Jasmin: Language Reference

Language Guild Documentation

Formal Verification & Implementation Standards

Ji, Yong-hyeon
hacker3740@kookmin.ac.kr

Department of Cyber Security
Kookmin University

April 18, 2025

Contents

Chapter 1

A

1.1 About Jasmin

Jasmin is a workbench for high-assurance and high-speed cryptography. Jasmin implementations aim at being efficient, safe, correct, and secure.

The Jasmin **programming language** smoothly combines high-level and low-level constructs, supporting "assembly in the head" programming. Programmers can control many low-level details that are performance-critical—such as instruction selection and scheduling, and register spilling—while also leveraging high-level abstractions (variables, functions, arrays, loops, etc.) to structure their code and facilitate formal verification.

The **semantics** is formally defined to enable rigorous reasoning about program behavior. The Coq definitions can be found in the `proofs/lang/sem.v` file. This semantics is executable, allowing Jasmin programs to be directly interpreted.

Jasmin programs can be automatically checked for **safety** and **termination** using a trusted static analyzer.

The Jasmin **compiler** produces predictable assembly and guarantees that the use of high-level abstractions incurs no run-time penalty. It is formally verified for correctness (the precise Coq statement and the corresponding machine-checked proofs are located in the `proofs/compiler/compiler_proof.v` file). This ensures that properties proved on the source program—such as safety, termination, and functional correctness—hold for the corresponding assembly program.

The Jasmin workbench leverages the EasyCrypt toolset for **formal verification**. Jasmin programs can be extracted to corresponding EasyCrypt programs, facilitating proofs of functional correctness, cryptographic security, and constant-time security against side-channel attacks.

1.2 Arrays in Jasmin programs

Caveat: This description applies to Jasmin versions 2022.04.0 and more recent.

1.2.1 Declaration

Arrays may be allocated in registers, in the stack, or in global memory (i.e., in the code segment, which is immutable). A declaration of (local) array variables has the following shape:

```
1 stack u64[5] a b c;
```

- **Storage class:** reg, stack, reg ptr or stack ptr;
- **Type of the elements;**
- **Number of elements,** specified between square brackets;
- **Variable names.**

1.2.2 Access

The common way to access (read or write) an array cell is to use indexing, as in a[3], meaning "access the fourth element of array a". The first element has index zero.

Indexes into register arrays should be statically known (after inlining, unrolling, and constant propagation); otherwise, compilation fails because each array cell is allocated to a specific (named) register.

Arrays in memory (stack or global) may be indexed by run-time values (i.e., values that are not statically known). Such an index may be stored in a register as a machine word; it must be syntactically cast into an integer. For example:

```
1 reg u64 r i;
2 stack u64[4] a;
3 ...
4 r = a[i];
```

Explicit Scaling

When computing the address of an array element, the index is *implicitly* scaled by the declared size of the elements in the array. For example:

```
1 stack u64[4] a;
2 a[3] = 1;
```

In this snippet, the second line accesses the 64 bits in memory at offset 3×8 after the beginning of array a since each element is 8 bytes wide (thus, the fourth element starts at offset 24). If the index is statically known, scaling is computed at compile-time.

Jasmin allows disabling implicit scaling via the following syntax for direct (i.e., unscaled) access:

```
1 a.[24] = 1;
```

Notice the dot before the opening square bracket. This feature is especially useful when the underlying instruction set does not support the required scaling. For instance, on x86-64, scales are limited to 1, 2, 4, or 8. Arrays with elements that are 128-bit wide (or larger) cannot be accessed using run-time indices that are implicitly scaled. In the following snippet, the first array access is rejected ("invalid scale"), so the second form is used with explicit scaling as a separate instruction:

```
stack u128[2] b;
reg u64 i;
reg u128 x;
...
x = b[i]; // rejected: invalid scale
i ≪= 4;
x = b.[i];
```

Type Punning

Arrays in memory are essentially a contiguous sequence of bytes. Over the lifetime of an array, this byte sequence might be interpreted in several ways: as an array of u8, or as an array of u16 (if the total byte size is even), etc. The type specified at declaration provides the default view; an expression like a[i] means "access the *element* at position i in array a, according to its declared type."

However, on every access, an alternative view may be used. For example, the expression a[u128 0] represents an access to the first element of array a seen as an array of 128-bit values. The type written immediately after the left bracket specifies the type for that access and is also the type of the value read or written.

This facility can be used with run-time indices (e.g., a[u16 i]) as well as with explicit scaling (e.g., a.[u128 i]).

1.2.3 Intuition about 'reg ptr' and 'stack ptr'

A 'reg ptr' is essentially a pointer that is compiled into an actual pointer. Similar to C, if you declare a pointer (e.g., '`int* p;`') and then write to it (e.g., `*p = 4;`) without initialization, you will likely encounter a runtime error. Likewise in Jasmin, if you define a reg ptr (e.g., reg ptr u64 [2] r;) and write to it (e.g., r[i] = 4) without initializing it (e.g., `r = t`, where `t` can be a `stack` array, another `reg ptr`, or a `stack ptr`), the compiler will reject the code.

There are two main cases where `reg ptr` plays an important role:

1. **Accessing a global array:** Direct access to a cell in a global array may not be possible. Instead, an intermediate `reg ptr` is needed. For example, rather than writing `g[i]`, you must declare a `reg ptr u64[N] r;` and assign `r = g;` (with the appropriate `N`), then use `r[i]`.

2. **Argument passing:** It is not possible to pass a stack array directly to a function. Instead, you pass its address using a `reg ptr`. Rather than calling `f(s)`, you declare a `reg ptr u64[N] r;` and assign `r = s;` (with the appropriate `N`), and then call `f(r)`. (Note that there is a compiler pass that performs this transformation automatically, so manual intervention is typically not required unless you wish to control the process explicitly.)

A `stack ptr` is only useful for spilling a `reg ptr`, that is, temporarily storing the pointer on the stack. You can only copy a `reg ptr` into a `stack ptr` or vice versa. For example:

```
stack u64[2] s;
reg ptr u64[2] r;
stack ptr u64[2] sp;

r = s;   // r is compiled into a register holding the address of s
sp = r;  // the address is copied onto the stack, freeing the register
...      // other code not using r
r = sp;  // the address is retrieved into a register
...      // r can be used again
```