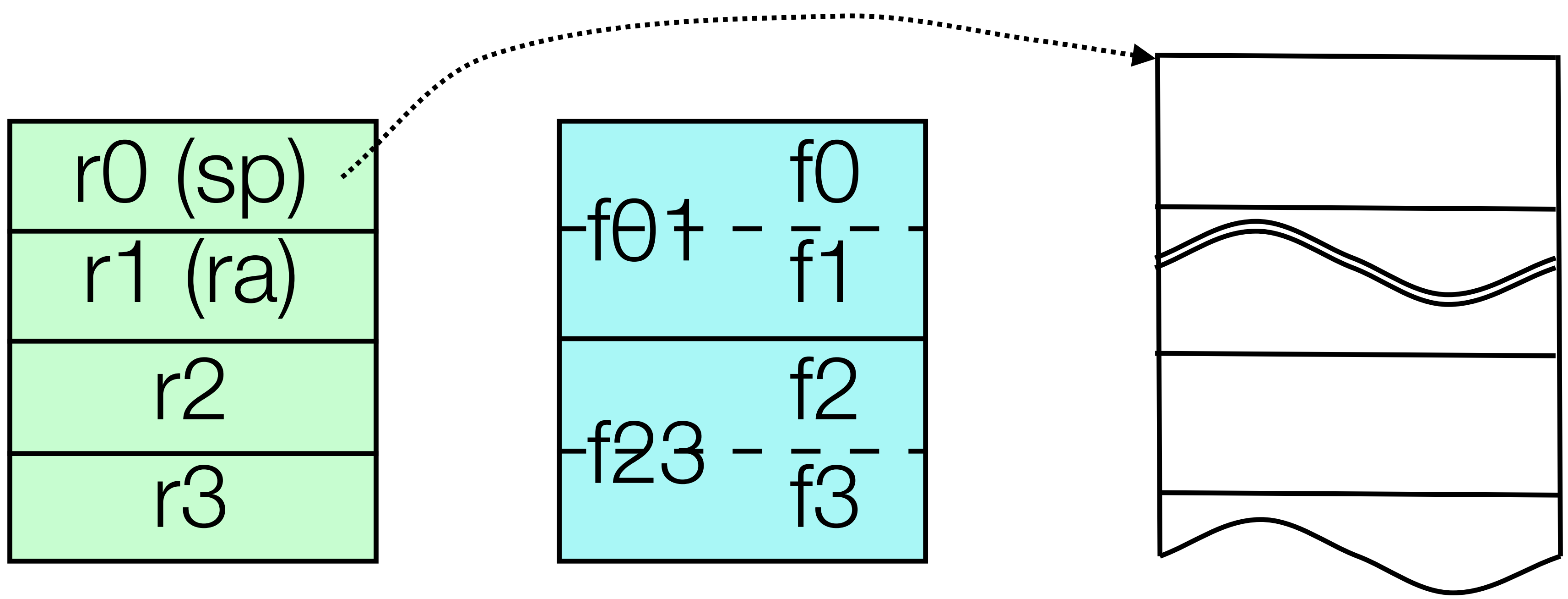
- ... we don't make use of the ML compiler's type information
  - Solution is not ML-specific.
  - It can be adapted even to dynamically typed settings.

