Application of SMT in a Meta-Compiler:
A Logic DSL for Specifying Type Systems

Romain Beguet, Raphaël Amiard

SMT 2023 Workshop

Some context first...

Langkit: Basics

- Language description language (a la JetBrains MPS, Spoofax, Eclipse Xtext)
- **State of the art in terms of expressing type-system complexity**
- Designed for Ada (main use case is Libadalang, an Ada language front-end)
- Libadalang: Ada front-end used industrially in most of AdaCore's Ada products (IDEs, style checkers, static analyzers, etc)

Langkit

```
lexer test_lexer {
    par_open <- "("
    par_close <- ")"
    id <- "\w+"
    keywords <- {"if", "then", "else", "fn"}
    separators <- {"(", ")", ":"}
    operators <- {"+", "-", "*", "="}
}
```

```
grammar test_grammar {
  fn_def <-
  FnDef("fn" id
        "(" list*(Param(id ":" id), ",") ")"
        "=" expr)
  expr <-
  IfExpr("if" expr "then" expr "else" expr)
  | OpExpr(expr @operator expr)
  | CallExpr(id "(" list+(Param(id), ",") ")")
}
```

```
class IfExpr {
    fun type_equation() : Equation =
        self.if_expr.type_equation
        and self.then_expr.type_equation
        and self.if_expr.type_var <-> self.then_expr.type_var
}
```

Libadalang: Semantic Analysis

- Ada supports function overloading on both arguments and return types
- Finding the correct declarations is a complex and non local process
  - Requires looking at the whole expression

```
procedure Test is
   function A return Boolean is (True);
   function A return Integer is (1);
   procedure B (X : Float) is null;
   procedure B (X : Integer) is null;
begin
   B (A);
end Test;
```

Libadalang: Example

```ada
procedure Test is
   function A return Boolean is (True);
   function A return Integer is (1);
   procedure B (X : Float) is null;
   procedure B (X : Integer) is null;
begin
   B (A);
end Test;
```

And(

$Or(A_{ref} \leftarrow$ <A test.adb:2>, $A_{ref} \leftarrow$ <A test.adb:3>),

$Or(B_{ref} \leftarrow$ <B test.adb:4>, $B_{ref} \leftarrow$ <B test.adb:5>),

$A_{expected\_type} \leftarrow arg\_type(B_{ref})$,

$A_{actual\_type} \leftarrow ret\_type(A_{ref})$,

$matching\_type(A_{actual\_type}, A_{expected\_type})$

)

A_{ref} = <A test.adb:3>

B_{ref} = <B test.adb:5>

$A_{expected\_type}$ = <Integer>

A_{actual_type} = <Integer>

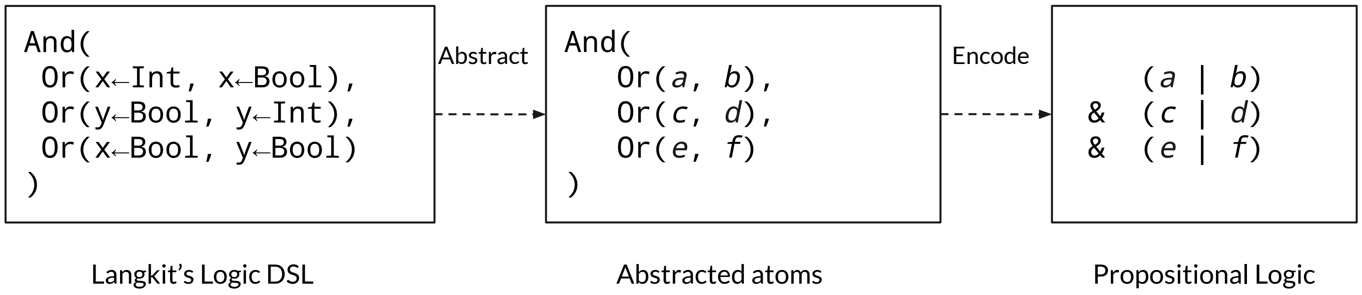
Naive Solver(s)

- Several iterations of naive solvers
- Last one: Expand disjunctions & prune early
- Order-dependent:
  - Equations are hard to write and easy to break
  - Kind of defeats the declarative, “modeling” aspect of our logic DSL
- Too slow for some problems

SMT-based solver for Langkit

SAT + Lazy encoding of theory

- Encode the high-level relation in a boolean formula, abstracting away atoms



SAT + Lazy encoding of theory

- Encode the high-level relation in a boolean formula, abstracting away atoms
- Ask SAT solver for a model, run the theory solver on it
- If we find a contradiction, integrate it back in the original problem and repeat

```
And(
 Or(x←Int, x←Bool),
 Or(y←Bool, y←Int),
 Or(x←Bool, y←Bool)
)

contradictions: [
  [x←Int, x←Bool]
]
```

