Verification Condition Generator for Qhasm

Ming-Hsien Tsai

February 17, 2014

1 Grammar

annotations	::=	*annotation annotations* | *annotation*
annotation	::=	`var` *auxvars* | `predicate` *predicates* | `inv` *bexp* | `assume` *bexp* | `assert` *bexp* | `cut` *bexp*
auxvars	::=	*auxvar auxvars* | *auxvar*
auxvar	::=	*VAR* `=` *exp* | *VAR* `@` *NUM* | VAR `(` formals `)` `=` exp
predicates	::=	*predicate predicates* | *predicate*
predicate	::=	*VAR* `(` formals `)` `=` bexp
formals	::=	*fparams* | ϵ
fparams	::=	*fparam* `,` *fparams* | *fparam*
fparam	::=	*VAR* `@u` *NUM* | *VAR* `@` *NUM*
bexp	::=	`true` | *exp2* `=` *exp2* | *exp2* `!=` *exp2* | *exp2 lessop atomics* | *exp2 lessop atomics lessop exp2* | *exp2 moreop atomics* | *exp2 moreop atomics moreop exp2* | *bexp* `->` *bexp* | *bexp* `||` *bexp* | *bexp* `&&` *bexp* | `~` *bexp* | `(` *bexp* `)` | `(` *exp* `=` *exp* `)` `$` *NUM* | *VAR* `(` *actuals* `)`
atomics	::=	*atomic* | *atomic* `,` *atomics*
exp	::=	*bexp* `?` *exp2* `:` *exp2* | *exp2*
exp2	::=	*atomic* | *exp2* `+` *exp2* | *exp2* `-` *exp2* | *exp2* `*` *exp2* | *exp2* `&` *exp2* | *exp2* `|` *exp2* | *exp2* `^` *exp2* | *exp2* `%` *exp2* | *exp2* `%u` *exp2* | *exp2* `<<` *exp2* | *exp2* `>>` *exp2* | *exp2* `>>u` *exp2* *exp2* `**` *exp2* | `-` *exp2* | `~` *exp2*
atomic	::=	*NUM* | `carry` | *var* | *atomic* `.` *atomic* | *atomic* `@` *NUM* | *atomic* `@u` *NUM* | *atomic* `@l` | *atomic* `@h` | *atomic* `[` *NUM* `,` *NUM* `]` | *VAR* `(` actuals `)` | `(` *exp* `)`
actuals	::=	*aparams* | ϵ
aparams	::=	*exp* `,` *aparams* | *exp*
var	::=	*VAR* | `mem64` `[` *qvar* `+` *NUM* `]` | `mem64` `[` *VAR* `]` | *qvar* `[` *NUM* `]`
qvar	::=	*VAR*
lessop	::=	`<` | `<=` | `<u` | `<=u`
moreop	::=	`>` | `>=` | `>u` | `>=u`

2 Annotations

There are six kinds of annotations, namely `var`, `predicate`, `inv`, `assume`, `assert`, and `cut`. The symbol ϵ below denotes an empty string.

annotations	::=	*annotation annotations* | *annotation*
annotation	::=	`var` *auxvars* | `predicate` *predicates* | `inv` *bexp* | `assume` *bexp* | `assert` *bexp* | `cut` *bexp*
auxvars	::=	*auxvar auxvars* | *auxvar*
auxvar	::=	*VAR* `=` *exp* | *VAR* `@` *NUM* | VAR `(` formals `)` `=` exp
predicates	::=	*predicate predicates* | *predicate*
predicate	::=	*VAR* `(` formals `)` `=` bexp
formals	::=	*fparams* | ϵ
fparams	::=	*fparam* `,` *fparams* | *fparam*
fparam	::=	*VAR* `@u` *NUM* | *VAR* `@` *NUM*

2.1 Var

Define logical variables or functions. Neither the name of a logical variable nor the name of a logical function can appear in the Qhasm code. The value of a logical variable is evaluated when the variable is defined. The scope of a logical variable does not cross a cut. If the initial value of a logical variable is not given, the bit-width of the variable should be specified as an extension. If the initial value of a logical variable is given, the bit-width should be able to be deduced from the initial value. The expression of a logical function is evaluated when the function is invoked. A logical function can still be invoked after a cut. The bit-widths of function parameters should be specified explicitly.

2.1.1 Examples

```
var a = x@u128 * y@u128
    b = x.y
    c@u128
    f(d@u128) = d + x.y
```

The bit-widths of `a`, `b`, and `c` are respectively 128, 256, 128.

2.2 Predicate

Define predicates. Similar to logical functions, predicates can still be used after cuts.

2.2.1 Examples

```
predicate p(x@u64) = 0 <=u x[0], x[8] <u 2**51
```

2.3 Inv

Specify invariants, which are mainly used as the assumption of input variables that never change. A defined invariant will be replaced by an assumption and will be inserted to every cut in the following annotations. For example,

```
inv 0 <=u x <=u 2**52
...
cut e1
...
cut e2
...
assert e3
```

is equivalent to

```
assume 0 <=u x <=u 2**52
...
cut e1 && 0 <=u x <=u 2**52
...
cut e2 && 0 <=u x <=u 2**52
...
assert e3
```

.

