Noname manuscript No.
(will be inserted by the editor)

ICE-based Refinement Type Discovery for Higher-Order Functional Programs

Adrien Champion · Tomoya Chiba · Naoki Kobayashi · Ryosuke Sato

the date of receipt and acceptance should be inserted later

Abstract We propose a method for automatically finding refinement types of higher-order function programs. Our method is an extension of the ICE framework of Garg *et al.* for finding invariants. In addition to the usual positive and negative samples in machine learning, their ICE framework uses *implication constraints*, which consist of pairs (x, y) such that if x satisfies an invariant, so does y. From these constraints, ICE infers *inductive* invariants effectively. We observe that the implication constraints in the original ICE framework are not suitable for finding invariants of recursive functions with multiple function calls. We thus generalize the implication constraints to those of the form $(\{x_1, \ldots, x_k\}, y)$, which means that if all of $x_1, \ldots, x_k$ satisfy an invariant, so does y. We extend their algorithms for inferring likely invariants from samples, verifying the inferred invariants, and generating new samples. We have implemented our method and confirmed its effectiveness through experiments.

Keywords Higher-Order Program Verification · Machine-Learning · Formal Verification · Refinement Types

1 Introduction

Higher-order functional program verification is an interesting and challenging problem. Over the past two decades, several approaches have been proposed: refinement types with manual annotations [11,33], liquid types [24], and reduction to higher-order recursion schemes [26]. These approaches face the same problem found in imperative and synchronous data-flow program verification: the need for predicates describing how loops and components behave for the verification and/or abstraction method to work in practice [8, 13, 18]. This paper

Adrien Champion
The University of Tokyo, now OCamlPro
E-mail: adrien.champion@email.com

Tomoya Chiba, Naoki Kobayashi
The University of Tokyo
E-mail: tomo.asleep@gmail.com, E-mail: koba@is.s.u-tokyo.ac.jp

Ryosuke Sato
Kyushu University
E-mail: sato@ait.kyushu-u.ac.jp

proposes to address this issue by combining refinement types with the recent machine-learning-based, invariant discovery framework Ice from [12, 13].

Consider for instance a function `f` from integers to integers such that if its input `n` is less than or equal to 101, then its output is 91, otherwise it is `n` − 10. (This is the case of the `mc_91` function on Figure 1.) Then our objective is to automatically discover, by using an *adaptation* of Ice, the refinement type

$$\texttt{f} \;:\; \{n : \text{int} \mid \mathit{true}\} \;\rightarrow\; \{r : \text{int} \mid (n > 101 \wedge r = n - 10) \;\vee\; r = 91\}.$$

That is, function `f` accepts any integer n that satisfies *true* as input, and yields an integer r equal to $n - 10$ when $n > 101$, and equal to 91 otherwise. The traditional Ice framework is not appropriate for our use-case. We briefly summarize it below, and then discuss how this approach needs to be extended for the purpose of functional program verification.

Brief review of the Ice framework.
Let $\mathscr{S}$ be a transition system $\langle\ \vec{s}, \mathcal{I}(\vec{s}), \mathcal{T}(\vec{s}, \vec{s}')\ \rangle$, with $\vec{s}$ its vector of state variables, $\mathcal{I}(\vec{s})$ its initial predicate, and $\mathcal{T}(\vec{s}, \vec{s}')$ the transition relation between consecutive states. Suppose we wish to prove that $\mathit{Prop}(\vec{s})$ is an invariant, i.e., that a property $\mathit{Prop}(\vec{s})$ holds for any state $\vec{s}$ reachable from an initial state. Then it suffices to find a predicate $\mathit{Inv}(\vec{s})$ that satisfies the following conditions.

$$\mathcal{I}(\vec{s}) \models \mathit{Inv}(\vec{s}) \tag{1}$$
$$\mathit{Inv}(\vec{s}) \;\wedge\; \mathcal{T}(\vec{s}, \vec{s}') \models \mathit{Inv}(\vec{s}') \tag{2}$$
$$\mathit{Inv}(\vec{s}) \models \mathit{Prop}(\vec{s}) \tag{3}$$
$$\tag{4}$$

The predicate $\mathit{Inv}(\vec{s})$ is an invariant that is *inductive* in that it is preserved by the transition relation, as guaranteed by (2). We call such an $\mathit{Inv}(\vec{s})$ a *strengthening inductive invariant* for $\mathit{Prop}(\vec{s})$. It serves as a *certificate* that $\mathit{Prop}(\vec{s})$ is a (plain) invariant. Given a candidate for $\mathit{Inv}(\vec{s})$, the conditions (1)–(2) can be checked by an SMT [2] solver. In the rest of this section, “invariant” will always mean “strengthening inductive invariant”.

The Ice framework is a machine-learning-based method combining a *learner* that incrementally produces candidate invariants, and a *teacher* that checks whether the candidates are such that (1), (3) and (2) hold. If a given candidate is not an invariant, the teacher produces *learning data* as follows, so that the learner can produce a better candidate. A candidate is an arbitrary Boolean combination of atomic predicates called *qualifiers*. Given a candidate $C_k(\vec{s})$, the teacher checks whether (1) holds — using an SMT solver for instance. If it does not, a concrete state $\vec{e}$ is extracted and will be given to the learner as an *example*: the next candidate C_{k+1} should be such that $C_{k+1}(\vec{e})$ holds, *i.e.* it must *include* the example. Conversely, if (3) does not hold, a concrete state $\vec{c}$ is extracted and will be given as a *counterexample*: the next candidate should be such that $C_{k+1}(\vec{c})$ does not hold, *i.e.* it must *exclude* the counterexample.

Unlike traditional machine-learning approaches, in Ice the teacher also extracts learning data from (2) when it does not hold. It takes the form of a pair of (consecutive) concrete states $(\vec{i}, \vec{i}')$, and is called an *implication constraint*: the next candidate should be such that $C_{k+1}(\vec{i}) \Rightarrow C_{k+1}(\vec{i}')$. Implication constraints are crucial for the learner to discover inductive invariants, as they let it know why its current candidate failed the induction check. The Ice framework does not specify how the learner generates candidates, but this is typically done by building a *classifier* consistent with the learning data, in the form of a decision tree — discussed further in Section 3.

```
let rec mc_91 n = if n > 100 then n - 10
                  else let tmp = mc_91 (n + 11) in mc_91 tmp
let main m =
  let res = mc_91 m in if m ≤ 101 then assert (res = 91)
```

Fig. 1: McCarthy's 91 function.

Refinement type inference as a predicate synthesis problem.
We now discuss why the original Ice framework is ill-suited for functional program verification. Consider McCarthy's 91 function from Figure 1. To prove this program correct in a refinement type setting, it is enough to find some refinement type

$$\{n : \text{int} \mid \rho_1(n)\} \;\rightarrow\; \{r : \text{int} \mid \rho_2(n, r)\}$$

for `mc_91`, where ρ_1 and ρ_2 are such that[1]

$$\rho_1(n) \;\wedge\; n > 100 \;\wedge\; r = n - 10 \;\models\; \rho_2(n, r) \tag{5}$$
$$\rho_1(n) \;\wedge\; n \leq 100 \;\models\; \rho_1(n + 11) \tag{6}$$
$$\rho_1(n) \;\wedge\; n \leq 100 \;\wedge\; \rho_2(n + 11, \mathit{tmp}) \;\models\; \rho_1(\mathit{tmp}) \tag{7}$$
$$\rho_1(n) \;\wedge\; n \leq 100 \;\wedge\; \rho_2(n + 11, \mathit{tmp}) \;\wedge\; \rho_2(\mathit{tmp}, r) \;\models\; \rho_2(n, r) \tag{8}$$
$$\mathit{true} \;\models\; \rho_1(m) \tag{9}$$
$$m \leq 101 \;\wedge\; \rho_2(m, \mathit{res}) \;\models\; \mathit{res} = 91 \tag{10}$$

We can observe some similarities between the Horn clauses above and (1)–(2). The constraints (9) and (10) respectively correspond to the constraints (1) and (3) on initial states and the property to be proved, whereas the constraints (5)–(8) correspond to the induction constraint (2). This observation motivates us to reuse the Ice framework for refinement type inference.

There are, however, two obstacles in adapting the Ice framework to refinement type inference. First, we must infer not one but several mutually-dependent predicates. Second, and more importantly, we need to generalize the notion of implication constraint because of the nested recursive calls found in functional programs. To illustrate, let us assume that we realized that `mc_91`'s precondition is $\rho_1(n) = \mathit{true}$. Then the third constraint from the `else` branch is

$$n \leq 100 \;\wedge\; \rho_2(n + 11, \mathit{tmp}) \;\wedge\; \rho_2(\mathit{tmp}, r) \;\models\; \rho_2(n, r).$$

Contrary to the ones found in the original Ice framework, this Horn clause is *non-linear*: it has more than one application of the same predicate (ρ_2, here) in its antecedents. Now, assuming we have a candidate for which this constraint is falsifiable, the implication constraint should have form $(\ \{(n_1, r_1),\ (n_2, r_2)\}, (n, r)\ )$, which means that the next candidate C should be such that $C(n_1, r_1) \wedge C(n_2, r_2) \;\Rightarrow\; C(n, r)$. This is because there are two occurrences of ρ_2 on the left-hand side of the implication.

The need to infer more than one predicate and support non-linear Horn clauses is not specific to higher-order functional program verification. After all, McCarthy's 91 function is first-order and is occasionally mentioned in first-order imperative program verification papers [4]. Sv-Comp [3], the main (imperative) software verification competition features 3247 verification problems in its linear arithmetic track which can be encoded as Horn clauses, 54 of which contain non-linear Horn clauses. In our context of higher-order functional program

[1] We discuss how to extract these verification conditions in Section 2.

verification the ratio is much higher, with 63 of our 164 OCaml [21] programs yielding non-linear Horn clauses.

The main contribution of this paper is to address the two issues aforementioned and propose a modified Ice framework suitable for higher-order program verification in particular. While adapting machine-learning techniques to higher-order program verification has been done before [37, 38], transposing implication constraints to this context is, to the best of our knowledge, new work. We also present various simplifications/optimizations for the encoding of the problem and the modified Ice framework, which prove extremely useful in practice. We have implemented our approach as a program verifier for a subset of OCaml and report on our experiments.

The rest of the paper is organized as follows. Section 2 introduces our target language and describes verification condition generation and simplification. The modified Ice framework is discussed in Section 3. We report on our implementation and experiments of the approach in Section 4. Section 5 describes and evaluates ongoing work for adapting our approach to Algebraic Data Types. Finally, we discuss related work in Section 6 before concluding in Section 7.

This article is an extended version of previous work [6]. This version adds information and examples that make the discussion more understandable, as well as a completely new section (Section 5) discussing preliminary work on adapting our approach to Algebraic Data Types. We implemented this work in our Horn clause solver HoIce [7] and report on our experimental evaluation.

2 Target Language and Verification Conditions

In this section, we first introduce the target language of our refinement type inference method. We then introduce a refinement type system and associated verification conditions (i.e., sufficient conditions for the typability of a given program).

2.1 Language

The target of the method is a simply-typed, call-by-value, higher-order functional language with recursion. Its syntax is given by:

$$\begin{aligned}
\mathbb{P}\ \text{(programs)} &::= \{f_1(\widetilde{z_1}) = e_1, \dots, f_n(\widetilde{z_n}) = e_n\}\\
e\ \text{(expressions)} &::= n \mid x \mid \oplus\{a_1 \Rightarrow e_1, \dots, a_n \Rightarrow e_n\} \mid \texttt{fail}\\
&\quad\ \mid \texttt{let}\ x = * \ \texttt{in}\ e \mid \texttt{let}\ x = a\ \texttt{in}\ e \mid \texttt{let}\ x = y\ z\ \texttt{in}\ e\\
a\ \text{(arith. expressions)} &::= n \mid x \mid op(a_1, a_2) \qquad v\ \text{(values)} ::= n \mid f_i\,\widetilde{v}\\
\tau\ \text{(simple types)} &::= \texttt{int} \mid \tau_1 \to \tau_2
\end{aligned}$$

We use the meta-variables $x, y, \dots, f, g, \dots$ for variables. We write $\widetilde{\cdot}$ for a sequence; for example, we write $\widetilde{x}$ for a sequence of variables. For the sake of simplicity, we consider only integers as base values. We represent Booleans using integers, and treat 0 as false and non-zero values as true. We sometimes write *true* for 1 and *false* for 0.