Langkit's Logic DSL

Abstract →

```
And(
   Or(a, b),
   Or(c, d),
   Or(e, f)
)

contradictions: [
  [a, e]
]
```

Abstracted atoms

Encode →

```
    (a | b)
&   (c | d)
&   (e | f)
&   (!a | !e)
```

Propositional Logic

Ordered Disjunctions

- In propositional logic, “`(A | B) & ...`” can be satisfied if **at least one** of A, B is satisfied
- In our logic, “`And(Or(A, B), ...)`” means: try with A first, and if it fails then try with B
- Allows conveying “preference”, e.g. :
    - In some languages like Ada or Scala, more local entities are preferred to more global ones

Ordered Disjunctions

- In the literature, solvers iteratively refine the model until it maximizes global satisfaction according to a given metric
- For our particular case, we can avoid the concept of global satisfaction

Ordered Disjunctions

- Consider:

```
And(
 Or(x←Int, y←Bool),
 Or(y←Int, x←Bool)
)
```

Ordered Disjunctions

- Consider:

```
And(
 Or(x←Int, y←Bool),
 Or(y←Int, x←Bool)
)
```

- Solutions:
  a) `{x←Int,  y←Int}`
  b) `{x←Bool, y←Bool}`
- Solution *a* is clearly preferred to solution *b*

Ordered Disjunctions

- Consider:

```
And(
 Or(x←Int,  y←Int),
 Or(x←Bool, y←Bool)
)
```

Ordered Disjunctions

- Consider:

```
And(
 Or(x←Int,  y←Int),
 Or(x←Bool, y←Bool)
)
```

- Solutions:
  a) `{x←Int, y←Bool}`
  b) `{y←Int, x←Bool}`
- None of them is better than the other, because:
  a) `x←Int` (from a) is preferred to `y←Int` (from b)
  b) `x←Bool` (from b) is preferred to `y←Bool` (from a)
- We say that the problem is *ambiguous*
  a) e.g. multiple overloads work for a given function call

Ordered Disjunctions

- Thanks to this restriction, we can compute an optimal model using only a SAT solver:
    a. For a given ordered disjunction, encode the fact that only one branch can be selected at the same time
    b. Make sure variables corresponding to left branches are decided first
- See proof in paper!

Exactly-One Constraints

- Encoding "`Or(A, B, C)`" in propositional logic:
  - At least one of A, B, C should be in the model: "A ∨ B ∨ C"
  - If A is in the model, B and C shouldn't: "A ⇒ ¬B & ¬C"
  - if B is in the model, A and C shouldn't: "B ⇒ ¬A & ¬C"
  - if C is in the model, A and B shouldn't: "C ⇒ ¬B & ¬C"
- This corresponds to a pairwise encoding of an Exactly-One (or one-hot) constraints, which enforces the fact that only one branch can be selected at once

Exactly-One Constraints

- Original transformation:

```
And(
 Or(x←Int, x←Bool),
 Or(y←Bool, y←Int),
 Or(x←Bool, y←Bool)
)
```

Encode →

```
    (a | b)
&   (c | d)
&   (e | f)
```

- Produces models in which atoms from the left **and** the right branch might appear
  - E.g. {a, b, c, d, e, f} is a valid model

Exactly-One Constraints

- New transformation:

```
And(
 Or(x←Int, x←Bool),
 Or(y←Bool, y←Int),
 Or(x←Bool, y←Bool)
)
```

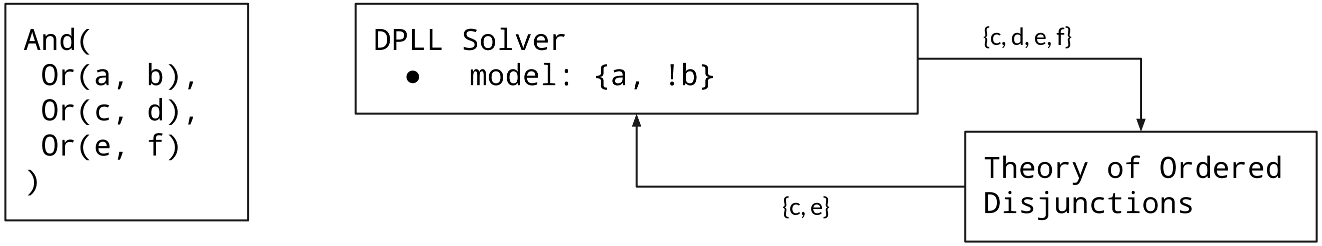
Encode →

```
    (a | b) & (!a | !b)
&   (c | d) & (!c | !d)
&   (e | f) & (!e | !f)
```

- Produces models in which **either** atoms from the left or from the right appear, but **not both**!
  - {a, c, e}, {a, c, f}, {a, d, e}, {a, d, f}, {b, c, e}, {b, c, f}, {b, d, e}, {b, d, f}
- However, we still have a problem with the **order**
  - E.g. either {a, c, e} or {b, c, e} might be found by the solver depending on its branching algorithm

Theory-Driven Decisions

- Extend SAT interface to allow the theory to have a word on variable decisions
  - When DPLL needs to branch, it asks the theory which literals it can make its choice on
- In our case: pick unassigned variables of left-most branches of ordered disjunctions



- This produces a sequence of models in which atoms from the left branches are tried first!

