Formalization and Test Generation for a CPU Architecture using Agda

Tatsuya Abe

CVS, AIST

Aim (1/2)

Aim: automatic test generation of a CPU

test = correctly work?

Aim (1/2)

Aim: automatic test generation of a CPU

test = correctly work?

To be concrete, we give a test generator of the CPU such that

input: one instruction specification of the CPU

output: test items for the instruction

Aim (1/2)

Aim: automatic test generation of a CPU

test = correctly work?

To be concrete, we give a test generator of the CPU such that

input: one instruction specification of the CPU

output: test items for the instruction

E.g., for an instruction ADD

input: addition, binary, receives immediate values or general registers, etc.

output: equations $0 + 0 = ?$, $? + ? = 0$, . . .

Aim (2/2)

In generating test items, we usually use implicit information except specifications of instructions.

Aim (2/2)

In generating test items, we usually use implicit information except specifications of instructions.

In the previous example (ADD), the implicit information is that equations w.r.t. 0 are useful for test.

Aim (2/2)

In generating test items, we usually use implicit information except specifications of instructions.

In the previous example (ADD), the implicit information is that equations w.r.t. 0 are useful for test.

In this study, we clarify such information (usually in skilled engineers' brain), and give how to write specifications of instructions (nicely using Agda).

Aim (2/2)

In generating test items, we usually use implicit information except specifications of instructions.

In the previous example (ADD), the implicit information is that equations w.r.t. 0 are useful for test.

In this study, we clarify such information (usually in skilled engineers' brain), and give how to write specifications of instructions (nicely using Agda).

In our writing style, specification of instruction has sufficient information for auto-generationg test items.

How to write specifications of instructions

One instruction specification is expected to be written at one page in a specification sheet.

How to write specifications of instructions

One instruction specification is expected to be written at one page in a specification sheet.

In our style, one instruction specification is exactly one record.

How to write specifications of instructions

One instruction specification is expected to be written at one page in a specification sheet.

In our style, one instruction specification is exactly one record.

Advantages of use of records:

How to write specifications of instructions

One instruction specification is expected to be written at one page in a specification sheet.

In our style, one instruction specification is exactly one record.

Advantages of use of records:

human readable each field has one job.

How to write specifications of instructions

One instruction specification is expected to be written at one page in a specification sheet.

In our style, one instruction specification is exactly one record.

Advantages of use of records:

human readable each field has one job.

change spec. of instr. enough to change a value of a field.

How to write specifications of instructions

One instruction specification is expected to be written at one page in a specification sheet.

In our style, one instruction specification is exactly one record.

Advantages of use of records:

human readable each field has one job.

change spec. of instr. enough to change a value of a field.

implement a test generatior our test generator chooses fields appropriately, gets values using the fields, combines the values, and generates test items.

Dependency in specification of instruction

Any instruction has arity (e.g., ADD has arity 2). Arguments are usually of the type [[ARG]] —arity information is lost!

Dependency in specification of instruction

Any instruction has arity (e.g., ADD has arity 2). Arguments are usually of the type [[ARG]] —arity information is lost!

Using dependent types, arguments are of the type as follows,

Dependency in specification of instruction

Any instruction has arity (e.g., ADD has arity 2). Arguments are usually of the type [[ARG]] —arity information is lost!

Using dependent types, arguments are of the type as follows,

```
record SpecOfInstr (instr : Instr) : Set where
  field
    numOfArgs : ℕ
    arguments : [ Vec ARG numOfArgs ] -- exact implementation!

spec : (instr : Instr) -> SpecOfInstr instr
spec ADD = record { numOfArgs = 2;
                    arguments = (IMM :: GR :: [])
                              ▷ (GR :: GR :: []) ▷ ε }
```

Readability of update function

In a standard manner, a state of general registers is defined as

```
State : Set           -- GR is the set of general registers
state = GR -> V       -- V is the set of values
```

Readability of update function

In a standard manner, a state of general registers is defined as

```
State : Set            -- GR is the set of general registers
state = GR -> V        -- V is the set of values
```

Getting a value is function application (e.g., state R1 = 13149).

