Typing P4 v1.1
- problem statement and proposed solution -

Oct 2015

Part 1

PROBLEM STATEMENT

Assessment based on

- P4 v.1 spec
- P4 open-source v.1 compiler
- P4 behavioral model generated code, v.1

An example

```
header_type ht
{
  fields {
     fiveBits : 5;
     sum : 5;
  }
}

action sum()
{
  add(m.sum, m.fiveBits, m.fiveBits);
}
```

I will use a shorthand notation:

```
bit<5> fiveBits;
sum = fiveBits + fiveBits;
```

Legal program fragment, accepted by P4 v.1 compiler.

Q: What is fiveBits?

```
bit<5> fiveBits = 0xFF;
```

A) 0x1F
B) 0xFF (8 bits)
C) -1
D) An arbitrary value

Answer:
The spec says nothing, so the right answer could be D.
According to the behavioral simulator it is B
(the intention was A, but there is a bug in the mask computation).

Q: What is sum?

```
bit<5> fiveBits = 0x10;
bit<5> sum = fiveBits + fiveBits;
```

A) 0
B) 0x20 (six bits)
C) -32
D) An arbitrary value

Answer:
According to the spec it is not defined.
According to the behavioral simulator it is B.

Q: Which branch is taken?

```
signed<5> fiveBits = 0;
fiveBits = fiveBits + -1;

if (fiveBits < 0) {
    ...
}
else {
   ...
}
```

A) True
B) False

Answer:
According to the spec it is not defined.
According to the behavioral simulator it is B: all values are unsigned.
(The signed declaration produces no change in generated code).

Other integer manipulation problems

- Operations between values with different widths
- Operations on different signs
- Operations on values with different signs and widths
- Comparisons <, >=: are they signed or unsigned?
- Negation: is the result signed or unsigned?
- Overflow: what happens when operations produce values out of range?
- Masking: are results signed or unsigned?
- Shifting right: arithmetic or logical.
- Shifting with negative amounts.
- Sign-extension: enlarging signed values.

Ambiguity is evil

- If the spec is ambiguous or undefined, it can be implemented in many ways.
- Not all implementations will produce the same answer.
- We can debate endlessly about the right answer (see the definition of addition of signed integers in C).

Solution

- The spec should describe the legal behaviors in detail
- Incorrect programs should be rejected
- Correct programs should produce the spec-mandated answer
- Programs should not have *surprising* behaviors

- v1.1 rc0 spec draft fixed some of these, but not all

Types

- **All** modern programming languages solve this problem with types.
  - (This is true for both statically-typed and dynamically-typed languages).
  - P4 must be statically typed, because dynamic types require runtime support.
- Each value has a compile-time type.
- The type specifies:
  - all legal values
  - all legal operations
  - the meaning of all legal operations
- Type conversions (casts) may be needed.
- The **simple** types need to be defined: aka. integers (i.e., we don't need any deep type theory)

Integer type behaviors to be specified

- Legal operations
- Overflow
- Sign-extensions
- Conversion rules
- Literal interpretation (how to write a constant)
- A cast operator may need to be introduced

Part 2

A DESIGN PROPOSAL FOR ADDING TYPES IN P4 V1.1

Core guiding principle

- “Least surprising behavior”
- Behavior modeled after the well-defined parts of C
- Fixed all undefined parts of C
- Prefer to forbid rather than surprise

High-level view of proposal

- Eliminate saturated types
- Types: bool, bit<N>, int<N>, varbit<N>, infint
- infint: infinite precision, compile-time only, for literals
- All behaviors specified
- No runtime exceptions
- All binary operations require both operands of the same type (except shifts)
- Restricted set of explicit casts
  - No ambiguity in how casts work
- Very few implicit casts allowed

Portability and types

- No target can support all possible operations
  - e.g. bit<2312312> x;
- Targets **will** impose restrictions. Examples:
  - Maximum width supported
  - Shift with a very large number
  - Arithmetic only on some widths (e.g., size % 8 = 0)
  - Constraints on operands of *
- It is OK to reject properly typed programs
- However, if a program is accepted, result should conform to the spec

Saturated types

- Can be implemented using black-boxes operating on bit<*> values
  (e.g., saturated_adder)
- Highly unlikely to be portable
- Most operations are probably not needed
- For symmetry we should probably have both signed and unsigned saturated types

bool type

- Not an integer or a bit
- Result of comparisons <= >= == != < >
- Operations: && || and !
- Also, first operand of ?:
- Two constants: **true** and **false**
- No implicit casts from/to bool
- The C program **if (x)** is written as **if (x != 0)**
  - This is the main reason
    bool should be different from bit<1>

varbit<N>

- Variable-length bitstring with up to N bits
- N must evaluate to a compile-time constant
- Has a dynamic length, which is <= N
- Initially length is 0
- Only two operations:
  - extract (parser), emit (deparser)
  - cannot be used in the match-action pipeline
- Length set by extract
- Length cannot be changed once set
- There are no casts to/from varbit<N>

bit<N> type

- Unsigned bitstring with exactly N bits
- N must evaluate to a compile-time constant
- Most binary operations require both operands of the same exact type (same sign and same width)
  - shift is the only exception

bit<N> operations

Operation	Description
unary + -	Result is bit<N> even for negation (C-style).
binary + -	Both operands same type; result has same type; overflow and underflow wrap around (C-style)
~	Result is bit<N>; complement all bits
& | ^	Both operands same type
!= == < > <= >=	Both operands same type; unsigned comparisons; result is bool
<< >>	Right operand must be unsigned, of any width; result has the same type as the left operand; logical shift; shift by an amount greater than N produces 0.
*	Both operands same type; result has same type. (Must extend prior to * to avoid overflow.)
?:	First operand is bool, other two operands must have same type.