We briefly explain programs and expressions; the formal semantics is given later. We use let-normal-form-style for simplicity. A program $\mathbb{P}$ is a set of mutually recursive function definitions $f(\widetilde{z}) = e$. The expression $\oplus\{a_i \Rightarrow e_i\}_{1\leq i\leq n}$ evaluates e_i non-deterministically if

$$\frac{}{E[\texttt{let}\ x = *\ \texttt{in}\ e] \longrightarrow_{\mathbb{P}} E[[n/x]e]} \qquad \frac{n \text{ is the value of } a}{E[\texttt{let}\ x = a\ \texttt{in}\ e] \longrightarrow_{\mathbb{P}} E[[n/x]e]}$$

$$\frac{f(\widetilde{y}) = e' \in \mathbb{P} \qquad |\widetilde{v}| < |\widetilde{y}|}{E[\texttt{let}\ x = f\,\widetilde{v}\ \texttt{in}\ e] \longrightarrow_{\mathbb{P}} E[[f\,\widetilde{v}/x]e]} \qquad \frac{f(\widetilde{y}) = e' \in \mathbb{P} \qquad |\widetilde{v}| = |\widetilde{y}|}{E[\texttt{let}\ x = f\,\widetilde{v}\ \texttt{in}\ e] \longrightarrow_{\mathbb{P}} E[\texttt{let}\ x = [\widetilde{v}/\widetilde{y}]e'\ \texttt{in}\ e]}$$

$$\frac{E \neq [\,]}{E[\texttt{fail}] \longrightarrow_{\mathbb{P}} \texttt{fail}} \qquad \frac{\text{the value of } a_i \text{ is non-zero}}{E[\oplus\{a_1 \Rightarrow e_1, \dots, a_n \Rightarrow e_n\}] \longrightarrow_{\mathbb{P}} E[e_i]}$$

$$E ::= [\,] \mid \texttt{let}\ x = E\ \texttt{in}\ e$$

Fig. 2: Operational semantics of the language

the value of a_i is non-zero, which can be also used to generate non-deterministic Booleans/integers. We also write $(a_1 \Rightarrow e_1) \oplus \cdots \oplus (a_n \Rightarrow e_n)$ for $\oplus\{a_i \Rightarrow e_i\}_{1 \le i \le n}$, and write $\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ for $(a \Rightarrow e_1) \oplus (\neg a \Rightarrow e_2)$. The expression $\texttt{let}\ x = *\ \texttt{in}\ e$ generates an integer, then binds x to it, and evaluates e. The expression $\texttt{let}\ x = a\ \texttt{in}\ e$ ($\texttt{let}\ x = y\ z\ \texttt{in}\ e$, resp.) binds x to the value of a ($y\ z$, resp.), and then evaluates e. The expression `fail` aborts the program. An assert expression $\texttt{assert}(a)$ can be represented as $\texttt{if}\ a\ \texttt{then}\ 0\ \texttt{else fail}$. An arithmetic expression consists of an integer constant, an (integer) variable, and primitive operators, denoted by *op*; we assume that the set of primitive operators contains standard integer/Boolean operations/relations like $+, <, \vee, \wedge, \cdots$. A value v is either an integer constant or a function closure; the latter is a partial function application of the form $f_i\,\widetilde{v}$; here, the length $|\widetilde{v}|$ of arguments $\widetilde{v}$ must be (strictly) smaller than $|\widetilde{x_i}|$ where $(f_i(\widetilde{x_i}) = e_i) \in \mathbb{P}$.

We assume that a program is well-typed under the standard simple type system. We also assume that every function in $\mathbb{P}$ has a non-zero arity, the body of each function definition has the integer type, and $\mathbb{P}$ contains a distinguished function symbol $\texttt{main} \in \{f_1, \dots, f_n\}$ whose simple type is $\texttt{int} \to \texttt{int}$.

The operational semantics of the target language is given on Figure 2, where we extend the syntax of expressions with $\texttt{let}\ x = v\ \texttt{in}\ e$ and $\texttt{let}\ x = e'\ \texttt{in}\ e$. The goal of our verification is to find an invariant (represented in the form of refinement types) of the program that is sufficient to verify that, for every integer n, $\texttt{main}\ n$ does not fail (i.e., is not reduced to `fail`).

2.2 Refinement Type System

We present a refinement type system for the target language. The syntax of refinement types is given by:

$$T(\text{refinement types}) ::= \{x : \texttt{int} \mid a\} \mid (x : T_1) \to T_2.$$

The refinement type $\{x{:}\texttt{int} \mid a\}$ denotes the set of integers that satisfy a, *i.e.*, the value of a is non-zero. For example, $\{x : \texttt{int} \mid x \geq 0\}$ represents natural numbers. The type $(x : T_1) \to T_2$ denotes the set of functions that take an argument x of type T_1 and return a value of type T_2. Here, note that x may occur in T_2. We write $\texttt{int}$ for $\{x : \texttt{int} \mid \mathit{true}\}$, and $T_1 \to T_2$ for $(x : T_1) \to T_2$ when x does not occur in T_2. By abuse of notation, we sometimes (as in Section 1) write $\{x : \texttt{int} \mid a\} \to T$ for $(x : \{x : \texttt{int} \mid a\}) \to T$.

Figure 3 shows the typing rules, which are the standard ones. We have three kinds of type judgments: $\Gamma \vdash t : T$ for expressions, $\vdash \mathbb{P} : \Gamma$ for programs, and $\Gamma \vdash T <: T'$ for subtyping. A

$$\frac{}{\Gamma \vdash n : \{x : \mathtt{int} \mid x = n\}}\ (\text{T-Const}) \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ (\text{T-Var}) \qquad \frac{[\![\Gamma]\!] \models \mathit{false}}{\Gamma \vdash \mathtt{fail} : T}\ (\text{T-Fail})$$

$$\frac{\Gamma, a_i \vdash e_i : T \quad \text{for each } i \in \{1,\ldots,n\}}{\Gamma \vdash \oplus\{a_1 \Rightarrow e_1, \ldots, a_n \Rightarrow e_n\} : T}\ (\text{T-Branch}) \qquad \frac{\Gamma, x : \mathtt{int} \vdash e : T}{\Gamma \vdash \mathtt{let}\ x = *\ \mathtt{in}\ e : T}\ (\text{T-Rand})$$

$$\frac{\Gamma, x : \{y : \mathtt{int} \mid y = a\} \vdash e : T}{\Gamma \vdash \mathtt{let}\ x = a\ \mathtt{in}\ e : [a/x]T}\ (\text{T-AExp}) \qquad \frac{\Gamma \vdash e : T' \qquad \Gamma \vdash_s T' <: T}{\Gamma \vdash e : T}\ (\text{T-Sub})$$

$$\frac{[\![\Gamma]\!], a_1 \models a_2}{\Gamma \vdash_s \{x : \mathtt{int} \mid a_1\} <: \{x : \mathtt{int} \mid a_2\}}\ (\text{S-Int}) \qquad \frac{\Gamma \vdash_s T_{21} <: T_{11} \qquad \Gamma, x : T_{21} \vdash_s T_{12} <: T_{22}}{\Gamma \vdash_s (x : T_{11}) \to T_{12} <: (x : T_{21}) \to T_{22}}\ (\text{S-Fun})$$

$$\frac{\Gamma \vdash y : (z : T_1) \to T_2 \qquad \Gamma \vdash z : T_1 \qquad \Gamma, x : T_2 \vdash e : T}{\Gamma \vdash \mathtt{let}\ x = y\ z\ \mathtt{in}\ e : T}\ (\text{T-App})$$

$$\frac{\begin{array}{c}\Gamma(\mathtt{main}) = (x : \mathtt{int}) \to \mathtt{int} \\ x_1 : T_1, \ldots, x_k : T_k \vdash e : T \text{ for each } f : (x_1 : T_1) \to \cdots \to (x_k : T_k) \to T \in \Gamma \qquad \text{where } f(x_1, \ldots, x_k) = e \in \mathbb{P}\end{array}}{\vdash \mathbb{P} : \Gamma}$$

(T-Prog)

$$[\![\emptyset]\!] = \mathit{true}, \qquad [\![\Gamma, x : \{y : \mathtt{int} \mid a\}]\!] = [\![\Gamma]\!] \wedge [x/y]a, \qquad [\![\Gamma, a]\!] = [\![\Gamma]\!] \wedge a, \qquad [\![\Gamma, x : (y : T_1) \to T_2]\!] = [\![\Gamma]\!]$$

Fig. 3: Typing rules of refinement type system

judgment $\Gamma \vdash e : T$ means that the expression e has the refinement type T under refinement type environment Γ, which is a sequence of refinement type bindings and guard predicates: $\Gamma \ ::= \ \emptyset \mid \Gamma, x : T \mid \Gamma, a$. Here, $x : T$ means that x has refinement type T, and a means that a holds. When $\Gamma = \Gamma_1, x : T, \Gamma_2$ where Γ_2 does not contain a binding of the form $x : T'$, we write $\Gamma(x)$ for T. A judgment $\vdash \mathbb{P} : \Gamma$ means that the program $\mathbb{P}$ is well-typed, where Γ describes the type of each function defined in $\mathbb{P}$. A subtyping judgment $\Gamma \vdash T <: T'$ means that a value of type T may be used as a value of type T'. In the rules, we implicitly assume that all the variables occurring in an arithmetic expression a have type `int`. Though typing rules are fairly standard, we explain a few typing rules. In rule T-Fail, $[\![\Gamma]\!]$ expresses the constraint implied by the type environment Γ. The premise $[\![\Gamma]\!] \models \mathit{false}$ ensures that there is no environment that conforms to Γ, so that `fail` is unreachable. In rule T-AExp, the information that x is bound to a is propagated to the type of x. Since the type T of e may contain x, we substitute a for x in the conclusion. The rule T-Sub allows the type of an expression to be weakened. For example, if we have $\Gamma \vdash e : \{x : \mathtt{int} \mid x = 1\}$, it can be weakened to $\Gamma \vdash e : \{x : \mathtt{int} \mid x > 0\}$ by using T-Sub.

The type system is sound in the sense that if $\vdash \mathbb{P} : \Gamma$ holds for some Γ, then `main` n does not fail for any integer n. We omit to prove this type system sound as it is a rather well-known system [24, 30, 31]. The type system is, however, incomplete: there are programs that never fail but are not typable in the refinement type system. Implicit parameters are required to make the type system complete [32].