Readability of update function

In a standard manner, a state of general registers is defined as

```
State : Set           -- GR is the set of general registers
state = GR -> V       -- V is the set of values
```

Getting a value is function application (e.g., state R1 = 13149).

Updating a state by a value is as follows,

Readability of update function

In a standard manner, a state of general registers is defined as

```
State : Set            -- GR is the set of general registers
state = GR -> V        -- V is the set of values
```

Getting a value is function application (e.g., state R1 = 13149).

Updating a state by a value is as follows,

$$f[x := v](y) = \begin{cases} v & \text{if } x = y \\ f(y) & \text{otherwise} \end{cases}$$

```
_[_:=_]_                : State -> GR -> V -> State
state [ r := v ] r' = if r == r' then v
                                 else (state r')
```

Test generator

```
spec BNOT = record { numOfArgs = 2;
                     arguments = (IM :: GR :: []) ▷ ··· ▷ ε;
                     rdFlags   = ε;
                     wrFlags   = ε;
                     rdInterp  = toFin 32 :: toVec 32 :: [];
                     wrInterp  = toVec 32 :: [];
                     rdknowhow = seeAll :: seeFancyBitPat :: [];
                     wrknowhow = seeFancyBitPat :: [];
                     ⋮
                   }
```

Test generator decides form of test items using values of numOfArgs, rdFlags, and wrFlags.

Test generator

```
spec BNOT = record { numOfArgs = 2;
                     arguments = (IM :: GR :: []) ▷ ··· ▷ ε;
                     rdFlags   = ε;
                     wrFlags   = ε;
                     rdInterp  = toFin 32 :: toVec 32 :: [];
                     wrInterp  = toVec 32 :: [];
                     rdknowhow = seeAll :: seeFancyBitPat :: [];
                     wrknowhow = seeFancyBitPat :: [];
                     ⋮
                   }
```

Test generator also gives test items using arguments, rdInterp, wrInterp, rdknowhow, and wrknowhow.

Test generator

```
spec BNOT = record { numOfArgs = 2;
                     arguments = (IM :: GR :: []) ▷ ··· ▷ ε;
                     rdFlags   = ε;
                     wrFlags   = ε;
                     rdInterp  = toFin 32 :: toVec 32 :: [];
                     wrInterp  = toVec 32 :: [];
                     rdknowhow = seeAll :: seeFancyBitPat :: [];
                     wrknowhow = seeFancyBitPat :: [];
                     ⋮
                   }
```

Test items are $\begin{array}{ll} 0 \cdot 0 = ?, & 0 \cdot (2^{32} - 1) = ?, \\ 0 \cdot (2^{32} - 2) = ?, & ? \cdot ? = 0, \end{array}$ etc.

Conclusion

We proposed how to write specification of instruction.

Conclusion

We proposed how to write specification of instruction.

In the writing style, we implemented a test generator (and a CPU emulator) using Agda.

Conclusion

We proposed how to write specification of instruction.

In the writing style, we implemented a test generator (and a CPU emulator) using Agda.

Advantages:

Conclusion

We proposed how to write specification of instruction.

In the writing style, we implemented a test generator (and a CPU emulator) using Agda.

Advantages:

- Specifications of instructions are not scattered all over the modules—one record for one instruction.

Conclusion

We proposed how to write specification of instruction.

In the writing style, we implemented a test generator (and a CPU emulator) using Agda.

Advantages:

- Specifications of instructions are not scattered all over the modules—one record for one instruction.
- Test items for the CPU are automatically generated.

Conclusion

We proposed how to write specification of instruction.

In the writing style, we implemented a test generator (and a CPU emulator) using Agda.

Advantages:

- Specifications of instructions are not scattered all over the modules—one record for one instruction.
- Test items for the CPU are automatically generated.
- We have possibility of showing some properties of test generator (written by Agda) using Agda in future.