AdaCore TECH PAPER

Application of SMT in a Meta-Compiler:

A Logic DSL for Specifying Type Systems

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

Romain Béguet[1], Raphaël Amiard[1]

[1] *AdaCore, 46 Rue d'Amsterdam, 75009 Paris, France*

Abstract

Building a compiler for a programming language is a notoriously hard task, especially when starting from scratch. That's why we developed Langkit, a meta-compiler which allows language designers to focus on the high-level aspects of their language. In particular, the specification of type systems is done in a declarative manner by writing equations in a high-level logic DSL. Those equations are then resolved by a custom solver embedded in the generated compiler front-end whenever a code fragment of the target language needs to be analyzed. This framework is successfully used at AdaCore to implement name and type resolution of Ada, powering navigation in IDEs and enabling development of deep static analysis tools.

We discuss the implementation of our solver, and how a recent switch to using a DPLL(T) solver backend with a custom theory allowed us both to address long-standing combinatorial explosion problems we had with our previous approach, but also to gain new insights in how to emit human-readable diagnostics for type errors.

Keywords

Satisfiability Modulo Theory, Meta-Compilation, Logic Programming

1. Introduction

In a typical multi-pass compiler pipeline for a statically typed language, the result of parsing a code fragment of the target language is an abstract syntax tree which does not carry any semantic information: identifiers are mere symbols that are not yet linked to their corresponding definition, expressions are untyped, etc. Thus, one or several semantic analysis phases take place to compute cross-references and validate or infer types.

In some languages (such as C), figuring out the definition a name refers to, or the type of an expression, is not context-sensitive: it merely requires a single name lookup in its lexical environment. In other words, resolving an identifier does not require resolving the name or the type of any other adjacent nodes. In languages such as Java or C++ that support function overloading, the function definition targeted by a function call may depend on the type of one or several of its arguments: therefore resolving this call requires a more involved analysis, often done in cooperation with a type analysis working on several sub-parts of the expression at once. Some languages such as Ada or Swift even allow overloading functions on their return-type, which may cause type information to flow forward and backward through sub-expressions of a given statement.

SMT 2023: 21st International Workshop on Satisfiability Modulo Theories, Rome, Italy, July 5-6, 2023

beguet@adacore.com (R. Béguet); amiard@adacore.com (R. Amiard)

CEUR Workshop Proceedings (CEUR-WS.org)

In such complex languages, it is often useful to compute the static semantics of a given program in two stage [1, 2], by first accumulating local constraints specified by each construct appearing in the program according to its syntactic category and lexical context, and then have a dedicated solver try to satisfy a fixed group of constraints at once in order to find a solution – the exact scope of collection and solving depending on the language. This is the approach taken by compiler implementations such as GHC [3] or Swiftc [4], but also by our meta-compiler framework Langkit [5], which we describe in the next section.

This technique offers several advantages: writing, understanding and debugging the type system is easier because typing rules are described explicitly instead of being implied by the execution of an algorithm. Elaborating an optimized implementation of a type system dedicated to a particular language often requires significant efforts (many articles have been published over the course of several years about optimization of overload resolution in Ada [6, 7, 8, 9]) and such implementations are hardly reusable. A constraint solver on the other hand is generally an independent component that can be reused and optimized without leaking in the implementation of the type system, making the latter easier to maintain and extend. However using a constraint solver to type programs raises an issue: how to produce good error messages for sets of constraints that have no solution, given that producing good error messages for complex type systems is already a complex problem [10]. We thus present in the following sections our main contributions, which are:

- The description and formalization of Langkit's high-level domain-specific language (DSL) for specifying static semantics of programming languages that support complex features such as subtyping, return-type overloading and preferred interpretations.
- Its implementation, as a lazy encoding of a custom *theory of ordered disjunctions* on top of the DPLL(T) architecture with minor extensions, and a demonstration that the resulting framework allows generating an efficient semantic analyzer for Ada.
- Extensions to our DSL to allow language implementations to produce concise and relevant error messages for object language expressions that are semantically incorrect – even in presence of overloading – based on explanations produced by our theory.

2. Logic Framework

Our Langkit meta-compiler allows users to specify a language frontend, from syntax to semantics, including the specification of the type system through methods that build sets of logic constraints. The main use-case of Langkit so far has been to implement Libadalang [11], demonstrating that the logic framework we developed is sufficient to implement the type system of Ada, a complex industrial language spanning multiple revisions and featuring generics, ad-hoc polymorphism (including return-type overloading), subtyping, object oriented features, etc. The entire specification for the Ada language frontend, including description of the syntax tree, the scope graph definition, and the name and type resolution specification, is around 25000 SLOCs, which we estimate to be at least 10 times smaller than what is needed by the main Ada compiler GNAT [12] to implement the equivalent functionalities.

As far as we know, Langkit is the only *generic* language framework that is able to express complex type systems such as those of Swift, Ada or C# (which is known to be NP-hard [13]),

$$
\begin{array}{lcll}
\mathcal{E} & ::= & \mathcal{A} & \textit{Atom} \\
 & | & \mathcal{E} \wedge \mathcal{E} & \textit{Conjunction} \\
 & | & \mathcal{E} \mathbin{\vec{\times}} \mathcal{E} & \textit{Ordered disjunction} \\
\mathcal{A} & ::= & \mathcal{V} \leftarrow \mathcal{C} & \textit{Assignment} \\
 & | & \mathcal{V} \leftarrow \mathcal{F}(\mathcal{V}, \ldots, \mathcal{V}) & \textit{Propagation} \\
 & | & \mathcal{V} \leftrightarrow \mathcal{V} & \textit{Aliasing} \\
 & | & \mathcal{P}(\mathcal{V}, \ldots, \mathcal{V}) & \textit{Assertion} \\
 & | & \top & \textit{True} \\
 & | & \bot & \textit{False} \\
\mathcal{V} & ::= & x, y, \ldots & \textit{Variables} \\
\mathcal{C} & ::= & \mathrm{A}, \mathrm{B}, \ldots & \textit{Constants} \\
\mathcal{F} & ::= & \mathrm{F}, \mathrm{G}, \ldots & \textit{Function symbols} \\
\mathcal{P} & ::= & \mathrm{P}, \mathrm{Q}, \ldots & \textit{Predicate symbols}
\end{array}
$$

Grammar 1: Formal syntax of Langkit's logic DSL

where types and names flow in both directions in the expression tree, and in some particular cases even require looking deeper down the tree [14]. In particular, we think that one of the features that makes our framework powerful is that there is no distinct stage between name binding and type inference: every bit of semantic information is represented by a logic variable to be resolved. Inter-dependency between name resolution and type inference is represented by type variables being defined as functions of name variables (as in $foo_{type} \leftarrow \texttt{ReturnType}(foo_{ref})$) and vice-versa.