2.3 Verification Conditions

Our goal has now been reduced to finding Γ such that $\vdash \mathbb{P} : \Gamma$, if such Γ exists. To this end, we first infer simple types for the target program by using the Hindley-Milner type inference algorithm. From the simple types, we construct a template for the refinement type environment Γ, by adding predicate variables, and then generate the verification conditions, *i.e.*, constraints on the predicate variables that describe a sufficient condition for $\vdash \mathbb{P} : \Gamma$. The construction of the verification conditions is also rather standard [24, 30, 31], we present it

$$
\begin{aligned}
VC_e(\Gamma \vdash x : T) &= VC_{<:}(\Gamma \vdash_s \Gamma(x) <: T) \\
VC_e(\Gamma \vdash n : T) &= VC_{<:}(\Gamma \vdash_s \{x : \mathtt{int} \mid x = n\} <: T) \\
VC_e(\Gamma \vdash \oplus\{a_1 \Rightarrow e_1, \dots, a_n \Rightarrow e_n\} : T) &= \textstyle\bigwedge_{i\in\{1,\dots,n\}} VC_e(\Gamma, a_i \vdash e_i : T) \\
VC_e(\Gamma \vdash \mathtt{fail} : T) &= [\![\Gamma]\!] \Rightarrow \mathit{false} \\
VC_e(\Gamma \vdash \mathtt{let}\ x = * \ \mathtt{in}\ e : T) &= VC_e(\Gamma, x : \mathtt{int} \vdash e : T) \\
VC_e(\Gamma \vdash \mathtt{let}\ x = a\ \mathtt{in}\ e : T) &= VC_e(\Gamma, x : \{y : \mathtt{int} \mid y = a\} \vdash e : T) \\
VC_e(\Gamma \vdash \mathtt{let}\ x = y\ z\ \mathtt{in}\ e : T) &= VC_{<:}(\Gamma \vdash_s \Gamma(z) <: T_1) \wedge VC_e(\Gamma, x : T_2 \vdash e : T) \\
&\quad \text{where } (z : T_1) \to T_2 = \Gamma(y) \\
VC_{<:}(\Gamma \vdash_s \{x : \mathtt{int} \mid a_1\} <: \{x : \mathtt{int} \mid a_2\}) &= ([\![\Gamma]\!] \wedge a_1) \Rightarrow a_2 \\
VC_{<:}(\Gamma \vdash_s (x : T_{11}) \to T_{12} <: (x : T_{21}) \to T_{22}) &= VC_{<:}(\Gamma \vdash_s T_{21} <: T_{11}) \wedge VC_{<:}(\Gamma, x : T_{21} \vdash_s T_{12} <: T_{22})) \\
VC_f(\Gamma \vdash f(x_1, \dots, x_k) = e) &= VC_e(\Gamma, x_1 : T_1, \dots, x_k : T_k \vdash e : T) \\
&\quad \text{where } \Gamma(f) = (x_1 : T_1) \to \cdots \to (x_k : T_k) \to T \\
VC(\vdash \{f_1(\widetilde{x_1}) = e_1, \dots, f_n(\widetilde{x_n}) = e_n\} : \Gamma) &= \ VC_{<:}(\ \vdash_s \Gamma(\mathtt{main}) <: (x : \mathtt{int}) \to \mathtt{int}) \\
&\quad \wedge \textstyle\bigwedge_{i\in\{1,\dots,n\}} VC_f\big(\Gamma \vdash f_i(\widetilde{x_i}) = e_i\big)
\end{aligned}
$$

Fig. 4: Verification condition generation

here in the form of a function $VC(\vdash \mathbb{P} : \Gamma)$ defined on Figure 4. We note that the verification conditions generated by the function $VC(\vdash \mathbb{P}{:}\Gamma)$ can be normalized to a set of Horn clauses [4].

In Figure 4, $VC(\vdash \mathbb{P} : \Gamma)$ is defined by using three sub-procedures VC_f, $VC_{<:}$, and VC_e. The procedure $VC_f(\Gamma \vdash f(x_1, \dots, x_k) = e)$ generates a condition for the function definition $f(x_1, \dots, x_k) = e)$ to be well-typed. The procedures $VC_{<:}(\Gamma \vdash_s T <: T')$ and $VC_e(\Gamma \vdash e : T)$ respectively generate conditions for $\Gamma \vdash_s T <: T'$ and $\Gamma \vdash e : T$ to be derivable by the rules in Figure 3. Each of the computation rules for $VC_{<:}(\Gamma \vdash_s T <: T')$ and $VC_e(\Gamma \vdash e : T)$ follows from the corresponding typing rule in Figure 3.

Example 1 Consider the following program and its associated simple types:

```
let incr n = n + 1 in let twice f x = f (f x) in
let main m = assert (twice incr m > m)
```

$$\mathtt{main} : \mathtt{int} \to \mathtt{int}, \quad \mathtt{incr} : \mathtt{int} \to \mathtt{int}, \quad \mathtt{twice} : (\mathtt{int} \to \mathtt{int}) \to \mathtt{int} \to \mathtt{int}$$

By assigning a unique predicate variable to each integer type, we can obtain the following refinement type templates.

$$
\begin{aligned}
&\mathtt{main} : \mathtt{int} \to \mathtt{int}, \qquad \mathtt{incr} : \{n : \mathtt{int} \mid \rho_1(n)\} \to \{k : \mathtt{int} \mid \rho_2(n, k)\}, \\
&\mathtt{twice} : (\{y : \mathtt{int} \mid \rho'_1(y)\} \to \{z : \mathtt{int} \mid \rho'_2(y, z)\}) \to \{x : \mathtt{int} \mid \rho'_3(x)\} \to \{r : \mathtt{int} \mid \rho'_4(x, r)\}.
\end{aligned}
$$

We then extract the following verification conditions from the body of the program:

$$
\begin{array}{rclrclrcl}
\rho_1(n) & \models & \rho_2(n, n+1) & \rho'_3(x) & \models & \rho'_1(x) & \rho'_3(x) \wedge \rho'_2(x, z_1) & \models & \rho'_1(z_1) \\
\multicolumn{4}{r}{\rho'_3(x) \wedge \rho'_2(x, z_1) \wedge \rho'_2(z_1, z_2)} & \models & \rho'_4(x, z_2) & \rho'_1(n) & \models & \rho_1(n) \\
\mathit{true} & \models & \rho'_3(m) & \rho'_4(m, r) & \models & r > m & \rho'_1(y) \wedge \rho_2(y, z) & \models & \rho'_2(y, z).
\end{array}
$$

2.4 Simplifying Verification Conditions

The number of unknown predicates to infer is critical to the efficiency of our algorithm in Section 3, because the algorithm succeeds only when the learner comes up with correct solutions for *all* the unknown predicates. We discuss here a couple of techniques to reduce the number of unknown predicates.

The first one takes place at the level of Horn clauses and is not limited to refinement type inference over functional programs. Suppose that some predicate ρ occurs in the clauses $\varphi \models \rho$ and $C[\rho] \models \varphi'$, where $C[\rho]$ is a formula having only positive occurrences of ρ, and ρ does not occur in φ, φ', nor any other clauses of the verification condition. Then, we can replace the two clauses above with $C[\varphi] \models \varphi'$ and $\rho \equiv \varphi$. For example, recall the `incr` / `twice` from the example above. The predicate ρ_1 occurs only in the clauses $\rho_1'(n) \models \rho_1(n)$ and $\rho_1(n) \models \rho_2(n, n+1)$. Thus, we can replace them with $\rho_1'(n) \models \rho_2(n, n+1)$ and $\rho_1(n) \equiv \rho_1'(n)$. In this manner we can reduce the number of unknown predicate variables. This optimization itself is not specific to our context of functional program verification; similar (and more sophisticated) techniques are also discussed in [4]. We found this optimization particularly useful in our context, because the standard verification condition generation for higher-order functional programs discussed above introduces too many predicate variables.

The other optimization is specific to our context of refinement type inference. Suppose that the simple type of a function f is $\mathtt{int} \to \mathtt{int}$. Then, in general, we prepare the refinement type template $\{x : \mathtt{int} \mid \rho_1(x)\} \to \{r : \mathtt{int} \mid \rho_2(x, r)\}$. If the evaluation of $f(n)$ does not fail for any integer n, however, then the above refinement type is equivalent to $\{x : \mathtt{int} \mid \mathit{true}\} \to \{r : \mathtt{int} \mid \rho_1(x) \Rightarrow \rho_2(x, r)\}$. Thus, the template $(x : \mathtt{int}) \to \{r : \mathtt{int} \mid \rho_3(x, r)\}$ suffices, with $\rho_3(x, r)$ encoding the $\rho_1(x) \Rightarrow \rho_2(x, r)$ we had previously, resulting in fewer predicates to infer. For instance, in the `mc_91` example from Section 1, it is obvious that $\mathtt{mc\_91}(n)$ never fails as its body contains no assertions and contains only calls to itself. Thus, we can actually set $\rho_1(n)$ to *true*.

In practice we use effect analysis [23] to check whether a function can fail. To this end, we extend simple types to effect types defined by: $\sigma \ ::= \ \mathtt{int} \mid \sigma_1 \xrightarrow{\xi} \sigma_2$, where ξ is either an empty effect ϵ, or a failure $\mathtt{f}$. The type $\sigma_1 \xrightarrow{\xi} \sigma_2$ describes functions that take an argument of type σ_1 and return a value of type σ_2, but with a possible side effect of ξ. We can infer these effect types using a standard effect inference algorithm [23]. A function with effect type $\mathtt{int} \xrightarrow{\epsilon} \sigma$ takes an integer as input and returns a value of σ without effect, i.e., without failure. For this type, we then use the simpler refinement type template $\{x : \mathtt{int} \mid \mathit{true}\} \to \cdots$ instead of $\{x : \mathtt{int} \mid \rho(x)\} \to \cdots$. For example, since `mc_91` has effect type $\mathtt{int} \xrightarrow{\epsilon} \mathtt{int}$, we assign the template $(x : \mathtt{int}) \to \{r : \mathtt{int} \mid \rho(x, r)\}$ for the refinement type of `mc_91`.

3 Modified Ice Framework

This section discusses our modified Ice framework tackling the predicate synthesis problem extracted from the input functional program as detailed in Section 2. Algorithm 1 details how the teacher supervises the learning process. Following the original Ice approach, teacher and learner only communicate by exchanging guesses for the predicates (from the latter to the former) and positive ($\mathcal{P}$), negative ($\mathcal{N}$) and implication ($\mathcal{I}$) data — from the former to the latter. These three sets of learning data are incrementally populated as long as the verification conditions are falsifiable, as discussed below.

Algorithm 1: Teacher supervising the learning process.

```
Input: the set $VC$ of verification conditions with predicate variables $\rho_1, \ldots, \rho_n$
Result: concrete predicates for $\rho_1, \ldots, \rho_n$ for which $\bigwedge VC$ is valid
1 $(\mathcal{P}, \mathcal{N}, \mathcal{I}) = (\emptyset, \emptyset, \emptyset)$ ;
2 $(p_1, \ldots, p_n) = learn($quals$, \mathcal{P}, \mathcal{N}, \mathcal{I})$ ;          (see Alg. 2)
3 while $\bigwedge VC[\rho_1 := p_1, \ldots, \rho_n := p_n]$ is falsifiable do
4     $(\mathcal{P}', \mathcal{N}', \mathcal{I}') = extract\_data(VC, p_1, \ldots, p_n)$ ;   (discussed in Sec. 3.1)
5     $(\mathcal{P}, \mathcal{N}, \mathcal{I}) = (\mathcal{P} \cup \mathcal{P}', \mathcal{N} \cup \mathcal{N}', \mathcal{I} \cup \mathcal{I}')$ ;
6     $(p_1, \ldots, p_n) = learn($quals$, \mathcal{P}, \mathcal{N}, \mathcal{I})$ ;      (see Alg. 2)
7 $(p_1, \ldots, p_n)$
```

3.1 Teacher

We now describe our modified version of the Ice teacher that, given some candidate predicates for $\Pi = \{\rho_1, \ldots, \rho_n\}$, returns learning data if the verification conditions instantiated on the candidates are falsifiable. Since there are several predicates to discover, the positive, negative and implication learning data (concrete values) will always be annotated with the predicate(s) concerned.

Now, all the constraints from the verification condition set VC have one of the following shapes, reminiscent of the original Ice's (1)–(2) from Section 1:

$$\alpha_1 \wedge \ldots \wedge \alpha_m \wedge C \models \alpha_{m+1} \tag{11}$$

$$\alpha_1 \wedge \ldots \wedge \alpha_m \wedge C \models \mathit{false} \qquad m \geq 1 \tag{12}$$

where each $\alpha_1, \ldots, \alpha_{m+1}$ is an application of one of the $\rho_1, \ldots, \rho_n$ to variables of the program, and C is a concrete formula ranging over the variables of the program. In the following, we write $\rho(\alpha_i)$ for the predicate α_i is an application of. To illustrate, recall constraint (8) of the example from Figure 1:

$$\underbrace{\rho_1(n)}_{\alpha_1} \wedge \underbrace{\rho_2(n+11, \mathit{tmp})}_{\alpha_2} \wedge \underbrace{\rho_2(\mathit{tmp}, r)}_{\alpha_3} \wedge \underbrace{n \leq 100}_{C} \models \underbrace{\rho_2(n, r)}_{\alpha_4}.$$

It has the same shape as (11), with $\rho(\alpha_1) = \rho_1$ and $\rho(\alpha_2) = \rho(\alpha_3) = \rho(\alpha_4) = \rho_2$.