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

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

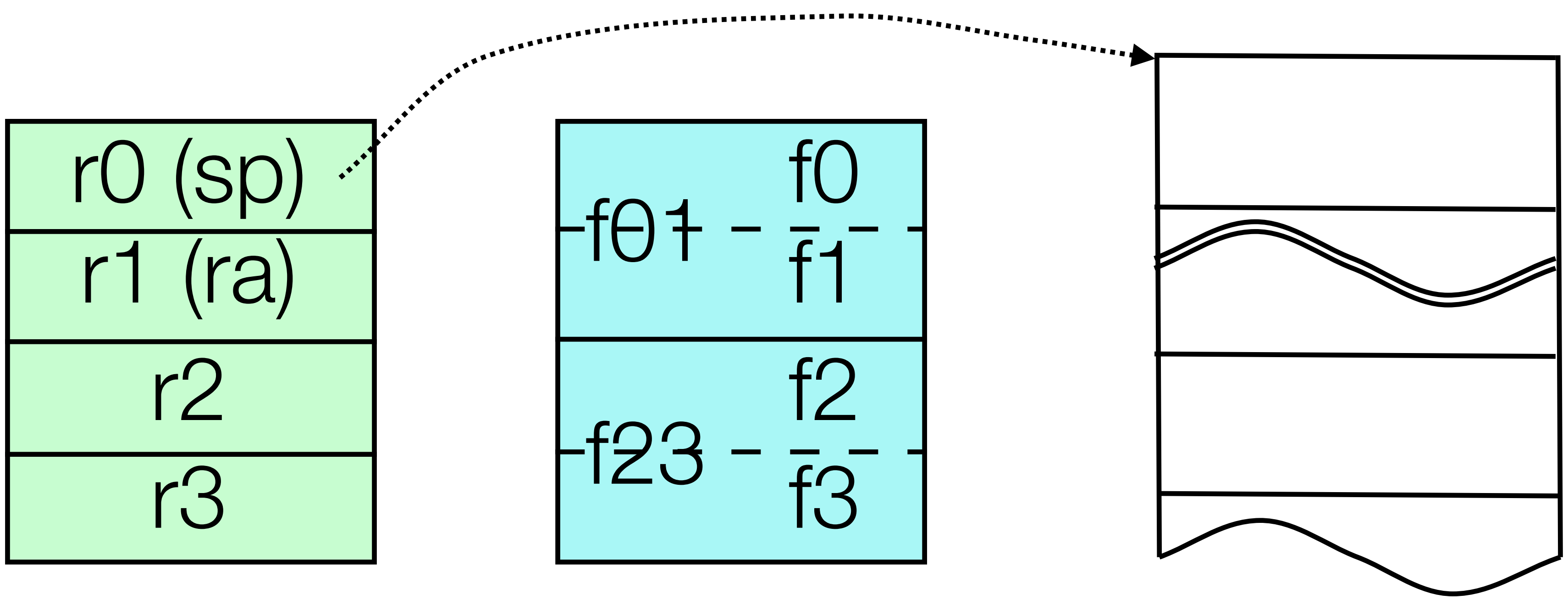


Calling a variadic function

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

implied

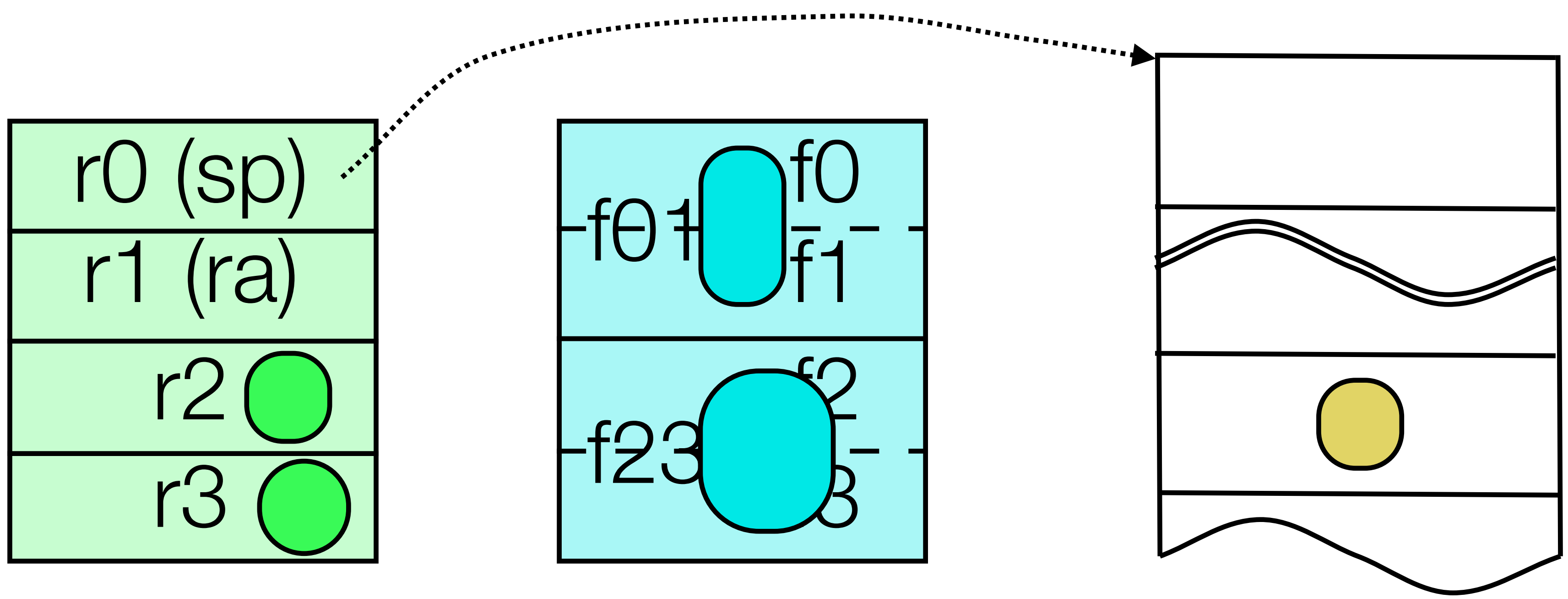
Prototype: `int f (int, double, double, int, void *);`