Note: no division – to avoid runtime exceptions.

int<N> type

- Bitstring with N bits interpreted as a signed number
- Represented using two's complement
- Most binary operations require both operands of the same exact type (same sign and same width)
  - shift is the only exception

int<N> operations

Operation	Description
unary + -	Result has is int<N>; overflow (negation) wraps around
binary + -	Both operands same type; result has same type; overflow and underflow wrap around
~	Result is int<N>: complement all bits.
& | ^	Both operands same type
!= == < > <= >=	Both operands same type; signed comparisons; result is bool
<< >>	Right operand must be an unsigned value. Result has the same type as the left operand; arithmetic shift; shift by an amount greater than N is well-defined.
*	Both operands same type; result has same type
?:	First operand is bool, other two operands must have same type.

Shifts

- Tricky for several reasons
  - different for signed and unsigned numbers
  - shift with negative amount is strange => illegal
  - shift work is exponential in size of RHS operand
- Targets may reject shift operations with very large amounts, e.g.:
  - bit<8> x; bit<16> y; y = y << x;
  - Target may require
    width of shift amount <= ceil(log2(width(lhs)))
  - Targets may even reject shifts by variable amounts

infint type

- Signed compile-time constant value, infinite precision
- A literal with no type specification has an infint type
  - e.g. **7**, bit<**5**>, stack[**9**], a << **3**
- infint type is only used for compile-time constant values
  - No runtime value can have an infint type
- All infint operations are arbitrary-precision; they are all performed at compile-time
- No infint bit-level manipulations (~ | & ^): must specify width
- Cannot have operations mixing infint with any other type
  - Explicit casts may be used
  - Or the compiler inserts implicit casts from infint

infint operations

Operation	Description
+ - (unary and binary)	Result is infint; no truncation
& | ~ ^	**Illegal**: cannot infer width
!= == < > <= >=	Signed comparisons; result is infint
<< >>	Right hand-side must be positive. a << b is a * 2^b a >> b is floor(a / 2^b)
*	Both operands same type; result has same type
?:	First operand is bool, other two operands must be infint

Explicit casts

- Cast syntax C-like: (type)
- Permitted casts:
  - bit<1> <-> bool
  - int<N> -> bit<N> same representation
  - int<N> -> int<M> (truncate/sign-extend)
  - bit<N> -> int<N> same representation
  - bit<N> -> bit<M> (truncate/0-extend at MSB)
  - infint -> bit<N> (overflow should give warning)
  - infint -> int<N> (overflow should give warning)
- Truncating cast is C-like:
  truncate to keep lsb, drop most significant bits
- Casts are not free; explicit cast makes cost apparent

Implicit casts

- Allow implicit casts only in 2 circumstances:
  - To convert an infint value to another type
  - In assignment statements, when RHS has a different type from LHS
- Compiler converts implicit casts to explicit
- Rules for conversion must be very clear
- A binary operation of an infint and another type will implicitly cast infint to the other type

Typed Integer Literals

- Simple integer literals have type infint
- Must be able to declare type in literal
- Proposed syntax:
  - *width* **w** *value* – unsigned numbers
  - *width* **s** *value* – signed number
  - *value* can be negative for signed number

EXAMPLES

Literal examples

Literal	Interpretation
10	type is infint, value is 10
-10	type is infint, value is -10
8w10	type is bit<8>, value is 10
8w-10	Illegal: negative unsigned number
8s10	type is int<8>, value is 10
8s-10	type is int<8>, value is -10
2s3	type is int<2>, value is -1, overflow warning
1w10	type is bit<1>, value is 0, overflow warning
1s10	type int<1>, value is 0, overflow warning

Illegal operations examples

bit<8> x; bit<16> y; int<8> z;

Operation	Why is it illegal	Alternatives
x+y	Different widths	((bit<16>)x) + y or x + ((bit<8>)y)
x+z	Different signs	((int<8>)x) + y or x + (bit<8>)z
(int<8>)y	Cast cannot change both size and width	(int<8>)(bit<8>)y
y + z	Different widths and signs. Note that the 4 alternatives produce all different results.	(int<8>)(bit<8>)y + z or y + (bit<16>)(bit<8>)z or (bit<8>)y + (bit<8>)z or (int<16>)y + (int<16>)z
x << z	RHS of shift cannot be signed	x << (bit<8>)z
x ? y : 0	First operand must be bool.	(x != 0) ? y : 0 or ((bit<1>)x) ? y : 0
x == z	Different signs	x == (bit<8>)z or (int<8>)x == z
1 << x	Width cannot be inferred	((bit<32>)1) << x or 32w1 << x
~1	Width cannot be inferred	~32w1
5 & -3	Width cannot be inferred	32w5 & -3

Implicit casts examples

bit<8> x; bit<16> y; int<8> z;

Operation	Interpretation
x+1	x + (bit<8>)1
z < 0	z < (int<8>)0
(x == 0) ? y : 0	(x == (bit<8>)0) ? y : (bit<16>)0
x << 13	0; overflow warning
x | 0xFFF	x | (bit<8>)0xFFF; overflow warning