Given some guesses $p_1, \ldots, p_n$ for the predicates $\rho_1, \ldots, \rho_n$, the teacher can check whether $VC[\rho_1 := p_1, \ldots, \rho_n := p_n]$ (the verification conditions obtained from VC by substituting p_i for each ρ_i) is falsifiable using an SMT solver. If it is, then function *extract_data* (Algorithm 1 line 4) extracts new learning data as follows. If a verification condition with shape (11) and $m = 0$ can be falsified, then we extract some values $\widetilde{x}$ from the model produced by the solver. This constitutes a *positive example* $(\rho(\alpha_1), \widetilde{x})$ since $\rho(\alpha_1)$ should evaluate to *true* for $\widetilde{x}$. From a counterexample model for a verification condition of the form (12), we extract a *negative constraint* $\{ (\rho(\alpha_1), \widetilde{x}_1), \ \ldots, \ (\rho(\alpha_m), \widetilde{x}_m) \}$. It means that *at least one* of the $(\rho(\alpha_i), \widetilde{x}_i)$ should be such that $\rho(\alpha_i)(\widetilde{x}_i)$ evaluates to *false*. Last, an *implication constraint* comes from a counterexample model for a verification condition of shape (11) with $m > 0$ and is a pair

$$\big(\{ (\rho(\alpha_1), \widetilde{x}_1), \ \ldots, \ (\rho(\alpha_m), \widetilde{x}_m) \}, \quad (\rho(\alpha_{m+1}), \widetilde{x}_{m+1}) \big).$$

Similarly to the original Ice implication constraints, this constraint means that if $\rho(\alpha_1)(\widetilde{x}_1) \wedge \ldots \wedge \rho(\alpha_m)(\widetilde{x}_m)$ evaluates to *true*, then so should $\rho(\alpha_{m+1})(\widetilde{x}_{m+1})$. Those positive examples, negative constraints, and implication constraints are accumulated in $\mathcal{P}$, $\mathcal{N}$, and $\mathcal{I}$, respectively, in Algorithm 1.

Remark 1 Note that negative examples and implication constraints in the original Ice framework are special cases of the negative constraints and implication constraints above. A negative example of the original Ice is just a singleton set $\{(\rho(\alpha_1), \widetilde{x}_1)\}$, and an implication constraint of Ice is a special case of the implication constraint where $m = 1$. Due to the generalization of learning data, negative constraints also contain unclassified data (unless they are singletons).

3.2 Learner: Building Candidates

We now start describing the learning part of our approach, which is an adaptation of the decision tree construction procedure from the original Ice framework [13]. The main difference is that the unclassified data can also contain values from negative constraints, as explained in Remark 1. This impacts decision tree construction as we now need to make sure the negative constraints are respected, in addition to checking that the implication constraints hold. Also, we adapted the qualifier selection heuristic (discussed in Section 3.4) to fit our context.

The learner takes, in addition to learning data ($\mathcal{P}$, $\mathcal{N}$, and $\mathcal{I}$), a mapping `quals` from predicate variables to sets of qualifiers as input. The learner then tries to find solutions for Horn clauses as Boolean combinations of qualifiers, by running Algorithm 2, as explained below. For the moment, we assume that the qualifier mapping `quals` is given a priori; how to find it is discussed in Section 3.5.

The learner needs to synthesize predicates for the variables $\rho_1, \ldots, \rho_n$ that respect the learning data. To do so, the learning data is *projected* on the different predicates and partially classified in the `class` mapping (in Algorithm 2 lines 1–5) following the semantics of the learning data given in Section 3.1. Notice the way each element of $\mathcal{N}$ is classified depending on whether it only has one predicate/values pair, as a consequence of Remark 1.

The algorithm also maintains a partial classification `class` (line 3) of the data from `unc`. This mapping encodes the choices made on the unclassified data: if $(\rho, \widetilde{x}) \mapsto$ *true* (*resp. false*), then a previous choice forced $(\rho, \widetilde{x})$ to be considered a positive (*resp.* negative) example.

It then calls `build_tree` (Algorithm 3) for each unknown predicate ρ, to construct a decision tree that encodes a candidate solution for ρ. A *tree* T is defined by $T ::= Node(q, T_+, T_-) \mid Leaf(b)$ where b is a Boolean. The formula it corresponds to is given by function f, defined inductively by

$$f(\ Node(q, T_+, T_-)\ ) = (\ q \wedge f(T_+)\ ) \vee (\ \neg q \wedge f(T_-)\ ) \quad \text{and} \quad f(\ Leaf(b)\ ) = b.$$

Algorithm 3 shows the decision tree construction process for a given $\rho \in \Pi$. We now discuss the algorithm formally and will illustrate it on an example in Section 3.3. Building a decision tree consists in choosing qualifiers splitting the learning data until there is no negative (positive) data left and the unclassified data can be classified as positive (negative) in each branch. The main difference with the tree construction from the original Ice framework is that the classification checks now take into account the negative constraints introduced earlier. Qualifier selection is discussed separately in Section 3.4.

Function `can_be_pos` checks whether all the unclassified data can be classified as positive. This consists in making sure that negative and implication constraints are verified or contain unclassified data — meaning future choices are able to (and will) verify the constraints. Given unclassified data U, constraint sets $\mathcal{N}$ and $\mathcal{I}$, and classifier mapping `class`, `can_be_pos` checks that the following conditions hold for every $u \in U$:

$$\forall N \in \mathcal{N}, \quad (\rho, u) \in N \quad \Rightarrow \exists (\rho', n) \in N \setminus \{(\rho, u') | u' \in U\},\ \texttt{class}(\rho', n) \simeq \mathit{false}$$

Algorithm 2: $\texttt{learn}(\texttt{quals}, \mathcal{P}, \mathcal{N}, \mathcal{I})$

```
Input: Qualifiers quals and positive ($\mathcal{P}$), negative ($\mathcal{N}$), and implication ($\mathcal{I}$) learning data
Result: concrete predicates for $\{\rho_1, \ldots, \rho_n\} = \Pi$ consistent with the learning data
1  global class = (
2      $\{ (\rho, \widetilde{x}) \mapsto true \mid (\rho, \widetilde{x}) \in \mathcal{P} \} \cup \{ (\rho, \widetilde{x}) \mapsto false \mid \{(\rho, \widetilde{x})\} \in \mathcal{N} \}$
3  );
4  foreach $(\rho, \widetilde{x})$ appearing in the elements of $\mathcal{I}$ and $\mathcal{N}$ do
5      if class$(\rho, \widetilde{x})$ is undefined then class$(\rho, \widetilde{x}) \leftarrow unknown$;
6  {
7      $\rho_i \mapsto$ build_tree(
8          $\rho_i$, quals$(\rho_i)$, $\{\widetilde{x} \mid$ class$(\rho_i, \widetilde{x})\}$, $\{\widetilde{x} \mid \neg$class$(\rho_i, \widetilde{x})\}$, $\{\widetilde{x} \mid$ class$(\rho_i, \widetilde{x}) = unknown\}$
9      ) | $i \in \{1, \ldots, n\}$
10 }
```

Algorithm 3: $\texttt{build\_tree}(\rho, Q, P, N, U)$

```
Input: Predicate variable $\rho$, qualifiers $Q$, positive ($P$), negative ($N$) and unclassified ($U$) projected learning data.
1  if $N = \emptyset \wedge$ can_be_pos$(U,$ class$)$ then
2      foreach $u \in U$ do class$(\rho, u) \leftarrow true$ ;
3      $Leaf(true)$
4  else if $P = \emptyset \wedge$ can_be_neg$(U,$ class$)$ then
5      foreach $u \in U$ do class$(\rho, u) \leftarrow false$ ;
6      $Leaf(false)$
7  else
8      choose $q$ in $Q$ that best divides the data
9      $(P_+, N_+, U_+)$ = data $\widetilde{x}$ from $(P, N, U)$ such that $q(\widetilde{x})$ ;
10     $(P_-, N_-, U_-)$ = data $\widetilde{x}$ from $(P, N, U)$ such that $\neg q(\widetilde{x})$ ;
11     $T_+$ = build_tree$(\rho,\ Q \setminus q,\ P_+,\ N_+,\ U_+, )$ ;
12     $T_-$ = build_tree$(\rho,\ Q \setminus q,\ P_-,\ N_-,\ U_-, )$ ;
13     $Node(q, T_+, T_-)$
```

$$\forall (LHS, rhs) \in \mathcal{I}, (\rho, u) \in LHS \Rightarrow \begin{cases} \texttt{class}(rhs) \simeq true \\ \vee\ \exists (\rho', l) \in LHS \setminus \{(\rho, u') | u' \in U\}, \\ \qquad\qquad\qquad \texttt{class}(\rho', l) \simeq false \end{cases}$$

where $\texttt{class}(n) \simeq b$ means that $\texttt{class}(n)$ is *unknown* or equal to b. Conversely, function `can_be_neg` checks that all the unclassified data can be classified as negative:

$$\forall u \in U,\ \forall (LHS, rhs) \in \mathcal{I}, \quad (\rho, u) = rhs \Rightarrow \exists (\rho', l) \in LHS,\ \texttt{class}(\rho', l) \simeq false.$$

The next section unfolds this algorithm on a simple example.

While we did not specify the order in which the trees are constructed (Algorithm 2 line 10), it can impact performance greatly because the classification choices influence later runs of `build_tree`. Hence, it is better to treat the elements of Π that have the least amount of unclassified data first. Doing so directs the choices of the qualifier q (Algorithm 3 line 8, discussed below) on as much classified data as possible. The data is then split (lines 9 and 10) using q: more classified data thus means more informed splits, leading to more relevant classifications of unclassified data in the terminal cases of the decision tree construction.

Remark 2 Because the functions `can_be_pos` and `can_be_neg` only locally check the constraints to decide whether the data can be classified to be true or false, the algorithm above may end up with inconsistent classification of data even if the learning data are consistent.[2] In such a case, we should either backtrack and reclassify the data, or use a SAT solver to get a globally consistent classification of data.

[2] Thanks to Uki Ryuu for identifying this problem.

3.3 Learner: Example

We now illustrate the decision tree building process discussed above using the 91 function from Figure 1, with verification conditions (5)–(10). Again, say that we realize that $\rho_1(n) = true$, so that we only need to synthesize $\rho_2(n, r)$. Suppose that the learner is called with $\mathcal{P} = \{(\rho_2, [101, 91])\}$ from verification condition (5), $\mathcal{N} = \{\{(\rho_2, [100, 90])\}\}$ from (10), and $\mathcal{I} = \{\ (\ \{(\rho_2, [103, 102]),\ (\rho_2, [102, 102])\},\ (\rho_2, [92, 102])\ )\ \}$ from (8).

Below we omit to write which predicate the input values are for, since there is only ρ_2 here. Also, we use +, − or ? superscripts on the samples to denote whether it is positive, negative or unclassified respectively. The learner starts working on the data $[101, 91]^+$, $[100, 90]^-$, $[103, 102]^?$, $[102, 102]^?$, $[92, 102]^?$; say the first qualifier it chooses is $n \geq 101$. This splits the data in two as shown on Figure 5, the data on which the qualifier evaluates to *true* (top branch) and the data where it evaluates to *false* (bottom branch). Then, the algorithm recurses on the branch where $n \geq 101$ with data $[102, 92]^+$, $[103, 102]^?$, $[102, 102]^?$. There is no negative example and the unclassified data can be classified as positive, since the implication constraint it comes from mentions [92, 102] which is still unclassified.

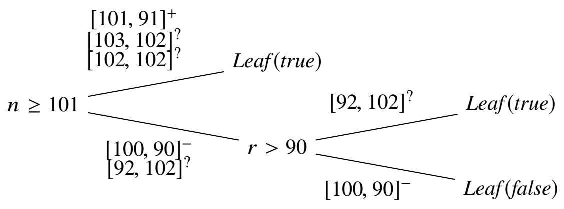


Fig. 5: Decision tree example.