The Big Picture

```
And(
 Or(x←Int, x←Bool),
 Or(y←Bool, y←Int),
 Or(x←Bool, y←Bool)
)
```

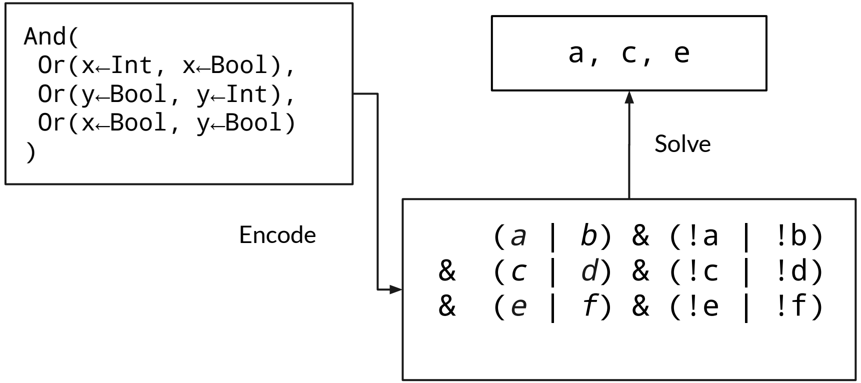
The Big Picture

```
And(
 Or(x←Int, x←Bool),
 Or(y←Bool, y←Int),
 Or(x←Bool, y←Bool)
)
```

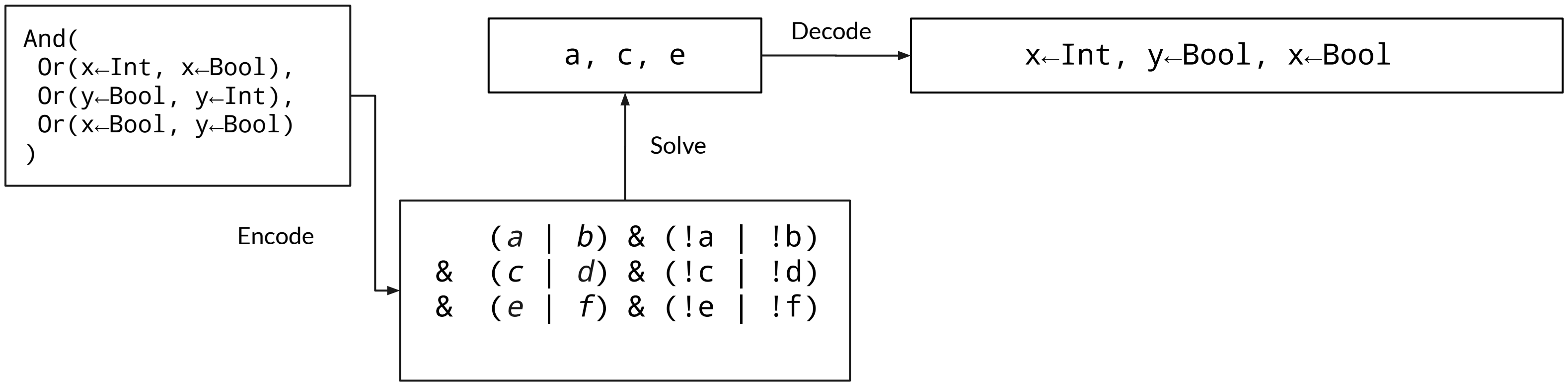
Encode

```
    (a | b) & (!a | !b)
&   (c | d) & (!c | !d)
&   (e | f) & (!e | !f)
```