Calling a variadic function

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

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

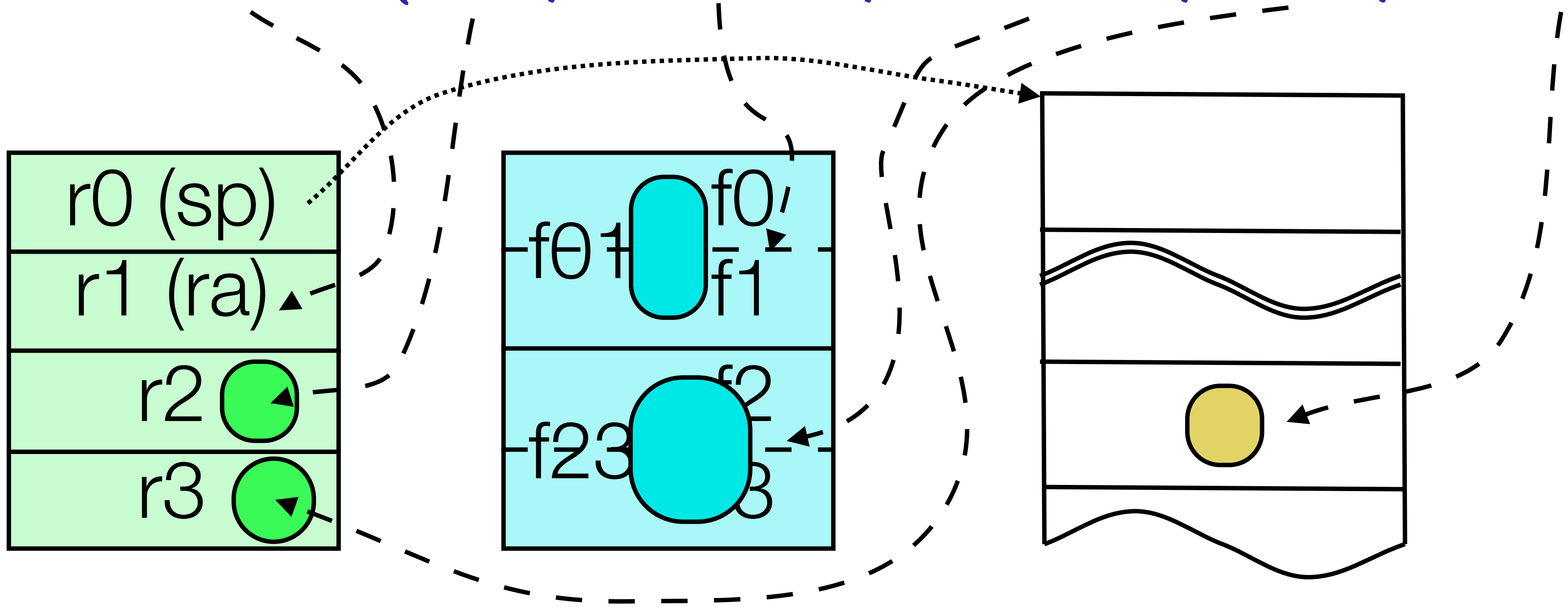


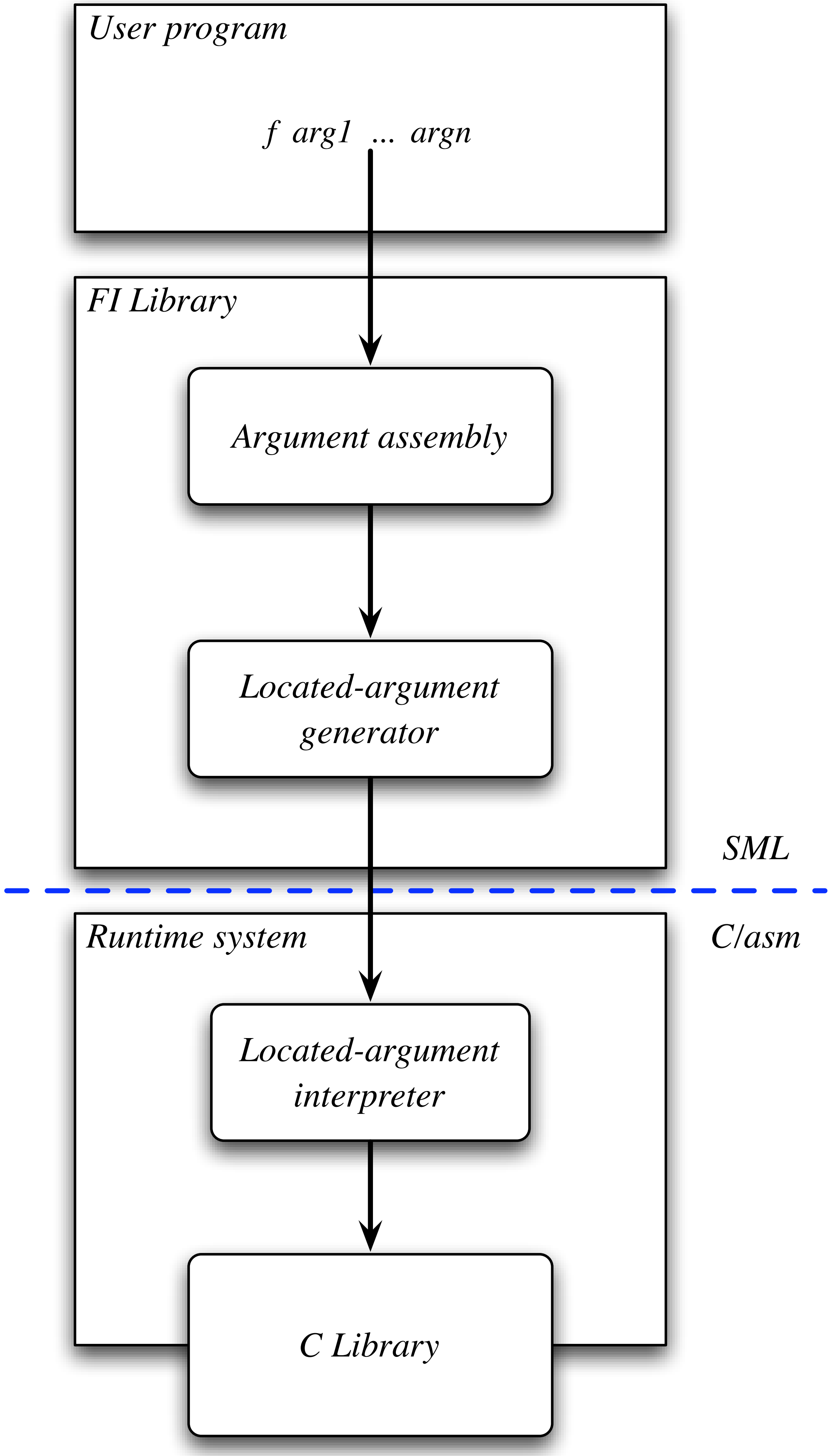
Calling a variadic function

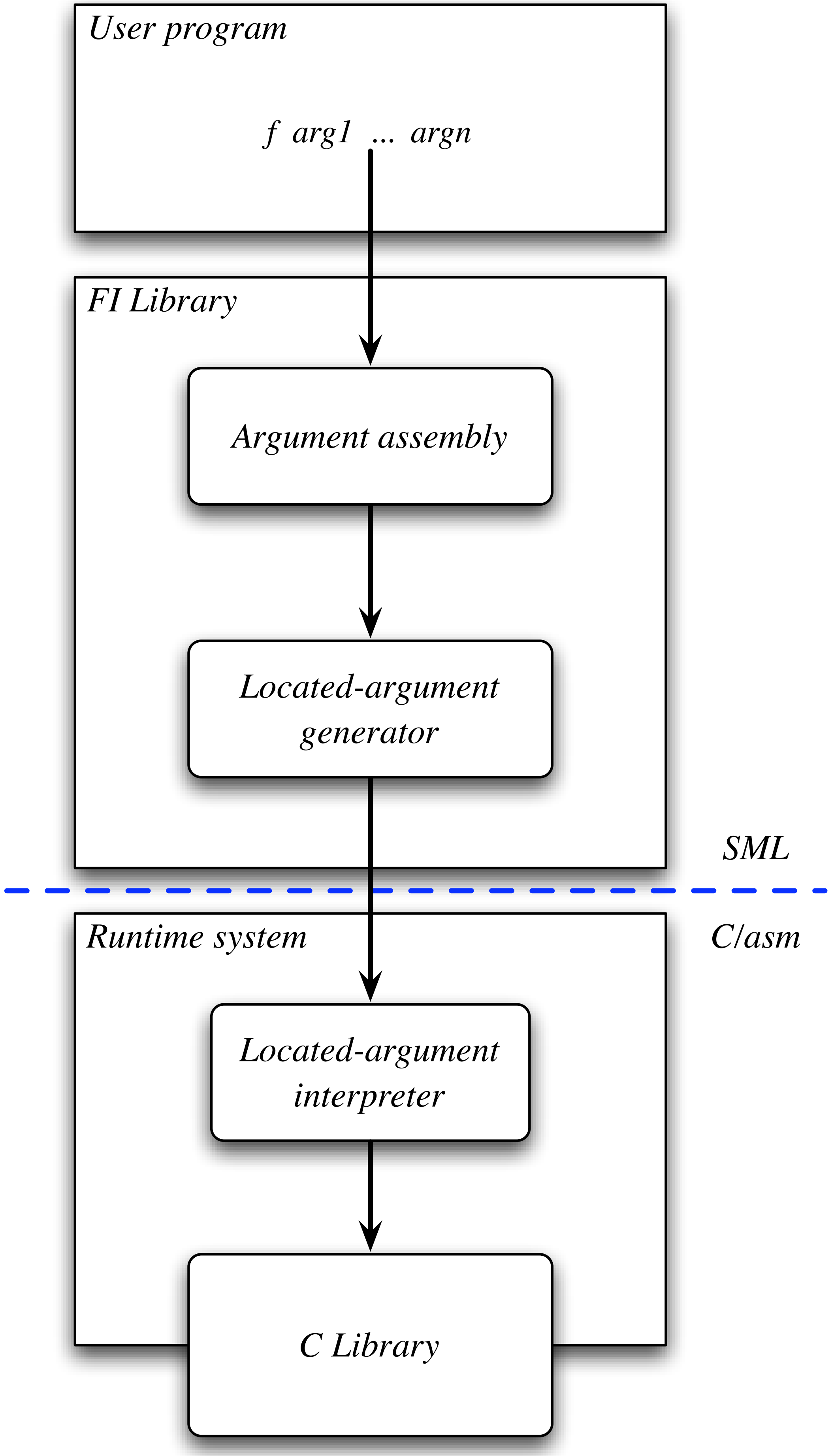
```
dispatch (f, [INT i, DOUBLE x, FLOAT w, CHAR c, PTR p])
```