2.4 Assume

Assume that a Boolean expression holds. This can be used to add assumptions about the input variables.

2.4.1 Examples

```
assume 0 <=u x <=u 2**52 && 0 <=u y <=u 2**52
```

2.5 Assert

Verify if an assertion holds.

2.5.1 Examples

```
assert (x@u512 - y@u512) % (2**255 - 19) = 0
```

2.6 Cut

Verify if an assertion holds and make the assertion the assumption of the following Qhasm code. Qhasm code before a cut will not be considered in the Qhasm code after the cut. Thus, the cut can be viewed as an abstraction of the Qhasm code before the cut.

2.6.1 Examples

```
cut (x@u512 = y@u512) % (2**255 - 19) = 0
```

3 Boolean Expressions

bexp	::=	`true` | *exp2* `=` *exp2* | *exp2* `!=` *exp2* |
		exp2 lessop atomics | *exp2 lessop atomics lessop exp2* |
		exp2 moreop atomics | *exp2 moreop atomics moreop exp2* |
		bexp `->` *bexp* | *bexp* `||` *bexp* | *bexp* `&&` *bexp* | `~` *bexp* |
		`(` *bexp* `)` | `(` *exp* `=` *exp* `)` `$` *NUM* | *VAR* `(` *actuals* `)`
actuals	::=	*aparams* | ϵ
aparams	::=	*exp* `,` *aparams* | *exp*
lessop	::=	`<` | `<=` | `<u` | `<=u`
moreop	::=	`>` | `>=` | `>u` | `>=u`

Boolean expressions contain `true`, equality (`=`), inequality (`!=`), number comparison (`<`, `<=`, `<u`, `<=u`, `>`, `>=`, `>u`, `>=u`), implication (`->`), disjunction (`||`), conjunction (`&&`), negation (`~`), and instantiated predicates. The Boolean expression `e1 <= v1, v2 <= e2` is a shorthand of `e1 <= v1 <= e2 && e1 <= v2 <= e2`. The Boolean expression `(e1 = e2)$n` is a comparison of `n` smaller chunks of `e1` and `e2`. For example, given that both `e1` and `e2` have a bit-width 64, `(e1 = e2)$2` is equivalent to the following expression.

```
e1[31,0] = e2[31,0] && e1[63,32] = e2[63,32]
```

4 Expressions

exp	::=	*bexp* `?` *exp2* `:` *exp2* | *exp2*
exp2	::=	*atomic* | *exp2* `+` *exp2* | *exp2* `-` *exp2* | *exp2* `*` *exp2* |
		exp2 `&` *exp2* | *exp2* `|` *exp2* | *exp2* `^` *exp2* |
		exp2 `%` *exp2* | *exp2* `%u` *exp2* |
		exp2 `<<` *exp2* | *exp2* `>>` *exp2* | *exp2* `>>u` *exp2*
		exp2 `**` *exp2* | `-` *exp2* | `~` *exp2*
atomic	::=	*NUM* | `carry` | *var* | *atomic* `.` *atomic* |
		atomic `@` *NUM* | *atomic* `@u` *NUM* | *atomic* `@l` | *atomic* `@h` |
		atomic `[` *NUM* `,` *NUM* `]` | *VAR* `(` actuals `)` | `(` *exp* `)`
actuals	::=	*aparams* | ϵ
aparams	::=	*exp* `,` *aparams* | *exp*
var	::=	*VAR* | `mem64` `[` *qvar* `+` *NUM* `]` | `mem64` `[` *VAR* `]` | *qvar* `[` *NUM* `]`
qvar	::=	*VAR*

Expressions contain addition (`+`), subtraction (`-`), multiplication (`*`), bit-wise and (`&`), bit-wise or (`|`), bit-wise xor (`^`), signed mod (`%`), unsigned mod

(`%`), left-shifting (`<<`), arithmetic right-shifting (`>>`), logical right-shifting (`>>u`), two's complement (`-`), one's complement (`~`), extension (`@`, `@u`), concatenation (`.`), extraction (`[,]`, `@h`, `@l`), exponentiation (`**`), and function invocation. There are two kinds of extensions.

- `e@n` extends `e` to `n` bits arithmetically.
- `e@un` extends `e` to `n` bits logically.

There are three kinds of extractions.

- `e[n,m]` with `n` $\geq$ `m` extracts the bits between position n (included) and position m (included). The position starts from 0.
- `e@h` extract the higher bits of `e`.
- `e@l` extract the lower bits of `e`.

Memory access is written as `mem64[v + n]` or `v[n]`, where `v` is the base and `n` is the offset (multiple of 8). A variable is a *qvar* if it is a program variable in the Qhasm code.