Next, we recursively go in the branch where $n \ngeq 101$ with data $[101, 92]^-$, $[91, 102]^?$. There is no positive data left, but [91, 102] cannot be classified negative as it is the consequent of implication constraint ({[103, 102], [102, 102]}, [92, 102]) which antecedents are entirely classified positively. The algorithm is forced to choose another qualifier, say $r > 90$. The data is split in two and both recursive calls are terminal cases. The learner thus returns the tree from Figure 5, which represents the formula

$$n \geq 101 \lor (\neg(n \geq 101) \land r > 90).$$

3.4 Qualifier Selection in Algorithm 3

We now discuss how to choose qualifier $q \in Q$ on line 8 in Algorithm 3. The choice of the qualifier q used to split the learning data $D = (P, N, U)$ in $D_q = (P_q, N_q, U_q)$ and $D_{\neg q} = (P_{\neg q}, N_{\neg q}, U_{\neg q})$ is crucial. In [13], the authors introduce two heuristics based on the notion of *Shannon Entropy* ε:

$$\varepsilon(D) = -\frac{|P|}{|P| + |N|} \log_2 \frac{|P|}{|P| + |N|} - \frac{|N|}{|P| + |N|} \log_2 \frac{|N|}{|P| + |N|} \tag{13}$$

which yields a value between 0 and 1. This entropy rates the ratio of positive and negative examples: it gets close to 1 when $|P|$ and $|N|$ are close. A small entropy is preferred as it indicates that the data contains significantly more of one than the other. The *information gain* γ of a split is

$$\gamma(D, q) = \varepsilon(D) - \left(\frac{|D_q|\varepsilon(D_q)}{\lfloor D \rceil} + \frac{|D_{\neg q}|\varepsilon(D_{\neg q})}{\lfloor D \rceil} \right) \tag{14}$$

where $\lfloor D = (P, N, U) \rceil = |P| + |N|$. A high information gain means q separates the positive examples from the negative ones. Note that the information gain ignores unclassified data, a shortcoming the Ice framework [13] addresses by proposing two qualifier selection heuristics. The first subtracts a penalty to the information gain. It penalizes qualifiers separating data coming from the same implication constraint — called *cutting the implication*. The second heuristic changes the definition of entropy by introducing a function approximating the probability that a non-classified example will eventually be classified as positive. We present here our adaptation of this second heuristic, as it is much more natural to transfer to our use-case.

The idea is to create a function Pr that approximates the probability that some values from the projected learning data $D = (P, N, U)$ end up classified as positive. More precisely, $Pr(v)$ approximates the ratio between the number of *legal* (constraint-abiding) classifications in which v is classified positively and the number of all legal classifications. Computing this ratio for the whole data is impractical: it falls in the *counting problems* class and it is $\#P$-complete [1]. The approximation we propose uses the following notion of *degree*:

$$Degree(v) \;=\; \sum_{(\widetilde{x}, v) \in \mathcal{I}} \frac{1}{1 + |\widetilde{x}|} - \sum_{(\widetilde{x}, y) \in \mathcal{I},\, v \in \widetilde{x}} \frac{1}{1 + |\widetilde{x}|} - \sum_{\widetilde{x} \in \mathcal{N},\, v \in \widetilde{x}} \frac{1}{|\widetilde{x}|}$$

The three terms appearing in function *Degree* are based on the following remarks. Let v be some value in the projected learning data. If $(\widetilde{x}, v) \in \mathcal{I}$, there is only one classification for $\widetilde{x}$ to force v to be true: the classification where all the elements of $\widetilde{x}$ are classified positively. More elements in $\widetilde{x}$ generally mean more legal classifications where one of them is false and v need not be true: $Pr(v)$ should be higher if $\widetilde{x}$ has few elements. If v appears in the antecedents of a constraint $(\widetilde{x}, y)$, then $Pr(v)$ should be lower. Still, if $\widetilde{x}$ has many elements it means v is less constrained. There are statistically more classifications in which v is true without triggering the implication, and thus more legal classifications where v is true. Last, if v appears in a negative constraint $\widetilde{x}$ then it is less likely to be true. Again, a bigger $\widetilde{x}$ means v is less constrained, since there are statistically more legal classifications where v is true.

Our Pr function compresses the degree between 0 and 1, and we define a new multi-predicate-friendly entropy function ε to compute the information gain (where $D = (P, N, U)$):

$$Pr(D) = \frac{\sum_{v \in P \cup N \cup U} Pr(v)}{|P|+|N|+|U|} \qquad Pr(v) = \begin{cases} 1 & \text{if } v \in P \\ 0 & \text{if } v \in N \\ \dfrac{1}{2} + \dfrac{\arctan Degree(v)}{\pi} & \text{otherwise} \end{cases}$$

$$\varepsilon(D) = -Pr(D) \log_2 Pr(D) \;-\; (1 - Pr(D)) \log_2 (1 - Pr(D))$$

Note that it can happen that none of the qualifiers can split the data, *i.e.* there is no qualifier left or they all have an information gain of 0. In this case we synthesize qualifiers that we know will split the data as described in the next subsection.

3.5 Mining and Synthesizing Qualifiers

We now discuss how to prepare the set Q of qualifiers used in Algorithm 3. The learner in both the original Ice approach and our modified version spend a lot of time evaluating qualifiers. Having too many of them slows down the learning process considerably, while not considering enough of them reduces the expressiveness of the candidates. The compromise we propose is to *i)* mine for (few) qualifiers from the clauses, and *ii)* synthesize (possibly many) qualifiers when needed, driven by the data we need to split.

To mine for qualifiers, for every clause C and for every predicate application of the form $\rho(\widetilde{v})$ in C, we add every atomic predicate a in C as a qualifier for ρ as long as all the free variables of a are in $\widetilde{v}$. All the other qualifiers are synthesized during the analysis.

Based on our experience, we have chosen the following synthesis strategy. With $v_1, \ldots, v_n$ the formal inputs of ρ, for all $(x_1, \ldots, x_n) \in P \cup N \cup U$, we generate the set of new qualifiers

$$\begin{array}{rl} & \{\ v_i \diamond x_i \mid 1 \le i \le n,\ \diamond \in \{\le, \ge\}\ \} \\ \cup & \{\ v_i + v_j \diamond x_i + x_j \mid 1 \le i < j \le n,\ \diamond \in \{\le, \ge\}\ \} \\ \cup & \{\ v_i - v_j \diamond x_i - x_j \mid 1 \le i < j \le n,\ \diamond \in \{\le, \ge\}\ \} \end{array}$$

Adding these qualifiers allows to split the data on these (strict, when negated) inequalities, and encode (dis)equalities by combining them in the decision tree. Also, notice that when no qualifier can split the data we have in general *small* P, N and U sets, and the number of new qualifiers is quite tractable. The learning process is an iterative one where relatively few new samples are added at each step, compared to the set of all samples. Since we could split the samples from the previous iteration, it is very often the case that P, N and U contain mostly new samples. Last, our approach shares the limitation of the original Ice: it will not succeed if a particular relation between the variables is needed to conclude, but no qualifier of the right shape is ever mined for or synthesized.

4 Experimental Evaluation

Our implementation consists of two parts: first, RType is a frontend (written in OCaml) generating Horn clauses from programs written in a subset of OCaml as discussed in Section 2. It relies on an external Horn clause solver for actually solving the clauses, and post-processes the solution (if any) to yield refinement types for the original program. HoIce[3], written in Rust[4], is one such Horn clause solver and implements the modified Ice framework presented in this paper. All experiments in this section use RType `v1.0` and HoIce `v1.0`. Under the hood, HoIce relies on the Z3[5] SMT solver [22] for satisfiability checks. In the following experiments, RType uses HoIce as the Horn clause solver. Note that the input OCaml programs are not annotated: the Horn clauses correspond to the verification conditions encoding the fact that the input program cannot falsify its assertion(s). RType supports a subset of OCaml including (mutually) recursive functions and integers, without algebraic data types.

We now report on our experimental evaluation. Our benchmark suite of 162 programs[6] includes the programs from [26] and [37] in the fragment RType supports, along with programs automatically generated by the termination verification tool from [19], and 10 new

[3] Hosted at `https://github.com/hopv/r_type` and `https://github.com/hopv/hoice`.
[4] For more details, see `https://www.rust-lang.org/en-US/`.
[5] The revision of Z3 in all the experiments is the latest at the time of writing: `5bc4c98`.
[6] Hosted at `https://github.com/hopv/benchmarks`.

benchmarks written by ourselves. We only considered programs that are safe since RType is not refutation-sound.
These benchmarks range from very simple to relatively complex, with in particular a program computing a solution to the N Queen problem using arrays. The verification challenge for this program is to prove it does not perform out-of-bound array accesses. Here are a few statistics on the number of lines in the original OCaml program, and the number of predicates and clauses in the Horn clause problems:

	max	**mean**	**variance**	**standard deviation**
lines	95	31.671	515.160	22.697
predicates	38	9.829	58.581	7.654
clauses	364	55.707	3604.963	60.041

Note that the Horn clause data was generated from the encoding discussed in Section 2, *including* optimizations.

All the experiments presented in this section ran on a machine running Ubuntu (Xeon E5-2680v3, 64GB of RAM) with a timeout of 100 seconds. The number between parentheses in the keys of the graphs is the number of benchmarks solved. We begin by evaluating the optimizations discussed in Section 2, followed by a comparison against automated verification tools for OCaml programs. Last, we evaluate our predicate synthesis engine against other Horn-clause-level solvers.

4.1 Evaluation of the Optimizations

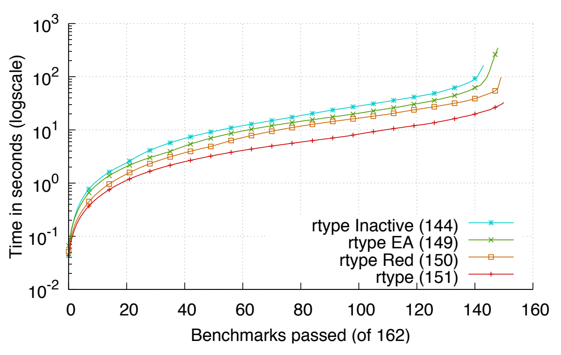
Figure 6a shows our evaluation of the effect analysis (**EA**) and clause reduction (**Red**) simplifications discussed in Section 2. It is clear that both effect analysis and Horn reduction speedup the learning process significantly. They work especially well together and can reduce drastically the number of predicates on relatively big synthesis problems, as shown on Figure 6c.

The 11 programs that we fail to verify show inherent limitations of our approach. Two of them require an invariant of the form $x + y \geq z$. Our current compromise for qualifier mining and synthesis (in Section 3.4) does not consider such qualifiers unless they appear explicitly in the program. We are currently investigating how to alter our qualifier synthesis approach to raise its expressiveness with a reasonable impact on performance. The remaining nine programs are not typable with refinement types, meaning the verification conditions generated by RType are actually unsatisfiable. An extension of the type system is required to prove these programs correct [32].

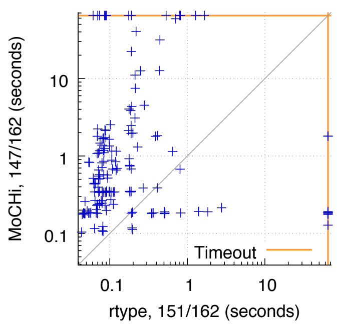
4.2 Comparison with other OCaml Program Verifiers

The first tool we compare RType to is the higher-order program verifier MoCHi from [26] (Figure 6b). MoCHi infers intersection types, which makes it more expressive than RType. The nine programs that MoCHi proves but RType cannot verify are the (refinement-)untypable ones discussed above. While this shows a clear advantage of intersection types over our approach in terms of expressiveness, the rest of the experiments make it clear that, when applicable, RType outperforms MoCHi on a significant part of our benchmarks.