implied

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

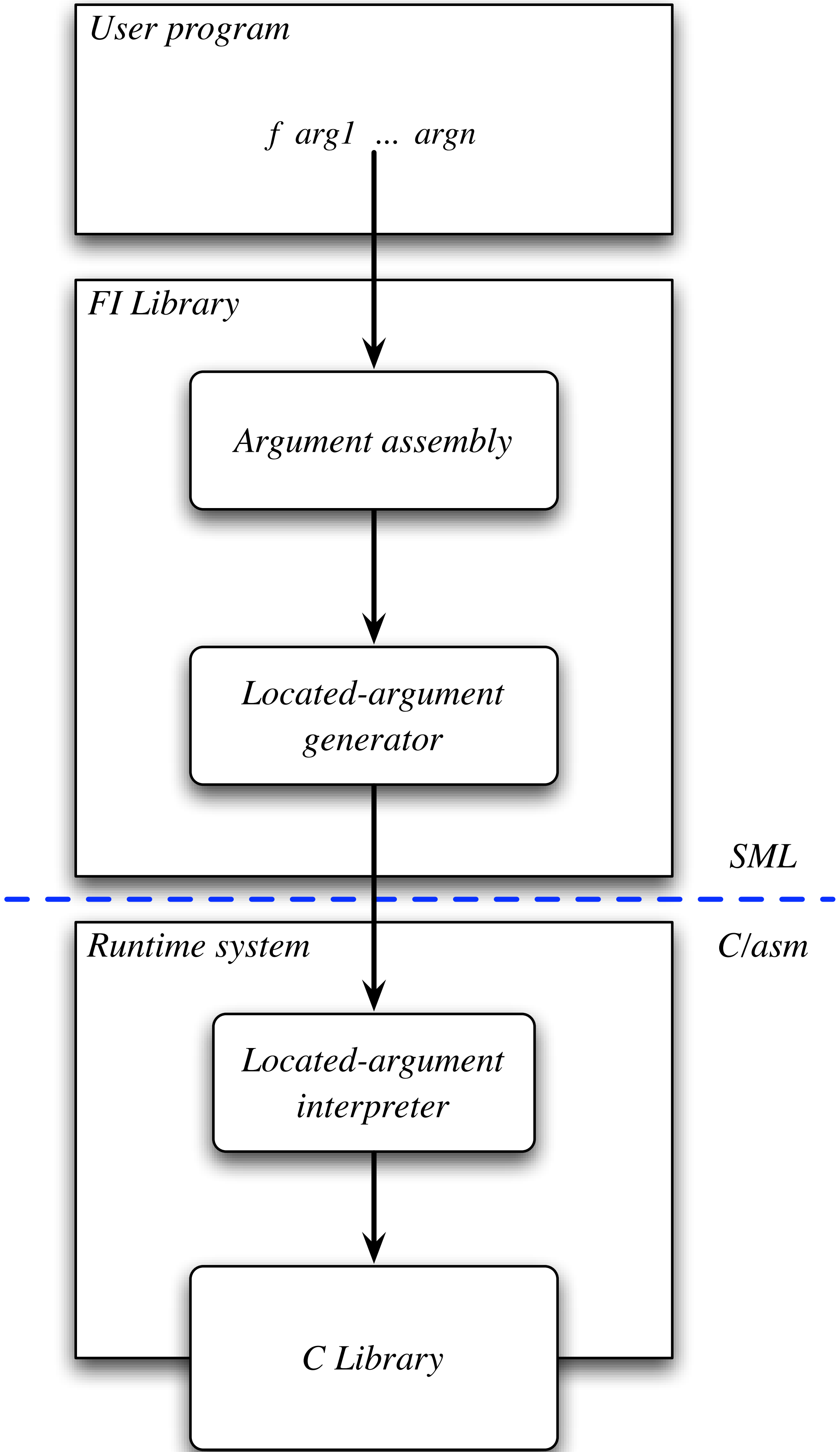



User program
f arg1 ... argn
FI Library
Argument assembly
Located-argument generator
SML
C/asm
Runtime system
Located-argument interpreter
C Library



"Staged Allocation"
(Olinsky, Lindig, Ramsey; POPL'06)

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



“Staged Allocation”
(Olinsky, Lindig, Ramsey; POPL’06)

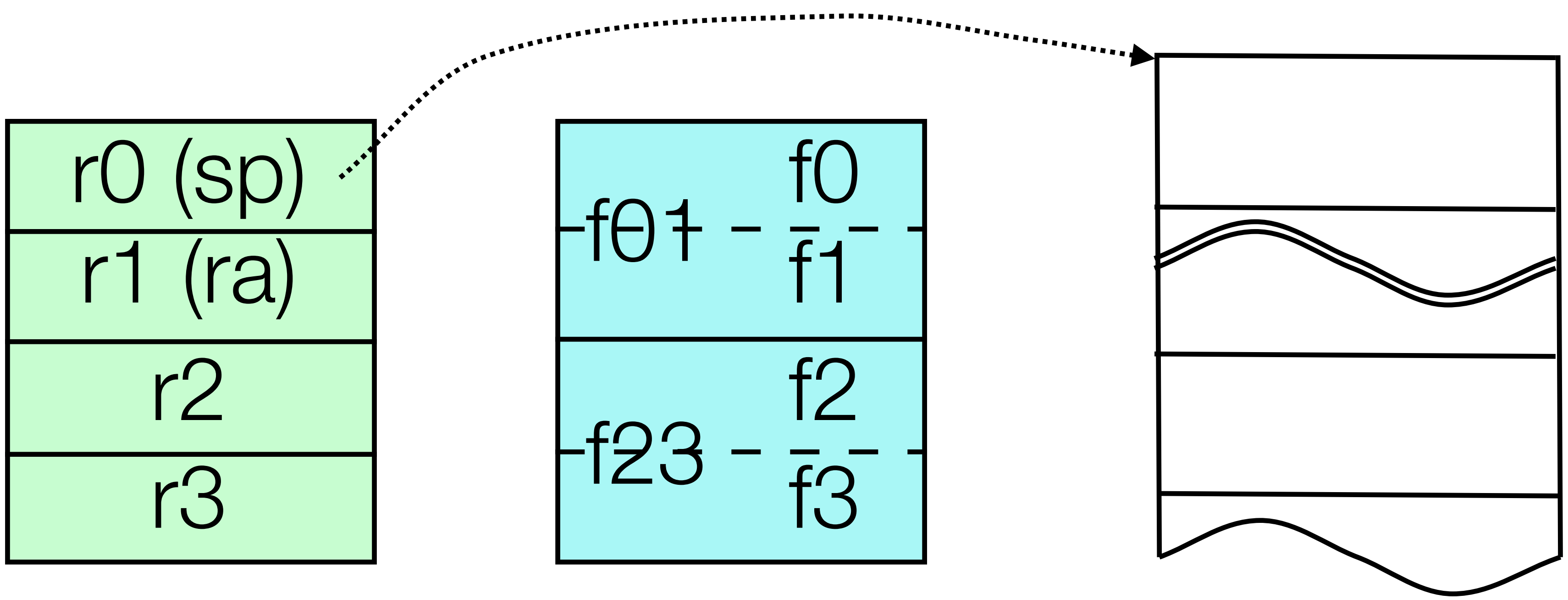
Reuses existing specs;
< 600 lines of (new) ML code

Generated for different architectures from a single MLRISC template

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

Located arguments

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



Staged Allocation

Staged Allocation

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

Staged Allocation

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

↓

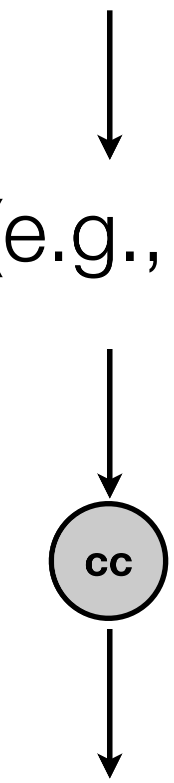
allocation request (e.g., **`(64, FPR, 8)`**, ...)

