OBRA: Oracle-based, Relational, Algorithmic Type Verification

Elizaveta Vasilenko[1 [0000−0001−5983−3347]], Niki Vazou[1 [0000−0003−0732−5476]], and Gilles Barthe[1,2 [0000−0002−3853−1777]]

[1] IMDEA Software Institute, Madrid, Spain
[2] MPI-SP, Bochum, Germany

Abstract. Relational logics aim to stablish properties of two expressions by combining synchronous proof rules, which reason about structurally equivalent expressions, with asynchronous proof rules, which only reason about one of the two expressions. As a result, relational logics are not syntax-directed and their algorithmic implementation is challenging. In this work, we design OBRA an algorithmic, relational, and bidirectional type system that only has synchronous rules to preserve predictability and relies on an external oracle to handle syntactic differences. We formalize OBRA and prove that it is equivalent to Relational Higher-Order Logic (RHOL). We implement OBRA by extending Liquid Haskell with synchronous relational rules and using user-provided unary proofs as the external oracle. Further, OBRA automatically translates relational properties to unary theorems with proof templates that can be manually augmented, debugged, and verified using Liquid Haskell. We evaluate OBRA on 12 benchmarks out of which 7 were proved automatically and the rest required smaller or equal proofs than the unary case.

Keywords: refinement types, relational types, liquid types

1 Introduction

Relational higher-order type systems [**?**] and logics [**?**,**?**] have been designed to simplify the verification of relational properties, *i.e.* properties that relate two programs or two runs of the same program. Such systems have been developed to reason about a wide variety of properties, including cost analysis [**?**], differential privacy [**?**], and information flow control [**?**]. As a simple relational property, below we relate how `map` and `filter` modify the length of their input lists:

```
filter ~ map |  ∀f1 ∼ f2. true ⇒ ∀xs1 ∼ xs2. len xs1 ≤ len xs2
                ⇒ len (r1 f1 xs1) ≤ len (r2 f2 xs2)
```

The property states that for each pair of functions `f1` and `f2` and pair of lists `xs1` and `xs2` such that the length of `xs1` is less than or equal to the length of `xs2`, the length of `filter f1 xs1` is less than or equal to the length of `map f2 xs2`, where the special symbols `r1` and `r2` are used to refer to the related functions.

Relational verification of such statements proceeds by unfolding the left-hand side and right-hand side expressions, in our example `filter f1 xs1` and `map f2 xs2`, according to the rules of an inductive system. In type-based systems, the

rules are usually *synchronous* and relate expressions of the same structure, *e.g.* they relate a function application to a function application or an if-expression to an if-expression, but not a function application to an if-expression. Such rules add up to a syntax-directed type system that can automatically establish very precise properties when comparing two similar expressions, like two runs of the same function. However, analysis of syntactically different expressions via type systems is limited [**?**]. Some works, such as [**?**] use example-driven heuristics to deal with program differences in common cases, but as a drawback, such systems lose predictability and hence, user-friendly error reporting.

To systematically analyze expressions of different structure, the Relational Higher-Order Logic (RHOL) [**?**,**?**,**?**] supports *asynchronous* rules. Such rules can choose just one of the related expressions for unfolding, *e.g.* they can compare an application to an if-expression, as required to relate the `map` with the guarded `filter` in our example. The co-existence of both synchronous and asynchronous rules in RHOL leads to a set of typing rules that is not syntax directed, thus reasoning in RHOL is non-algorithmic. Such systems focus on the theoretical aspects of relational logics and are inherently not syntax directed. As a consequence, their implementations have not yet been really explored.

Yet, the proofs conducted in RHOL can be translated to and checked by an algorithmic system, such as the unary refinement type system of Liquid Haskell [**?**]. In theory, this is possible because RHOL is equivalent to HOL (Theorem 3 of [**?**]) and HOL proofs can be encoded in Liquid Haskell (Theorem 3.1 of [**?**]). In practice, both [**?**] and [**?**] manually encoded RHOL proofs in Liquid Haskell and found that the process is feasible but tedious. Most of the times the syntax directed reasoning is used to deconstruct the related terms. Thus, we conjectured that the extension of Liquid Haskell with algorithmic, synchronous rules would ease the proof process of relational properties.

In this work, we design and implement OBRA, an oracle-based, relational, and algorithmic verifier that extends Liquid Haskell with synchronous relational rules. OBRA has a bidirectional, relational type system that not only verifies relational properties, but also translates them to unary Liquid Haskell theorems and automatically generates synchronous proofs templates for them. These proofs are developed using Liquid Haskell as theorem-prover [**?**] and can be interactively adjusted as necessary. This interaction of unary and relational systems, established a novel method of proving relational properties that takes the best of both worlds: our system has the full expressiveness of RHOL via unary proofs, yet supports automation of synchronous reasoning. Moreover, OBRA is the first relational system that generates proofs for relational properties and utilizes an underlying existing unary system. To measure the automation of our method, we used OBRA to prove 12 properties and compared our proof sizes against the existing manual proofs in plain Liquid Haskell [**?**].

Concretely, our contributions are the following:

1) We design an algorithmic, bidirectional, relational, refinement type system on top of a unary refinement type checker that uses a novel oracle rule (rule T-Obl of fig. 4) to make the system complete (theorem 1) *w.r.t.* RHOL.

2) We design a translation from the relational typing statements to unary theorems (§ 3.3) to ease the development of relational proofs. Our system reduces the proof of a relational property to the verification of an equivalent unary theorem. *3)* We implement our system on top of LIQUID HASKELL and evaluate it on 12 examples (table 1). Our evaluation shows that the synchronous rules are sufficient to prove 58% of our benchmarks, while for the rest, our translation generates unary theorems that can be manually completed. In total, our method reduces manual proof effort by 36%, compared to purely manual unary proofs.

2 Overview of OBRA

OBRA receives from the user a pair of expressions and a relational property (§ 2.1), then generates for the property a unary proof term that is potentially incomplete (§ 2.2), and returns it to the user who completes the proof as an oracle (§ 2.3). § 2.4 summarizes the workflow of OBRA.

2.1 Relational Properties

Relational properties express relations between two expressions or two runs of the same expression. As an example, consider the following `map` function such that `map f xs` applies `f` to all the elements of `xs`:

```
map :: (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Our first relational property is that `map` preserves the length inequality when applied to two different lists, expressed as a *relational signature*:

```
relational map ~ map :: (Int → Int) → [Int] → [Int]
                     ~ (Int → Int) → [Int] → [Int]
| ∀f1 ∼ f2. (true ⇒ true) ⇒ ∀xs1 ∼ xs2. len xs1 ≤ len xs2
⇒ len (r1 f1 xs1) ≤ len (r2 f2 xs2)
```

where `len` is inductively defined to return the length of a list. Reading the relational signature from left to right, the `relational` keyword declares that a relational specification follows. Then, `map ~ map` states that the property relates two runs of the `map` function. Next, we provide unary types of related expressions separated by a tilde. To keep our system simple (and reducible to RHOL § 3.4), these types are monomorphic. So, here, we relate two runs of `map` on integers. The final part of the signature is the assertion that states the preservation of length inequality. The assertion is of the form $\forall \texttt{x1} \sim \texttt{x2}.\phi_x \Rightarrow \phi$ and contains one pair of quantifiers for each pair of arguments of the related functions. The first assumption $\forall \texttt{f1} \sim \texttt{f2}.$ `(true ⇒ true)` relates the pair of functions `f1` and `f2` and it is trivial, meaning that their input and output values can be arbitrarily related. Next is the non-trivial assumption $\forall \texttt{xs1} \sim \texttt{xs2}.$ `len xs1 ≤ len xs2` about `xs1` and `xs2`. Under this condition, we prove `len (r1 f1 xs1) ≤ len (r2 f2 xs2)` where `r1` and `r2` are reserved variables for the left and right instances of the related expressions, which here are the two runs of `map`.

Automatic verification of the above relational assertion is very challenging. relSTLC [?] is an automatic verifier that relies on synchronous rules and will fail to prove the above property. Since the assumption `len xs1` $\leq$ `len xs2` does not ensure that the length of the two input lists are equal, there is no guarantee that the two runs of `map` have the same syntactic structure thus, the synchronous rules of relSTLC will fail to apply. relSTLC would trivially prove a length preservation property for `map` that assumes equal length of the two input lists, but fails for inequality. Similarly, the synchronous rules of RHOL [?] will fail to apply, since the input lists do not have the same structure. However, it is still possible to complete the proof in RHOL using asynchronous rules. Concretely, one can first case split on the left list and, at each case, case split on the right list, and at all the four cases evoke higher-order logic to prove the property.

OBRA adopts this exhaustive case split approach as the default synchronous rule and is able to prove the `map` property. Concretely, the T-Case rule of fig. 4 of § 3 will generate four cases for all the structure combinations of the two input lists. Thus, our system is more general than relSTLC, and still algorithmic.

Hence, OBRA verifies the `map` property by case splitting on all combinations of the input list structure. This information alone is not enough to understand why the property holds and, more importantly, how to repair a potentially failing proof. To gain understanding and repair capabilities, OBRA generates a unary proof term that can be checked by Liquid Haskell and inspected by the user.

2.2 Unary Verification Of Relational Properties

As evidence that the `map~map` property holds, OBRA automatically generates a unary theorem, named `mapRmap` with its proof term. Using Liquid Haskell as a theorem prover [?], the theorem is encoded as a refinement type of the `mapRmap` function and its proof as the body definition of `mapRmap`.

Relational to Unary Specification Below is the generated unary theorem `mapRmap`.

