Mechanizing the P4 Language Specification with P4-SpecTec

Jaehyun Lee, Seokhun Jeong, Sukyoung Ryu
@ P4 Developer Days

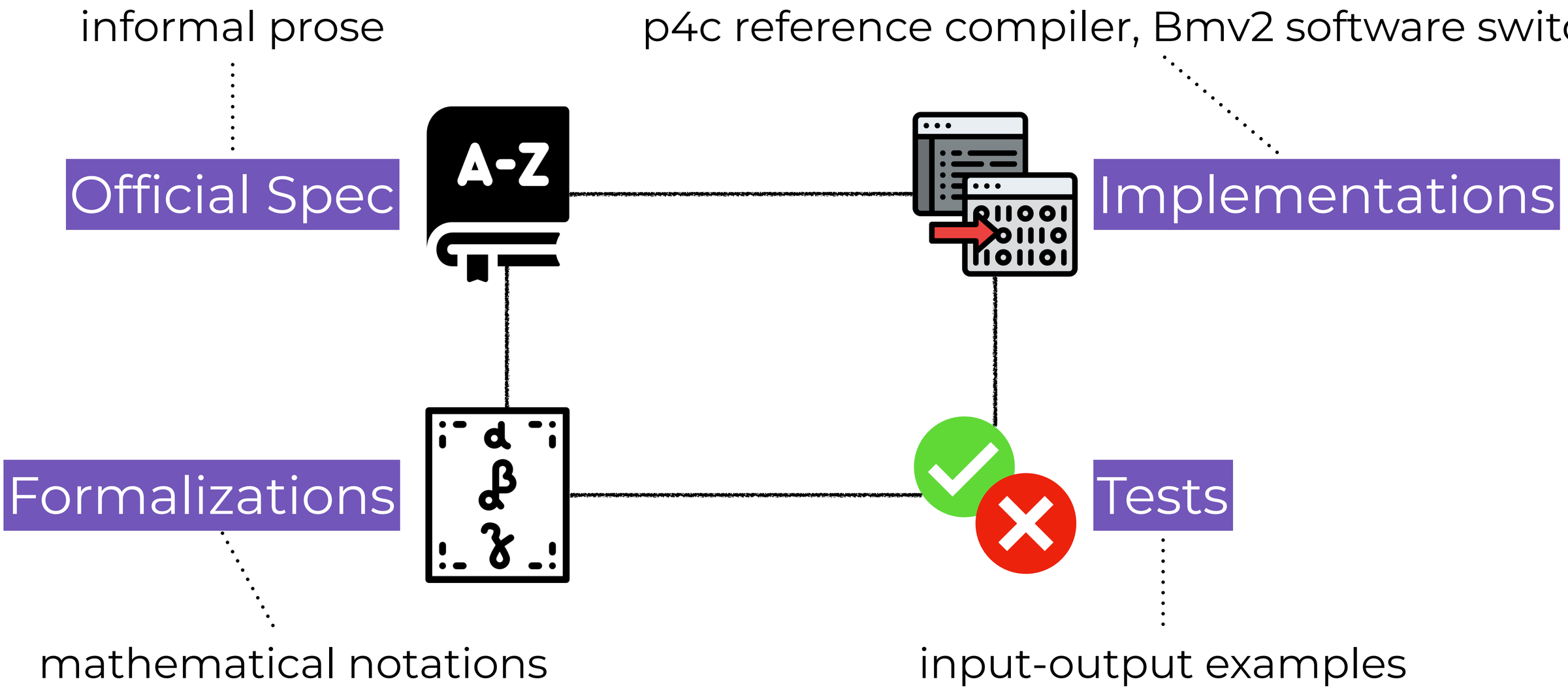
Mechanizing the P4 Language Specification with P4-SpecTec

What is mechanization, and **why** do we need it?

How do we mechanize the P4 language specification?

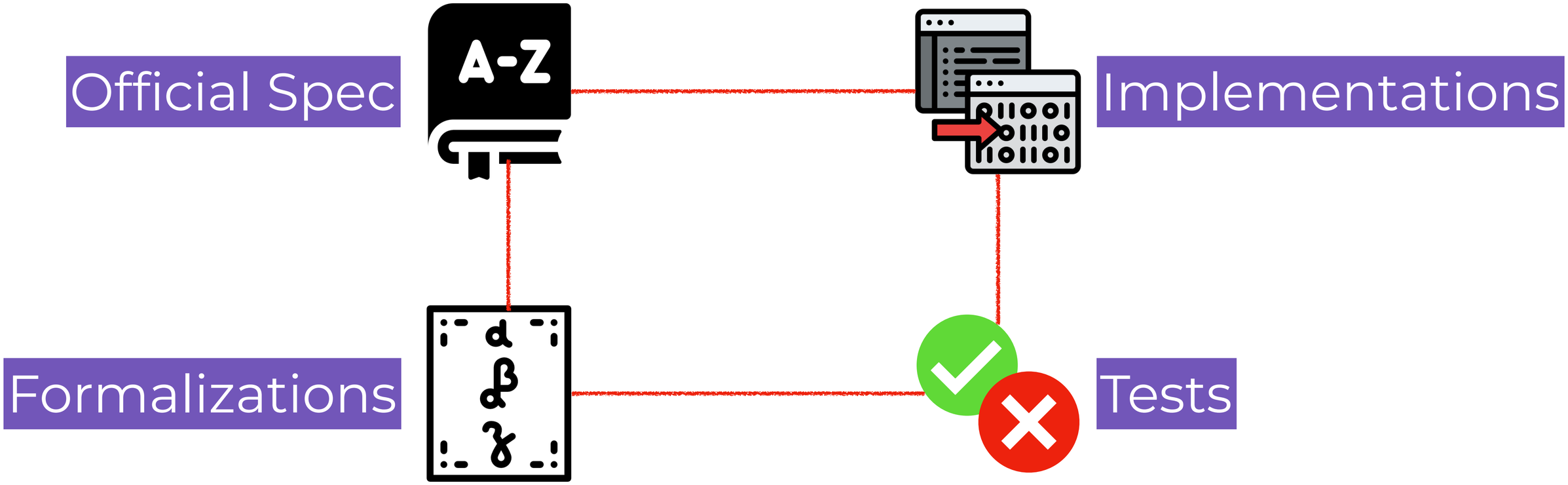
What are the benefits?

P4 from a PL Point of View: The Four Representations



Are the four representations truly in sync?

Are they **consistent** with one another? in principle, YES; in practice, NO



Maintained by different parties, and at different paces

Part of the Reason: P4 is Evolving

Official Spec Introduced generic types @ May 2021

A.3. Summary of changes made in version 1.2.2, released May 17, 2021

- Added support for accessing tuple fields (Section 8.12).
- Added support for generic structures (Section 7.2.11).

Implementations Generics + Type Inference? Patched @ March 2025

Compiler Bug: `Could not find type of <Type_Header> ...` on specialized generic struct type #4835

Closed Bug #5133

Formalizations Remains pre-2021, the recent spec is based 2024

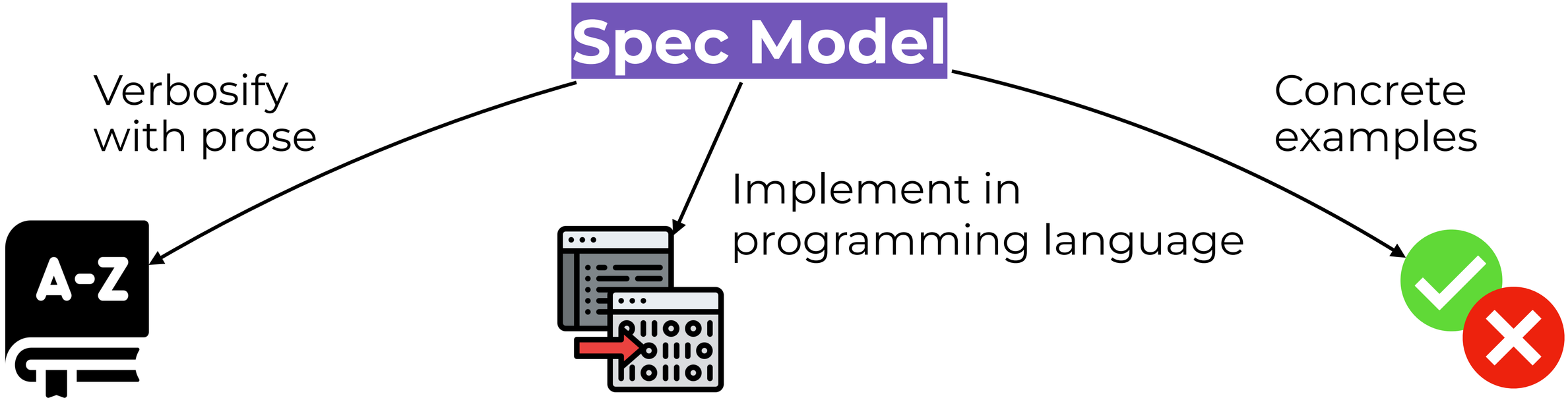
Can we, at the same time,
evolve a programming language,
and maintain the consistency between representations?