Staged Allocation

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

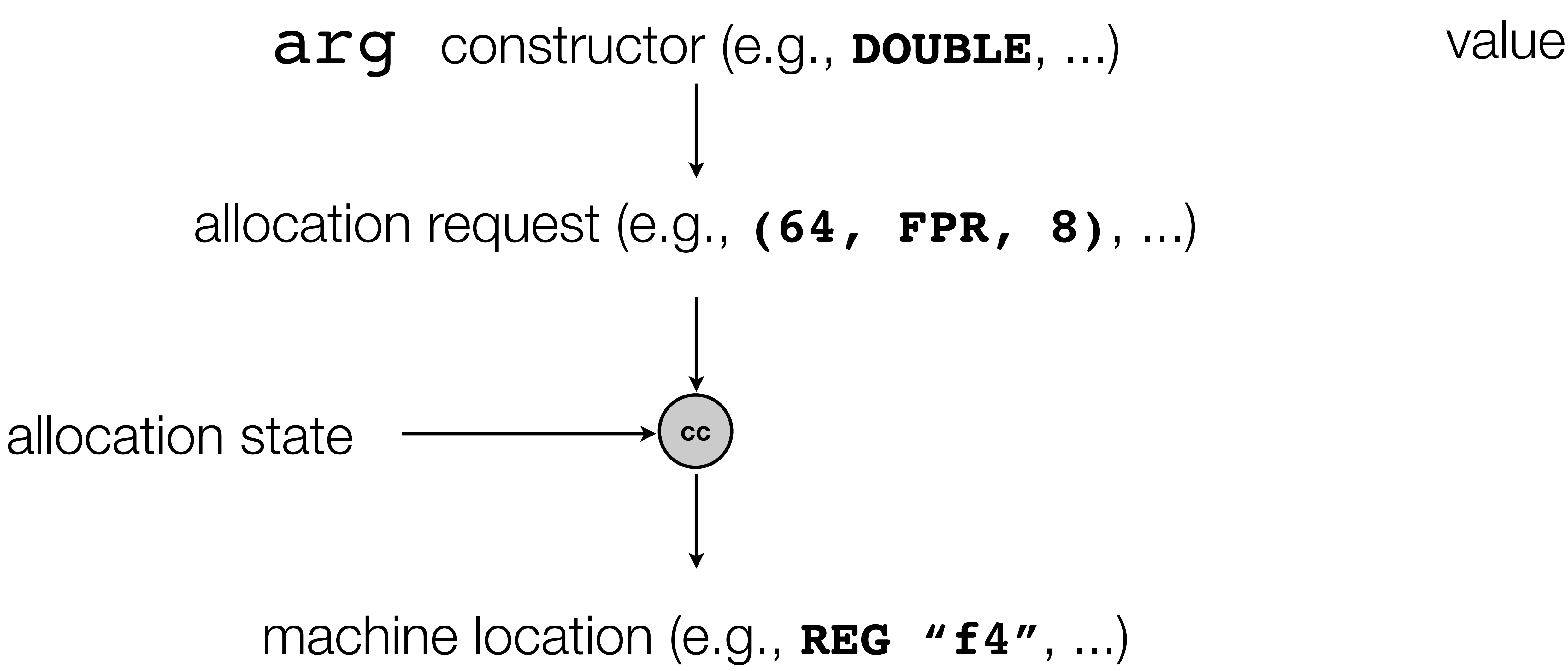
value

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

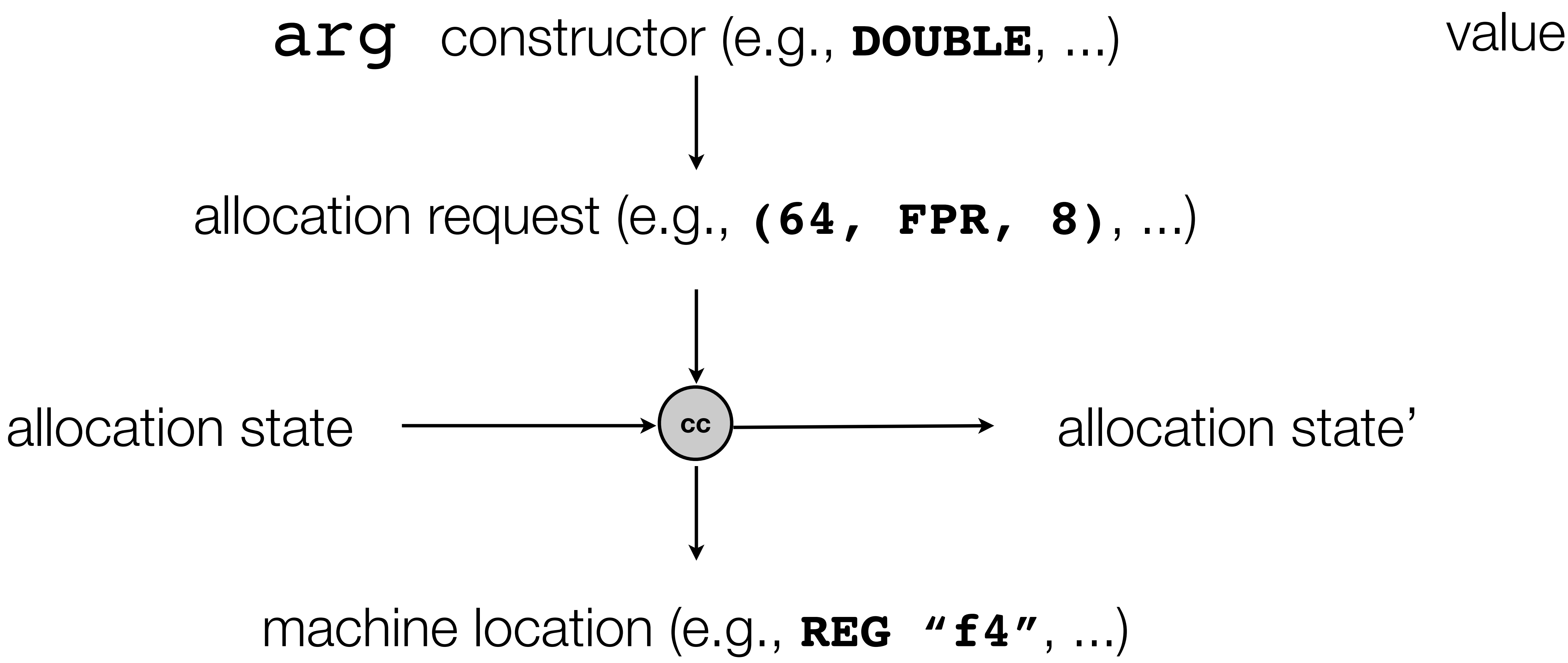


machine location (e.g., **`REG “f4”`**, ...)