```
mapRmap :: f1:(Int → Int) → f2:(Int → Int)
        → f1f2:(x1:Int → x2:Int → x1x2:() → ())
        → xs1:[Int] → xs2:[Int] → xs1xs2:{len xs1 ≤ len xs2}
        → {len (map f1 xs1) ≤ len (map f2 xs2)}
```

In general, for a relational assertion $\forall$`x1` $\sim$ `x2`.$\phi_x \Rightarrow \phi$, OBRA will generate three arguments. The first two arguments are the quantification binders `x1` and `x2` that turn the universal quantification of the relational assertion into a lambda abstraction in the classic "propositions as types" [?] style. The third argument captures the assumption ϕ_x, it is named as the combination of the names of the two quantified variables, *i.e.* `x1x2`, and we call it *relational argument*. The type of the relational argument depends on the types of the arguments it relates. For example, in the refinement type of `mapRmap`, the relational argument for the list arguments `xs1` and `xs2` is `xs1xs2:{len xs1 ≤ len xs2}`, which is a shorthand for `{() | len xs1 ≤ len xs2}`. The relational argument for the function arguments `f1` and `f2` is itself a function. In general, relations on non functional arguments are captured by a refined unit type. Relations on functional arguments

are captured by a function that has three arguments, two for the arguments of the related functions and one for the relation between them. The generation of the unary theorem proceeds until it reaches the non-quantified base case of the relational assertion, which is encoded as a refined unit type. Note that, this encoding of the relational assertion as a unary type eliminates all the relational quantifiers, thus the derived refinement type belongs in the decidable logic of Liquid Haskell. The relational to unary translation is formalized in § 3.4.

The Unary Proof Term Next, let's see how one can prove the `mapRmap` property. The below Haskell definition gets accepted by Liquid Haskell, *i.e.* providing a proof of the `mapRmap` property.

```
mapRmap f1 f2 f1f2 []     []     xs1xs2 = ()
mapRmap f1 f2 f1f2 []     (x:xs) xs1xs2 = () ? map f2 xs
mapRmap f1 f2 f1f2 (x:xs) []     xs1xs2 = ()
mapRmap f1 f2 f1f2 (x:xs) (y:ys) xs1xs2 = mapRmap f1 f2 f1f2 xs ys ()

(?) :: x:a → b → { v:a | x == v }
x ? _ = x
```

We split four cases comparing the structure of the input lists. In the first case both lists are empty, because Liquid Haskell knows that `map f [] = []` and `len [] = 0`, the property holds trivially. Trivial proofs are encoded using the `()` value and are automated using Liquid Haskell's SMT-logic and restricted function unfolding (via the PLE algorithm [**?**]). In the second case we need to prove that `0 ≤ len (map f2 (x:xs))`. There, the missing piece for the proof is that the length of `map f2 xs` is non-negative and to complete the proof we need to write the expression `map f2 xs` and let Liquid Haskell infer its properties. This help is provided by the question mark operator `?` which strengthens the proof environment. The third case is trivial, since the assumption `len (x:xs) ≤ len []` is a contradiction under which the property holds. The last case is completed by an inductive call to `mapRmap`, with a trivial proof as its last argument. Thus the proof is complete.

The Generated Proof Term OBRA generates a proof term for the `mapRmap` theorem that is similar to the above. Before presenting the proof term, we need to emphasize an integral feature of how refinement type checking operates. Unlike type theory style theorem provers, Liquid Haskell is an SMT-based verifier, meaning that the proof term will type check, if the SMT is given enough information to decide the property. Taking this into account, OBRA's automatically generated terms collect all the information gathered by the two related terms, even if they are not required to show the validity of the property.

The generated proof term for `mapRmap` is verbose, but has the same structure and contains the same information as the previous, user-defined proof.

```
mapRmap f1 f2 f1f2 xs1 xs2 xs1xs2
  = case xs1 of
      [] → case xs2 of
        [] → () ? [] ? []
        x2 : xs2 → () ? [] ? (f2 x2 : map f2 xs2)
```

```
x1 : xs1 → case xs2 of
  [] → () ? (f1 x1 : map f1 xs1) ? []
  x2 : xs2 →
   (\x1 x2 x1x2 x3 x4 x3x4 → () ? x1x2 ? x3x4 ? (x1:x3) ? (x2:x4))
   (f1 x1) (f2 x2) (f1f2 x1 x2 (() ? x1 ? x2))
   (map f1 xs1) (map f2 xs2)
   (mapRmap f1 f2 f1f2 xs1 xs2 (() ? xs1 ? xs2))
```

The proof case splits on the four potential structures of the input lists. In the first three cases, OBRA ensures that all the subexpressions that appear in the related expressions, *e.g.* `f2 x2:map f2 xs`, `[]`, *etc.*, appear in the proof. In the last case, generated by the rule T-App of fig. 3, a lambda abstraction strengthens the proof environment with both the higher-order relational argument and the inductive hypothesis, both fully applied. The proof term is machine generated, thus it is verbose, but it is validated by Liquid Haskell. The complete generation rules are presented in § 3 and follow the program product method of [**?**].

2.3 Oracle-based Proofs

In many cases (5 out of the 12 benchmarks in table 1), synchronous rules are not sufficient to prove the relational property. This happens when the two related programs have different control flows. For example, let's compare `map` to `filter`:

```
filter :: (a → Bool) → [a] → [a]
filter _ []     = []
filter f (x:xs) = if f x then x : filter f xs else filter f xs
```

Now we can revisit the relational property of `map` and `filter`, from § 1:

```
relational filter ~ map :: (Int → Bool) → [Int] → [Int]
        ~ (Int → Int) → [Int] → [Int]
        | ∀f1 ∼ f2. (true ⇒ true) ⇒ ∀xs1 ∼ xs2. len xs1 ≤ len xs2
        ⇒ len (r1 f1 xs1) ≤ len (r2 f2 xs2)
```

The above property of length inequality preservation holds, but its proof is not possible using only synchronous rules. Thus, OBRA will translate the relational signature to a refinement type that is equivalent to the relational property, and will also provide an inhabitant of this type that is "incomplete" because the inductive step is proved by calling the below lemma `oblig`:

```
oblig :: f1:(Int → Bool) → f2:(Int → Int) → (Int → Int → () → ())
      → x1:Int → xs1:[Int] → x2:Int → xs2:[Int]
      → xs1xs2:{len (x1:xs1) ≤ len (x2:xs2)}
      → {len (filter f1 (x1:xs1)) ≤ len (map f2 (x2:xs2))}
```

The generated proof term is similar to the `mapRmap` proof term in that it also has four cases and the three cases follow the same pattern. The difference is that in the last case, *i.e.* when both lists are non-empty, the structure of the `filter` and `map` expressions is different thus no synchronous rule can be applied. In this case (rule T-Obl of fig. 4), OBRA generates a call to the `oblig` lemma.

Generation of Obligations The obligation is an expression that proves the missing requirement. In our example, an expression that proves that `len (filter f1 (x1:xs1)) ≤ len (map f2 (x2:xs2))`. This proof can be done in the current

proving environment, *i.e.* using all the arguments of `filterRmap`, the inductive hypothesis, and the variables introduced by the case splitting. For user interactivity, instead of requiring an in-place proof, OBRA captures this current proving environment as arguments of the obligation lemma and asks the user to complete the proof, without the need to understand the complete proof term.

In this example, the user can complete the proof manually, as follows:

```
oblig f1 f2 f1f2 x1 xs1 x2 xs2 xs1xs2 =
  if f1 x1 then filterRmap f1 f2 f1f2 xs1 xs2 ()
           else filterRmap f1 f2 f1f2 xs1 xs2 ()
```

The body of the obligation handles the structural difference between `filter` and `map`, namely the branching that is only present in `filter`. In the case of the branching when `f1 x1` is true, both `filter` and `map` append a new element to the recursive results. Hence, the property is reduced to the comparison between lengths `len (filter f1 xs1)` $\leq$ `len (map f2 xs2)`. This holds by the inductive hypothesis `filterRmap` which concludes the proof. In the other case, only `map` appends the element to the list, while `filter` skips the element for which `f1 x1` is false. The same inductive hypothesis can be used here to show that `len (filter f1 xs1)` $\leq$ `len (map f2 xs2)` and thus `len (filter f1 xs1)` $\leq$ `len (map f2 xs2) + 1` which is the desired property in the second case.

2.4 OBRA Workflow

Figure 1 presents the workflow of OBRA that given a pair of expressions $e_1 \sim e_2$ and a relational property ϕ interacts with an oracle to decide if the pair satisfies ϕ. First, OBRA is using a relational analysis system (`RA`; which is essentially the synchronous subset of RHOL) to generate a unary refinement type u, a unary proof term e, and a set of proof obligations O. Second, a unary analysis system (`UA`; *e.g.* Liquid Haskell) is used to check that e has type u. In this step also, OBRA interacts with the oracle to provide inhabitants for the proof obligations O. The final result of OBRA is the result of `UA`.

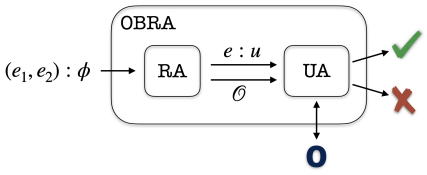

Fig. 1: OBRA workflow. `RA` is relational analysis, `UA` is unary refinement type checking, *e.g.* Liquid Haskell, and `O` is the oracle, *e.g.* a user.

The translation is designed so that e has type u, so when the proof obligations O are empty, *i.e.* in synchronous proofs, OBRA is automatic.

In § 3.4 we show that if the oracle has the proof power of higher-order logic, then OBRA is equivalent to RHOL. Yet, compared to RHOL, OBRA is algorithmic and reduces to a quantifier-free, unary refinement type checking. We claim that this reduction generates proof obligations that are simpler than the original relational property and thus, the proof effort is reduced. To evaluate

$$
\begin{array}{rrcl}
\textit{Constants} & c & ::= & \textit{true} \mid \textit{false} \mid () \mid i \in \mathbb{Z} \mid +, -, *, /, =, \wedge, \neg \mid \texttt{nil}_t \mid \texttt{cons}_t \\
\textit{Expressions} & e, p & ::= & c \mid x, f, o, \mathbf{r_1}, \mathbf{r_2} \in V \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{rec}\ f = (\lambda x.e) : t \\
& & & \mid \lambda x : t.e \mid e\ x \mid \texttt{if}\ x\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{case}\ x\ \{\texttt{nil} \mapsto e; \texttt{cons}\ x\ x \mapsto e\} \\
\textit{Base types} & b & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{list}\ t \\
\textit{Unrefined types} & t, s & ::= & b \mid t \to t \\
\textit{Refined types} & u, v & ::= & b\{\nu : p\} \mid x : u \to u \\
\textit{Assertion} & \phi, \psi & ::= & p \mid \forall x_1 : t_1, x_2 : t_2. \phi \Rightarrow \phi \\
\textbf{\textit{Environments}} & & & \\
\textit{Typing} & \Gamma & ::= & \emptyset \mid \Gamma; x : t \\
\textit{Relational} & \Phi & ::= & \emptyset \mid \Phi; x_1 \sim x_2 : t_1 \sim t_2 \mid \phi \\
\textit{Refined} & \mathrm{R} & ::= & \emptyset \mid \mathrm{R}; x : u \\
\textit{Translation} & \mathrm{T} & ::= & \emptyset \mid \mathrm{T}; x_1 \sim x_2 \rightsquigarrow x \\
\textit{Obligation} & \mathrm{O} & ::= & \emptyset \mid \mathrm{O}; \mathrm{R} \vdash o : u
\end{array}
$$

Fig. 2: Syntax of λ_{OBRA}.

this claim, in § 4, we use OBRA to prove 12 relational properties and conclude that OBRA can automatically prove 58% of them and reduces the proof effort by 36% in lines of code compared to the unary proofs of the same properties.

3 Formalization of OBRA

Here, we formalize OBRA as the core calculus λ_{OBRA} that is *relational*, *synchronous*, *bidirectional*, and *SMT-aided*. § 3.1 and § 3.2 respectively present the syntax and typing rules of λ_{OBRA}. In § 3.3 we translate λ_{OBRA} to a unary, refined system. Finally, in § 3.4 we prove that λ_{OBRA} is equivalent to the sound RHOL.

3.1 Syntax

Figure 2 introduces the syntax of λ_{OBRA}. We define expressions, unrefined and refined types, assertions, and five environments. We use magenta color for elements of the language that only appear in the translation, described in § 3.3.

Constants in λ_{OBRA} include booleans, unit, integers and operators for arithmetic $(+, -, *, /)$, equality $(=)$, and boolean logic $(\wedge$ and $\neg)$. Finally, constants include the type indexed list constructors $\texttt{nil}_t$ and $\texttt{cons}_t$.

Expressions include constants and variables, including the special relational variables $\mathbf{r_1}$ and $\mathbf{r_2}$ and variables o that capture proof obligations. Expressions are in A-normal form (ANF), *i.e.* function arguments and branching conditions must be variables. Recursive and lambda functions are type annotated.

Types of λ_{OBRA} are either unrefined or refined. *Unrefined types* t, s can either be the base types b, that is, unit, boolean, integer, and list or function types. In *refined types* u, v the base type $b\{\nu : p\}$ is refined by a predicate p that comes from the language of expressions and in the function type $x : u_x \to u$ the argument is bound by a variable x that is used in the refinement of the return type u.

Notation: We use an unrefined type to denote the corresponding refined type with only *true* refinements, *e.g.* `unit` is shorthand for $\texttt{unit}\{\nu : \mathit{true}\}$.

Assertions ϕ, ψ are logical predicates that encode relational properties. As such, they are always quantified by two typed variables x_1, x_2 that represent the two sides of the relation. The body of the assertion is an implication and its base case a boolean expression. Since two expressions can be related by two unary types and an assertion, we call the triplet $t_1 \sim t_2 \mid \phi$ a *relational type. Notation:* We write $\forall x_1 x_2.\phi \Rightarrow \psi$ for $\forall x_1 : t_1 x_2 : t_2.\phi \Rightarrow \psi$ when the types are implied.

λ_{OBRA} has five *environments*. The typing environment Γ binds variables to unrefined types $x : t$ and the refined environment R binds variables to refined types $x : u$. The relational environment Φ binds the pair of variables $x_1 \sim x_2$ to their relational type, while the translation environment T binds pairs of variables to a new variable, that captures their relation. Finally, the obligation environment O collects a set of obligations, each of which is a unary typing judgment.

Synthesis and Translation $\boxed{\Gamma \mid \Phi \mid \mathrm{T} \vdash e_1 \sim e_2 \Rightarrow t_1 \sim t_2 \mid \phi \mathbin{+} e : u \mid \mathrm{O}}$

$$\frac{\text{constTy}(c_1) = t_1 \quad \text{constTy}(c_2) = t_2 \quad \text{constPr}(t_1, t_2, \mathbf{r_1} = c_1 \wedge \mathbf{r_2} = c_2) = \phi}{\Gamma \mid \Phi \mid \mathrm{T} \vdash c_1 \sim c_2 \Rightarrow t_1 \sim t_2 \mid \phi \mathbin{+} \text{constTr}(t_1, t_2) : \texttt{trTy}(t_1, t_2) \mid \emptyset}\ \text{T-Const}$$

$$\frac{x_1 \sim x_2 : t_1 \sim t_2 \mid \phi \in \Phi \quad x_1 \sim x_2 \rightsquigarrow x \in \mathrm{T}}{\Gamma \mid \Phi \mid \mathrm{T} \vdash x_1 \sim x_2 \Rightarrow t_1 \sim t_2 \mid \phi \mathbin{+} x : \lceil \phi \rceil [x_1/\mathbf{r_1}][x_2/\mathbf{r_2}] \mid \emptyset}\ \text{T-Var}$$

$$\frac{\begin{array}{c}\Gamma \mid \Phi \mid \mathrm{T} \vdash x_1 \sim x_2 \Leftarrow s_1 \sim s_2 \mid \psi[\mathbf{r_1}/x_1][\mathbf{r_2}/x_2] \mathbin{+} e_t : u_t \mid \mathrm{O}_2 \\ t_t \doteq x_1 : s_1 \to x_2 : s_2 \to x_t : u_t \to u \quad \phi' \doteq \phi[\mathbf{r_1}\ x_1/\mathbf{r_1}][\mathbf{r_2}\ x_2/\mathbf{r_2}] \\ \Gamma \mid \Phi \mid \mathrm{T} \vdash e_1 \sim e_2 \Rightarrow s_1 \to t_1 \sim s_2 \to t_2 \mid \forall x_1 : s_1, x_2 : s_2.\psi \Rightarrow \phi' \mathbin{+} e : t_t \mid \mathrm{O}_1\end{array}}{\Gamma \mid \Phi \mid \mathrm{T} \vdash e_1\ x_1 \sim e_2\ x_2 \Rightarrow t_1 \sim t_2 \mid \phi \mathbin{+} e\ x_1\ x_2\ e_t : u \mid \mathrm{O}_1, \mathrm{O}_2}\ \text{T-App}$$