Single Source of Truth Underlying the Representations

The **Specification Model**,

an unambiguous and complete definition of the syntax and semantics.

e.g., a complete formalization

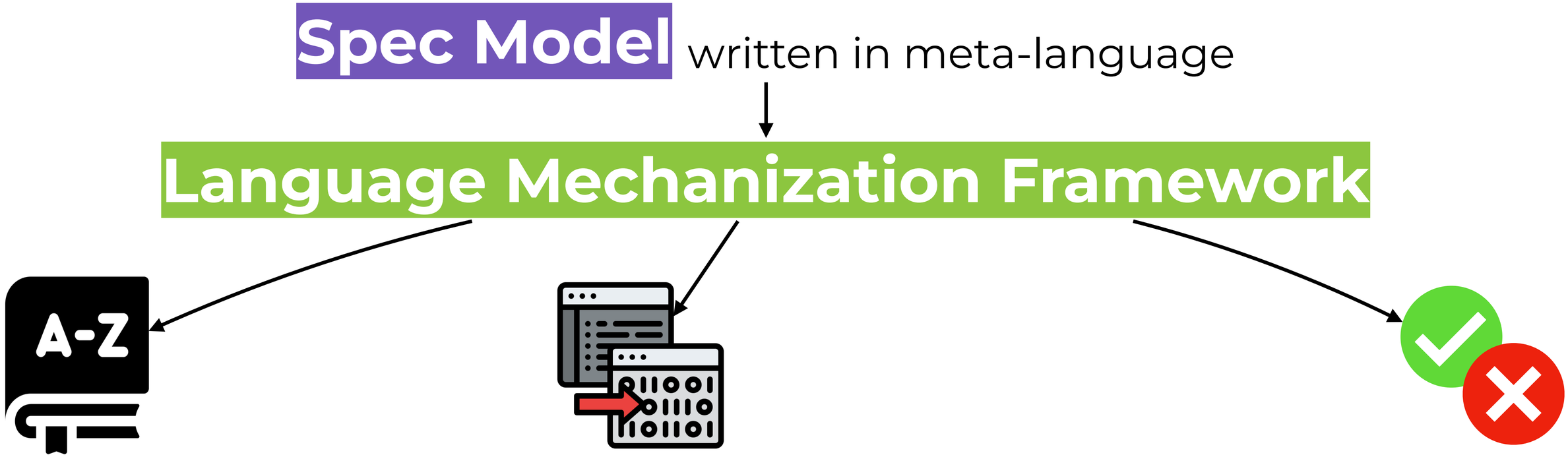


Language Mechanization Frameworks for Consistency

Language mechanization frameworks

provide a DSL (called meta-language) for defining the spec model

and automate generation of various backends from it



```
rule Stmt_ok/conditionalStatement-without-else:
  TC |- IF `( expression_c ) statement_then: TC conditionalStatementIR
  -- Expr_ok: TC |- expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  -- Stmt_ok: TC |- statement_then : TC_t statementIR_then
 -- if conditionalStatementIR = IF `( expressionIR_c ) statementIR_then
```

Mechanized P4 spec for conditional statement

12.6. Conditional statement

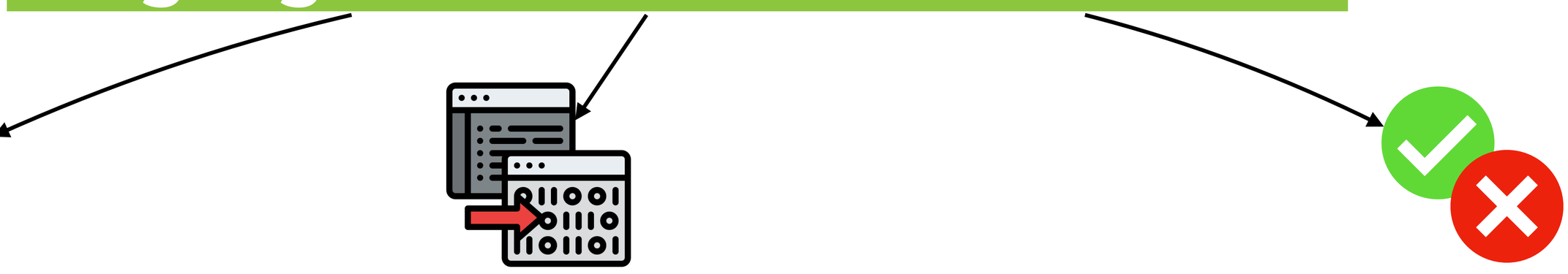
The conditional statement uses standard syntax and semantics familiar from many programming languages.

However, the condition expression in P4 is required to be a Boolean (and not an integer).

Current P4 prose spec for conditional statement

```
rule Stmt_ok/conditionalStatement-without-else:
  TC |- IF `( expression_c ) statement_then: TC conditionalStatementIR
  -- Expr_ok: TC |- expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  -- Stmt_ok: TC |- statement_then : TC_t statementIR_then
 -- if conditionalStatementIR = IF `( expressionIR_c ) statementIR_then
```

Spec Model written in meta-language



Mechanizing the P4 Language Specification with P4-SpecTec

What is mechanization, and **why** do we need it?

How do we mechanize the P4 language specification?

What are the benefits?

Other Languages Share the Same Problem

JavaScript and **WebAssembly (Wasm)** are evolving, but also in the presence of multiple representations: the specification, implementations, tests, ...

Mechanization Integrated into Real-World Languages

ESMeta

Sukyoung Ryu @sukyoungryu · Nov 30, 2022

ESMeta is integrated into the CI of both ECMA-262 and Test262. Now, each ECMA-262 PR will execute the ESMeta type checker, and any new or changed tests in a Test262 PR will be executed using the ESMeta interpreter!

Wasm-SpecTec

SpecTec has been adopted

Published on March 27, 2025 by Andreas Rossberg.

Story 1. ESMeta for the ECMAScript Standard

14.6.2 Runtime Semantics: Evaluation

IfStatement : `if (` *Expression* `)` *Statement* `else` *Statement*

1. Let *exprRef* be ? Evaluation of *Expression*.
2. Let *exprValue* be ToBoolean(? GetValue(*exprRef*)).
3. If *exprValue* is **true**, then
    a. Let *stmtCompletion* be Completion(Evaluation of the first *Statement*).
4. Else,
    a. Let *stmtCompletion* be Completion(Evaluation of the second *Statement*).
5. Return ? UpdateEmpty(*stmtCompletion*, **undefined**).