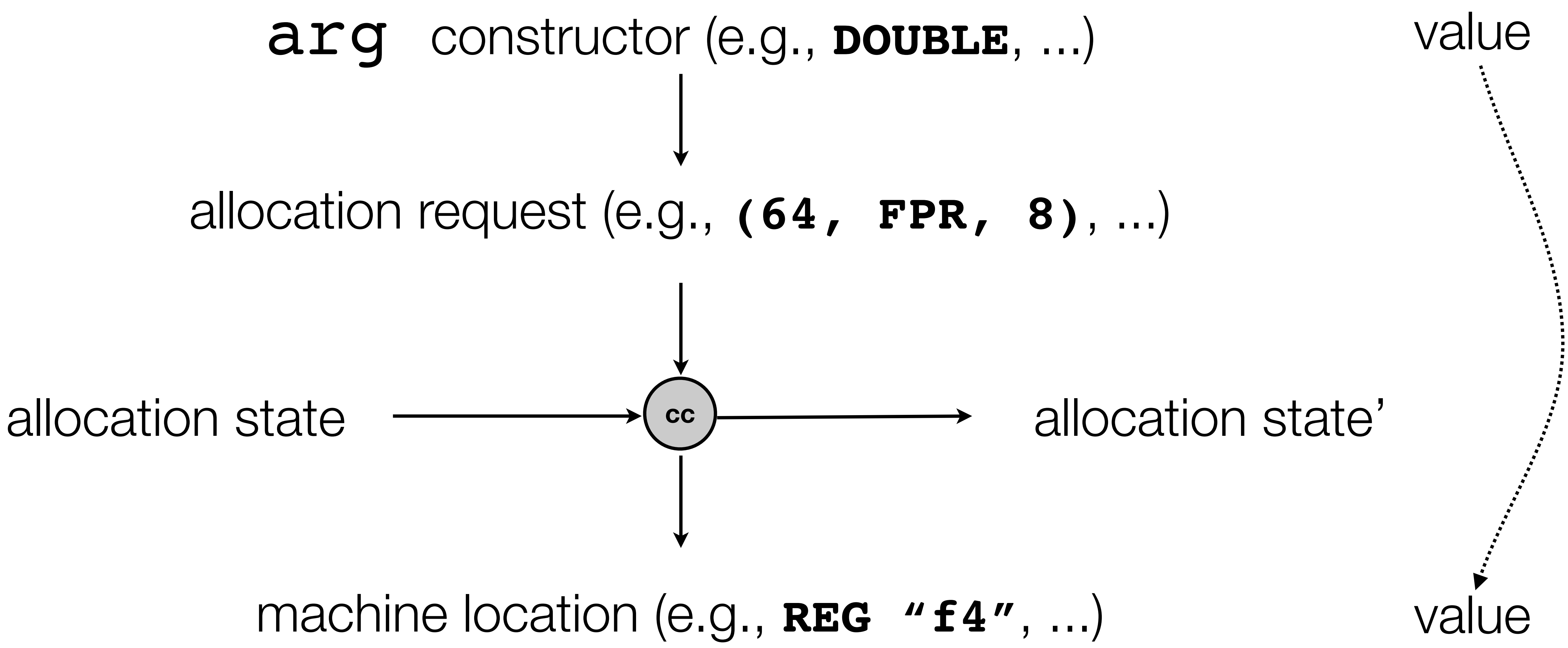
Staged Allocation



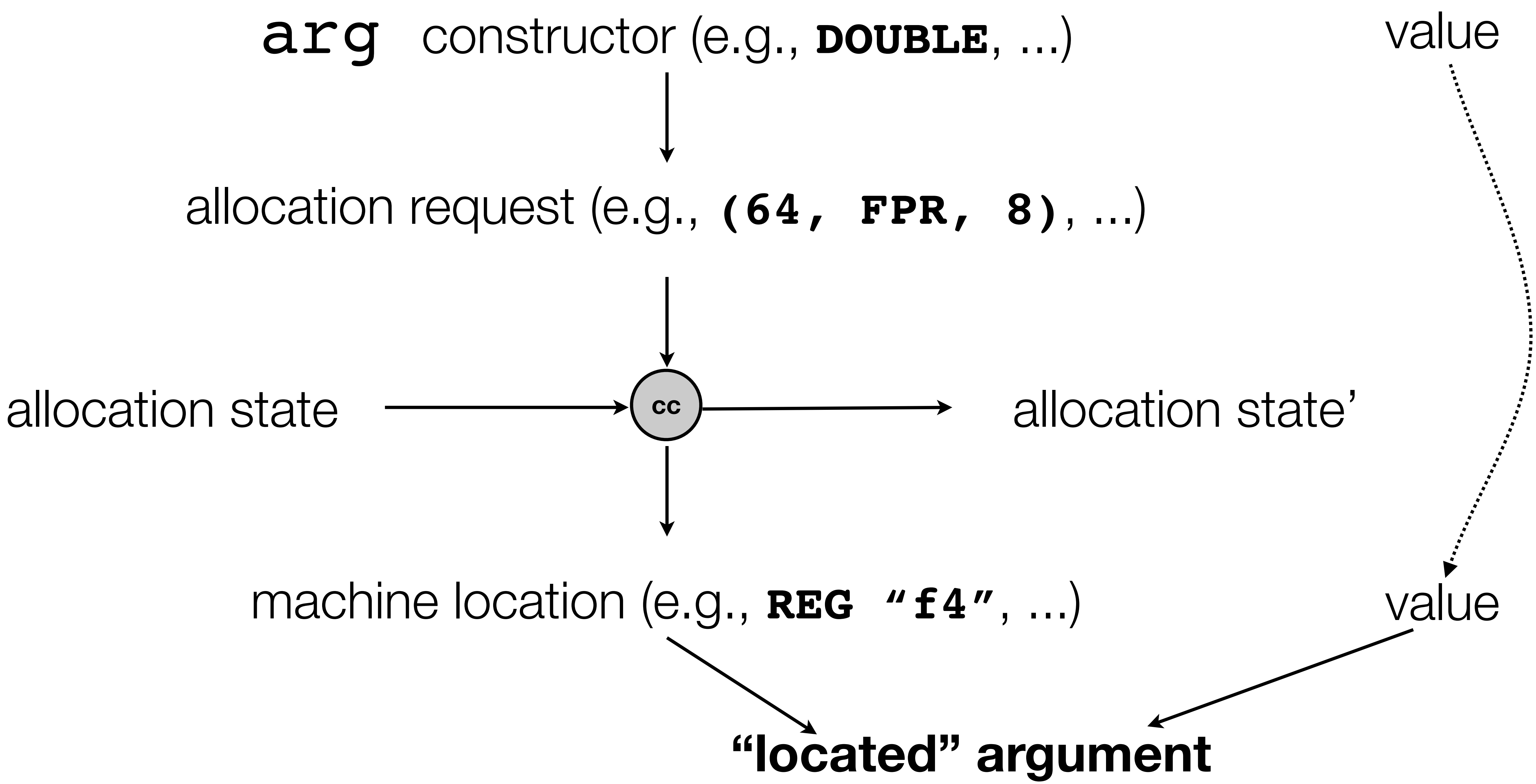
Staged Allocation



Staged Allocation

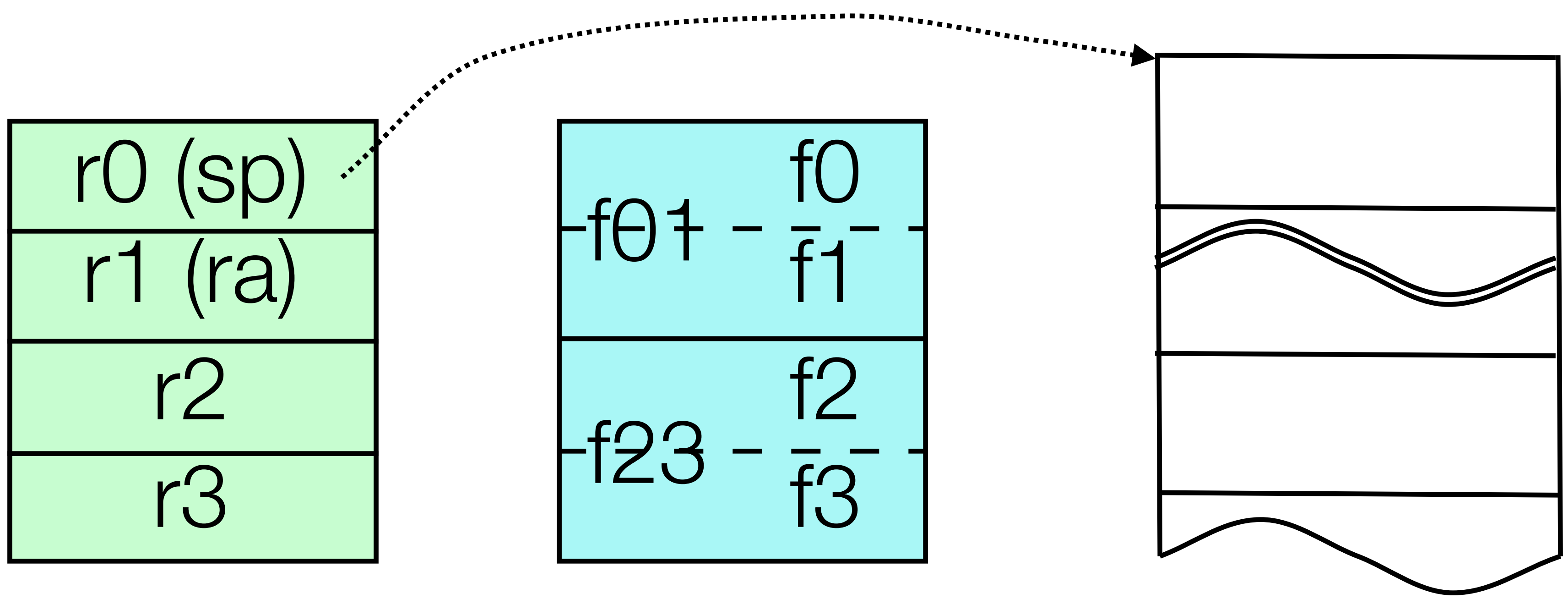