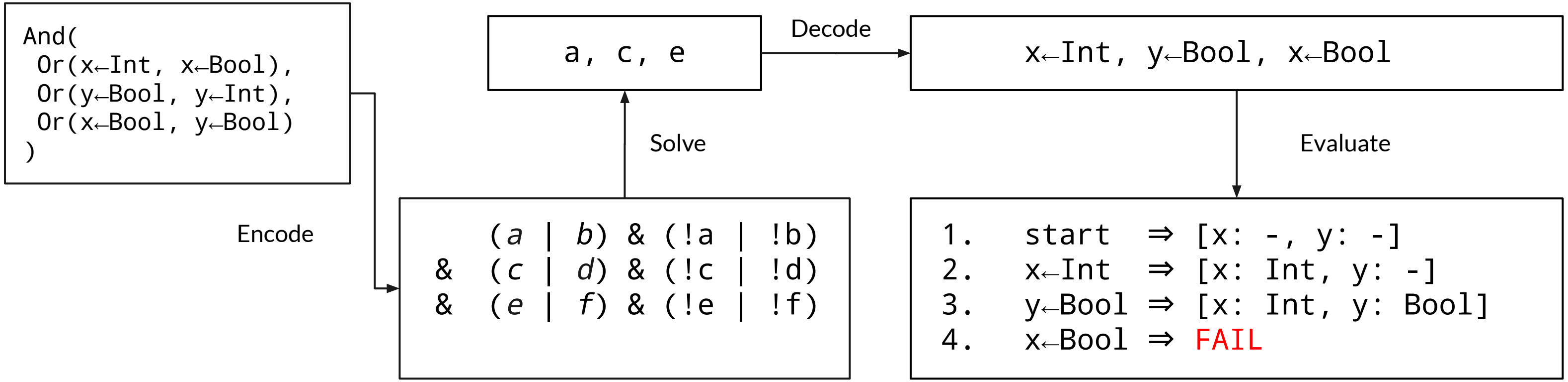
The Big Picture



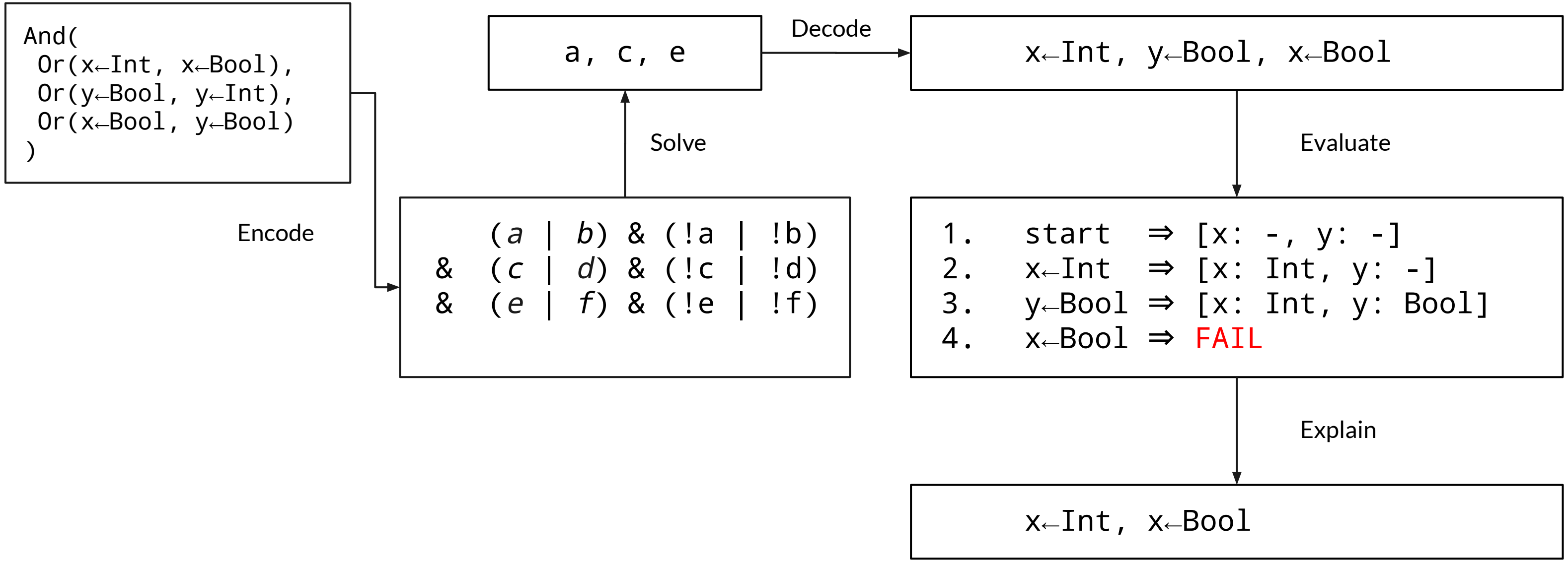
The Big Picture



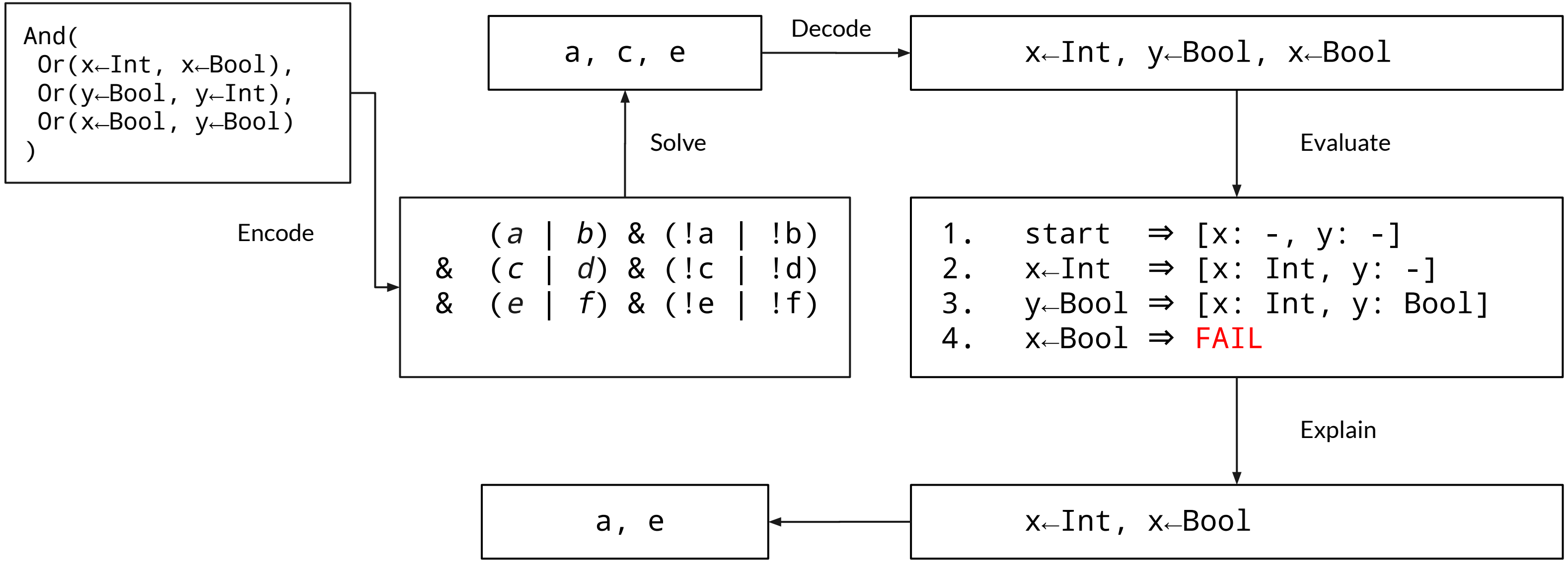
The Big Picture



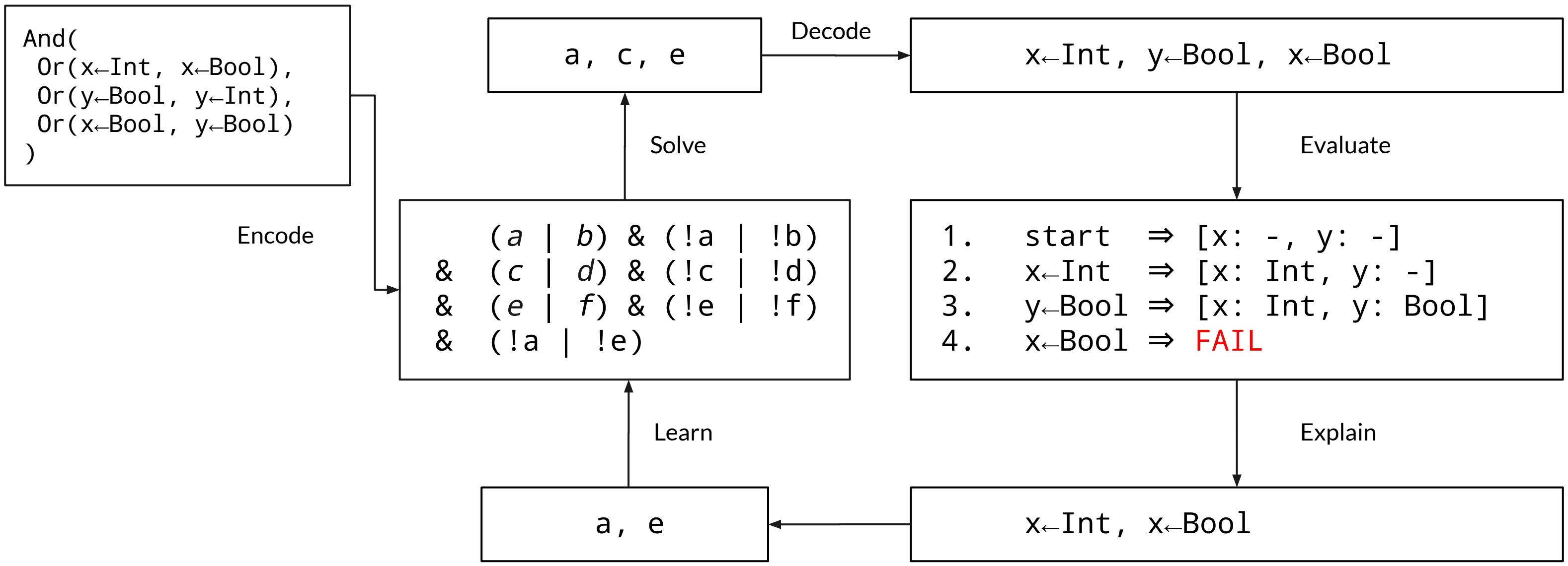
The Big Picture



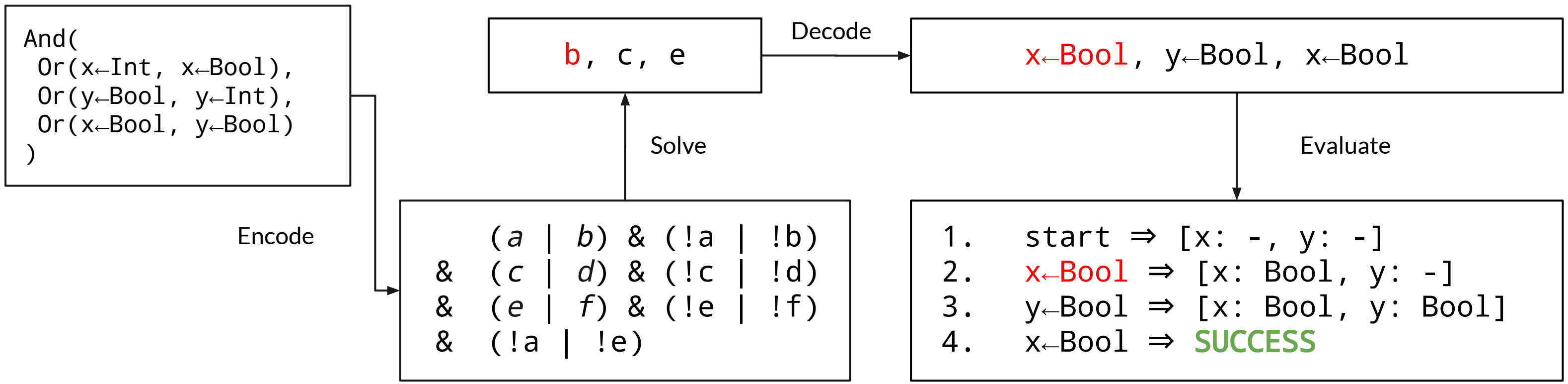
The Big Picture



The Big Picture



The Big Picture



AdaSAT

- Implementation of a SAT solver in Ada
  - Conflict Driven Clause Learning (CDCL)