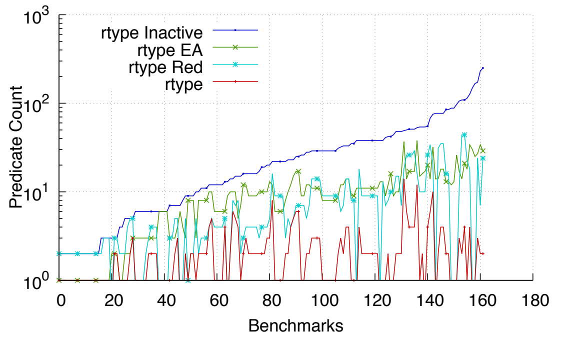
We also evaluated our implementation against DOrder from [37, 38]. This comparison is interesting as DOrder also uses machine-learning to infer refinement types, but does not



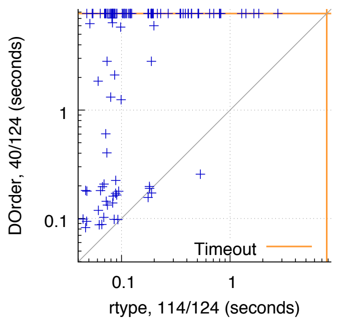
(**a**) Cumulative runtime comparison.



(**b**) Against MoCHi.



(**c**) Predicate reduction.



(**d**) Against DOrder.

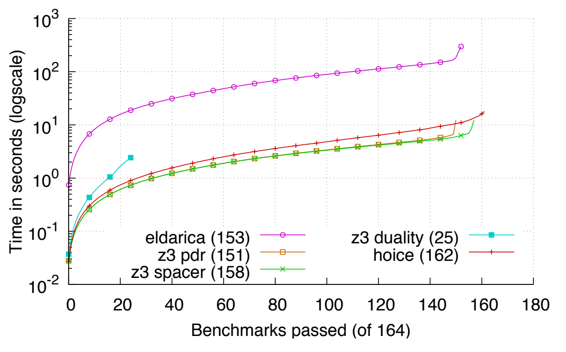
Fig. 6: Evaluation: verification of OCaml programs.

support implication constraints. DOrder compensates by conducting test runs of the program on random inputs to gather better positive data. It supports a different subset of OCaml than RType though, and after removing the programs it does not support, 124 programs are left. The results are on Figure 6d, and show that RType overwhelmingly outperforms DOrder. This is consistent with the results reported for the original Ice framework: the benefit gained by considering implication constraints is huge.

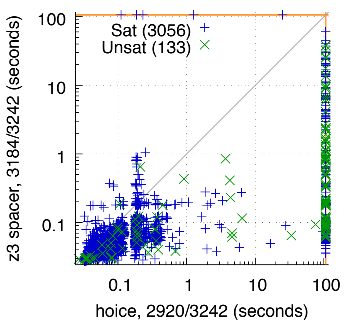
These results show that, despite its limitations, our approach is competitive and often outperforms other state-of-the-art automated verification tools for OCaml programs.

4.3 Horn-clause-level Evaluation

Last, we compare our Horn clause solver HoIce to other solvers (Figure 7): Spacer [17], Duality [20], Z3's PDR [14], and Eldarica [15]. The first three are implemented in Z3 (C++) while Eldarica is implemented in Scala. The benchmarks are the Horn clauses encoding the safety of the 162 programs aforementioned with additional two programs, omitted in the previous evaluation as they are unsafe.



(a) Cumulative comparison.



(b) Against Spacer on SV-COMP.

Fig. 7: Comparison with Horn clause solvers.

HoIce solves the most benchmarks at 162.[7] The fastest tool overall is Z3's Spacer which solves slightly fewer benchmarks. The two timeouts for HoIce come from the programs discussed above for which HoIce does not have the appropriate qualifiers to conclude. Because it mixes IC3 [5] with interpolation, Spacer infers the right predicates quite quickly. Thus, in our use-case, our approach is competitive with state-of-the-art Horn clause solvers in terms of speed, in addition to being more precise. We also include a comparison on the SV-COMP with Spacer on Figure 7b. HoIce is generally competitive, but timeouts on a significant part of the benchmarks. Quite a few of them are unsatisfiable; the Ice framework is not made to be efficient at proving unsatisfiability. The rest of the timeouts require qualifiers we do not mine for nor synthesize, showing that some more work is needed on this aspect of the approach.

In our experience, it is often the case that HoIce's models are significantly simpler than those of Spacer's and PDR's (as illustrated in Appendix A). Note that simple models are useful if the Horn clause solver is placed inside a CEGAR loop such as the one in MoCHi [26]; indeed, Sato et al. [25] have recently employed CHC solving as a backend of MoCHi, and observed that HoIce was more effective than Spacer as the backend CHC solver in that context.

5 Algebraic Data Types

This section discusses how to extend our approach to deal with functional programs that manipulate algebraic data types (ADTs) such as lists and trees. We assume here that ADTs do not contain functions; we do not consider, for example, a list of functions. As we discuss below, most of our framework need not be changed, including verification condition generations, and the main procedures for teacher and learner, as long as the backend SMT solver supports ADTs; the main new issue is how to find/synthesize appropriate qualifiers. Below, after briefly discussing the overall extension of our framework in Section 5.1, we explain our approach to qualifier synthesis in Section 5.2. We then report preliminary experiments on the extension in Section 5.3.

[7] This is consistent with the OCaml results: 151 sat results, 9 unsat from programs RType cannot verify, and 2 unsat from unsafe programs.

5.1 Overview

As mentioned above, except for the fact that a new method is required for qualifier discovery, our framework can be smoothly extended to deal with ADTs; this is an advantage of our ICE-based approach. Below we sketch the extension of each component through the following running example.

```
let rec ins (i : int) (lst : int list) = match lst with
| [] → i :: []
| hd :: tl → if i ≤ hd then i :: lst else hd :: (ins i tl)
let main (i : int) (lst : int list) = assert (ins i lst != [])
```

The function `ins` takes an integer `i` and an integer list `lst` as arguments, and returns a list obtained by inserting `i` into `lst`.

Refinement Types and Verification Condition Generation. Under the assumption that ADTs do not contain functions, the simplest way to extend the refinement type system and verification condition in Section 2 is to treat ADTs just like ordinary base types. The syntax of refinement types is extended by:

$$\begin{aligned} T \text{ (refinement types)} &::= \{x : \mathbb{A} \mid a\} \mid (x : T_1) \to T_2 \\ \mathbb{A} \text{ (base and algebraic data types)} &::= \texttt{int} \mid \texttt{int list} \mid \cdots \end{aligned}$$

Here, the set of expressions ranged over by a is extended to allow operations on algebraic data types. No change is required on the verification condition generation procedure in Figure 4 (except the extension of the syntax of types and expressions).

For the running example above. we obtain the following refinement type templates and verification conditions.

$$\begin{array}{l} \texttt{main} : \texttt{int} \to \texttt{int list} \to \texttt{int} \\ \texttt{ins} : i : \texttt{int} \to lst : \texttt{int list} \to \{r : \texttt{int list} \mid \rho_{\texttt{ins}}(i, lst, r)\} \\ \hline \end{array}$$

$$\begin{array}{rcl} \rho_{\texttt{ins}}(i, lst, r) \land r = [\,] & \models & false \\ i \leq hd & \models & \rho_{\texttt{ins}}(i, hd :: tl, i :: hd :: tl) \\ lst = [\,] & \models & \rho_{\texttt{ins}}(i, lst, i :: [\,]) \\ i > hd \land \rho_{\texttt{ins}}(i, tl, r) & \models & \rho_{\texttt{ins}}(i, hd :: tl, hd :: r) \end{array}$$

Teacher. Assuming that the backend SMT solver supports ADTs, the teacher procedure described in Section 3.1 can be used as it is. For the example above, given a candidate solution:$\rho_{\texttt{ins}}(x, \ell, r) \equiv r \neq [\,]$, the teacher just needs to check that the candidate satisfies the verification conditions by using the SMT solver.

Learner. The learner procedure described in Section 3.2 can also be used as it is. In fact, for the example above, the learner can easily find the valid candidate solution $\rho_{\texttt{ins}}(x, \ell, r) \equiv r \neq [\,]$, by using equality constraints as qualifiers.

The remaining issue is how to find appropriate qualifiers. In Section 3.5, we have discussed how to mine and synthesize qualifiers on integers. That is not sufficient for programs manipulating ADTs, like the following one:

```
let rec ins (i : int) (lst : int list) = ... (* the same as function ins above *)
let rec sorted : int list → bool = function
| [] | _ :: [] → true
| hd_1 :: hd_2 :: tl → if hd_1 > hd_2 then false else sorted (hd_2 :: tl)
let main (i : int) (lst : int list) = if sorted lst then assert (sorted (ins i lst))
```

From the example above, we obtain the following refinement type templates and verification conditions.

$$\begin{array}{l}
\mathtt{main} : \mathtt{int} \to \mathtt{int\ list} \to \mathtt{int} \\
\mathtt{ins} : i : \mathtt{int} \to lst : \mathtt{int\ list} \to \{r : \mathtt{int\ list} \mid \rho_{\mathtt{ins}}(i, lst, r)\} \\
\mathtt{sorted} : lst : \mathtt{int\ list} \to \{b : \mathtt{bool} \mid \rho_{\mathtt{sorted}}(lst, b)\} \\
\hline
\end{array}$$

$$\begin{array}{rcl}
\rho_{\mathtt{sorted}}(lst, \mathtt{true}) \wedge \rho_{\mathtt{ins}}(i, lst, r) \wedge \rho_{\mathtt{sorted}}(r, \mathtt{false}) & \models & \mathit{false} \\
lst = [\,] & \models & \rho_{\mathtt{ins}}(i, lst, i :: [\,]) \\
i \le hd & \models & \rho_{\mathtt{ins}}(i, hd :: tl, i :: hd :: tl) \\
i > hd \wedge \rho_{\mathtt{ins}}(i, tl, r) & \models & \rho_{\mathtt{ins}}(i, hd :: tl, hd :: r) \\
lst = [\,] & \models & \rho_{\mathtt{sorted}}(lst,\ \mathtt{true}) \\
tl = [\,] & \models & \rho_{\mathtt{sorted}}(hd :: tl,\ \mathtt{true}) \\
hd_1 > hd_2 & \models & \rho_{\mathtt{sorted}}(hd_1 :: hd_2 :: tl,\ \mathtt{false}) \\
hd_1 \le hd_2 \wedge \rho_{\mathtt{sorted}}(hd_2 :: tl,\ res) & \models & \rho_{\mathtt{sorted}}(hd_1 :: hd_2 :: tl,\ res)
\end{array}$$

In order for the above clauses to be valid, $\rho_{\mathtt{sorted}}(x, b)$ must express the property: "$b = \mathtt{true}$ just if x is sorted," which clearly cannot be expressed by the qualifiers mined/synthesized by the method in Section 3.5. This issue is addressed in the next subsection.

5.2 Qualifier Synthesis for ADTs

Our approach to qualifier synthesis is, given a set of CHCs representing verification conditions, to extract (possibly recursive) functions that take ADTs as input and return base type values, and to allow them to be used in the qualifier mining and synthesis discussed in Section 3.5 (thus, v_i in Section 3.5 may now be a function application $f(v_1, \dots, v_n)$). Let us explain this through the last example. We first gather CHCs on $\rho_{\mathtt{sorted}}$:

$$\begin{array}{rcl}
lst = [\,] & \models & \rho_{\mathtt{sorted}}(lst,\ \mathtt{true}) \\
tl = [\,] & \models & \rho_{\mathtt{sorted}}(hd :: tl,\ \mathtt{true}) \\
hd_1 > hd_2 & \models & \rho_{\mathtt{sorted}}(hd_1 :: hd_2 :: tl,\ \mathtt{false}) \\
hd_1 \le hd_2 \wedge \rho_{\mathtt{sorted}}(hd_2 :: tl,\ res) & \models & \rho_{\mathtt{sorted}}(hd_1 :: hd_2 :: tl,\ res)
\end{array}$$

We turn them to the following function definition for the underlying SMT solver.

```
(define-fun-rec sorted ((i Int) (lst Intlist)) Bool
    (ite (or (= lst nil) (tl lst)) true
        (ite (> (hd lst) (hd (tl lst))) false)
            (ite (≤ (hd lst) (hd (tl lst))) (sorted (tl lst)))))
```

We call this process *function reconstruction*, which will be explained later.