Staged Allocation



Located arguments

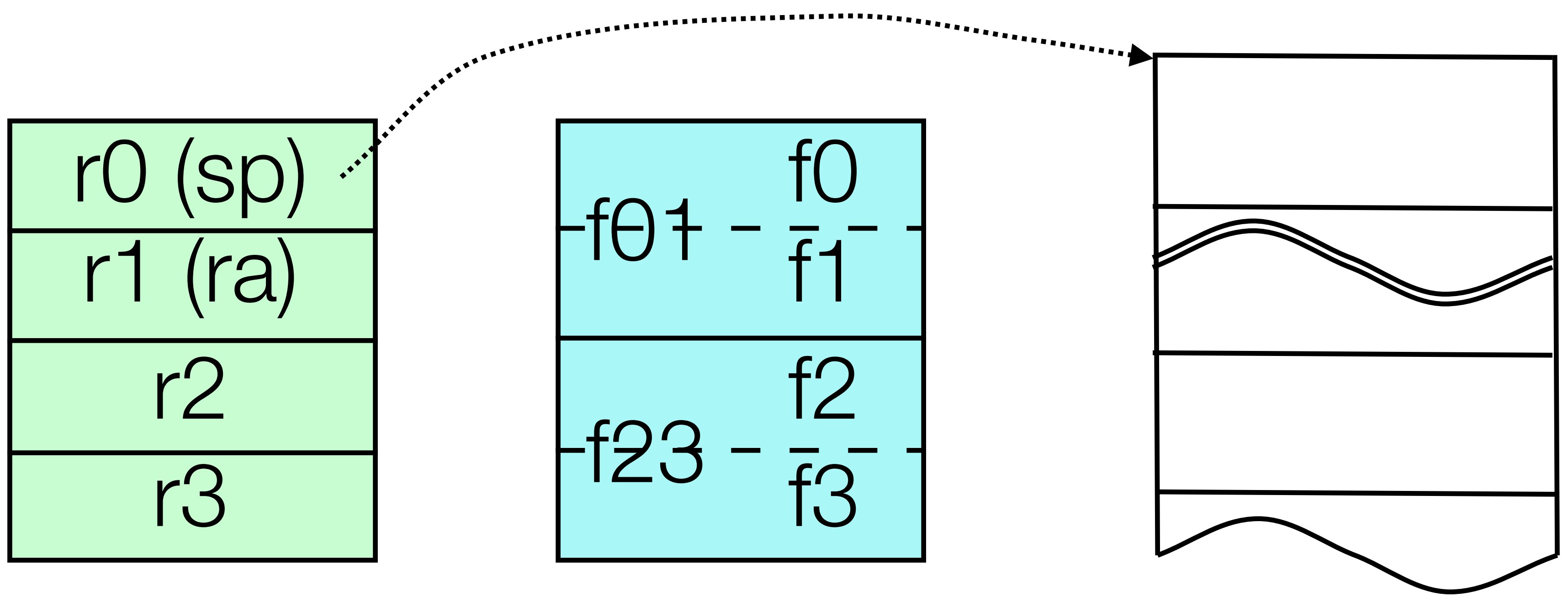
```
dispatch (f, [INT i, DOUBLE x, FLOAT w, CHAR c, PTR p])
```