Langkit has also been used to implement more complex type system features in proof-of-concepts projects such as Dependz [15], implementing a small dependently typed calculus. However, it remains to be seen how adaptable our approach is to languages with type systems that contain more structural features, like structural subtyping.

2.1. Formalization

In our Langkit meta-compiler, an element of *unresolved* semantic information – such as a type to be inferred – can be represented by a *logic variable*. Using Langkit's logic DSL – whose grammar is formalized in 1, users can define constraints on those variables: an *Assignment* allows assigning constant values to variables, such as $foo_{type} \leftarrow \texttt{Integer}$. A *Propagation* assigns to a variable a function of another variable – as in $bar_{type} \leftarrow \texttt{ReturnType}(bar_{ref})$ – by evaluating that function at solve-time once all the argument variables have been solved. Such functions are defined by the user in the pure (side-effect free) functional language fragment provided by Langkit. An *Aliasing* allows unifying two variables together, forcing them to hold the same value. An *Assertion* can be used to enforce that a predicate over given variables must hold – as in $\texttt{IsSubtype}(foo_{type}, bar_{type})$, and is evaluated as soon as all input variables are solved. Predicate symbols are also defined by the user as functions returning booleans in Langkit's functional DSL. Finally, it is possible to combine all of these constraints in arbitrarily nested conjunctions and (ordered) disjunctions.

Ordered Disjunctions. The ordered disjunction operator ($\vec{\times}$) resembles the one introduced by Qualitative Choice Logic [16] and LPODs [17, 18]. Intuitively, the meaning of $\mathcal{E}_1 \vec{\times} \mathcal{E}_2$ is: try $\mathcal{E}_1$ first, but if it doesn't work then try $\mathcal{E}_2$. This behavior allows language designers to encode static *preference* semantics in their language. For example, in a binary operation `X + Y` in Ada, if there are multiple possible interpretations of $+$ among which one of them is the $+$ operator defined on root integers (one of the primitive types in Ada), then this interpretation is preferred [19]. This can also for example be used to encode the preferred order for overload resolution in C++, or even the order in which implicit values should be considered when synthesizing an implicit parameter in Scala.

Definition 2.1 (Partially Ordered Split Programs)**.** Formally, we define the semantics of ordered disjunctions by reasoning in terms of *partially ordered split programs*, where a split program is only a conjunction of atoms. For that, we define the operator $\psi(\mathcal{E})$ reducing an equation $\mathcal{E}$ to its partially ordered set of split programs $\langle \{P_1, \ldots, P_n\}, \leq \rangle$, using the rules from figure 1.

$$\frac{}{\psi(\mathcal{A}) = \langle \{\mathcal{A}\}, \{\mathcal{A} \leq \mathcal{A}\} \rangle}\ \textsc{Atom}$$

$$\frac{\psi(\mathcal{E}_1) = \langle S_1, \leq_1 \rangle \qquad \psi(\mathcal{E}_2) = \langle S_2, \leq_2 \rangle}{\psi(\mathcal{E}_1 \vec{\times} \mathcal{E}_2) = \langle S_1 \cup S_2, \leq_1 \cup \leq_2 \cup \{P_1 \leq P_2\}_{P_1 \in S_1, P_2 \in S_2} \rangle}\ \textsc{Disjunction}$$

$$\frac{\psi(\mathcal{E}_1) = \langle S_1, \leq_1 \rangle \qquad \psi(\mathcal{E}_2) = \langle S_2, \leq_2 \rangle}{\psi(\mathcal{E}_1 \wedge \mathcal{E}_2) = \langle \{P_1 \wedge P_2\}_{P_1 \in S_1, P_2 \in S_2}, \{P_1 \wedge P_2 \leq Q_1 \wedge Q_2\}_{P_1 \leq_1 Q_1, P_2 \leq_2 Q_2} \rangle}\ \textsc{Conjunction}$$

Figure 1: Reduction rules for ψ, mapping a DSL equation to a partially ordered set of split programs.

Intuitively, the resulting partial order allows reasoning about preference on split programs that do not contain ordered disjunctions anymore – as if the ordering information had been extracted out of them. We have for example that:

$$\psi((v \leftarrow \textsc{a} \vec{\times} v \leftarrow \textsc{b}) \wedge (w \leftarrow \textsc{c} \vec{\times} w \leftarrow \textsc{d})) = \langle \{P_1, P_2, P_3, P_4\}, P_1 \leq P_{2,3} \leq P_4 \rangle$$

$$P_1 = v \leftarrow \textsc{a} \wedge w \leftarrow \textsc{c}, \qquad P_2 = v \leftarrow \textsc{a} \wedge w \leftarrow \textsc{d},$$
$$P_3 = v \leftarrow \textsc{b} \wedge w \leftarrow \textsc{c}, \qquad P_4 = v \leftarrow \textsc{b} \wedge w \leftarrow \textsc{d}.$$

In particular, P_2 and P_3 are not comparable because P_2 contains $v \leftarrow \textsc{a}$ which is preferred over $v \leftarrow \textsc{b}$, but P_3 contains $w \leftarrow \textsc{c}$ which is preferred over $w \leftarrow \textsc{d}$. However, both P_2 and P_3 are smaller than P_4 (which contains all least preferred alternatives). P_1 on the other hand is smaller than all of them, as it contains all most preferred alternatives.

Definition 2.2 (Evaluation of Split Programs)**.** We now give a method for evaluating equations that do not contain ordered disjunctions. By definition, such an equation is of the form $A_1 \wedge \cdots \wedge A_n$ (with $n \geq 1$). Since conjunctions are commutative and associative, we can actually reason about such equations as an unordered set of atoms $\{A_1, \ldots, A_n\}$.