The learner may now use, as qualifiers, arithmetic constraints involving the function `sorted` above, and may return the following candidate solution:

$$\begin{array}{rcl}
\rho_{\mathtt{ins}}(i, lst, r) & \equiv & \mathtt{sorted}(lst) \wedge \mathtt{sorted}(r) \\
\rho_{\mathtt{sorted}}(lst, b) & \equiv & \mathtt{sorted}(lst) = b
\end{array}$$

The teacher can verify it as a valid solution, as long as the underlying SMT solver can properly deal with recursive functions (and indeed, Z3 can verify the validity of the solution above instantly).

Remark 3 In the example above, we picked $\rho_{\texttt{sorted}}$ for the function reconstruction. Our criterion for a predicate ρ to be eligible for function reconstruction is that it only appears in *i)* negative clauses and *ii)* clauses that only mention ρ (called the *defining* clauses of ρ). When there are more than one candidate predicate, we heuristically choose one with the simplest signature (lowest arity and lowest number of ADT-valued parameters) and complexity (lowest number of non-negative clauses mentioning ρ).

5.2.1 Function Reconstruction

We now explain how to reconstruct a function definition from the defining clauses of a predicate ρ. We can assume that each of the defining clauses is of the form:

$$C \wedge \rho(t_{1,1}, \dots, t_{1,k}, y_1) \wedge \cdots \rho(t_{\ell,1}, \dots, t_{\ell,k}, y_\ell) \models \rho(x_1, \dots, x_k, t)$$

where C is a conjunction of atomic constraints without predicate variables. We first eliminate variables other than $x_1, \dots, x_k, y_1, \dots, y_\ell$ in a heuristic manner. For example, if C contains $x = z_1 :: z_2$, we eliminate z_1 and z_2 by adding $\texttt{is-cons}(x)$ and replacing z_1 and z_2 with $\texttt{hd}(x)$ and $\texttt{tl}(x)$ respectively. If the variable elimination fails, then we give up the function reconstruction for ρ.

Example 2 Consider one of the defining clauses for $\rho_{\texttt{sorted}}$:

$$hd_1 \leq hd_2 \wedge \rho_{\texttt{sorted}}(hd_2 :: tl,\ r) \models \rho_{\texttt{sorted}}(hd_1 :: hd_2 :: tl,\ r).$$

It can first be transformed to:

$$hd_1 \leq hd_2 \wedge x = hd_1 :: hd_2 :: tl \wedge \rho_{\texttt{sorted}}(hd_2 :: tl,\ r) \models \rho_{\texttt{sorted}}(x,\ r).$$

By eliminating hd_1, hd_2 and tl, we obtain:

$$\begin{aligned}&\texttt{is-cons}(x) \wedge \texttt{is-cons}(\texttt{tl}(x)) \wedge \texttt{hd}(x) \leq \texttt{hd}(\texttt{tl}(x))\\ &\qquad \wedge \rho_{\texttt{sorted}}(\texttt{hd}(\texttt{tl}(x)) :: \texttt{tl}(\texttt{tl}(x)),\ r) \models \rho_{\texttt{sorted}}(x,\ r).\end{aligned}$$

□

Using this variable elimination, the defining clauses are normalized to:

$$\begin{gathered}C_1 \wedge \rho(\vec{t}_{1,1}, y_1) \wedge \cdots \wedge \rho(\vec{t}_{1,\ell_1}, y_{\ell_1}) \models \rho(\vec{x}, t_1)\\ \cdots\\ C_n \wedge \rho(\vec{t}_{n,1}, y_1) \wedge \cdots \wedge \rho(\vec{t}_{n,\ell_n}, y_{\ell_n}) \models \rho(\vec{x}, t_n).\end{gathered}$$

For the sake of simplicity, we assume below that $C_1, \dots, C_n$ do not contain variables other than $\vec{x}$. We now check that *i)* $C_1, \dots, C_n$ are mutually exclusive and exhaustive, and *ii)* the term $\vec{t}_{i,j}$ contains none of the variables $y_j, \dots, y_{\ell_i}$. We then construct the function definition:

$$\begin{aligned}f_\rho(\vec{x}) = {}& \text{if } C_1 \text{ then let } y_1 = f_\rho(\vec{t}_{1,1}) \text{ in ... let } y_{\ell_1} = f_\rho(\vec{t}_{1,\ell_1}) \text{ in } t_1\\ & \text{else if} \cdots\\ & \ \text{else let } y_n = f_\rho(\vec{t}_{n,1}) \text{ in} \dots \text{let } y_{\ell_n} = f_\rho(\vec{t}_{n,\ell_n}) \text{ in } t_n\end{aligned}$$

Finally, we check that the above definition makes f_ρ total by requiring that C_i implies that $\vec{t}_{i,\ell_i} < \vec{x}$ with respect to a certain well-founded order $<$.[8]

[8] Since the totality of f_ρ is undecidable in general, this check is necessarily heuristic. The check is omitted in the current implementation reported in Section 5.3. The implementation is thus unsound, although the unsoundness did not show up in the reported experiments.

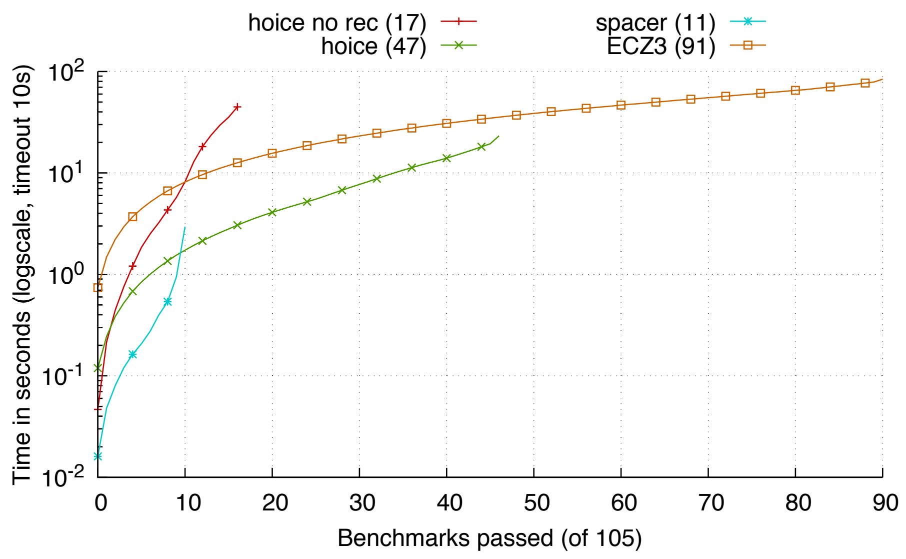


Fig. 8: Comparison on benchmarks with ADTs.

Remark 4 Our approach above relies on the assumption that the underlying SMT solver can effectively reason about recursive functions. We have used Z3 and applied some optimizations to help it reason about recursive functions. Omitting solver-specific tweaks[9], the most rewarding optimization was to add an invariant discovery step at the end of function reconstruction. The *length* function over lists for instance, without the invariant that its output is always positive, can be a problem for solvers even on relatively simple queries. This invariant discovery step generates candidate invariants based on the signature and definition of the function, which it attempts to prove by checking they are preserved in each branch of the definition. If some invariant *inv* is discovered for function *f*, then whenever the definition of *f* is given to a solver then it is followed by the assertion $\forall \vec{v}, f(\vec{v}) \Rightarrow inv(\vec{v})$.

5.3 Evaluation

While the work on ADTs presented in this section is still in an early stage, we implemented and evaluated it against Spacer [17] and recent work [9] where the authors reduce Horn clause problems over ADTs to an equisatisfiable problem over basic sorts. Unfortunately, as far as we know this latter approach does not always allow to produce models for the predicates if the problem is satisfiable, while Spacer and our approach do.

Also, we were not able to retrieve a binary for the implementation of [9] called ECZ3. The results presented on Figure 8 for ECZ3 are the results reported in [9] where the experiments ran on a different machine. As a consequence, the runtimes reported below are not comparable and readers should really focus on the number of benchmarks solved. This evaluation uses the set of benchmarks from [9].[10]

[9] Such as the use of conditional “check-sat” in the check `https://rise4fun.com/Z3/cXot4`.

[10] Available at `https://fmlab.unich.it/iclp2018`.

The two versions of our implementation[11] presented, HoIce and “HoIce no rec”, run with and without function reconstruction respectively. The difference in precision is quite noticeable, despite the fact that the approach and the implementation are still ongoing work. The benchmarks HoIce is not able to solve fail for various reasons and indicate future directions of research. In a few cases, the problem is that the underlying solver (Z3) returns *unknown*, at which point HoIce is forced to give up. In other cases the solver does not return in reasonable time on a query. This often happens in the teacher while checking a valid candidate, meaning all clauses are unsatisfiable which the solver struggles to verify. In a few other cases the function reconstructed are not enough for HoIce to reach a conclusion and it keeps trying to find a model forever.

ECZ3 from [9] is by far the best in terms of precision, with the drawback that models for the original predicates are not available. Spacer yields performance similar to HoIce without function reconstruction which suggests that it would also benefit from function reconstruction. In fact, we ran spacer on a handful of problems in which we manually forced the definition given by function reconstruction, and spacer was able to solve the modified version. This is a good indication that the approach we suggest in this section extends beyond sampling- and template-based techniques such as our generalized Ice framework.

Last, for the sake of reproducibility we should mention that we used Z3 version 4.7.1[12] as both HoIce’s underlying SMT solver and in the evaluation of spacer. More recent versions of Z3 (4.8.* at the time of writing) seem far less efficient when it comes to dealing with ADTs and recursive functions. Running HoIce with Z3 4.8.* on the benchmarks mentioned in this section yields a huge number of timeouts and unknown result (meaning the Z3 cannot answer one the teacher’s queries).

6 Related Work

There has been a lot of work on sampling-based approaches to program invariant discoveries during the last decade [12,13,27–29,36–38]. Among others, most closely related to this paper are Garg et al.’s Ice framework [12,13] (which this paper extends) and Zhu et al.’s refinement type inference methods [36–38]. To the best of our knowledge, Zhu et al. [36–38] were the first to apply a sampling-based approach to refinement type inference for higher-order functional programs. They did not, however, consider implication constraints. As discussed in Section 4, their tool fails to verify some programs due to the lack of implication constraints.

There are other automated/semi-automated methods for verification of higher-order functional programs [16,24,30–32,34,37,38], based on some combinations of Horn clause solving, automated theorem proving, counterexample-guided abstraction refinement, (higher-order) model checking, etc. As a representative of such methods, we have chosen MoCHi and compared our tool with it in Section 4. As the experimental results indicate, our tool often outperforms MoCHi, although not always. Thus, we think that our learning-based approach is complementary to the aforementioned ones; a good integration of our approach with them is left for future work. Liquid types [24], another representative approach, is semi-automated in that users have to provide qualifiers as hints. By preparing a fixed, default set of qualifiers, Liquid types may also be used as an automated method. From that viewpoint, the main advantage of our approach is that we can infer arbitrary Boolean combinations of qualifiers as refinement predicates, whereas Liquid types can infer only conjunctions of qualifiers. On

[11] This evaluation uses HoIce 1.8.1.

[12] `https://github.com/Z3Prover/z3/releases/tag/z3-4.7.1`

the downside, since we synthesize (a potentially infinite number of) quantifiers, and build candidates which are arbitrary Boolean combinations of these qualifiers, our approach has no guarantee to terminate, unlike Liquid types.

Since the publication of our original paper on which this extended version is based, at least two related approaches to Horn clause solving were published [10,35]. Both approaches rely on ideas similar to ours: produce candidate for the predicates based on data accumulated by refuting previous candidates. The main difference with our work is that neither use implication constraints, which we believe to be important for the learning of inductive invariants. Also, they seem to mainly target verification problems stemming from imperative programs while our approach was designed with functional program verification in mind.

7 Conclusion

In this paper we proposed an adaptation of the machine-learning-based, invariant discovery framework Ice to refinement type inference. The main challenge was that implication constraints and negative examples were ill-suited for solving Horn clauses of the form $\rho(\widetilde{x_1}) \wedge \cdots \wedge \rho(\widetilde{x_n}) \wedge \ldots \models \rho(\widetilde{x})$, which tend to appear often in our context of functional program verification because of nested recursive calls.