Prose pseudocode for the dynamic semantics of JavaScript if statement

... prose, but more or less an algorithm

The ESMeta Spec Model

14.6.2 Runtime Semantics: Evaluation

IfStatement : **`if (`** *Expression* **`)`** *Statement* **`else`** *Statement*

1. Let *exprRef* be ? Evaluation of *Expression*.

Compile the prose into intermediate representation (IR_{ES})

Spec Model

Syntax and

semantics in IR_{ES}

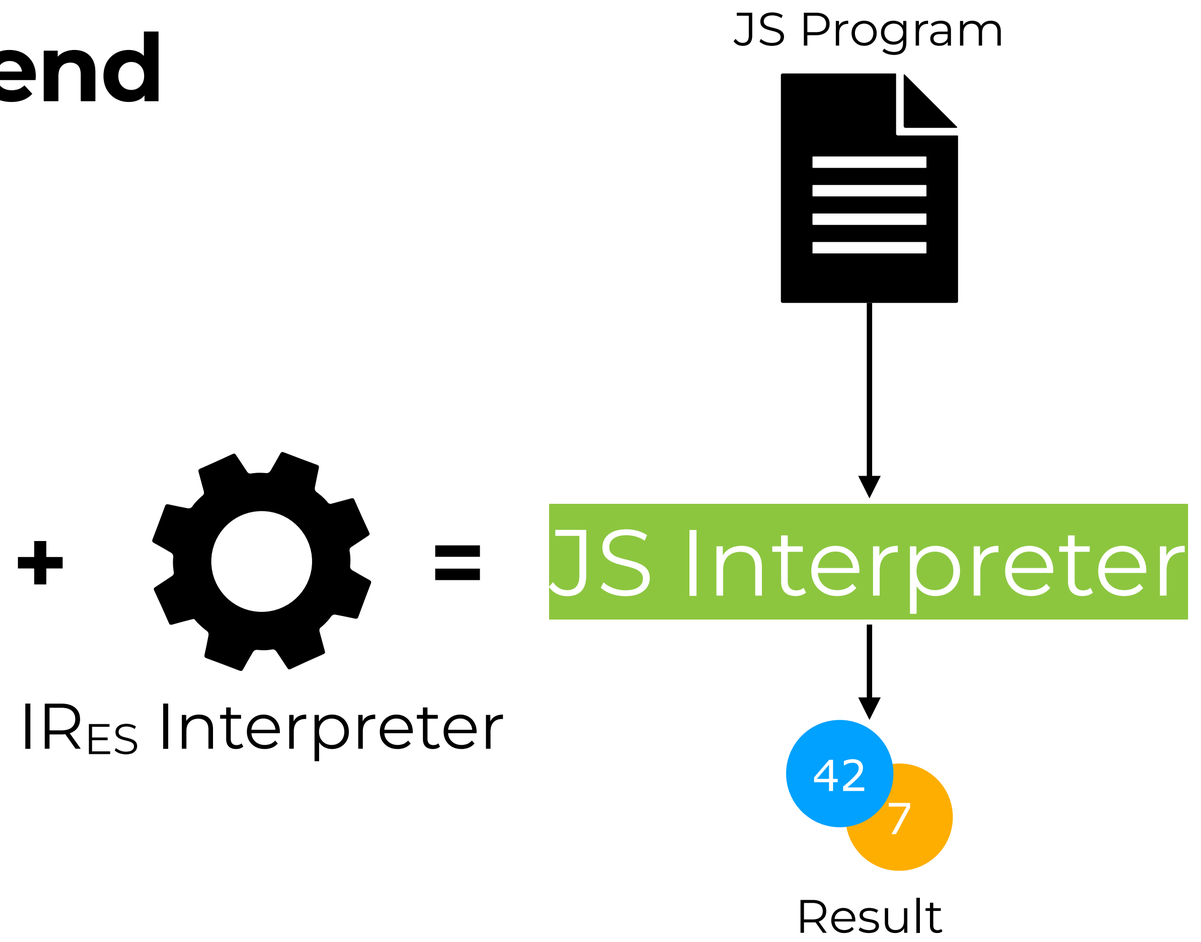
```
def <SYNTAX>:IfStatement[0,0].Evaluation(
  this: Ast[IfStatement[0]],
): Unknown = {
  sdo-call %0 = this[0]->Evaluation()
  assert (? %0: Completion)
  if (? %0: Abrupt) return %0 else %0 = %0.Value
  let exprRef = %0
  ...
}
```

ESMeta Interpreter Backend

Spec Model

IR_{ES} program

```
def <SYNTAX>:IfStatement[0,0].Evaluation(
  this: Ast[IfStatement[0]],
): Unknown = {
  sdo-call %0 = this[0]->Evaluation()
  ...
}
```



JavaScript semantics is an IR_{ES} program

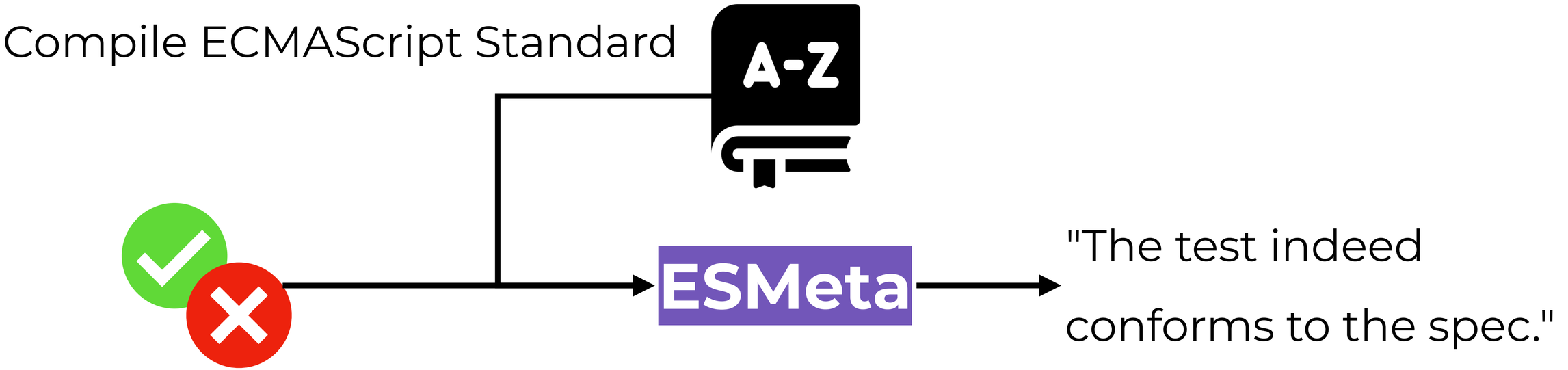
when executed by an IR_{ES} interpreter, becomes a JavaScript interpreter

Sukyoung Ryu and Jihyeok Park. 2024. JavaScript Language Design and Implementation in Tandem. Commun. ACM 67, 5 (May 2024), 86–95. https://doi.org/10.1145/3624723

ESMeta, Integrated into the ECMAScript CI

Sukyoung Ryu @sukyoungryu · Nov 30, 2022

ESMeta is integrated into the CI of both ECMA-262 and Test262. Now, each ECMA-262 PR will execute the ESMeta type checker, and any new or changed tests in a Test262 PR will be executed using the ESMeta interpreter!



Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. Proc. ACM Program. Lang. 8, PLDI, Article 210 (June 2024), 26 pages. https://doi.org/10.1145/3656440

Story 2. SpecTec for the WebAssembly Standard

Prose Spec step-by-step pseudocode-style explanation

if $blocktype$ $instr_1^*$ **else** $instr_2^*$ **end**

1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. If c is non-zero, then:
   a. Execute the block instruction **block** $blocktype$ $instr_1^*$ **end**.
4. Else:
   a. Execute the block instruction **block** $blocktype$ $instr_2^*$ **end**.