  - Two-watched literals
  - Blocking literals
  - ...
- Low overhead in both directions (memory layout, exceptions)
- Fastest possible on trivial cases (because most cases solved will be trivial)
- Theory-driven variable decisions
- Optimized handling of AMO constraints

AdaSAT: Optimized AMO Constraints

- Pairwise encoding requires quadratic number of clauses

```
Or(a, b, c)  --Encode-->  (a | b | c) & (!a | !b) & (!a | !c) & (!b | !c)
```

- Tried other encodings (bitwise encoding requires $\log_2$ extra vars & linear extra clauses)

AdaSAT: Optimized AMO Constraints

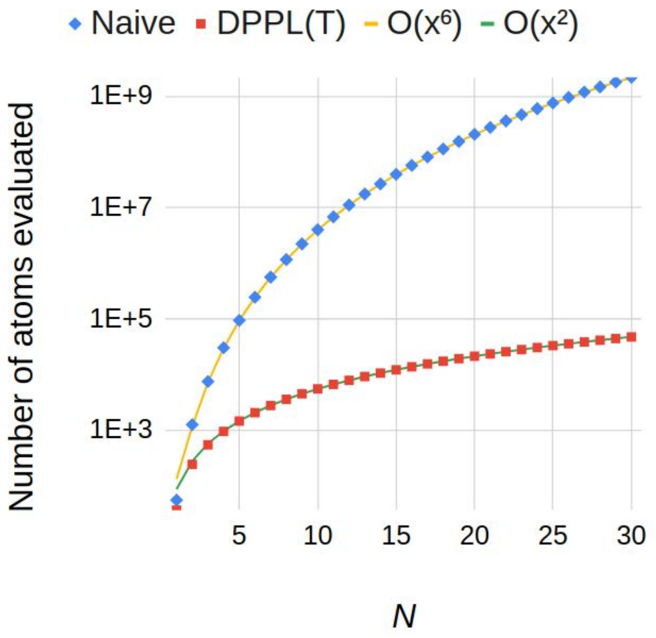
- Make sure indices for branches of a given disjunction are contiguous
- Represent an AMO constraint of variables in range `a .. b` using a special clause shape

```
Or(a, b, c)  --Encode-->  (a | b | c) & (AMO, a, c)
```