We addressed this issue by generalizing Ice's notion of implication constraint. For similar reasons, we also adapted negative *examples* by turning them into negative *constraints*. This means that, unlike the original Ice framework, our learner might have to make classification choices to respect the negative learning data. We have introduced a modified version of the Ice framework accounting for these adaptations, and have implemented it, along with optimizations based on effect analysis. Our evaluation on a representative set of programs show that it is competitive with state of the art OCaml model-checkers and Horn clause solvers.

We also reported on preliminary work on adapting our approach to Algebraic Data Types by reconstructing, when relevant, functions that the framework can leverage to build useful qualifiers. The evaluation of our prototype implementation show that doing so is rewarding but, in its current state, fails to outperform a recent technique that encodes Horn clause over ADT verification as ADT-free Horn clauses, with the drawback of not being able to generate models for the original problem when the problem is satisfiable.

Acknowledgements We would like to thank anonymous referees for useful comments. We would also like to thank He Zhu for his help in benchmarking DOrder, Hiroshi Unno for his help and insight on Horn clauses over ADTs and the reviewers for their constructive feedback. This work was supported by JSPS KAKENHI Grant Number JP15H05706 and JP20H00577.

References

1. Arora, S., Barak, B.: Computational Complexity - A Modern Approach. Cambridge University Press (2009)
2. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability, *Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 825–885. IOS Press (2009). DOI 10.3233/978-1-58603-929-5-825
3. Beyer, D.: Competition on software verification - (SV-COMP). In: C. Flanagan, B. König (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7214,

pp. 504–524. Springer (2012). DOI 10.1007/978-3-642-28756-5_38. URL https://doi.org/10.1007/978-3-642-28756-5_38

4. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn clause solvers for program verification. In: L.D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, W. Schulte (eds.) Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday, *Lecture Notes in Computer Science*, vol. 9300, pp. 24–51. Springer (2015). DOI 10.1007/978-3-319-23534-9_2
5. Bradley, A.R.: SAT-based model checking without unrolling. In: R. Jhala, D.A. Schmidt (eds.) Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings, *Lecture Notes in Computer Science*, vol. 6538, pp. 70–87. Springer (2011). DOI 10.1007/978-3-642-18275-4_7
6. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: Ice-based refinement type discovery for higher-order functional programs. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I, pp. 365–384 (2018). DOI 10.1007/978-3-319-89960-2_20. URL https://doi.org/10.1007/978-3-319-89960-2_20
7. Champion, A., Kobayashi, N., Sato, R.: Hoice: An ice-based non-linear horn clause solver. In: S. Ryu (ed.) Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings, *Lecture Notes in Computer Science*, vol. 11275, pp. 146–156. Springer (2018). DOI 10.1007/978-3-030-02768-1_8. URL https://doi.org/10.1007/978-3-030-02768-1_8
8. Champion, A., Mebsout, A., Sticksel, C., Tinelli, C.: The Kind 2 model checker. In: Proceedings of CAV 2016, *LNCS*, vol. 9780, pp. 510–517. Springer (2016). DOI 10.1007/978-3-319-41540-6_29
9. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Solving horn clauses on inductive data types without induction. TPLP **18**(3-4), 452–469 (2018). DOI 10.1017/S1471068418000157. URL https://doi.org/10.1017/S1471068418000157
10. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained horn clauses using syntax and data. In: N. Bjørner, A. Gurfinkel (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018, pp. 1–9. IEEE (2018). DOI 10.23919/FMCAD.2018.8603011. URL https://doi.org/10.23919/FMCAD.2018.8603011
11. Freeman, T.S., Pfenning, F.: Refinement types for ML. In: Proceedings of PLDI'91, pp. 268–277. ACM (1991). DOI 10.1145/113445.113468
12. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Proceedings of CAV 2014, *LNCS*, vol. 8559, pp. 69–87. Springer (2014). DOI 10.1007/978-3-319-08867-9_5
13. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Proceedings of POPL 2016, pp. 499–512. ACM (2016). DOI 10.1145/2837614.2837664
14. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: A. Cimatti, R. Sebastiani (eds.) Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7317, pp. 157–171. Springer (2012). DOI 10.1007/978-3-642-31612-8_13
15. Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems - tool paper. In: D. Giannakopoulou, D. Méry (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7436, pp. 247–251. Springer (2012). DOI 10.1007/978-3-642-32759-9_21
16. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: verifying functional programs using abstract interpreters. In: Proceedings of CAV 2011, *LNCS*, vol. 6806, pp. 470–485. Springer (2011). DOI 10.1007/978-3-642-22110-1_38
17. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (2016). DOI 10.1007/s10703-016-0249-4
18. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Proceedings of FASE 2009, *LNCS*, vol. 5503, pp. 470–485. Springer (2009). DOI 10.1007/978-3-642-00593-0_33
19. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Proceedings of ESOP 2014, *LNCS*, vol. 8410, pp. 392–411. Springer (2014)
20. McMillan, K., Rybalchenko, A.: Computing relational fixed points using interpolation. Tech. rep. (2013)
21. Minsky, Y.: Ocaml for the masses. ACM Queue **9**(9), 43 (2011). DOI 10.1145/2030256.2038036
22. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of TACAS 2008, *LNCS*, vol. 4963, pp. 337–340. Springer (2008). DOI 10.1007/978-3-540-78800-3_24
23. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999). DOI 10.1007/978-3-662-03811-6

24. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proceedings of PLDI 2008, pp. 159–169. ACM (2008). DOI 10.1145/1375581.1375602
25. Sato, R., Iwayama, N., Kobayashi, N.: Combining higher-order model checking with refinement type inference. In: Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, pp. 47–53. ACM (2019)
26. Sato, R., Unno, H., Kobayashi, N.: Towards a scalable software model checker for higher-order programs. In: Proceedings of PEPM 2013, pp. 53–62. ACM (2013). DOI 10.1145/2426890.2426900
27. Sharma, R., Aiken, A.: From invariant checking to invariant inference using randomized search. In: Proceedings of CAV 2014, *LNCS*, vol. 8559, pp. 88–105. Springer (2014). DOI 10.1007/978-3-319-08867-9_6
28. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., Nori, A.V.: A data driven approach for algebraic loop invariants. In: Proceedings of ESOP 2013, *LNCS*, vol. 7792, pp. 574–592. Springer (2013). DOI 10.1007/978-3-642-37036-6_31
29. Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Nori, A.V.: Verification as learning geometric concepts. In: Proceedings of SAS 2013, *LNCS*, vol. 7935, pp. 388–411. Springer (2013). DOI 10.1007/978-3-642-38856-9_21
30. Terauchi, T.: Dependent types from counterexamples. In: Proceedings of POPL 2010, pp. 119–130. ACM (2010). DOI 10.1145/1706299.1706315
31. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: Proceedings of PPDP 2009, pp. 277–288. ACM (2009). DOI 10.1145/1599410.1599445
32. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: Proceedings of POPL '13, pp. 75–86. ACM (2013). DOI 10.1145/2429069.2429081
33. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of POPL '99, pp. 214–227. ACM (1999). DOI 10.1145/292540.292560
34. Zhu, H., Jagannathan, S.: Compositional and lightweight dependent type inference for ML. In: Proceedings of VMCAI 2013, *LNCS*, vol. 7737, pp. 295–314. Springer (2013). DOI 10.1007/978-3-642-35873-9_19
35. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: J.S. Foster, D. Grossman (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018, pp. 707–721. ACM (2018). DOI 10.1145/3192366.3192416. URL https://doi.org/10.1145/3192366.3192416
36. Zhu, H., Nori, A.V., Jagannathan, S.: Dependent array type inference from tests. In: Proceedings of VMCAI 2015, *LNCS*, vol. 8931, pp. 412–430. Springer (2015). DOI 10.1007/978-3-662-46081-8_23
37. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: Proceedings of ICFP 2015, pp. 400–411. ACM (2015). DOI 10.1145/2784731.2784766
38. Zhu, H., Petri, G., Jagannathan, S.: Automatically learning shape specifications. In: Proceedings of PLDI 2016, pp. 491–507. ACM (2016). DOI 10.1145/2908080.2908125

Appendix

A Comparing Model Complexity

This section illustrates the difference in complexity between Z3's Spacer and PDR models. We use the Horn clause problem

https://github.com/hopv/benchmarks/blob/master/clauses/lia/termination/up_down01.smt2.

There are 11 predicates to infer in this satisfiable problem. Our predicate synthesis engine HoIce immediately returns the following model where all variables are of type `Int`, and we write • for variables that do not appear in the predicate's definition

predicate	variables	definition
app_1030_{12}	•, •, v_3, •, •, •, •, •	$v_3 = 0$
$fail_{20}$	•	*false*
app_1030_8	v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8	$app\_1030_{12}(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$
app_1030_9	•, •, •, •, •, •, •, •, •	*true*
app_1030_{13}	•, •, •, •, •, •, •, •, •	*true*
$down\_1031_{19}$	•, •, •, •	*true*
up_1032_{24}	v_1, v_2, v_3	$app\_1030_8(v_1, v_2, v_3, 0, 0, 0, 0, 0)$
up_1032_{25}	•, •, •, •	*true*
bot_{15}	•, •	*false*
$fail_{21}$	•, •	*false*
up_1115_{29}	•, •, •, •	*true*

Note that we have abbreviated the name of the predicates slightly for the sake of readability. While this model could be made slightly simpler by inlining the predicate applications, it is already quite concise.

Z3's Spacer also returns immediately with a model that is much more complex. We only look at two of the predicates. First,

$$app\_1030_{12}(v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8)$$

defined as

$$\begin{array}{lll} & \exists (v, v'), & (\,(v = 0) = (0 \le v_1)\,) \wedge \neg(v = 0) \wedge v' \le 0 \\ & & \wedge\ v_2 = 0 \wedge v_3 = 0 \wedge v_4 = 0 \wedge v_5 = 0 \wedge v_6 = 0 \wedge v_7 = 0 \wedge v_8 = 0 \\ \vee & \exists v, & (\,(v = 0) = (v_1 \le 0)\,) \wedge \neg(v = 0) \\ & & \wedge\ v_2 = 0 \wedge v_3 = 0 \wedge v_4 = 0 \wedge v_5 = 0 \wedge v_6 = 0 \wedge v_7 = 0 \wedge v_8 = 0 \end{array}$$

Another example is $fail_{20}(v_1)$, defined as $\exists (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)$,

$$\begin{array}{l} (\,\mathbf{A} \vee \mathbf{B}\,) \wedge \neg(x_1 = 0) \\ \wedge\ (x_7 = -x_3) \wedge (x_5 = -x_3) \wedge (x_4 = -x_2) \\ \wedge\ (\,(x_6 = 0) = (x_4 \le x_5)\,) \\ \wedge\ (\,(x_6 = 0) \vee (x_8 = 0)\,) \\ \wedge\ (\,\neg(x_8 = 0) = (x_7 \ge 0)\,) \\ \wedge\ v_1 = 0 \end{array}$$

where $\mathbf{A}$ is $\exists (x_9, x_{10}, x_{11})$,

$$\begin{array}{l} \neg(x_9 = 0) \wedge (\,(x_{11} = 0) = (0 \le x_{10})\,) \\ \wedge\ (\,(x_9 = 0) = (\mathbf{0} \le \mathbf{x_3})\,) \wedge \neg(x_{11} = 0) \\ \wedge\ x_2 = 0 \wedge x_1 = 0 \end{array}$$

and $\mathbf{B}$ is $\exists (x_9, x_{10}, x_{11})$,

$$\begin{array}{l} \neg(x_9 = 0) \wedge (\,(x_{11} = 0) = (0 \le x_{10})\,) \\ \wedge\ (\,(x_9 = 0) = (\mathbf{x_3} \le \mathbf{0})\,) \wedge \neg(x_{11} = 0) \\ \wedge\ x_2 = 0 \wedge x_1 = 0 \end{array}$$

which is equivalent to *false*: https://rise4fun.com/Z3/XPN.