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_1^*\ \text{end} \quad (\text{if } c \neq 0)$$

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_2^*\ \text{end} \quad (\text{if } c = 0)$$

Formal Spec dynamic semantics as formal reduction rules

The Wasm-SpecTec Spec Model

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_1^*\ \text{end} \quad (\text{if } c \neq 0)$$

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_2^*\ \text{end} \quad (\text{if } c = 0)$$

Resemble in ASCII characters

Spec Model Reduction rules described with SpecTec metalang

```
rule Step_pure/if-true:
  (CONST I32 c) (IF bt instr_1* ELSE instr_2*) ~> (BLOCK bt instr_1*)
  -- if c =/= 0

rule Step_pure/if-false:
  (CONST I32 c) (IF bt instr_1* ELSE instr_2*) ~> (BLOCK bt instr_2*)
  -- if c = 0
```

Wasm-SpecTec Prose Backend

Spec Model

```
LHS ~> RHS
-- Premise(s)
```

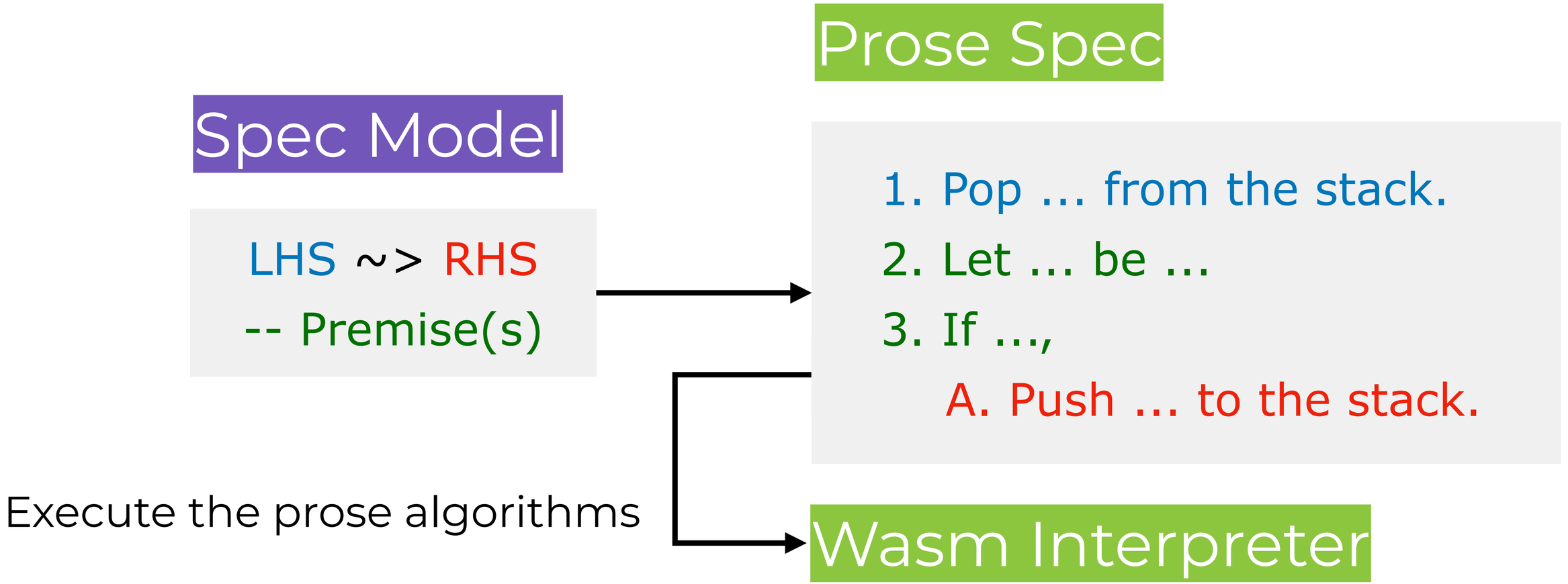
Prose Spec

1. Pop ... from the stack.
2. Let ... be ...
3. If ...,
    A. Push ... to the stack.

"What" reduction takes place in the stack machine

"How" to carry out the reduction, computationally

Wasm-SpecTec Interpreter Backend



Wasm-SpecTec Spec Document Backend

Spec Model

```
LHS ~> RHS
-- Premise(s)
```

"Splice" the reduction rule

and the generated prose

Prose Spec

1. Pop ... from the stack.
2. Let ... be ...
3. If ...,
    A. Push ... to the stack.

Skeleton Document

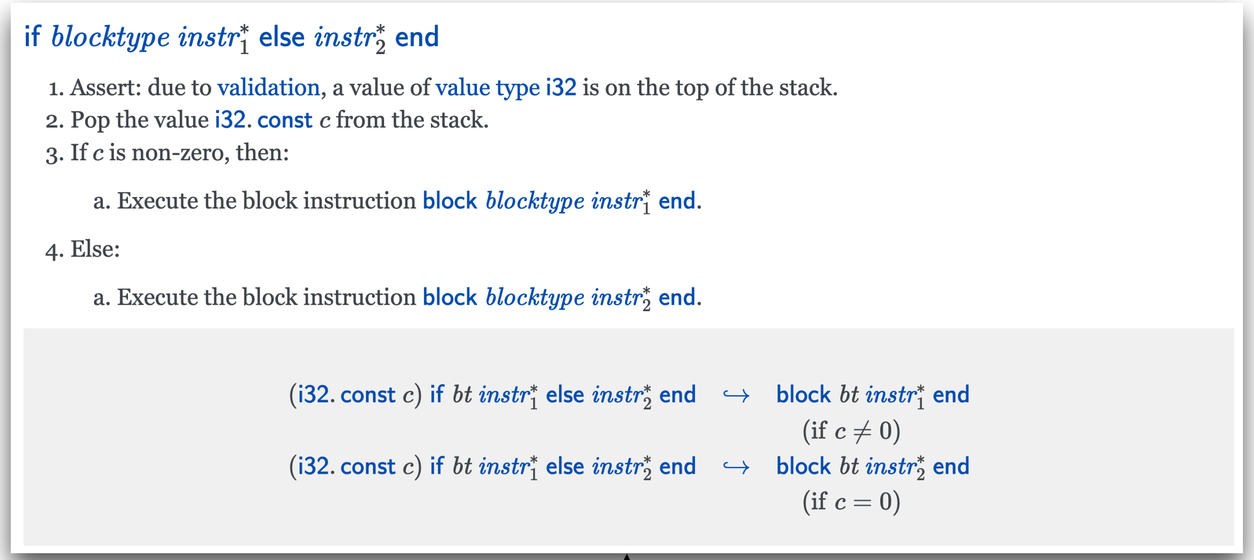
```
.. _exec-if:

$${rule-prose: Step_pure/if}

$${rule: {Step_pure/if-*}}
```

Wasm-SpecTec Spec Document Backend

Official Spec Document



if $blocktype$ $instr_1^*$ **else** $instr_2^*$ **end**

1. Assert: due to validation, a value of value type i32 is on the top of the stack.
2. Pop the value i32.const c from the stack.
3. If c is non-zero, then:
   a. Execute the block instruction **block** $blocktype$ $instr_1^*$ **end**.
4. Else:
   a. Execute the block instruction **block** $blocktype$ $instr_2^*$ **end**.

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_1^*\ \text{end} \quad (\text{if } c \neq 0)$$

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_2^*\ \text{end} \quad (\text{if } c = 0)$$

Skeleton Document

```
.. _exec-if:

$${rule-prose: Step_pure/if}

$${rule: {Step_pure/if-*}}
```

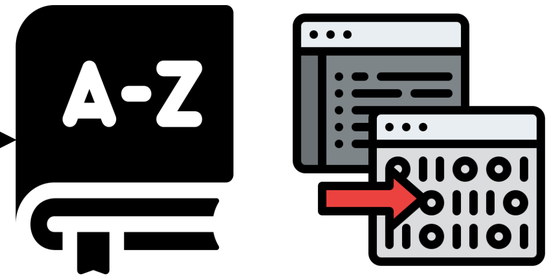
Render the spliced document

Wasm-SpecTec, Integrated into the Wasm Spec

SpecTec has been adopted

Published on March 27, 2025 by Andreas Rossberg.

"Two weeks ago, the Wasm Community Group voted to adopt SpecTec for authoring future editions of the Wasm spec."