Fig. 3: Relational Typing Synthesis and Translation.

3.2 RA: Relational Algorithmic Typing

OBRA's main typing judgment $\Gamma \mid \Phi \mid \mathrm{T} \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi \mathbin{+} e : u \mid \mathrm{O}$ checks that under the typing environment Γ and the relational environment Φ, the expressions $e_1 \sim e_2$ have, *resp.* types t_1 and t_2 and satisfy the relational property ϕ. The judgement further keeps track of the translation environment T to generate a unary expression e of refinement type u that is equivalent to the relational property ϕ, assuming the set of proof obligations O holds. For clarity of exposition, we separate the two functionalities of the judgements. Here, we describe the relational part of the judgement, in § 3.3 we describe the translation part (marked with magenta color in the rules and for simplicity omitted in this subsection), and in § 3.4 we combined them again to prove correctness of OBRA.

Checking and Translation $\boxed{\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e : u \,|\, \mathrm{O}}$

$$\frac{\begin{array}{c}\Gamma; x_1 : s_1; x_2 : s_2 \,|\, \Phi; x_1 \sim x_2 : \psi \,|\, \mathrm{T}; x_1 \sim x_2 \rightsquigarrow x \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e : u \,|\, \mathrm{O} \\ e_t \doteq \lambda x_1 : s_1,\ x_2 : s_2,\ x : \mathtt{trTy}(s_1, s_1).e \quad \phi' \doteq \phi[\mathbf{r_1}\ x_1/\mathbf{r_1}][\mathbf{r_2}\ x_2/\mathbf{r_2}] \\ u_t \doteq x_1 : s_1 \to x_2 : s_2 \to x : \lceil \psi[x_1/\mathbf{r_1}][x_2/\mathbf{r_2}] \rceil \to u\end{array}}{\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash \lambda x_1 : s_1.e_1 \sim \lambda x_2 : s_2.e_2 \Leftarrow s_1 \to t_1 \sim s_2 \to t_2 \,|\, \forall x_1 x_2.\psi \Rightarrow \phi' \mathbin{+} e_t : u_t \,|\, \mathrm{O}}\ \text{T-Lam}$$

$$\frac{\begin{array}{c}t_1 = s_{x1} \to s_1 \qquad t_2 = s_{x2} \to s_2 \qquad \mathrm{Def}(f_1, x_1, e_1) \qquad \mathrm{Def}(f_2, x_2, e_2) \\ \Gamma_r \,|\, \Phi_r \,|\, \mathrm{T}_r \vdash e_1 \sim e_2 \Leftarrow s_1 \sim s_2 \,|\, \phi \mathbin{+} e : u \,|\, \mathrm{O} \qquad \Gamma_r \doteq \Gamma; f_1 : t_1; f_2 : t_2; x_1 : s_{x1}; x_2 : s_{x2} \\ \phi_r \doteq \forall y_1, y_2.(|\,y_1\,|, |\,y_2\,|) < (|\,x_1\,|, |\,x_2\,|) \wedge \psi[y_1/x_1][y_2/x_2] \Rightarrow \phi \\ \Phi_r \doteq \Phi; f_1 \sim f_2 : \phi_r; x_1 \sim x_2 : s_{x1} \sim s_{x2} \mid \psi \qquad \mathrm{T}_r \doteq \mathrm{T}; f_1 \sim f_2 \rightsquigarrow f; x_1 \sim x_2 \rightsquigarrow x \\ e_r \doteq \mathtt{rec}\ f = (\lambda x_1 : s_{x1}, x_2 : s_{x2}, x : \mathtt{trTy}(s_{x1}, s_{x2}).e) : t_r \\ t_r \doteq x_1 : s_{x1} \to x_2 : s_{x2} \to x : \lceil \phi[x_1/\mathbf{r_1}][x_2/\mathbf{r_2}] \rceil \to u \\ \phi_t \doteq \forall x_1 : s_{x1}, x_2 : s_{x2}.\psi \Rightarrow \phi[\mathbf{r_1}\ x_1/\mathbf{r_1}][\mathbf{r_2}\ x_2/\mathbf{r_2}]\end{array}}{\Gamma \,|\, \Psi \,|\, \mathrm{T} \vdash \mathtt{rec}\ f_1 = (\lambda x_1.e_1) : t_1 \sim \mathtt{rec}\ f_2 = (\lambda x_2.e_2) : t_2 \Leftarrow t_1 \sim t_2 \,|\, \phi_t \mathbin{+} e_r : t_r \,|\, \mathrm{O}}\ \text{T-Rec}$$

$$\frac{\begin{array}{c}\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash e_{x1} \sim e_{x2} \Rightarrow s_1 \sim s_2 \,|\, \psi \mathbin{+} e_x : u_x \,|\, \mathrm{O}_1 \qquad e_t \doteq \mathtt{let}\ (x_1, x_2, x) = (e_{x1}, e_{x2}, e_x)\ \mathtt{in}\ e \\ \Gamma; x_1 : s_1; x_2 : s_2 \,|\, \Phi; x_1 \sim x_2 : s_1 \sim s_2 \,|\, \psi \,|\, \mathrm{T}; x_1 \sim x_2 \rightsquigarrow x \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e : u \,|\, \mathrm{O}_2\end{array}}{\Gamma \,|\, \Psi \,|\, \mathrm{T} \vdash \mathtt{let}\ x_1 = e_{x1}\ \mathtt{in}\ e_1 \sim \mathtt{let}\ x_2 = e_{x2}\ \mathtt{in}\ e_2 \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_t : u \,|\, \mathrm{O}_1, \mathrm{O}_2}\ \text{T-Let}$$

$$\frac{\begin{array}{c}\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash x_1 \sim x_2 \Rightarrow \mathtt{bool} \sim \mathtt{bool} \,|\, p \mathbin{+} x : u_x \,|\, \mathrm{O}_x \\ \Gamma \,|\, \Phi; x_1 \sim x_2 : \mathtt{bool} \sim \mathtt{bool} \,|\, p \wedge x_1 \wedge x_2 \,|\, \mathrm{T} \vdash e_{t1} \sim e_{t2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{tt} : u \,|\, \mathrm{O}_{tt} \\ \Gamma \,|\, \Phi; x_1 \sim x_2 : \mathtt{bool} \sim \mathtt{bool} \,|\, p \wedge x_1 \wedge \neg x_2 \,|\, \mathrm{T} \vdash e_{t1} \sim e_{f2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{tf} : u \,|\, \mathrm{O}_{tf} \\ \Gamma \,|\, \Phi; x_1 \sim x_2 : \mathtt{bool} \sim \mathtt{bool} \,|\, p \wedge \neg x_1 \wedge x_2 \,|\, \mathrm{T} \vdash e_{f1} \sim e_{t2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{ft} : u \,|\, \mathrm{O}_{ft} \\ \Gamma \,|\, \Phi; x_1 \sim x_2 : \mathtt{bool} \sim \mathtt{bool} \,|\, p \wedge \neg x_1 \wedge \neg x_2 \,|\, \mathrm{T} \vdash e_{f1} \sim e_{f2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{ff} : u \,|\, \mathrm{O}_{ff} \\ e_t \doteq \mathtt{if}\ x_1\ \mathtt{then}\ (\mathtt{if}\ x_2\ \mathtt{then}\ e_{tt}\ \mathtt{else}\ e_{tf})\ \mathtt{else}\ (\mathtt{if}\ x_2\ \mathtt{then}\ e_{ft}\ \mathtt{else}\ e_{ff}) \\ \mathrm{O} \doteq \mathrm{O}_x, \mathrm{O}_{tt}, \mathrm{O}_{tf}, \mathrm{O}_{ft}, \mathrm{O}_{ff}\end{array}}{\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash \mathtt{if}\ x_1\ \mathtt{then}\ e_{t1}\ \mathtt{else}\ e_{f1} \sim \mathtt{if}\ x_2\ \mathtt{then}\ e_{t2}\ \mathtt{else}\ e_{f2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_t : u \,|\, \mathrm{O}}\ \text{T-If}$$

$$\frac{\begin{array}{c}\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash x_1 \sim x_2 \Rightarrow \mathtt{list}\ s_1 \sim \mathtt{list}\ s_2 \,|\, p \mathbin{+} e : \_ \,|\, \mathrm{O}_x \\ \Gamma_t \doteq \Gamma; y_1 : s_1; y_2 : s_1; z_1 : \mathtt{list}\ s_1; z_2 : \mathtt{list}\ s_2 \qquad \Phi_t \doteq \Phi; y_1 \sim y_2 : true; z_1 \sim z_2 : true \\ \mathrm{T}_t \doteq \mathrm{T}; y_1 \sim y_2 \rightsquigarrow y; z_1 \sim z_2 \rightsquigarrow z \\ \Gamma_t \,|\, \Phi_t; x_1 \sim x_2 : p \wedge x_1 = \mathtt{nil} \wedge x_2 = \mathtt{nil} \,|\, \mathrm{T}_t \vdash e_{n1} \sim e_{n2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{nn} : u \,|\, \mathrm{O}_{nn} \\ \Gamma_t \,|\, \Phi_t; x_1 \sim x_2 : p \wedge x_1 = \mathtt{nil} \wedge x_2 = \mathtt{cons}\ y_2\ z_2 \,|\, \mathrm{T}_t \vdash e_{n1} \sim e_{c2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{nc} : u \,|\, \mathrm{O}_{nc} \\ \Gamma_t \,|\, \Phi_t; x_1 \sim x_2 : p \wedge x_1 = \mathtt{cons}\ y_1\ z_1 \wedge x_2 = \mathtt{nil} \,|\, \mathrm{T}_t \vdash e_{c1} \sim e_{n2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{cn} : u \,|\, \mathrm{O}_{cn} \\ \Gamma_t \,|\, \Phi_t; x_1 \sim x_2 : p \wedge x_1 = \mathtt{cons}\ y_1\ z_1 \wedge x_2 = \mathtt{cons}\ y_2\ z_2 \,|\, \mathrm{T}_t \vdash e_{c1} \sim e_{c2} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_{cc} : u \,|\, \mathrm{O}_{cc} \\ e_t \doteq \mathtt{case}\ x_1\ \{\mathtt{nil} \mapsto e_{tn}; \mathtt{cons}\ y_1\ z_1 \mapsto e_{tc}\} \\ e_{tn} \doteq \mathtt{case}\ x_2\ \{\mathtt{nil} \mapsto e_{nn}; \mathtt{cons}\ y_2\ z_2 \mapsto e_{nc}\} \\ e_{tc} \doteq \mathtt{case}\ x_2\ \{\mathtt{nil} \mapsto e_{cn}; \mathtt{cons}\ y_2\ z_2 \mapsto e_{cc}\} \\ \mathrm{O} \doteq \mathrm{O}_x, \mathrm{O}_{nn}, \mathrm{O}_{nc}, \mathrm{O}_{cn}, \mathrm{O}_{cc}\end{array}}{\begin{array}{l}\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash \mathtt{case}\ x_1\ \{\mathtt{nil} \mapsto e_{n1}; \mathtt{cons}\ y_1\ z_1 \mapsto e_{c1}\} \\ \qquad \sim \mathtt{case}\ x_2\ \{\mathtt{nil} \mapsto e_{n2}; \mathtt{cons}\ y_2\ z_2 \mapsto e_{c2}\} \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e_t : u \,|\, \mathrm{O}\end{array}}\ \text{T-Case}$$