- In unit propagation, as soon as any literal in `a .. b` is set to True, set every other to False
- In conflict resolution, simulate a pairwise encoding (but never synthesize binary clauses)

Results

Performance



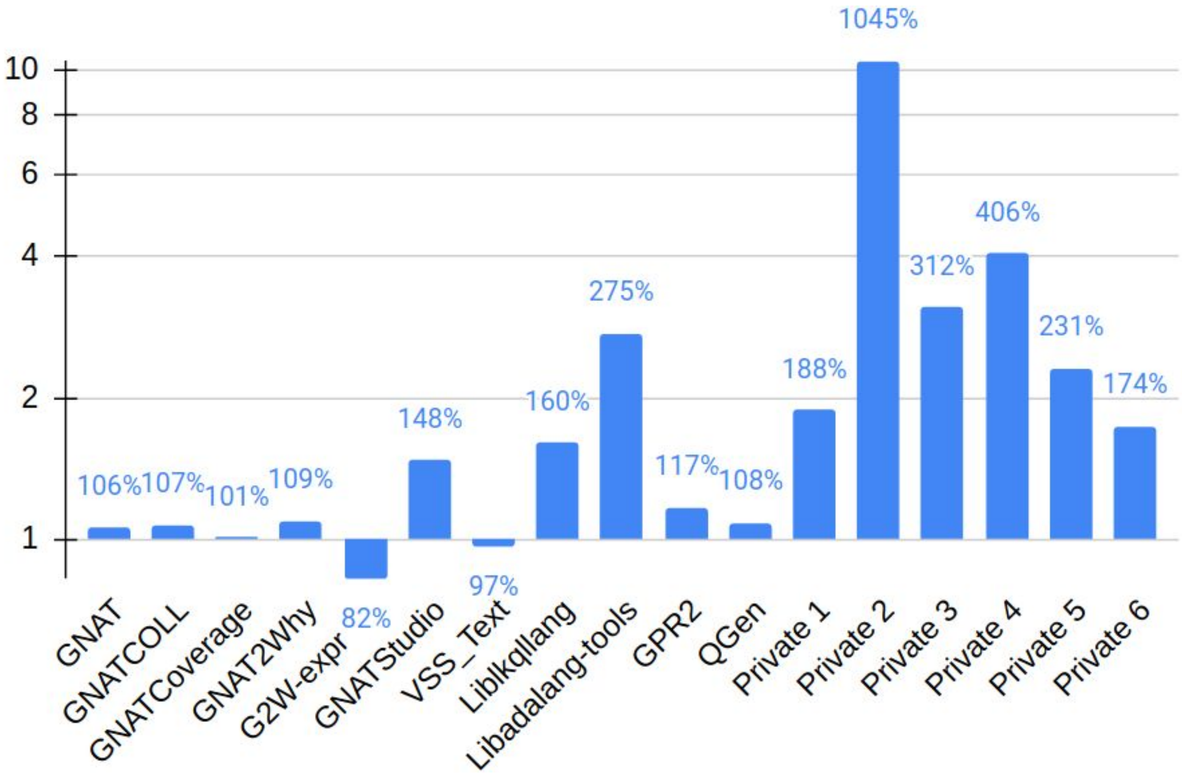
Number of atoms evaluated when varying number of overloads & number of calls to F

```
procedure Test is
   type T1 is null record;
   type T2 is null record;
   type T3 is null record;

   function F (X : T1) return T1 is (null record);
   function F (X : T1) return T2 is (null record);
   function F (X : T1) return T3 is (null record);

   function F (X : T2) return T2 is (null record);
   function F (X : T2) return T3 is (null record);

   function F (X : T3) return T3 is (null record);

   procedure P (X : T1) is null;
   procedure P (X : T2) is null;
   procedure P (X : T3) is null;

   X : T1;
begin
   P (F (F (F (X))));
end Test;
```

Speedup

Impact resolving all names & types over several codebases.

This is **total** run-time speedup (including parsing & scope construction). Real solver speedup is marginally higher.