Spec Model → Wasm-SpecTec → 

Edit the spec model,

using Wasm-SpecTec metalang

Underlying Recipe: An Authoritative Spec Model

THE standard

14.6.2 Runtime Semantics: Evaluation

IfStatement : **if** (*Expression*) *Statement* **else** *Statement*

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_1^*\ \text{end} \quad (\text{if } c \neq 0)$$

$$(\text{i32.const } c)\ \text{if } bt\ instr_1^*\ \text{else } instr_2^*\ \text{end} \hookrightarrow \text{block } bt\ instr_2^*\ \text{end} \quad (\text{if } c = 0)$$

drives the spec model design

IR$_{ES}$ Program

Formal Reduction Rules

which are also executable, giving assurance

JS Interpreter

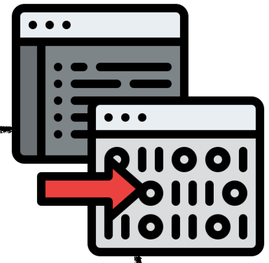
Wasm Interpreter

P4 Lacks an Authoritative Spec Model

Informal, not systematic

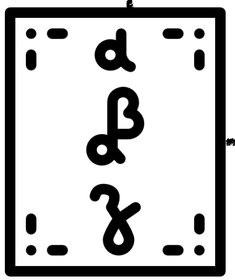
Nano-pass compiler

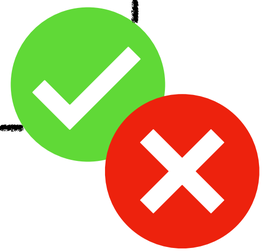
Official Spec



Implementations

Formalism





Tests

Incomplete, not up-to-date

Incomplete

P4 Spec Model Desideratas

The **Specification Model**,

an unambiguous and complete definition of the syntax and semantics.

(1) executable s.t. designers can cross-check with the tests

(2) help language designers work out every detail

(3) but also in a way that hides implementation details

P4 Spec Model in Terms of Inference Rules

Define the P4 spec model in terms of **inference rules**,

i.e., derivations of conclusions from premises

```
relation Stmt_ok:
  typingContext ⊢ statement : typingContext statementIR
```

A relation is a collection of inference rules:

Under **typingContext**,

we type a P4 **statement**,

to get an updated **typingContext** and a typed P4 **statementIR**

Mechanized P4 Syntax in P4-SpecTec

Define the complete formal syntax of P4 in P4-SpecTec

```
syntax baseType = BOOL | ERROR | ...
syntax type = baseType | namedType | ...
syntax expression = ...

syntax conditionalStatement =
  | IF `( expression ) statement
  | IF `( expression ) statement ELSE statement

syntax statement =
  | ...
  | conditionalStatement
```

P4 syntax, in P4-SpecTec

```
conditionalStatement
    : IF "(" expression ")" statement
    | IF "(" expression ")" statement ELSE statement
    ;
```

P4 syntax, from the official spec

An inference rule for conditional statement

```
relation Stmt_ok:
  typingContext
  ⊢ statement
   : typingContext statementIR
```

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  ----
  -- Stmt_ok: TC ⊢ statement_then : TC_then statementIR_then
  -- Stmt_ok: TC ⊢ statement_else : TC_else statementIR_else
  ----
  -- if conditionalStatementIR
    = IF `( expressionIR_c ) statementIR_then ELSE statementIR_else
```

1. Under context TC, we type

   IF `(expression_c)

   statement_then

   ELSE

   statement_else

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  ----
  -- Stmt_ok: TC ⊢ statement_then : TC_then statementIR_then
  -- Stmt_ok: TC ⊢ statement_else : TC_else statementIR_else
  ----
  -- if conditionalStatementIR
     = IF `( expressionIR_c ) statementIR_then ELSE statementIR_else
```

2. Check that the premises hold (in order)

Type the expression,

to get **expressionIR_c**

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  ----
  -- Stmt_ok: TC ⊢ statement_then : TC_then statementIR_then
  -- Stmt_ok: TC ⊢ statement_else : TC_else statementIR_else
  ----
  -- if conditionalStatementIR
    = IF `( expressionIR_c ) statementIR_then ELSE statementIR_else
```

2. Check that the
premises hold (in order)

Get its type by $typeof,
and check that it is BOOL

Check both branches,
getting TC_then, TC_else,
and typed branches

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  ----
  -- Stmt_ok: TC ⊢ statement_then : TC_then statementIR_then
  -- Stmt_ok: TC ⊢ statement_else : TC_else statementIR_else
  ----
  -- if conditionalStatementIR
     = IF `( expressionIR_c ) statementIR_then ELSE statementIR_else
```

2. Check that the premises hold (in order)

Bind the typed statement

IF `(expressionIR_c)
statementIR_then
ELSE
statementIR_else

to conditionalStatementIR

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  ----
  -- Stmt_ok: TC ⊢ statement_then : TC_then statementIR_then
  -- Stmt_ok: TC ⊢ statement_else : TC_else statementIR_else
  ----
  -- if conditionalStatementIR
    = IF `( expressionIR_c ) statementIR_then ELSE statementIR_else
```

3. We keep the context as-is,

and result in

a typed P4 statement,

conditionalStatementIR

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  ----
  -- Stmt_ok: TC ⊢ statement_then : TC_then statementIR_then
  -- Stmt_ok: TC ⊢ statement_else : TC_else statementIR_else
  ----
  -- if conditionalStatementIR
    = IF `( expressionIR_c ) statementIR_then ELSE statementIR_else
```

Inference rules in general,

do *not*

guarantee executability

But by imposing

constraints on the rules,

we can execute them

Implemented an

interpreter that runs the

inference rules

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  ----
  -- Stmt_ok: TC ⊢ statement_then : TC_then statementIR_then
  -- Stmt_ok: TC ⊢ statement_else : TC_else statementIR_else
  ----
  -- if conditionalStatementIR
    = IF `( expressionIR_c ) statementIR_then ELSE statementIR_else
```

https://www.youtube.com/watch?v=a97P7VvCg6k&ab_channel=PLSSWorkshop for more detailed discussion of the constraints

P4 Spec Model Desideratas: Inference Rules

The **Specification Model**,

an unambiguous and complete definition of the syntax and semantics.

(1) executable s.t. designers can cross-check with the tests

by implementing an interpreter for the inference rules

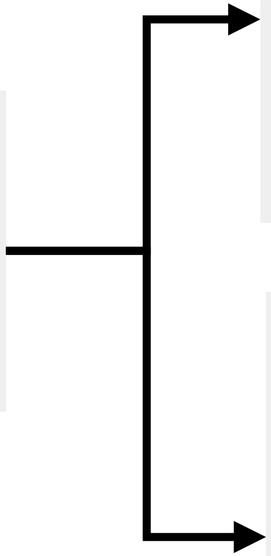
(2) help language designers work out every detail

(3) but also in a way that hides implementation details

What if we *don't* have an else branch?

We write *another* inference rule for the case without an else branch

```
syntax conditionalStatement =
  | IF `( expression ) statement
  | IF `( expression ) statement
    ELSE statement
```



```
rule Stmt_ok/conditionalStatement-without-else:
  TC ⊢ IF `( expression_c ) statement_then
       : TC conditionalStatementIR
  ----
  -- ...
```

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then
         ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- ...
```

Inference Rules Help Case-Analysis

Inference rule itself does *not* have nested control flow

Spec designer should work out every detail, including corner cases

```
TC ⊢ Input : Output
-- if cond:
    -- ... foo ...
-- else:
    -- ... bar ...
```

```
TC ⊢ Input : Output
-- if cond:
-- ... foo ...
```

```
TC ⊢ Input : Output
-- if not cond:
-- ... bar ...
```

Inference Rules Help Case-Analysis

Inference rule itself does *not* have nested control-flow

Spec designer should work out every detail, including corner cases

```
argument
    : expression  /* positional argument */
    | name "=" expression  /* named argument */
    | "_"
    | name "=" "_"
    ;
```

Things can get complex...!

Inference Rules Hide Implementation Details

Inference rules define what is valid; anything left unspecified is invalid

... no need to spell out all error conditions explicitly

```
-- Expr_ok: TC ⊢ expression_c : expressionIR_c
-- if BOOL = $typeof(expressionIR_c)
```

However, the condition expression in P4 is required to be a Boolean (and not an integer).