$$\frac{\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash e_1 \sim e_2 \Rightarrow t_1 \sim t_2 \,|\, \psi \mathbin{+} e : \_ \,|\, \mathrm{O} \quad \Gamma \mid \Phi \vdash t_1 \sim t_2 \,|\, \psi \prec : \phi}{\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} e : \lceil \phi \rceil [e_1/\mathbf{r_1}][e_2/\mathbf{r_2}] \,|\, \mathrm{O}}\ \text{T-Sub}$$

$$\frac{\text{no other rule applies} \qquad \Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad \text{fresh } o \qquad u = \lceil \phi[e_1/\mathbf{r_1}][e_2/\mathbf{r_2}] \rceil}{\Gamma \,|\, \Phi \,|\, \mathrm{T} \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \,|\, \phi \mathbin{+} o : u \,|\, \mathrm{O}; \lceil \Gamma; \Phi; \mathrm{T} \rceil \vdash o : u}\ \text{T-Obl}$$

Fig. 4: Relational Typing Checking and Translation.

Subtyping $\boxed{\Gamma \mid \Phi \vdash t_1 \sim t_2 \mid \psi \prec: \phi}$

$$\frac{\mathsf{SmtValid}(\langle\!|\Phi|\!\rangle \Rightarrow \langle\!|p_1|\!\rangle \Rightarrow \langle\!|p_2|\!\rangle)}{\Gamma \mid \Phi \vdash b_1 \sim b_2 \mid p_1 \prec: p_2}\text{S-Base}$$

$$\frac{\begin{array}{c}\Gamma \mid \Phi \vdash t_{x1} \sim t_{x2} \mid \phi_x[\mathbf{r_1}/x_1][\mathbf{r_2}/x_2] \prec: \psi_x[\mathbf{r_1}/x_1][\mathbf{r_2}/x_2] \\ \Gamma; x_1 : t_{x1}; x_2 : t_{x2} \mid \Phi; x_1 \sim x_2 : \phi_x \vdash t_1 \sim t_2 \mid \psi \prec: \phi\end{array}}{\begin{array}{r}\Gamma \mid \Phi \vdash x_1 : t_{x1} \to t_1 \sim x_2 : t_{x2} \to t_2 \mid \forall x_1 : t_{x1}, x_2 : t_{x2}.\psi_x \Rightarrow \psi[\mathbf{r_1}\ x_1/\mathbf{r_1}][\mathbf{r_2}\ x_2/\mathbf{r_2}] \\ \prec: \forall x_1 : t_{x1}, x_2 : t_{x2}.\phi_x \Rightarrow \phi[\mathbf{r_1}\ x_1/\mathbf{r_1}][\mathbf{r_2}\ x_2/\mathbf{r_2}]\end{array}}\text{S-Fun}$$

Fig. 5: Relational Subtyping.

Figures 3 and 4 present the rules of OBRA which are a bidirectional and synchronous variant of RHOL [**?**]. If the translation is ignored, our rules comprise a subset of RHOL: a weakening rule as well as all synchronous (two-sided) rules. Our branching (rule T-If) and pattern-matching (rule T-Case) rules are modified to be more permissive. That is, the T-If rule has 5 premises, one that derive the guard predicate p, and 4 that handle the four possible cases of the guard, unlike RHOL, where the cases of the guard should match. The same applies to the T-Case rule that has 5 cases in OBRA and 3 cases in RHOL. This design choice allows us to type more programs while maintaining the system syntax-directed. Both rules are still derivable in RHOL.

Unlike RHOL, our rules are *bidirectional* [**?**], namely the typing judgment is split into two judgments: type synthesis $\Gamma \mid \Phi \vdash e_1 \sim e_2 \Rightarrow t_1 \sim t_2 \mid \phi$ in fig. 3 and typechecking $\Gamma \mid \Phi \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi$ in fig. 4. Both of the judgments establish that type $t_1 \sim t_2 \mid \phi$ can be assigned to a pair of expressions $e_1 \sim e_2$ under Γ and Φ. Our bidirectional system is in the spirit of unary refinement types [**?**], which enabled decidable checking of properties within quantifier-free logic of linear arithmetic and uninterpreted functions (QF-EUFLIA) [**?**].

The second property of our system is that it is *synchronous*. All checking and synthesis rules of our system assume the same syntactic structure of expressions on the left- and right-hand side, *e.g.* it is possible to relate two variables or two lambda-expressions, but not a lambda to a variable.

Relational Subtyping Figure 5 describes relational subtyping. Informally, the judgment $\Gamma \mid \Phi \vdash t_1 \sim t_2 \mid \psi \prec: \phi$ states that the relational type $t_1 \sim t_2 \mid \phi$ is less specific than the type $t_1 \sim t_2 \mid \psi$, *i.e.* two expressions related by $t_1 \sim t_2 \mid \psi$ could also be related by $t_1 \sim t_2 \mid \phi$ with some loss of information.

Rule S-Base applies to base types related by quantification-free predicates p_1 and p_2. In the premise, an SMT-automated procedure checks that the predicate p_1 implies p_2 under the quantifier-free interpretation of the relational environment Φ. To get this interpretation, we conjunct all the predicates of the base

types in Φ, after substituting the aliases $\mathbf{r_1}$ and $\mathbf{r_2}$ with the variables x_1 and x_2:

$$(\!|\Phi|\!) \doteq \bigwedge \{(\!|p[x_1/\mathbf{r_1}][x_2/\mathbf{r_2}]|\!) \mid x_1 \sim x_2 : b_1 \sim b_2 \mid p \in \Phi\}$$

To ensure SMT-decidable implication checking, we further use a logical embedding for the predicates that substitutes the functions and the recursive definitions with uninterpreted functions and is a homomorphism for all other cases:

$$(\!|\lambda x\!:\!t.e|\!) \doteq f \quad (\!|\texttt{rec}\ f = (\lambda x.e)\!:\!t|\!) \doteq f \quad (\!|e\ x|\!) \doteq (\!|e|\!)\ x \quad \ldots$$

Rule S-Fun follows the contravariant behavior of function types, where the relational variables $\mathbf{r_1}$ and $\mathbf{r_2}$ are substituted to capture the related types.

Relational Type Synthesis Figure 3 summarizes the rules of relational type synthesis. Rule T-Var fetches the relation between the variables from the relational environment and rule T-App handles function application. Rule T-Const produces a relational type for two constants of compatible types. The function constTy returns the type of a constant. To both check the type compatibility of the constants and generate the relational predicate the T-Const rule uses the partial function constPr that is defined as follows:

$$\begin{aligned} \text{constPr}(b_1, b_2, p) &\doteq p \\ \text{constPr}(t_{x1} \to t_1, t_{x2} \to t_2, p) &\doteq \forall x_1 x_2.\text{constPr}(t_{x1}, t_{x2}, \mathit{true}) \Rightarrow \text{constPr}(t_1, t_2, p) \end{aligned}$$

For two base types, constPr produces a trivial predicate $\mathbf{r_1} = c_1 \wedge \mathbf{r_2} = c_2$ that says that the constants are equal to themselves. For a pair of function types, constPr generates fresh variables x_1, x_2 to bind the arguments and inductively calls itself on the argument types with the predicate *true*. In the return types of the inductive call the predicate p is unchanged. When the two types do not have the same structure, the function is undefined, thus the T-Const rule fails.

Relational Type Checking Figure 4 presents the type checking mode of the system. Rule T-Lam validates a pair of functions against the relation $\forall x_1 x_2.\psi \Rightarrow \phi$. We propagate ϕ to a checking judgment between the bodies e_1 and e_2 while extending the environments Γ and Φ with the assumptions about the arguments. In a similar manner, rule T-Rec handles definitions of recursive functions f_1 and f_2. We add f_1 and f_2 to the typing environment and extend Φ with a relational inductive hypothesis $f_1 \sim f_2 \mid \forall x_1 x_2.\psi \Rightarrow \phi$. For brevity, we omit the respective types $s_1 \to t_1$ and $s_2 \to t_2$ of f_1 and f_2 from the full syntax of relational typing. As in RHOL, the rule requires that the recursive functions terminate using the predicate Def that verifies that the function's domain is an inductive type or an integer and that the recursive call is made on a smaller argument.

Rule T-If analyses synchronous branching on x_1 and x_2 in the two compared programs respectively. There are four possible scenarios: conditions in both programs pass, only the condition on the left passes while the right one fails, only the right condition passes, and both conditions fail. In each of the cases, the rule preserves any additional relation p that was known about x_1 and x_2 prior

to branching. All in all, the rule has five premises, the first of which is used to synthesize p. In the other four premises, the rule extends Φ with an assumption about x_1 and x_2 in which p gets strengthened by the (un)satisfied branching conditions. Then, the rule contraposes the branches of the two programs according to the triggered conditions. Finally, it checks that all four resulting configurations conform to the relation ϕ. The premises of the rule T-Case comprise a very similar cross-product, but on the list constructors `nil` and `cons` instead of the booleans *true* and *false*. In the premises that correspond to the matching of pattern `cons` y_1 z_1 (or *resp.* `cons` y_2 z_2), the binders for the head y_1 (*resp.* y_2) and the tail z_1 (*resp.* z_2) of the list are added to the typing environment Γ.