Located arguments

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

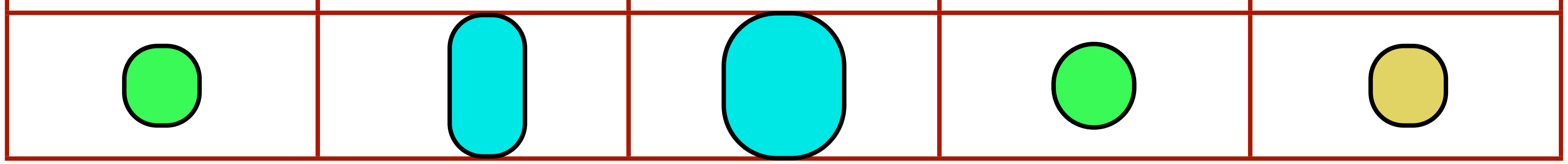
“r2”	“f01”	“f23”	“r3”	“(sp+24)”

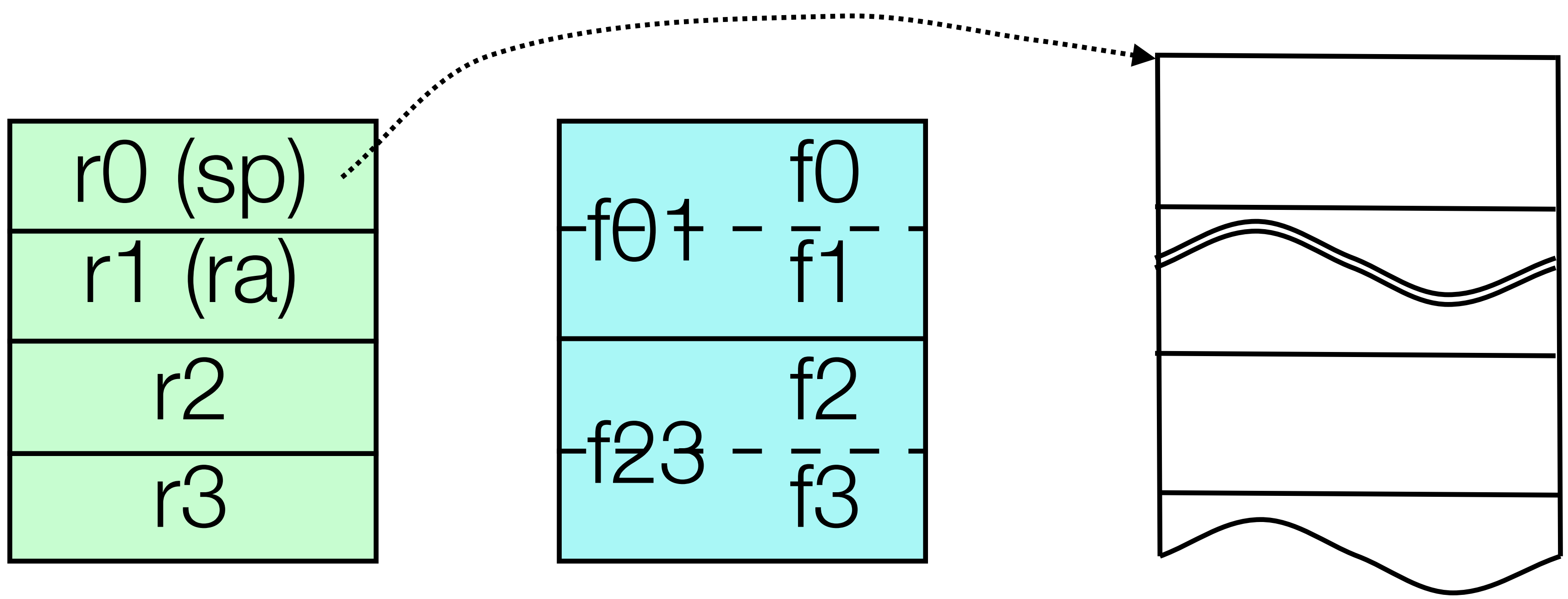


Located arguments

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

“r2”	“f01”	“f23”	“r3”	“(sp+24)”

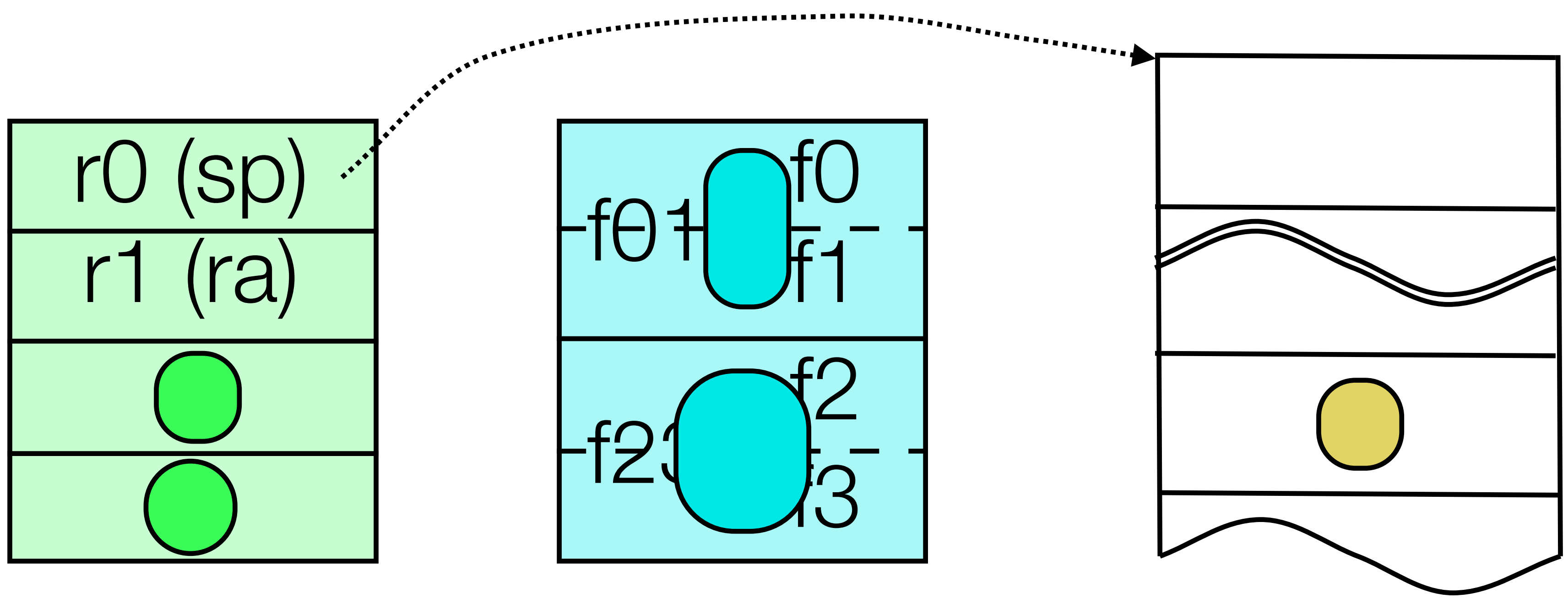




Located arguments

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

“r2”	“f01”	“f23”	“r3”	“(sp+24)”



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!