We therefore define $\Phi : \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{V} \times \mathcal{C}) \cup \{\bot\}$ as reducing a set of atoms to a variable-to-constant mapping μ or a failure term $\bot$ using the reduction rules from figure 2, where predicate calls $\textsc{p}(C_1, \ldots, C_n) \in \{\top, \bot\}$ and function calls $\textsc{f}(C_1, \ldots, C_n) = C$ may run arbitrary, side-effect free computations:

$$\frac{}{\Phi(P \cup \{\bot\}) = \bot}\ \textsc{Bot} \qquad \frac{\Phi(P) = \mu}{\Phi(P \cup \{\top\}) = \mu}\ \textsc{Top} \qquad \frac{\Phi(\{A[v_1 := v_2]\}_{A \in P}) = \mu}{\Phi(P \cup \{v_1 \leftrightarrow v_2\}) = \mu}\ \textsc{Alias}$$

$$\frac{C_1 \neq C_2}{\Phi(P \cup \{v \leftarrow C_1\} \cup \{v \leftarrow C_2\}) = \bot}\ \textsc{Fail} \qquad \frac{n \geq 0 \qquad v_1 \ldots v_n \text{ pairwise distinct}}{\Phi(\{v_i \leftarrow C_i\}_{1 \leq i \leq n}) = \{(v_i, C_i)\}_{1 \leq i \leq n}}\ \textsc{Success}$$

$$\frac{(v_1 \leftarrow C_1), \ldots, (v_n \leftarrow C_n) \in P \qquad \Phi(P \cup \{w \leftarrow \textsc{f}(C_1, \ldots, C_n)\}) = \mu}{\Phi(P \cup \{w \leftarrow \textsc{f}(v_1, \ldots, v_n)\}) = \mu}\ \textsc{Propagate}$$

$$\frac{(v_1 \leftarrow C_1), \ldots, (v_n \leftarrow C_n) \in P \qquad \Phi(P \cup \{\textsc{p}(C_1, \ldots, C_n)\}) = \mu}{\Phi(P \cup \{\textsc{p}(v_1, \ldots, v_n)\}) = \mu}\ \textsc{Assert}$$

Figure 2: Reduction rules for Φ, describing evaluation of split programs.

In particular, rule Success triggers when the only atoms left are assignments to distinct variables. Conversely, rule Fail triggers when there are two conflicting assignments to the same variable. Both rules Propagate and Assert require all their variable arguments to have corresponding assignments, which values are then used as arguments to dynamically invoke the used-defined operation denoted by the function or predicate symbol. Note that for the sake of brevity we have not included rules to explicitly fail when reduction is stuck (which can only happen if rules Propagate and Assert cannot be applied because some variables are never assigned), but this is easily detectable and we reduce to $\bot$ as well in these cases.

Definition 2.3 (Preferred Solutions)**.** Given an equation $\mathcal{E}$ and its partially ordered set of split programs $\psi(\mathcal{E}) = \langle S, \leq \rangle$, we define the set of *preferred solutions* $S^* \subseteq S$ of $\mathcal{E}$ as the set of split programs for which evaluation succeeds and for which there exists no smaller split program for which evaluation succeeds as well:

$$S^* = \{P^* \in S : \Phi(P^*) \neq \bot \wedge (\nexists Q \in S : Q < P^* \wedge \Phi(Q) \neq \bot)\}$$

Using this definition, we can already sketch a naive algorithm to compute the set of preferred solutions of any given equation: construct its set of split programs in a preferred order (i.e. construct P before Q if $P \leq Q$), applying the oracle Φ on each of them. As soon as we find a split program P^* that satisfies Φ, insert it in the resulting set. Now continue the process but only consider split programs that are not comparable to P^*, i.e. $Q \in S \wedge \neg(P^* \leq Q)$.

It is common in languages to consider an expression ambiguous when there are several possible non-comparable solutions. This case can be easily detected by checking whether all preferred solutions are equivalent or not: $|\{\Phi(P^*)\}_{P^* \in S^*}| = 1$.

3. Implementation and Results

Unsurprisingly, the algorithm sketched in definition 2.3 is very inefficient in practice. In particular, its run-time for an equation that has no solution is always exponential in its number of disjunctions, since it will expand and evaluate all split programs independently before determining that none are feasible. Consider also the following (solvable) equation:

$$x \leftarrow \text{A} \wedge (x \leftarrow \text{B} \mathbin{\vec{\times}} x \leftarrow \text{A}) \wedge (y \leftarrow \text{A} \mathbin{\vec{\times}} y \leftarrow \text{B} \mathbin{\vec{\times}} y \leftarrow \text{C})$$

The algorithm presented so far will compute the set preferred solutions (which only contains one solution evaluating to $\{(x, \text{A}), (y, \text{A})\}$) after evaluating 4 fully expanded options.

Early Pruning A first improvement can be made on this algorithm to avoid doing redundant work in many cases: instead of using our oracle Φ on fully expanded split programs, we can use it on intermediate split programs *during* the expansion of the equation. This requires a slight modification to Φ to make it output a different outcome when it fails because evaluation is stuck or because of an actual conflict: it only makes sense to prune the expansion in the second case, as the first case may simply be due to the incompleteness of the partial equation. In our example, we would get the following intermediate equations by splitting on the first ordered disjunction:

1. $\underline{(x \leftarrow \text{A} \wedge x \leftarrow \text{B})} \wedge (y \leftarrow \text{A} \mathbin{\vec{\times}} y \leftarrow \text{B} \mathbin{\vec{\times}} y \leftarrow \text{C})$
2. $\underline{(x \leftarrow \text{A} \wedge x \leftarrow \text{A})} \wedge (y \leftarrow \text{A} \mathbin{\vec{\times}} y \leftarrow \text{B} \mathbin{\vec{\times}} y \leftarrow \text{C})$

If we decide to run the oracle on the partial solutions (underlined above) – starting with the first one since it is preferred – we will find a contradiction early, and therefore we can avoid expanding its right-hand side. Thanks to this improvement, we now only require evaluating one partially expanded equation and one fully expanded equation (the solution).