Rule T-Let compares two let-expressions and rule T-Sub ascribes $t_1 \sim t_2 \mid \phi$ to e_1 and e_2 as long as ϕ subsumes the relation ψ, produced by synthesis.

Finally, rule T-Obl is used when no other rule applies and its only premises $\Gamma \vdash e_1 : t_1$ and $\Gamma \vdash e_2 : t_2$ require that the expressions have the correct unrefined types. Using relational type checking alone, *i.e.* while ignoring the translation, this rule is not correct, since the assertion ϕ is totally ignored. Essentially, reaching this rule means that the relational rules of OBRA failed and an oracle, called via the translation mechanism, is needed to prove the relational property.

3.3 OB: Oracle-Based Translation

Now, we revisit the OBRA judgment $\Gamma \mid \Phi \mid \mathrm{T} \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi + e : u \mid \mathrm{O}$ with a focus on the magenta translation-related parts. Intuitively (as formalized in § 3.4), the judgement generates an expression e, a refinement type u, and a set of obligations O; the expression e has type u and the obligations O are satisfiable *if and only if* the expressions $e_1 \sim e_2$ satisfy the relational predicate ϕ. The output e, also known as *program product* [**?**], combines e_1 and e_2 into a common control flow that simulates the simultaneous execution of the two programs. Similarly, we produce a proof of the relational property that is sensitive to mutual branching and case-splitting in the compared expressions.

The crux of the translation is the rule T-Obl. When nothing else can be done, T-Obl generates a refinement type u that translates the relational predicate ϕ. To turn assertions into refinement types, we follow [**?**,**?**] and turn forall-quantification and implication into functional arguments, and boolean predicates into refinements of the unit type, using the function $\lceil\phi\rceil$:

$$\lceil p \rceil \doteq \mathtt{unit}\{\nu : p\} \qquad \lceil \forall x_1 : t_1, x_2 : t_2 . \psi \Rightarrow \phi \rceil \doteq x_1 : t_1 \to x_2 : t_2 \to x : \lceil \psi \rceil \to \lceil \phi \rceil$$

The translated refinement type is added to the set of obligations O with a fresh obligation variable o under a refinement environment that captures the current state of the translation. The definition of o should be later (fig. 6) guessed by the oracle. To capture the state of the translation, we use the operator $\lceil \Gamma ; \Phi ; \mathrm{T} \rceil$ that returns a refinement type environment as follows:

$$\lceil \Gamma ; \Phi ; \mathrm{T} \rceil \doteq \{\, x_1 : t_1, x_2 : t_2, x : \lceil \phi[x_1/\mathbf{r_1}][x_2/\mathbf{r_2}] \rceil \mid x_1 \sim x_2 \rightsquigarrow x \in \mathrm{T}, \\ x_1 : t_1 \in \Gamma, x_2 : t_2 \in \Gamma, x_1 \sim x_2 : \phi \in \Phi \}$$

Namely, for each binding $x_1 \sim x_2 \rightsquigarrow x$ in the translation environment T, we add the bindings $x_1 : t_1, x_2 : t_2, x : \lceil \phi \rceil$ to the refinement type environment, where the types of the variables x_1 and x_2 are taken from the typing environment Γ and the relation between them from the relational environment Φ.

In the implementation, the refinement type environment is turned into arguments of the proof term e. For example, in the `filterRmap` example of § 2.3, the proof function `obligation` contains all the relevant arguments in the proof environment, while relational arguments with true refinements are omitted.

Translation Rules Figures 3 and 4 present the the translation rules, which are combined with the relational typing.

Rule T-Var looks up the translated expression from the context — that exactly translates a pair of variables to a unary variable — and generates the translated type by lifting the relational property to a unary type.

Rule T-Const produces a proof term $\mathrm{constTr}(t_1, t_2)$ that proves ϕ. By the definition of constPr, ϕ is equivalent to $\mathbf{r_1} = c_1 \wedge \mathbf{r_2} = c_2$ which trivially holds when $\mathbf{r_1}$ is in fact c_1 and $\mathbf{r_2}$ is c_2. However, to be compatible with the rest of the translation, we generate a proof via $\mathrm{constTr}(t_1, t_2)$:

$$\begin{aligned}
\mathrm{constTr}(b_1, b_2) &\doteq \texttt{()} \\
\mathrm{constTr}(t_{x_1} \to t_1, t_{x_2} \to t_2) &\doteq \lambda x_1 : t_{x_1}, x_2 : t_{x_2}, x : \texttt{trTy}(t_{x_1}, t_{x_2}).\mathrm{constTr}(t_1, t_2) \\
\texttt{trTy}(b_1, b_2) &\doteq \texttt{unit} \\
\texttt{trTy}(t_{x_1} \to t_1, t_{x_2} \to t_2) &\doteq t_{x_1} \to t_{x_2} \to \texttt{trTy}(t_{x_1}, t_{x_2}) \to \texttt{trTy}(t_1, t_2)
\end{aligned}$$

The function constTr is defined inductively. For constants of base types, it returns a trivial unit proof. Constants of function types are translated to lambda expressions with three arguments: the initial x_1 and x_2, and the relational argument x that carries the proof of a relational precondition about x_1 and x_2. The unary type u of the proof terms is constructed by `trTy` which is defined inductively and converts two base types into a unit. Two function types become a function type with three arguments: the initial x_1 and x_2, plus their relation.

Rule T-App translates function applications and is using a third argument e_t that relates the two arguments. Dually, rule T-Lam handles lambda-abstractions and it composes two lambda functions into a new lambda with three arguments. Rule T-Rec translates the recursive functions f_1 and f_2 to a new recursive function f and it adds the fresh symbol f to the relational context. Similarly, rules T-Let, T-If, and T-Case translate let-bindings, if-then-else, and case-expression respectively, by combining the two expressions into a single expression and generating the proof of the relation between them. Finally, rule T-Sub propagates the translated proof to the weaker property.

In short, the judgement $\Gamma \mid \Phi \mid \mathrm{T} \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \mid \phi \dashv e : u \mid \mathrm{O}$ turns the property ϕ into a unary refined type u, following "propositions as types" [**?**], and generates a proof term e that proves ϕ. The term e may be open because it can contain some obligation variables that should be guessed by an oracle.

$$\frac{}{\emptyset \models \emptyset}\text{O-Emp} \qquad \frac{\mathrm{O} \models \theta \qquad \exists e_o.\mathrm{R} \vdash e_o : u}{\mathrm{O}; \mathrm{R} \vdash o : u \models \theta, [e_o/o]}\text{O-Var}$$

$$\frac{\Gamma \,|\, \Phi \,|\, \mathrm{T}(\Phi) \vdash e_1 \sim e_2 \Leftarrow t_1 \sim t_2 \,|\, \phi + e{:}u \,|\, \mathrm{O} \qquad \mathrm{O} \models \theta \qquad \lceil \Gamma; \Phi \rceil \vdash \theta \cdot e : u}{\Gamma \,|\, \Phi \vdash_{\text{OBRA}} e_1 \sim e_2 : t_1 \sim t_2 \,|\, \phi}\text{T-OBRA}$$

Fig. 6: OBRA type checking.

3.4 Metatheory of OBRA

We combine the type checking rules of §3.2 with the translation of § 3.3 to design OBRA as an oracle-based, relational, algorithmic type checker that behaves as RHOL. Here, we describe OBRA and claim that it is equivalent to RHOL (theorem 1), the proofs are in the supplementary material.

Figure 6 presents the rule that designs the OBRA system. To check that the expressions $e_1 \sim e_2$ satisfy the relational refinement type $t_1 \sim t_2 \,|\, \phi$, OBRA has only one rule, T-OBRA, that performs three steps. First, it is using the bidirectional translation to generate the translated expression e, the unary type u, and the environment of proof obligations O. Next, it checks that the proof obligations O are satisfiable. The relation $\mathrm{O} \models \theta$ checks that the proof obligations O are satisfiable and returns a model, *i.e.* a substitution θ that maps each proof obligation variable to a witness expression: $\theta ::= \emptyset \mid \theta, [e/o]$. Satisfiability of O is defined by the two inductive rules O-Emp and O-Var that ensure, using unary refinement typing, that the model θ is indeed a valid witness. The final check ensures that the expression $\theta \cdot e$, *i.e.* the expression e under the substitution θ, indeed has the translated type u under the typing environment Γ refined with the relational predicates in Φ. In the absence of a translation environment, we turn the relational environment into a refined environment as follows:

$$\lceil \Gamma; \Phi \rceil \doteq \lceil \Gamma; \Phi; \mathrm{T}(\Phi) \rceil \qquad \mathrm{T}(\Phi) \doteq \{x_1 \sim x_2 \rightsquigarrow x \mid x_1 \sim x_2 : \phi \in \Phi, \text{fresh } x\}$$

Equivalence with RHOL Soundness of OBRA is shown by equivalence to RHOL, a relational type system that is sound and equivalent to HOL [**?**].

Let $\Gamma \mid \Psi \vdash_{\text{RHOL}} e_1 \sim e_2 : t_1 \sim t_2 \mid \phi$ be the RHOL judgment. The judgment is very similar to OBRA apart from that it is keeping track of an assertion environment Ψ instead of our relational environment Φ. We define the operation $\lfloor \cdot \rfloor$ that converts a relational environment to a set of assertions:

$$\lfloor \emptyset \rfloor \doteq \emptyset \qquad \lfloor \Phi; x_1 \sim x_2 : \phi \rfloor \doteq \lfloor \Phi \rfloor; \phi[x_1/\mathbf{r_1}][x_2/\mathbf{r_2}]$$

With this, we state soundness and completeness of OBRA *w.r.t.* RHOL.

Theorem 1. OBRA *is equivalent to* RHOL*:*

1. *If* $\Gamma \,|\, \Phi \vdash_{\text{OBRA}} e_1 \sim e_2 : t_1 \sim t_2 \,|\, \phi$*, then* $\Gamma \,|\, \lfloor \Phi \rfloor \vdash_{\text{RHOL}} e_1 \sim e_2 : t_1 \sim t_2 \,|\, \phi$.
2. *If* $\Gamma \,|\, \lfloor \Phi \rfloor \vdash_{\text{RHOL}} e_1 \sim e_2 : t_1 \sim t_2 \,|\, \phi$*, then* $\Gamma \,|\, \Phi \vdash_{\text{OBRA}} e_1 \sim e_2 : t_1 \sim t_2 \,|\, \phi$.

Table 1: Summary of Benchmarks. `Toy` are toy examples and `Tick` are the relational benchmarks of [**?**]. **Code** is the number of lines of executable code. In **Unary** columns, **Spec** is the number of lines of the refinement type specification and **Proof** is the number of lines of code to verify the specification. In **Relational** columns, **Spec** is the number of lines of the relational specification, **User** is the number of lines of code to verify the proof obligations, and **Auto** is the number of lines of auto-generated code.

	Benchmark	**Code**	**Unary**		**Relational**		
			Spec	**Proof**	**Spec**	**User**	**Auto**
Toy	`Increment`	3	2	1	3	0	17
	`Map`	15	10	9	6	0	36
	`Filter map`	6	3	16	5	16	29
	`Higher-order map`	12	8	16	6	16	29
Tick	`2D count`	22	2	21	9	0	113
	`Binary counters`	36	5	45	19	0	60
	`Boolean expressions`	35	1	27	4	21	257
	`Constant-time comparison`	13	5	1	4	0	112
	`Memory allocation`	22	1	1	3	0	50
	`Insertion Sort`	39	3	90	5	90	44
	`Merge sort`	30	6	73	5	52	186
	`Square and multiply`	14	8	9	9	0	240
	Total	247	54	309	78	195	1173

4 Evaluation

We implemented OBRA as an extension to Liquid Haskell. Concretely, we added two new features. First, the keyword `relational` allows the user to ascribe relational signatures to functions. OBRA will verify these signatures using the rules of § 3. The second feature is the flag `--relational-hints`, which tells Liquid Haskell to generate unary proof templates from relational signatures.

Benchmarks Table 1 evaluates our implementation using two sets of benchmarks. The first set, `Toy`, contains small examples like the `Map` and `Filter map` presented in § 2. `Increment` asserts the monotonicity of the increment function and `Higher-order map` tests function mapping with bounded differences. The second set, `Tick`, contains all the relational benchmarks from [**?**]. `2D count` compares two implementations of a 2D count function. `Binary counters` asserts that if two boolean lists are dual, then incrementing and decrementing them would result in the same cost and a dual value. `Boolean expressions` compares two implementations of a boolean expression evaluator. `Constant-time comparison` asserts that a cryptographic safe function that compares an input with a secret password, has the same cost for inputs of the same length. `Memory allocation` compares memory usage between lazy and forced evaluation. `Insertion sort` and `Merge sort` both assert that the difference of the cost of sorting two lists is bounded by the difference of the number of unsorted elements in the lists. Fi-

nally, `Square and multiply` asserts that the difference of the cost of two square and multiply runs is bounded by the difference of their list inputs.

Code Size Table 1 summarizes the number of lines of code (LoC) required for unary and relational verification. The **Code** column contains the number of lines of executable code, used both for the unary and relational verification. **Spec** is the number of lines required to express the main specification. In total, the relational specifications (78 LoC) are more verbose than the unary ones (54 LoC). This was expected, since in the relational form the types are separated from the properties. The column **Proof** contains the user provided LoC that prove the unary specifications. In the relational case, **Auto** contains the LoC automatically generated and **User** is the LoC the user provided to prove the generated obligations. Out of the 12 benchmarks, 7 were proved automatically, *i.e.* requiring 0 lines of user provided proof code. In all the benchmarks, the generated proofs are big, totalling 1173 LoC. This proof code is neither optimized nor is meant to be read by the user. Instead, when user code is required, the translation is generating top level proof obligations. In the 5 benchmarks that did require user proofs the user-provided proof size is smaller or equal to proofs in the unary case. In total the user provided proof code reduced from 309 to 195 LoC, *i.e.* a reduction of 36%. In short, the relational specifications are a little more verbose but the lines of user provided proof code is smaller.

Expressiveness Compared to relSTLC ([**?**]), which is the main implemented relational system, OBRA offers three key advantages. Firstly, it is more expressive. Unlike relSTLC, which only supports indices-based predicates making it impossible to express properties required by the `Boolean expressions` benchmark, OBRA supports arbitrary relational properties. Secondly, OBRA generates unary proofs and provides local error reporting, enabling easier debugging of relational proofs. Although relSTLC may automatically prove more relational proofs through heuristics, such as in the `merge sort` benchmark, it lacks debugging information when these heuristics fail. Finally, the generated proofs are checked by Liquid Haskell, increasing confidence in their validity.

5 Related Approaches

Our work is inspired by the rich literature on relational verification of higher-order programs. System R [**?**] is an early example of type-based relational verification for a polymorphic λ-calculus. However, its focus is limited to parametricity. Information flow typing [**?**,**?**] provides another case of type-based relational verification. However, it is limited to non-interference properties. Another example of a relational type system is Fuzz [**?**] and its dependently typed version Dfuzz [**?**]. Fuzz and Dfuzz use a rich system of linear types to reason about program sensitivity and differential privacy. In the case of DFuzz, types can be indexed over arithmetic expressions, making the type checking algorithm highly non-trivial. However, the type systems are syntax-directed (with the obvious exception of the subtyping rules, for which classic techniques can be used) and

as a consequence typing can be algorithmically reduced to constraint solving over natural numbers and the reals [**?**]. RelCost [**?**] makes a similar use of indexed types to reason about relational cost. In contrast to sensitivity, which can be proved for many examples without the use of non-synchronous rules, relative cost requires asynchronous rules, which are therefore an integral part of RelCost. This makes algorithmic typing for RelCost challenging. Nevertheless, [**?**] develops mechanisms to perform algorithmic typing, at the cost of losing completeness *w.r.t.* the declarative type system. Our approach avoids losing completeness, by generating proof obligations that need to be discharged in a unary type system.