No memory management, e.g., malloc/free

P4 Spec Model Desideratas: Inference Rules

The **Specification Model**,

an unambiguous and complete definition of the syntax and semantics.

(1) executable s.t. designers can cross-check with the tests

by implementing an interpreter for the inference rules

(2) help language designers work out every detail

b/c inference rules help think in terms of case-analysis

(3) but also in a way that hides implementation details

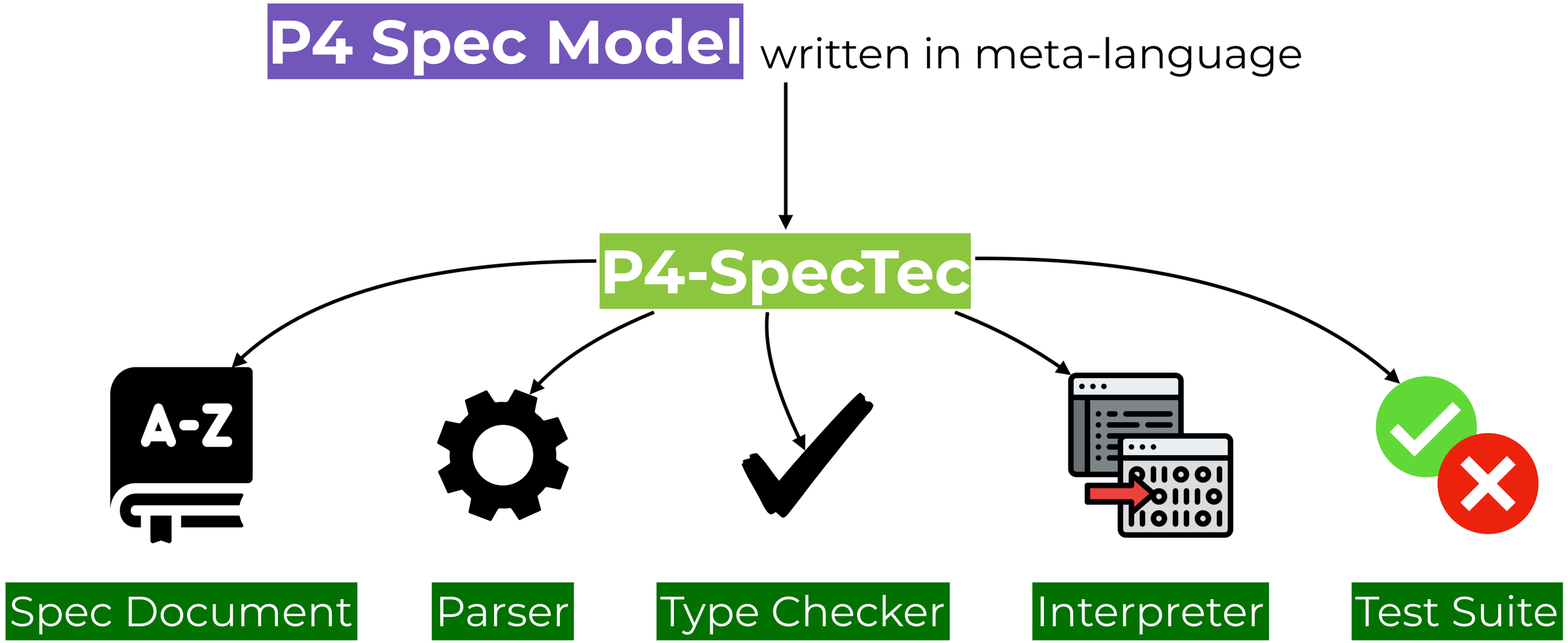
b/c inference rules abstract over the implementation details

Mechanizing the P4 Language Specification with P4-SpecTec

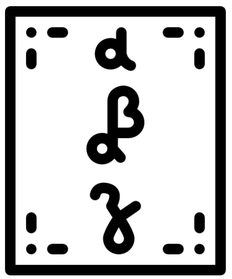
What is mechanization, and **why** do we need it?

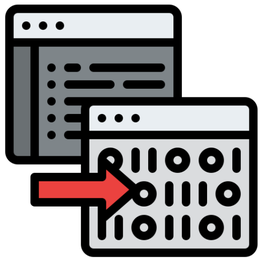
How do mechanize the P4 language specification?

What are the benefits?


P4 Spec Model
written in meta-language
P4-SpecTec
A-Z
Spec Document
Parser
Type Checker
Interpreter
Test Suite

Mechanizing the P4 Spec Model





Read the official spec

and also prior formalization efforts

Examine P4C behavior

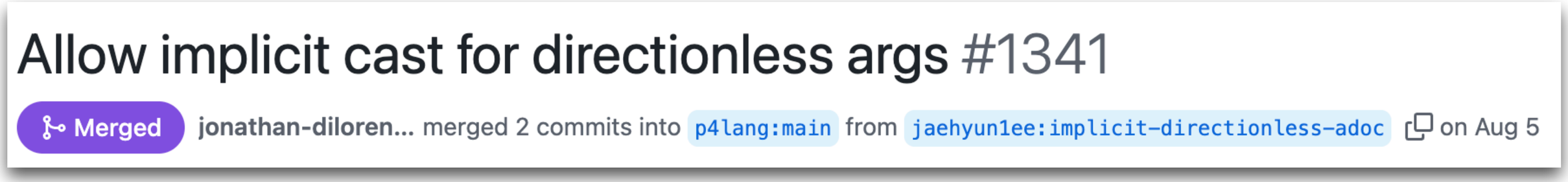
and its test programs

P4 Spec Model written in meta-language

Syntax | Type System | Instantiation | Dynamic Semantics

Identifying Inconsistencies and Ambiguities

6 PRs on the P4 spec repo, all merged after discussion



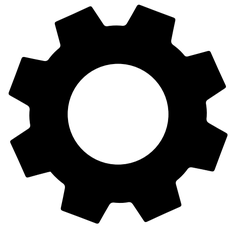
Official spec disallowed implicit cast on directionless arguments

But p4c allowed such behavior

P4-SpecTec Type Checker Backend

P4 Spec Model

```
relation Stmt_ok:
  typingContext ⊢ statement
                  : typingContext statementIR
```

\+  = **P4 Type Checker**

Metalang Interpreter

Validated with the p4c reference compiler test suite:

- Excluded 176 tests (ambiguous semantics, target-specific features, …)
- Positive tests: 1,166 / 1,170 (**99.7%**)
- Negative tests: 254 / 258 (**98.4%**)

Are the Negative Tests Enough?