However it has a serious flaw: the optimization can range from being extremely useful to completely ineffective depending on the order in which disjunctions are expanded: if we had decided in our example to expand the ordered disjunctions on the right first, we would not have discovered the contradiction until everything was fully expanded. This is inevitable because no predefined expansion order in the solver will be optimal for all input equations. Thus, the burden of choosing the right order is left to the language designer, which would make our logic DSL not purely declarative[1].

Conflict-Driven Traversal In order to design an optimal traversal strategy, we make the two following observations:

1. First, we notice that even if we do not choose the optimal disjunction to split on in our algorithm above, we can compute a posteriori the minimal set of atoms that would have been needed to reject this option. In our example, we can easily extract from option

[1]Note that this is precisely how our previous solver was implemented, and this not only caused maintenance problems in our Ada specification due to the order-dependent nature of the DSL being implicit, it was also sometimes a challenging task to end up with the optimal ordering when the semantics of multiple Ada constructs interfered, causing the performance of the generated semantic analyzer to degrade.

$\{x \leftarrow \text{A}, x \leftarrow \text{B}, y \leftarrow \text{A}\}$ that atoms $\{x \leftarrow \text{A}, x \leftarrow \text{B}\}$ were the cause of the conflict. We call this process Explain-Conflict.

2. If we knew in advance the complete set of atoms that are in conflict, we could reformulate the search problem as a satisfiability problem in a propositional logic augmented with ordered disjunctions (such as Qualitative Choice Logic [16]), in which atoms are abstracted by boolean variables. In our example, this would come down to finding an assignment that satisfies $a \wedge (b \vec{\times} a) \wedge (d \vec{\times} e \vec{\times} f) \wedge \underline{(\neg a \vee \neg b)}$, where the underlined clause prevents the conflicting atoms from being set at the same time.

By combining these two ingredients together, we can already devise a general solver scheme based on a *lazy encoding* [20] of our theory: iteratively refine the initial formula with conflicts found by the *SAT-with-ordered-disjunctions* solver on each candidate (here, to find one solution):

Algorithm 1 Lazy Encoding Scheme

$\mathcal{E}^{\#} \leftarrow \textsc{Encode}(\mathcal{E})$ ▷ *Encode the equation into a proposition with ordered disjunctions*
while $M \leftarrow \textsc{Solve}(\mathcal{E}^{\#})$ **do** ▷ *Invoke the SAT-with-ordered-disjunctions solver*
  $C \leftarrow \textsc{Decode}(M)$ ▷ *Reconstruct the candidate from the model*
  $R \leftarrow \Phi(C)$ ▷ *Evaluate the candidate*
  if $R = \bot$ **then**
    $\eta \leftarrow \textsc{Explain-Conflict}(C)$ ▷ *Build a clause that contradicts this candidate*
    $\mathcal{E}^{\#} \leftarrow \mathcal{E}^{\#} \wedge \textsc{Encode}(\eta)$ ▷ *Append it to the original formula to forbid this model*
  else
    return R ▷ *R is one of the preferred solutions*
  end if
end while
return Unsat

Encoding of Ordered Disjunctions Instead of designing an ad-hoc *SAT-with-ordered-disjunctions* solver, we propose a partial encoding σ to propositional logic, so as to leverage decades of research on optimizations in SAT solvers. The encoding is partial because the ordering information cannot easily be conveyed through the propositional formula itself [16] – thus we will handle it differently. We define σ using the two rules in figure 3, which reduce a DSL equation to a propositional formula alongside a mapping from each boolean variable appearing in the propositional formula to the set of atoms from the DSL equation that it represents. The mapping allows decoding models into candidate solutions that we can evaluate with Φ.

$$\frac{v \vdash \sigma(\mathcal{E}_1) = \pi_1 \parallel \mu_1 \qquad \dots \qquad v \vdash \sigma(\mathcal{E}_n) = \pi_n \parallel \mu_n}{v \vdash \sigma(\mathcal{A}_1 \wedge \dots \wedge \mathcal{A}_m \wedge \mathcal{E}_1 \wedge \dots \wedge \mathcal{E}_n) = \bigwedge_{i=1}^{n} \pi_i \parallel \{v \to \{\mathcal{A}_1, \dots, \mathcal{A}_n\}\} \cup \bigcup_{i=1}^{n} \mu_i} \quad (1)$$

$$\frac{n > 1 \qquad \mathit{fresh}(w_1, \dots, w_n) \qquad w_1 \vdash \sigma(\mathcal{E}_1) = \pi_1 \parallel \mu_1 \qquad \dots \qquad w_n \vdash \sigma(\mathcal{E}_n) = \pi_n \parallel \mu_n}{v \vdash \sigma(\mathcal{E}_1 \vec{\times} \dots \vec{\times} \mathcal{E}_n) = \bigwedge_{i=1}^{n} \pi_i \wedge \bigwedge_{i=1}^{n} (v \vee \neg w_i) \wedge (\neg v \vee \bigvee_{i=1}^{n} w_i) \wedge \mathsf{AMO}_{i=1}^{n}(w_i) \parallel \bigcup_{i=1}^{n} \mu_i} \quad (2)$$

Figure 3: Rules for the partial encoding σ, reducing a DSL equation to a propositional formula.

We give the following intuitions about the encoding rules:

- The context variable v in both rules is transferred from parent to child. Except from an initial context variable (say v_0), only disjunctions create variables – which correspond to the selection of each of their branch. Thus, v represents the current parent branch.
- In the first rule, the order in which atoms $\mathcal{A}_i$ and nested equations $\mathcal{E}_i$ occur in the conjunction does not matter. Note that a single variable is used to represent all the atoms in a conjunction.
- In the second rule, we append to the formulas generated by each branch these facts:
  1. A branch variable w_i can only be selected if its parent branch v is selected.
  2. At least one branch variable w_i must be selected if its parent branch v is selected.
  3. At most one branch variable w_i can be selected at the same time.

Remark. We do not impose a specific encoding for the generated at-most-one (AMO) constraints themselves, as there exists a wide variety of them [21] which may be more effective in certain scenarios than others – we discuss our approach in a later paragraph.