Diagnostics

Before:

```
procedure Test is
   X : Integer;

   function Foo (X : Integer) return Integer is (0);
   function Foo (X : Float)   return Integer is (0);
begin
   X := Foo (True);
end Test;
```

```
Resolving xrefs for node <AssignStmt test.adb:7:4-7:20>
*******************************************************

Resolution failed for node <AssignStmt test.adb:7:4-7:20>
```

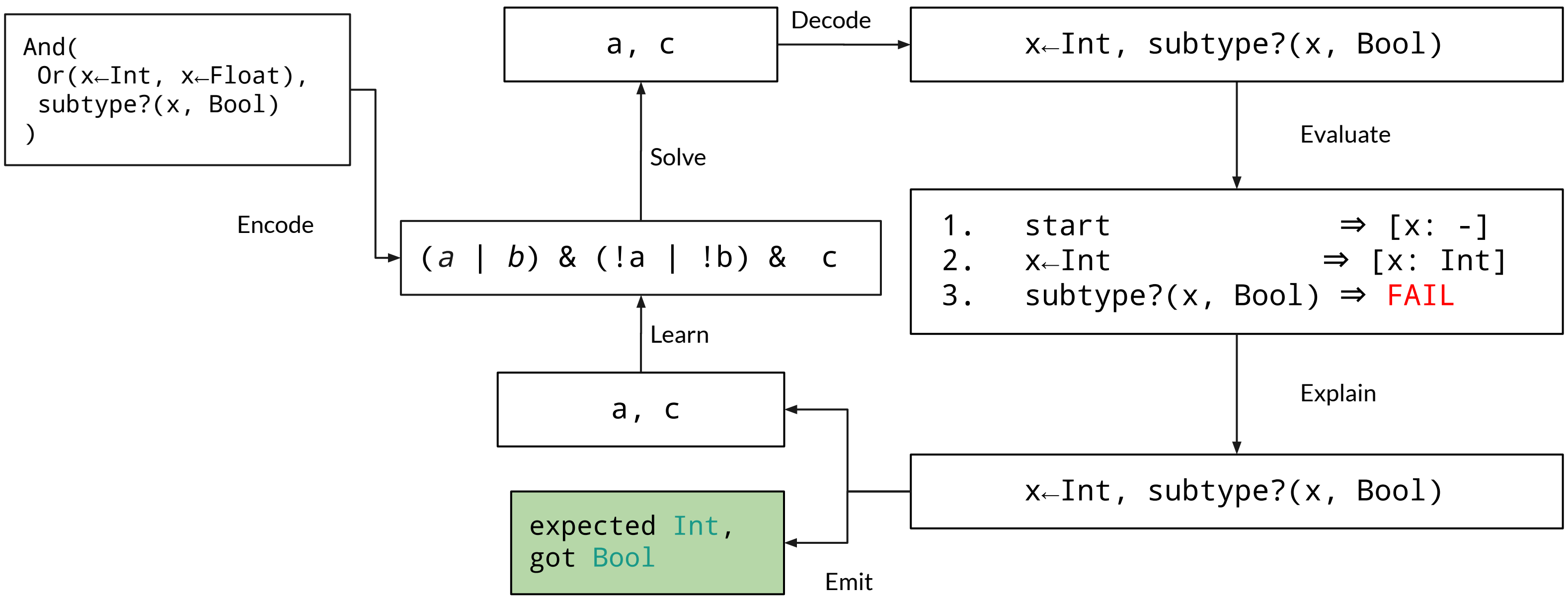
Diagnostics

- Realization: explanations produced by the theory look exactly like what we want to report
  - They only keep the relevant information out of a failure
  - Since that same explanation is used for the solver, we know we will never have duplicate diagnostics
- Allow attaching error message templates to atoms

```
@predicate_error("expected $expected_type, got $self")
fun subtype(self, expected_type: BaseTypeDecl) → bool = …
```

- Allow attaching context to atoms
- *Still work-in-progress*

Diagnostics Generation



Diagnostics

After:

```
procedure Test is
   X : Integer;

   function Foo (X : Integer) return Integer is (0);
   function Foo (X : Float)   return Integer is (0);
begin
   X := Foo (True);
end Test;
```

```
Resolving xrefs for node <AssignStmt test.adb:7:4-7:20>
*******************************************************

test.adb:7:9: error: no matching alternative (of 2 candidates)
7 |     X := Foo (True);
  |          ^^^

test.adb:5:22: info: expected Float, got Boolean
5 |     function Foo (X : Float) return Integer is (0);
  |                       ^^^^^

test.adb:4:22: info: expected Integer, got Boolean
4 |     function Foo (X : Integer) return Integer is (0);
  |                       ^^^^^^^
```

Future work

- Try plugging existing SAT solvers!
  - Maybe CaDiCaL via IPASIR-UP ?
- Encode some properties of our logic more eagerly
  - E.g. it might be possible to encode atom dependencies directly
- Investigate cost/benefit of implementing fine grained theory propagation
- Express other type systems with different paradigms
  - Structural subtyping
  - Advanced type inference
  - …

Questions?