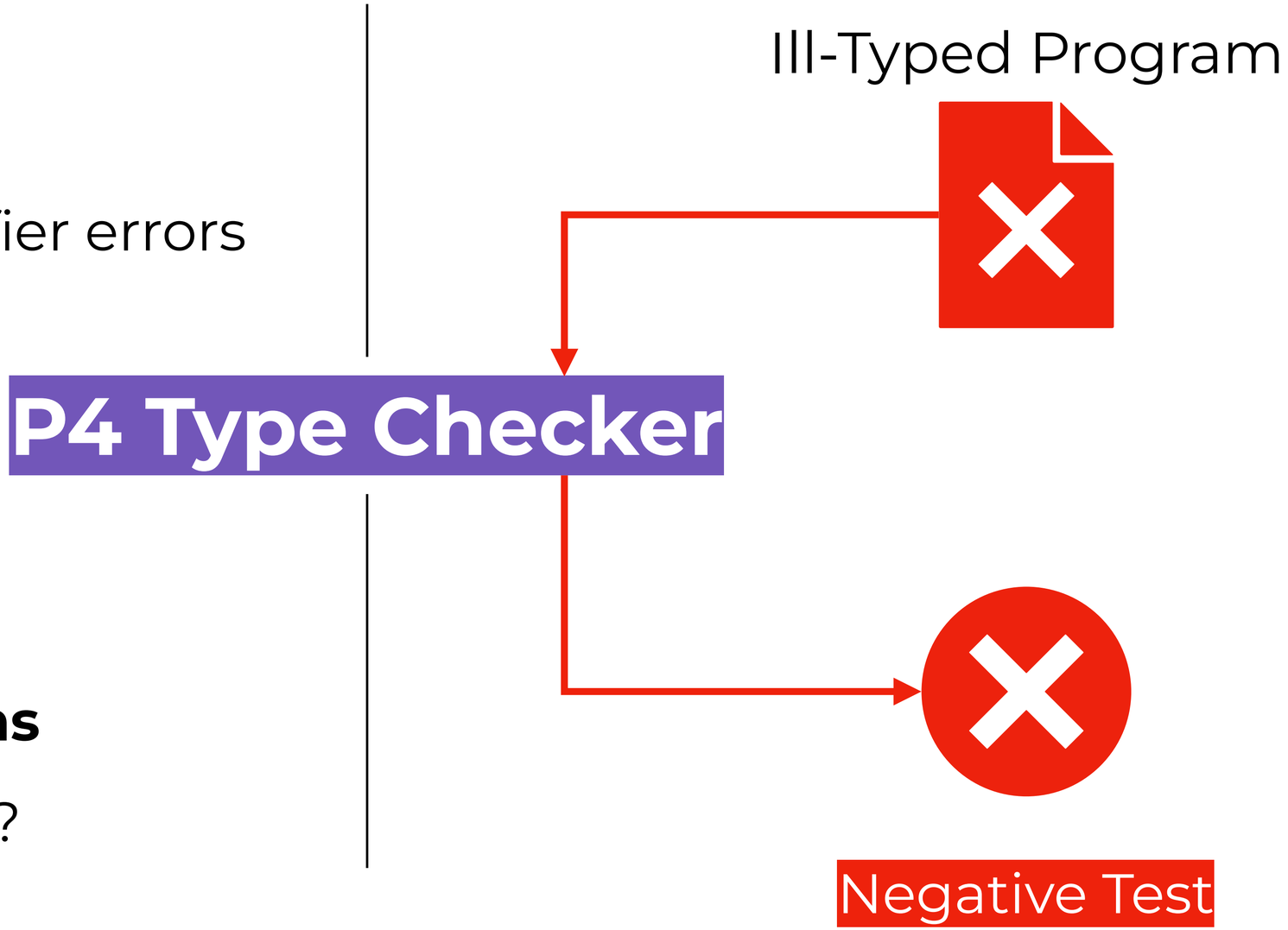
Don't want 100 tests

with trivial free identifier errors

Does it exercise

all ill-typed conditions

in the P4 type system?



Are the Negative Tests Enough?

12.6. Conditional statement

The conditional statement uses standard syntax and semantics familiar from many programming languages.

However, the condition expression in P4 is required to be a Boolean (and not an integer).

Does the p4c negative test suite trigger this violation?

What are the ill-typed conditions in the P4 type system?

... manually inspecting the spec document does not scale

P4 Spec Model to the Rescue

A complete, unambiguous spec model;

mechanized and executable in P4-SpecTec;

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then
         ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  -- ...
```

Ill-Typed Conditions are Violations of Premises

Inference rules omit the explicit error states

When a conditional premise is violated, becomes an ill-typed condition

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then
         ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  -- ...
```

when if condition

is *not* a boolean

collected 939

ill-typed conditions

over the spec model

Negative Test Generation by Mutation

Ill-typed programs are *one-off* from well-typed ones

Mutate well-typed programs to yield negative tests

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then
         ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  -- ...
```

Satisfied

```
void f() {
  bool b;
  if (b); else;
}
```

A promising candidate for triggering premise violation

Negative Test Generation by Mutation

Mutate the candidate well-typed program,

in hope of triggering the ill-typed condition

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then
         ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- Expr_ok: TC ⊢ expression_c : expressionIR_c
  -- if BOOL = $typeof(expressionIR_c)
  -- ...
```

Violated

```
void f() {
  bool b;
  if (b); else;
}
```

Random Mutation

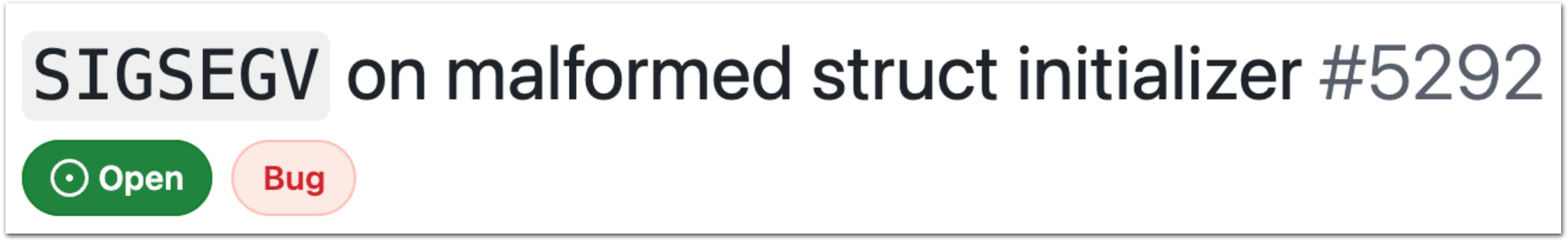
```
void f() {
  bit<32> b;
  if (b); else;
}
```

P4-SpecTec Negative Test Backend

Out of total 939 ill-typed conditions collected over the spec model,

- p4c test suite: 247 / 939 (26.30%)
- Generated by P4-SpecTec: **557 / 939 (59.32%)**

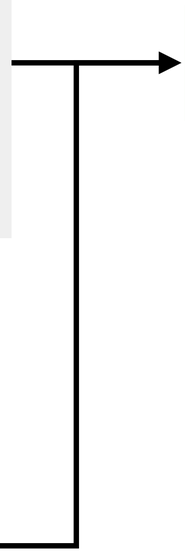
Discovered **23 bugs/discrepancies** in the p4c frontend



(WIP) P4-SpecTec Document Backend

```
rule Stmt_ok/conditionalStatement-without-else:
  TC ⊢ IF `( expression_c ) statement_then
       : TC conditionalStatementIR
  ----
  -- ...
```

```
rule Stmt_ok/conditionalStatement-with-else:
  TC ⊢ IF `( expression_c ) statement_then
       ELSE statement_else
       : TC conditionalStatementIR
  ----
  -- ...
```



Prose Spec

TC ⊢ conditionalStatement

1. If conditionalStatement matches pattern `if (%) %`, then
   a. Let if `(expression_cond) statement_then be conditionalStatement
   b. Let typedExpressionIR_cond be the result of typing expression_cond under TC
   c. Let expressionIR # `(typeIR ctk) be typedExpressionIR_cond
   d. If typeIR equals bool, then
      i. Let TC_then and statementIR_then be the result of typing statement_then under TC
   2. Result in TC and if `(typedExpressionIR_cond) statementIR_then
2. Else, ...

(WIP) P4-SpecTec Document Backend

Splice and render

Skeleton Asciidoc ← Prose Spec

1. If `conditionalStatement` matches pattern `if (%) %`, then
   a. Let `if ( expression_cond ) statement_then` be `conditionalStatement`
   b. Let `typedExpressionIR_cond` be the result of **typing** `expression_cond` **under** `TC`
   c. ...