This implementation of Encode can be directly plugged in algorithm 1, and since it generates clauses in CNF, Solve may be implemented by most available SAT solvers. Here, using a pairwise encoding for AMO constraints, our running example would be encoded into a formula equivalent to:

$$a \wedge (b \vee c) \wedge (\neg b \vee \neg c) \wedge (d \vee e \vee f) \wedge (\neg d \vee \neg e) \wedge (\neg d \vee \neg f) \wedge (\neg e \vee \neg f)$$

which correctly yields models $\{a, b, d\}, \{a, c, d\}, \{a, b, e\}, \ldots$ However, the order in which these models are generated is never specified, therefore a preferred order is not guaranteed.

Enforcing a Preferred Order Indeed, modern SAT solvers typically rely on complex heuristics for literal decisions [22], and since literals in our proposed encoding directly correspond to branches of ordered disjunctions, it is not possible to enforce that models must be generated in an order of our choosing. To remedy this situation, we propose to extend the DPLL(T) interface to allow the theory to interfere in literal decisions. In particular, we propose to replace the standard Decide [20] transition with a stricter rule which we call T-Decide:

$$\frac{l \text{ or } \neg l \text{ appears in } F \qquad l \text{ is undefined in } M \qquad M \rhd_T l}{M \parallel F \Rightarrow M\ l^d \parallel F}\ \text{T-Decide}$$

where $M \rhd_T l$ signifies that theory T allows literal l to be decided in the context of the current model M. Note that this rule does not prevent the theory from allowing multiple literals to be considered (or from forbidding any of them to be, for that matter), thus sophisticated decision heuristics of DPLL implementations may still apply. In our specific case, when the theory is presented with multiple possible literals decisions, it simply makes sure not to select a literal corresponding to the n'th branch of an ordered disjunction if a literal corresponding to selecting the m'th branch (with $m < n$) of that same disjunction is also among the possible decisions. Moreover, once a first solution P^* is found (if ever), the theory may switch its mode of decision in order to force the traversal of candidate solutions that are non-comparable to P^* (by forbidding literals that could only end up producing less satisfying candidate), so as to generate the set of *preferred solutions* and detect ambiguities in a smaller number of iterations.

Theorem 3.1. *Our literal decision scheme effectively generates split programs in a preferred order.*

Proof. Assume that our procedure does not implement a preferred order of traversal. This implies that at some point, it produced a split program C whereas it should have produced a split program $C' < C$. So, out of all the ordered disjunctions of the original equation, the branches selected in C' are the same as those selected in C excepted for $k \geq 1$ of them which must be strictly preferred. Consider one of those k ordered disjunction, encoded to literals $w_1 \dots w_l$ according to rule (2) from figure 3. Then for some $1 \leq m < n \leq l$, the model for C contains $\neg w_m$ and w_n while that of C' contains w_m and $\neg w_n$. By definition of our decision scheme, the theory could not have decided w_n if w_m was also a possible decision, thus $\neg w_m$ must have been implied by previous literal assignments in C, a subset of which must not exist in C' (otherwise they would have implied $\neg w_m$ in C' as well). Since literals directly map to branches of ordered disjunctions, these differences directly correspond to different branch selections between C and C'. As implication graphs are directed and acyclic, we can apply this reasoning at most k times until we have to consider the selection of a preferred branch which cannot be explained in terms of other preferred branches. At this stage, the literals that explain this selection necessarily correspond to least preferred branches in C', which makes C and C' not comparable, therefore contradicting our initial hypothesis. □

The Theory Solver On top of guiding the selection of literals, our theory solver must be as smart as possible in its implementation of the Explain-Conflict routine in order to extract minimal explanations out of invalid candidates and make the traversal converge as fast as possible. There are two failure cases to handle when evaluating a candidate solution:

- The first one is when the evaluation results in $\bot$ due to rules Fail or Assert. The explanation for such a failure is always a subset of the atoms from the candidate solution. In particular, it suffices to include in the explanation all the atoms that we encounter by traversing the dependency graph of atom assignments backwards (taking aliasing into account) starting from the conflicting assignments or the failing predicate.
- The second case to handle is when the evaluation is stuck because of a missing variable assignment (for example in equation $x \leftarrow C \wedge \text{P}(x, y)$, y is never assigned a value). In this case we build a clause that forbids the atoms with such unsatisfied dependencies to be part of a model if the model does not also contain atoms that satisfy these dependencies – which may involve atoms that were not part of the candidate solutions. In the case of a cyclic dependency (as in equation $x \leftarrow \text{P}(y) \wedge y \leftarrow \text{P}(x)$), we build a clause that breaks the cycle by requiring an assignment to any of the variables involved in the cycle (as this assignment will propagate throughout the cycle).

Optimizing At-Most-One Constraints We initially encoded our AMO constraints using a naive pairwise encoding, but quickly found instances where it required generating too many clauses, causing a non-negligible overhead. We experimented with the binary encoding and found it to be slightly less efficient on the average cases. In the end, we implemented built-in support for at-most-one constraints in our CDCL algorithm. The idea of adding native support for cardinality constraints in SAT solvers is not new [23, 24], but for our use case we exploited

the fact that our encoding produces AMO constraints over contiguous literals (see second rule in figure 3), and designed an even more compact in-memory representation for them, requiring only a marker flag and two integers determining the range of literals that participate in the constraint. Thus, a big constraint such as AMO$(x_1, x_2, \ldots, x_{998}, x_{999})$ only requires us 3 integers to represent, whereas a pairwise encoding would generate several hundred thousands of binary clauses. This allows keeping memory overhead to a minimum and enables fast iterations over the constraints' literals in the unit propagation and conflict resolution routines.

3.1. Performance Results

Our new implementation replaced a legacy solver based on the naive algorithm with early pruning described in the beginning of this section. We first give an overview of the performance improvements obtained on carefully crafted test cases exhibiting known pathological patterns that can be found in actual codebases.

Synthetic Test-Cases The main source of complexity in typechecking Ada comes from the use of function overloading. Thus, our first synthetic test case simply declares N different overloads and performs M nested calls to the overloaded function (see figure 4a).

Our second synthetic test case extends the first one by adding bridges between types (see figure 4b), such that while a chain of nested calls of length $K < M$ may be resolved in many different ways, the complete chain of length M has only one possible interpretation. This effectively tricks the solvers into consider candidates that later become dead ends.