The first general purpose relational logic for higher-order (stateful) programs is Relational Hoare Type Theory (RHTT) [**?**]. RHTT is implemented within the Coq proof assistant, giving its users great flexibility to interleave synchronous and asynchronous reasoning. To our best knowledge, the problem of algorithmic verification for RHTT has not been studied. Another example of general-purpose relational type system is RF$*$ [**?**]. RF$*$ is built on top of F$*$ [**?**], and follows similar principles. Type-checking generates proof obligations that are delegated to SMT-solvers. One limitation of RF$*$ is that it is primarily restricted to synchronous reasoning. In contrast, our approach allows combining synchronous and asynchronous reasoning via the generation of proof obligations.

Our work is most closely related to Relational Higher-Order Logic (RHOL) [**?**]. RHOL differs from previously mentioned approaches by imposing a stratification between computations, that are expressed in a type system, and properties, that are expressed in a higher-order logic over the type system. The stratification is key for achieving soundness and completeness *w.r.t.* standard HOL. Our work allows us to combine expressiveness of RHOL with algorithmic verification that comes for free with other approaches. There are several extensions of RHOL, in particular with probabilities [**?**] and monadic cost R^{C} [**?**]. Like RHOL, these systems are not supported by algorithmic verification.

An alternative approach is to carry out relational reasoning using an extrinsic proof approach. In this approach, relational properties are captured as non-relational properties, typically over unit types, and the proofs are constructed manually relying on a proposition-as-types encoding customized to the language at hand. This is the approach used in [**?**] for F$\star$ and in [**?**] for Liquid Haskell. The latter work was later extended to probabilistic computations in [**?**]. Our work builds upon [**?**] by providing additional automation while similarly using SMT-aided refinement types to reason about quantitative specifications.

6 Conclusion

OBRA is an oracle-based, relational, algorithmic type checker that proves relational properties of higher-order, functional programs. We formalized OBRA as a core calculus, λ_{OBRA}, and proved it equivalent to RHOL. Further, the system automatically translates relational properties to unary theorems with a proof template. We implemented OBRA as an extension of Liquid Haskell and evaluated it on 12 benchmarks and out of which 7 were proved automatically.

Acknowledgements. This work is founded by the Horizon Europe ERC Starting Grant CRETE (GA: 101039196), and the DECO grant PID2022-138072OB-I00, funded by MCIN/AEI/10.13039/501100011033/ FEDER, UE.

References

1. Abadi, M., Cardelli, L., Curien, P.: Formal Parametric Polymorphism. In: POPL (1993). https://doi.org/10.1145/158511.158622
2. Aguirre, A., Barthe, G., Birkedal, L., Bizjak, A., Gaboardi, M., Garg, D.: Relational Reasoning for Markov Chains in a Probabilistic Guarded Lambda Calculus. In: ESOP (2018), https://doi.org/10.1007/978-3-319-89884-1_8
3. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Katsumata, S., Sato, T.: Higher-Order Probabilistic Adversarial Computations: Categorical Semantics and Program Logics. In: ICFP (2021), https://doi.org/10.1145/3473598
4. Aguirre, A., Barthe, G., Gaboardi, M., Garg, D., Strub, P.: A Relational Logic for Higher-Order Programs. In: ICFP (2017), https://doi.org/10.1145/3110265
5. de Amorim, A.A., Arias, E.J.G., Gaboardi, M., Hsu, J.: Really Natural Linear Indexed Type Checking. In: Implementation and Application of Functional Languages (2014), https://dl.acm.org/doi/10.1145/2746325.2746335
6. Barthe, G., Fournet, C., Grégoire, B., Strub, P., Swamy, N., Béguelin, S.Z.: Probabilistic Relational Verification for Cryptographic Implementations. In: POPL (2014), https://doi.org/10.1145/2535838.2535847
7. Barthe, G., Gaboardi, M., Arias, E.J.G., Hsu, J., Roth, A., Strub, P.: Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In: POPL (2015), https://doi.org/10.1145/2676726.2677000
8. Barthe, G., Grégoire, B., Hsu, J., Strub, P.Y.: Coupling Proofs Are Probabilistic Product Programs. In: POPL (2017), https://doi.org/10.1145/3093333.3009896
9. Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., Hoffmann, J.: Relational Cost Analysis. In: POPL (2017). https://doi.org/10.1145/3093333.3009858, https://doi.org/10.1145/3093333.3009858
10. Çiçek, E., Qu, W., Barthe, G., Gaboardi, M., Garg, D.: Bidirectional Type Checking for Relational Properties. In: PLDI (2019), https://doi.org/10.1145/3314221.3314603
11. Çiçek, E., Qu, W., Barthe, G., Gaboardi, M., Garg, D.: Bidirectional Type Checking for Relational Properties. In: PLDI (2019), https://doi.org/10.1145/3314221.3314603
12. Dunfield, J., Krishnaswami, N.: Bidirectional Typing. In: ACM Comput. Surv. (2021), https://doi.org/10.1145/3450952
13. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear Dependent Types for Differential Privacy. In: POPL (2013), https://doi.org/10.1145/2429069.2429113
14. Grimm, N., Maillard, K., Fournet, C., Hritcu, C., Maffei, M., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N., Béguelin, S.Z.: A Monadic Framework for Relational Verification: Applied to Information Security, Program Equivalence, and Optimizations. In: CPP (2018), https://doi.org/10.1145/3167090
15. Handley, M.A.T., Vazou, N., Hutton, G.: Liquidate Your Assets: Reasoning about Resource Usage in Liquid Haskell. In: POPL (2019), https://doi.org/10.1145/3371092

16. Heintze, N., Riecke, J.G.: The SLam Calculus: Programming with Secrecy and Integrity. In: POPL (1998), http://doi.acm.org/10.1145/268946.268976
17. Jhala, R., Vazou, N.: Refinement types: A tutorial. In: Foundations and Trends in Programming Languages (2020), http://dx.doi.org/10.1561/2500000032
18. Nanevski, A., Banerjee, A., Garg, D.: Verification of Information Flow and Access Control Policies with Dependent Types. In: S&P (2011), https://doi.org/10.1109/SP.2011.12
19. Nanevski, A., Banerjee, A., Garg, D.: Dependent Type Theory for Verification of Information Flow and Access Control Policies. In: TOPLAS (2013), http://doi.acm.org/10.1145/2491522.2491523
20. Pottier, F., Simonet, V.: Information Flow Inference for ML. In: POPL (2002), http://doi.acm.org/10.1145/503272.503302
21. Radicek, I., Barthe, G., Gaboardi, M., Garg, D., Zuleger, F.: Monadic Refinements for Relational Cost Analysis. In: POPL (2018), http://doi.acm.org/10.1145/3158124
22. Reed, J., Pierce, B.C.: Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In: ICFP (2010), https://doi.org/10.1145/1863543.1863568
23. Swamy, N., Hritcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P., Kohlweiss, M., Zinzindohoue, J.K., Béguelin, S.Z.: Dependent Types and Multi-monadic Effects in F. In: POPL (2016), http://doi.acm.org/10.1145/2837614.2837655
24. Vasilenko, E., Vazou, N., Barthe, G.: Safe Couplings: Coupled Refinement Types. In: ICFP (2022), https://doi.org/10.1145/3547643
25. Vazou, N., Breitner, J., Kunkel, R., Van Horn, D., Hutton, G.: Theorem Proving for All: Equational Reasoning in Liquid Haskell (Functional Pearl). In: Haskell (2018), https://doi.org/10.1145/3242744.3242756
26. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement Types for Haskell. In: ICFP (2014), https://doi.org/10.1145/2692915.2628161
27. Vazou, N., Tondwalkar, A., Choudhury, V., Scott, R.G., Newton, R.R., Wadler, P., Jhala, R.: Refinement Reflection: Complete Verification with SMT. In: POPL (2017), https://doi.org/10.1145/3158141
28. Wadler, P.: Propositions as Types. In: Commun. ACM (2015). https://doi.org/10.1145/2699407, https://doi.org/10.1145/2699407