Readable, with consistent cross-references

Spec Maintenance Becomes Clear in P4-SpecTec

Allow implicit cast for directionless args #1341

Merged **jonathan-diloren...** merged 2 commits into `p4lang:main` from `jaehyun1ee:implicit-directionless-adoc` on Aug 5

```
4881  4881    The calling convention is copy-in/copy-out (<<sec-calling-convention>>).  For
                generic functions the type arguments
4882  4882    can be explicitly specified in the function call. The compiler only
4883        -   inserts implicit casts for direction `in` arguments to methods or
4884        -   functions as described in <<sec-casts>>. The types for all
      4883  +   inserts implicit casts for direction `in` or directionless arguments to methods,
      4884  +   functions, or constructors as described in <<sec-casts>>. The types for all
4885  4885    other arguments must match the parameter types exactly.
```

Spec Maintenance Becomes Clear in P4-SpecTec

Allow implicit casts for directionless arguments #2

Open **shyukahn** wants to merge 6 commits into `concrete` from `concrete-exclude-impl-cast-directionless`

```
@@ -117,8 +117,9 @@ rule Call_convention_expr_ok/empty-not-action:
117  117    p TC NOACTION |- (`EMPTY typeIR_param _ _) `@ typedExpressionIR_arg

120       -  -- if _ `# `( typeIR_arg ctk_arg ) = typedExpressionIR_arg
121       -  -- Type_alpha: typeIR_param ~~ typeIR_arg
     120  +  -- if typedExpressionIR_arg_cast
     121  +      = $coerce_unary(typedExpressionIR_arg, typeIR_param)
```

Instead of equating types for `EMPTY case with Type_alpha,

try inserting implicit cast with $coerce_unary

Spec Maintenance Becomes Clear in P4-SpecTec

Allow implicit casts for directionless arguments #2

Open **shyukahn** wants to merge 6 commits into `concrete` from `concrete-exclude-impl-cast-directionless`

```
268    268     >>> Running typing test on ../../../../p4c/testdata/p4_16_samples/constructor_cast.p4
269            - Excluding file: ../../../../p4c/testdata/p4_16_samples/constructor_cast.p4
       269     + Typecheck success: ../../../../p4c/testdata/p4_16_samples/constructor_cast.p4

3838           - Running typing: [EXCLUDE] 215/1278 (16.82%) [PASS] 1063/1278 (83.18%) [FAIL] 0/1278 (0.00%)
       3838    + Running typing: [EXCLUDE] 124/1278 (9.70%) [PASS] 1154/1278 (90.30%) [FAIL] 0/1278 (0.00%)
```

Consequences of spec edits become evident

… demonstrating their effects on tests with directionless arguments

… and also without breaking other tests

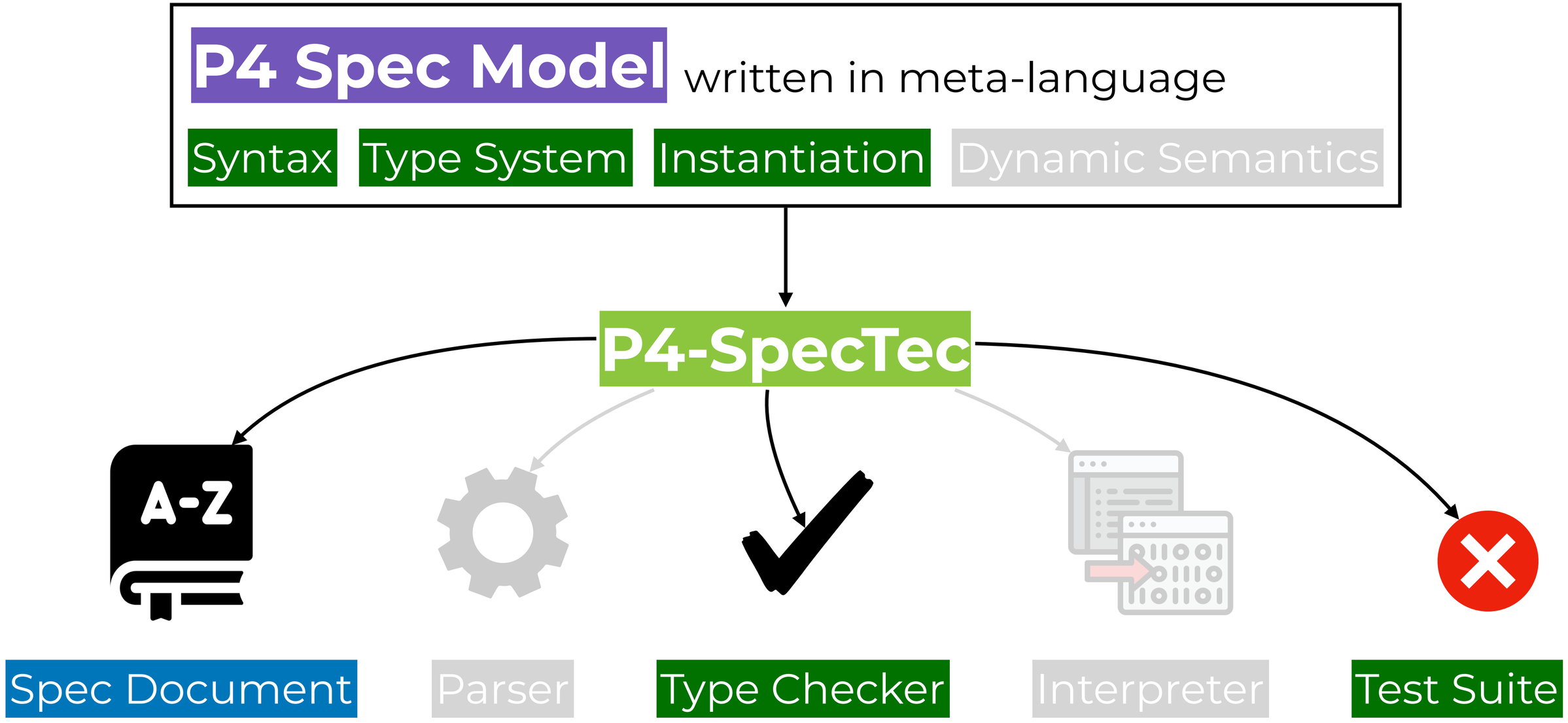
(WIP) Spec Maintenance Becomes Clear in P4-SpecTec

Allow implicit casts for directionless arguments #2

Open **shyukahn** wants to merge 6 commits into `concrete` from `concrete-exclude-impl-cast-directionless`

1. If **actctxt** equals ACTION, then ...
2. Else,
   a. Let **expressionIR** # `(**typeIR_arg** **ctk_arg**) be **typedExpressionIR_arg**
   b. If Type_alpha: **typeIR_param** $=_{\alpha}$ **typeIR_arg** holds, then
      i. If **ctk_arg** does not match pattern DYN, then
         1. Result in **typedExpressionIR_arg**
   b. Let **typedExpressionIR?** be $coerce_unary(**typedExpressionIR_arg**, **typeIR_param**)
   c. If **typedExpressionIR?** matches pattern (_), then
      i. Let **typedExpressionIR_arg_cast** be **typedExpressionIR?**
      ii. If **ctk_arg** does not match pattern DYN, then
         1. Result in **typedExpressionIR_arg_cast**

Prose is automatically updated


P4 Spec Model
written in meta-language
Syntax
Type System
Instantiation
Dynamic Semantics
P4-SpecTec
A-Z
Spec Document
Parser
Type Checker
Interpreter
Test Suite

Mechanizing the P4 Language Specification with P4-SpecTec

What is mechanization, and **why** do we need it?

How do we mechanize the P4 language specification?

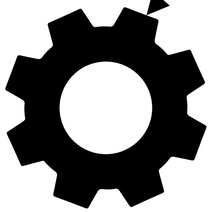
What are the benefits?

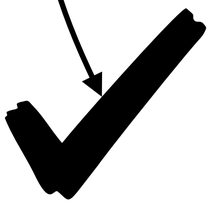
https://github.com/kaist-plrg/p4-spectec

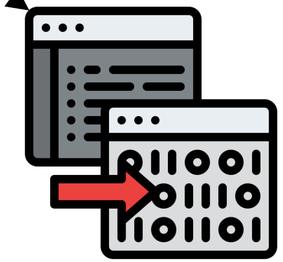
P4 Spec Model written in meta-language

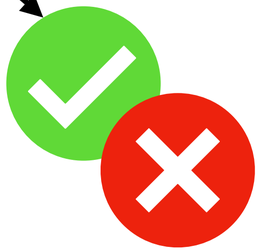
Syntax | Type System | Instantiation | Dynamic Semantics

P4-SpecTec









Spec Document | Parser | Type Checker | Interpreter | Test Suite