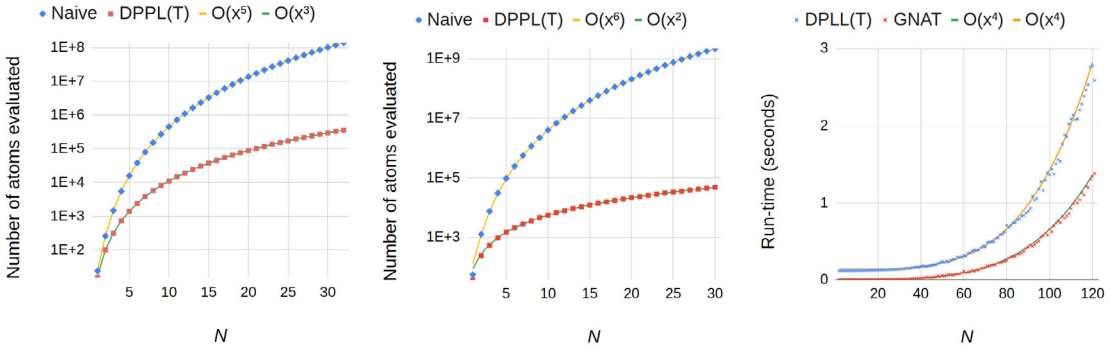
```
1  procedure Test is
2     type T1 is null record;
3     type T2 is null record;
4     type T3 is null record;
5
6     function F (X : T1) return T1 is (null record);
7     function F (X : T2) return T2 is (null record);
8     function F (X : T3) return T3 is (null record);
9
10    procedure P (X : T1) is null;
11    procedure P (X : T2) is null;
12    procedure P (X : T3) is null;
13
14    X : T1;
15 begin
16    P (F (F (F (X))));
17 end Test;
```

(a) Synthetic example 1, for $N = M = 3$.

```
1  procedure Test is
2     type T1 is null record;
3     type T2 is null record;
4     type T3 is null record;
5
6     function F (X : T1) return T1 is (null record);
7     function F (X : T1) return T2 is (null record);
8     function F (X : T1) return T3 is (null record);
9
10    function F (X : T2) return T2 is (null record);
11    function F (X : T2) return T3 is (null record);
12
13    function F (X : T3) return T3 is (null record);
14
15    procedure P (X : T1) is null;
16    procedure P (X : T2) is null;
17    procedure P (X : T3) is null;
18
19    X : T1;
20 begin
21    P (F (F (F (X))));
22 end Test;
```

(b) Synthetic example 2, for $N = M = 3$.

Figure 4: Patterns of code exhibiting known pathological solver behavior.



(a) Synthetic case 1 ($M = N$). (b) Synthetic case 2 ($M = 3$). (c) Synthetic case 2 ($M = N$).

Figure 5: Performance measurements on synthetic examples. A logarithmic scale is needed in plots (a) and (b) to visualize both curves on the same plot.

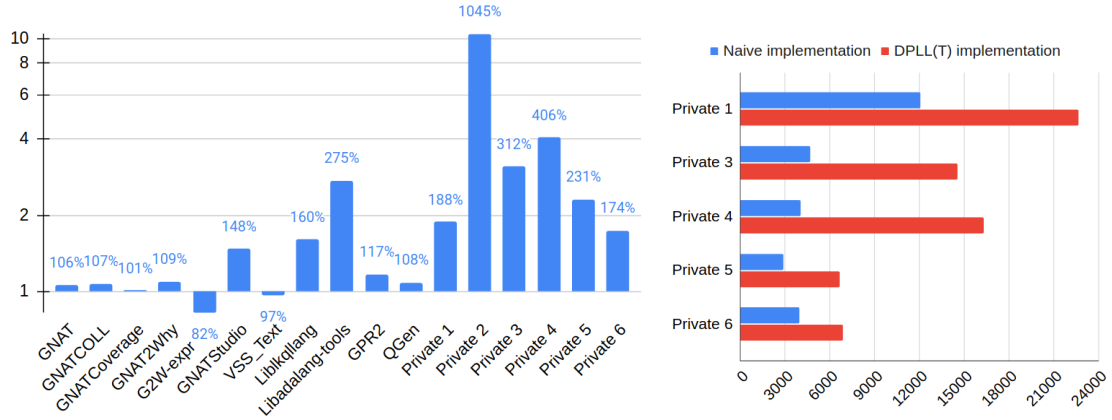
For both cases, we report the total *number of atoms evaluated* by each implementation, which is a deterministic measure of the work done that directly correlates to the actual run-time of the solvers. In 5a, we see that both solvers have polynomial complexity on the first synthetic case, but of very different factors and degrees ($O(N^5)$ versus $O(N^3)$). On the second synthetic case, the naive implementation completely blows up if we let $M = N$. Therefore in 5b, we fixed $M = 3$ and only let N vary. Even then, the naive implementation exhibits a $O(N^6)$ behavior, whereas our new implementation is only quadratic. Finally, in 5c we compare the run-time of our frontend to the run-time of the the main industrial Ada compiler GNAT [12] on that second synthetic example, with $M = N$. This comparison is by nature flawed as we have to compare run-time of the complete frontends including parsing, construction of lexical scopes, etc. However, we could verify that on the tested synthetic cases, name and type resolution do dominate the run-time in our frontend. What this measurement shows is that while our solution has much higher constant factors (probably explained by the generic nature of our framework), the asymptotic run-time of the two frontends is $O(N^4)$.

Real-World Codebases We have stress-tested our new implementation on a variety of actual Ada codebases featuring different coding styles, from internal 50k SLOCs codebases maintained by AdaCore to industrial codebases of several millions SLOCs to code excerpts exhibiting known pathological cases – including instances of those shown above. We measured the total time[2] spent by the frontend to analyze each complete codebase with both the legacy solver backend (t_{old}) and the DPLL(T)-based solver backend (t_{new}). In figure 6a, we plot the ratio t_{old}/t_{new} for each codebase, which shows significant performance improvements[3]. Figure 6b gives an idea of the effective performance of the frontends in terms of number of source lines of code resolved per second on five industrial codebases (averaging 1.8M SLOCs).

Another consequence of the new solver being able to significantly reduce the search space is

[2]Note that since the time spent in the frontend is not exclusively inside the solver, the increase in performance of the solver itself needed to justify these speedups is even greater.

[3]*G2W-expr* is actually one particular source file of the *GNAT2Why* codebase, and is the only instance which we have found to be negatively impacted by the transition. We have yet to investigate the underlying issue.



(a) Ratios of run-time between the new frontends (t_{old}/t_{new}). Logarithmic scale.

(b) Number of lines of code analyzed/s on various industrial codebases.

Figure 6: Performance comparison between the legacy frontend and the DPLL(T)-based frontend on real Ada codebases.

that the transition allowed fixing hundreds of timeouts over several codebases, which were due to our previous solver being stuck in an exponential exploration of the problem space (but the timeouts triggered fast enough to have no notable impact on the reported ratios).

Finally, while it is difficult to make a fair performance comparison between our analyzer and that of GNAT due to major architectural differences, we have found our implementation to be competitive in everyday usage.

3.2. Producing Error-Case Diagnostics

IDEs and language servers [25] being one of the primary targets of our meta-compiler framework, it is important for our name resolution framework to allow emitting helpful error message when analyzing user code that is semantically invalid, so as to implement live feedback features of modern IDEs. With this in mind, we recently extended our logic DSL to allow language designers to annotate their typing rules with minimal efforts so as to convey the information required for producing a relevant error message for each kind of error of their type system.

1. We extended the atom constructors to allow attaching abstract contextual information to them, which we call *logic contexts*. Intuitively, this can be used by the language-independent solver to regain language-specific knowledge, and can be exploited when a candidate solution is rejected to provide insight as to what was being tested.
2. We extended the interface of atoms to allow language designers to specify a location and an error message template that should be instantiated if this atom is part of a conflict which caused a candidate solution to be rejected.

We then extended the behavior of our solver in case it couldn't resolve a given equation, so as to run itself a second time in an *error-reporting* mode. In this mode, it will encounter the same conflicts as the first time, but will use the explanation produced after each conflict and

extract the *logic contexts* of the atoms that we know minimally explain the conflicts. It can then emit an error message by instantiating the failed atom's error message template with the actual solve-time values computed so far, and the extracted logic contexts. The key insight is that since explanations only contain atoms that were relevant for a failure, the error reported to the user will also only contain contexts that are relevant for the failure, instead of for example showing overloads that were selected in other sub-expression which cannot in any way intervene in the failure under investigation.

Note that we prioritize maximal performance on satisfiable equations, as they represent the majority of cases even during live code edition. Thus we favor the overhead of re-running the solver a second time when we find an unsatisfiable instance, rather than the overhead of consistently populating and keeping around all the data structures required to report an error message that in most cases will not be needed.

This work is still in progress (for instance, we cannot emit errors for cases with multiple solutions yet) but we already found concrete scenarios in which error messages produced by our generated Ada frontend were more helpful than messages emitted by the GNAT compiler. This is also a promising lead, using the custom theory we developed, to provide a general framework to produce useful error messages from the specification of type systems by Langkit users.

4. Related Work

Many specific language implementations, such as the aforementioned Swift's Swiftc [4] and Haskell's GHC [3], have a custom unification-based solver as their core. However, such solvers are usually specialized for a given type system, and contain built-in primitives that encode the typing rules of the language they intend to type.

- In the case of Swift, it was shown that it is possible to generate typing problems where the search space is truly pathological [14]. While the design of the solver is not completely formalized, it appears relatively clear from the developer documentation [4] that the solver has a relatively naive design that explores the tree of solutions, similar to the algorithm exposed at the beginning of section 3.
- In the case of GHC, even though Hindley-Milner type inference has an exponential worst case, careful design of the type system seems key in making sure that the solver works with a relatively simple unification algorithm while staying efficient in most of the real-world use cases.
- The Rust compiler team also has a project to reimplement the trait part of their type system using a constraint solver [26] based on Prolog, using a classical depth-first exploration algorithm for solving.

The takeaway is that languages with complex type systems seem to benefit from a solver-based approach where accumulation of constraints is separated from solving the constraint system. In that light, having a generalized approach based on SMT seems like a good foundation for future work on complex type systems.

On the meta-compiler front, the closest related work that we know of is Spoofax [27]. In [28], they present their framework for modeling static semantics of languages. However, they mention

that their framework is not capable of specifying languages with subtyping and type-based overloading. In a more recent paper [29], they address some of those issues, but it seems more limited in several aspects.

- The constraint unification algorithm seems to be a regular unification algorithm, and no effort is yet made to optimize exponential search-space problems, which is the very point of the approach described in this paper.
- No effort was yet made to see whether the approach would scale to real-world use cases. Finally, there is not yet a framework to generate error messages when the constraints are unsatisfiable.

MPS [30] (short for Meta-Programming System) is another very powerful framework and language development workbench by Jetbrains. It covers everything from IDE integration to code generation, and provides a language based on constraints to describe the type system of the object language. While the constraint language seems to be powerful enough to express type systems with subtyping, it seems to have built-in support for a lot of notions, and thus to have a *meta type theory* that is much more rigid than what is allowed by our framework. To the best of our knowledge there is no support for anything other than simple name-resolution rules, and thus anything like type-based overloading resolution seems impossible to express in their framework. Finally, while there is a user guide, there is no formal description of the capabilities of the high level constraint language, nor of the underlying solver, so a lot of observations we are able to make are hard to confirm/infirm.

5. Future Work

So far, Langkit has really been used to define one main type system, and a limited number of minor experiments. One of the big long term goals would be to express other type systems with different paradigms, such as structural-subtyping, or other systems with a lot of inference such as the Swift or Rust type systems, and see how efficient and expressive our framework is for those. In terms of solver optimizations, there are several leads to explore as well:

- Our current solver does not perform theory propagation, which has shown in some cases to be a game changer [31].
- Although we have been able to take advantage of extensive research on SAT solvers to implement various optimizations on our own implementation (conflict analysis, two-watched literals, blocking literals, etc.), we also want to try plugging existing state-of-the-art SAT solver backends, after modifying them to support our extensions.
- We want to experiment encoding some properties of our logic more eagerly, such as dependencies between atoms, in order to minimize the amount of rounds needed to converge.

However these three leads require careful experimentation, as the potential overhead brought by each of them on simple instances may affect overall performance negatively even if the benefit is high on complex cases, as semantic analyzers also spend a lot of time resolving trivial equations.

References

[1] R. Milner, A theory of type polymorphism in programming, J. Comput. Syst. Sci. 17 (1978) 348–375. URL: https://doi.org/10.1016/0022-0000(78)90014-4. doi:10.1016/0022-0000(78)90014-4.

[2] F. Pottier, D. Rémy, The Essence of ML Type Inference, 2005, pp. 389–489.

[3] S. Peyton Jones, Type inference as constraint solving: how GHC's type inference engine actually works, Zurihac keynote talk, 2019. URL: https://www.microsoft.com/en-us/research/publication/type-inference-as-constraint-solving-how-ghcs-type-inference-engine-actually-works/.

[4] Swift type checker design and implementation, https://apple-swift.readthedocs.io/en/latest/TypeChecker.html, 2017.

[5] Langkit, https://github.com/AdaCore/langkit, 2023.

[6] H. Ganzinger, K. Ripken, Operator identification in Ada: Formal specification, complexity, and concrete implementation, ACM Sigplan Notices 15 (1980) 30–42.

[7] T. Pennello, F. DeRemer, R. Meyers, A simplified operator identification scheme for Ada, SIGPLAN Not. 15 (1980) 82–87. URL: https://doi.org/10.1145/947680.947688. doi:10.1145/947680.947688.

[8] T. P. Baker, A one-pass algorithm for overload resolution in ada, ACM Trans. Program. Lang. Syst. 4 (1982) 601–614.

[9] D. A. Mundie, D. A. Fisher, Optimized overload resolution and type matching for ada, Ada Lett. XI (1991) 83–90. URL: https://doi.org/10.1145/112630.112640. doi:10.1145/112630.112640.

[10] D. Zhang, A. C. Myers, D. Vytiniotis, S. L. P. Jones, Diagnosing type errors with class, in: D. Grove, S. M. Blackburn (Eds.), Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, ACM, 2015, pp. 12–21. URL: https://doi.org/10.1145/2737924.2738009. doi:10.1145/2737924.2738009.

[11] Libadalang, https://github.com/AdaCore/libadalang, 2023.

[12] GNAT: The GNU Ada compiler, https://files.adacore.com/gnat-book/, 2004.

[13] Lambda expressions vs. anonymous methods, part five, https://learn.microsoft.com/en-us/archive/blogs/ericlippert/lambda-expressions-vs-anonymous-methods-part-five, 2007.

[14] Exponential time complexity in the Swift type checker, https://www.cocoawithlove.com/blog/2016/07/12/type-checker-issues.html, 2016.

[15] Dependz, https://github.com/Roldak/Dependz, 2020.

[16] G. Brewka, S. Benferhat, D. L. Berre, Qualitative choice logic, Artif. Intell. 157 (2004) 203–237. URL: https://doi.org/10.1016/j.artint.2004.04.006. doi:10.1016/j.artint.2004.04.006.

[17] G. Brewka, Logic programming with ordered disjunction, CoRR cs.AI/0207042 (2002). URL: https://arxiv.org/abs/cs/0207042.

[18] R. Confalonieri, J. Nieves, Nested Logic Programs with Ordered Disjunction, volume 677, 2010, pp. 55–66.

[19] Ada 2022 reference manual, chapter 8.6, paragraph 29, http://www.ada-auth.org/standards/22rm/html/RM-8-6.html, 2022.

[20] R. Nieuwenhuis, A. Oliveras, C. Tinelli, Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T), J. ACM 53 (2006) 937–977. URL: https://doi.org/10.1145/1217856.1217859. doi:10.1145/1217856.1217859.
[21] V.-H. Nguyen, V.-Q. Nguyen, K. Kim, P. Barahona, Empirical study on SAT-encodings of the at-most-one constraint, in: The 9th International Conference on Smart Media and Applications, SMA 2020, Association for Computing Machinery, New York, NY, USA, 2021, p. 470–475. URL: https://doi.org/10.1145/3426020.3426170. doi:10.1145/3426020.3426170.
[22] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in: Proceedings of the 38th Annual Design Automation Conference, DAC '01, Association for Computing Machinery, New York, NY, USA, 2001, p. 530–535. URL: https://doi.org/10.1145/378239.379017. doi:10.1145/378239.379017.
[23] J. Maglalang, Native cardinality constraints: More expressive, more efficient constraints (2012).
[24] M. H. Liffiton, J. C. Maglalang, A cardinality solver: More expressive constraints for free, in: A. Cimatti, R. Sebastiani (Eds.), Theory and Applications of Satisfiability Testing – SAT 2012, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 485–486.
[25] J. K. Rask, F. P. Madsen, N. Battle, H. D. Macedo, P. G. Larsen, The specification language server protocol: A proposal for standardised LSP extensions, in: J. Proença, A. Paskevich (Eds.), Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021, volume 338 of *EPTCS*, 2021, pp. 3–18. URL: https://doi.org/10.4204/EPTCS.338.3. doi:10.4204/EPTCS.338.3.
[26] The Chalk book, https://rust-lang.github.io/chalk/book/, 2023.
[27] L. C. L. Kats, E. Visser, The spoofax language workbench, in: W. R. Cook, S. Clarke, M. C. Rinard (Eds.), Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, part of SPLASH 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA, ACM, 2010, pp. 237–238. URL: https://doi.org/10.1145/1869542.1869592. doi:10.1145/1869542.1869592.
[28] H. Antwerpen, P. Neron, A. Tolmach, E. Visser, G. Wachsmuth, A constraint language for static semantic analysis based on scope graphs, 2016, pp. 49–60. doi:10.1145/2847538.2847543.
[29] H. van Antwerpen, C. B. Poulsen, A. Rouvoet, E. Visser, Scopes as types, Proc. ACM Program. Lang. 2 (2018) 114:1–114:30. URL: https://doi.org/10.1145/3276484. doi:10.1145/3276484.
[30] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, A. Pierantonio (Eds.), Domain-Specific Languages in Practice: with JetBrains MPS, Springer, 2021. URL: https://doi.org/10.1007/978-3-030-73758-0. doi:10.1007/978-3-030-73758-0.
[31] R. Nieuwenhuis, A. Oliveras, DPLL(T) with exhaustive theory propagation and its application to difference logic, volume 3576, 2005, pp. 321–334. doi:10.1007/11513988_33.