Relational Nullable Types with Boolean Unification

MAGNUS MADSEN and JACO VAN DE POL, Aarhus University, Denmark

We present a simple, practical, and expressive relational nullable type system. A relational nullable type system captures whether an expression may evaluate to null based on its type, but also based on the type of other related expressions. The type system extends the Hindley-Milner type system with Boolean constraints, supports parametric polymorphism, and preserves principal types modulo Boolean equivalence. We show how to support full Hindley-Milner style type inference with an extension of Algorithm W.

We conduct a preliminary study of open source projects showing that there is a need for relational nullable type systems across a wide range of programming languages. The most important findings from the study are: (i) programmers use programming patterns where the nullability of one expression depends on the nullability of other related expressions, (ii) such invariants are commonly enforced with run-time exceptions, and (iii) reasoning about these programming patterns requires not only knowledge of when an expression may evaluate to null, but also when it may evaluate to a non-null value. We incorporate these observations in the design of the proposed relational nullable type system.

CCS Concepts: • **Theory of computation → Program semantics**.

Additional Key Words and Phrases: relational nullable type system, relational pattern matching, choose construct, type inference, Algorithm W, Boolean unification, successive variable elimination algorithm

ACM Reference Format:
Magnus Madsen and Jaco van de Pol. 2021. Relational Nullable Types with Boolean Unification. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 110 (October 2021), 28 pages. https://doi.org/10.1145/3485487

1 INTRODUCTION

Null — infamously dubbed the "Billion Dollar Mistake" by its inventor Sir Tony Hoare — is a special value that is an inhabitant of every type. The null value serves two distinct purposes: to represent the absence of a value or as a place holder for an uninitialized value. Null is dangerous because any attempt to use it as an ordinary value, e.g. by dereference or invocation, triggers the dreaded `NullPointerException` crashing the program or, even worse, leading to undefined behavior, depending on the semantics of the programming language.

A *nullable* type system prevents such crashes or undefined behavior by tracking, for every expression, whether it may evaluate to null [Banerjee et al. 2019; Fähndrich and Xia 2007; Nieto et al. 2020b; Summers and Müller 2011]. If an expression is *nullable* then the programmer must explicitly check for null before the result of the expression can be used. If, on the other hand, an expression is *non-nullable* then the type system guarantees that the expression cannot evaluate to null at run-time. Thus the programmer does not have to check for null.

Recently, nullable type systems have become popular in mainstream programming languages. New programming languages such as Kotlin, Swift, and TypeScript have adopted nullable type systems from the start, whereas established languages such as C# and Scala are being retrofitted with nullable type systems [Fähndrich and Leino 2003; Nieto et al. 2020b].

Authors' address: Magnus Madsen, magnusm@cs.au.dk; Jaco van de Pol, jaco@cs.au.dk, Department of Computer Science, Aarhus University, Åbogade 34, Aarhus, 8210, Denmark.

2475-1421/2021/10-ART110
https://doi.org/10.1145/3485487

In this paper we propose a novel *relational nullable type system* that captures not only nullability of a single expression, but its nullability with respect to other related expressions. The type system can enforce invariants such as: "the expressions e_1 and e_2 cannot both be null", "the expressions e_1, e_2, and e_3 cannot all be non-null", or: "among the expressions e_1, e_2, and e_3 exactly one must be null". For a more concrete example, imagine a method that takes two arguments: file and encoding where *either* both arguments must be null *or* both arguments must be non-null.

We conduct a preliminary study of open source projects showing that there is a need for relational nullable type systems across a wide range of programming languages. The most important findings are that programmers use programming patterns where the nullability of one expression depends on the nullability of other related expressions. Such programming patterns occur in imperative, object-oriented, and functional languages, and without type system support, programmers currently rely on runtime checks to enforce these relational nullability invariants.

To the best of our knowledge, contemporary nullable type systems, in the research literature and in mainstream programming languages, only consider the nullability of a single expression. That is, while several existing systems take into account that the nullability of an expression may vary over time (e.g. during object construction), no existing system considers that the nullability of one expression may depend on the nullability of other related expressions.

We propose a simple, practical, and expressive relational nullable type system to ensure the safety of such programming patterns. The type system captures both *may* and *must* information about nullity and non-nullity, i.e. whether an expression *may* or *must* evaluate to null and whether it *may* or *must* evaluate to a non-null value. The type system enables a special form of pattern matching on a sequence of expressions and their nullity. Unlike in a traditional pattern match, the cases of such a match do not have to be exhaustive. Instead, the type system prevents matching on expressions whose values could lead to an unmatched case at runtime. This is really the simplest solution for programming with null: simply leave out cases that should not occur and the type system ensures that they cannot occur.

An important property of the type system is that it supports full type inference. The key insight is that the exhaustiveness condition can be translated into a Boolean formula which can be inferred by an extension of Algorithm W because Boolean formulas have most general unifiers [Boole 1847; Madsen and van de Pol 2020; Martin and Nipkow 1989].

While our work is presented in the context of null, it is equally applicable to relational uses of the Option data type and it gives precise types to combinators such as filter, map, and flatMap. The type system extends the Hindley-Milner type system, supports parametric polymorphism and preserves principal types modulo Boolean equivalence.

In summary, the contributions of this paper are:

- (**Preliminary Study**) We conduct a preliminary study on the use of relational nullability programming patterns in open source projects. We find that such patterns occur across a wide range of programming languages (cf. Section 2).
- (**Type System**) We present a minimal calculus with null and a special pattern matching construct named choose. We present a declarative type system for the calculus based on Hindley-Milner extended with Boolean constraints and we show how to perform principal type inference using an extension of Algorithm W (cf. Section 3 and 4).
- (**Implementation**) We provide two implementations of the calculus and type system: a proof-of-concept that exactly mirrors the formal system and an extension of the Flix programming language with relational nullable types (cf. Section 5).

In Section 6, we compare our contributions to related work on nullable type systems.

2 PRELIMINARY STUDY

We now present a preliminary study on the use of relational nullability programming patterns in real-world open source projects. The primary purpose of the study is twofold: to ascertain whether: (i) there is a use case for relational nullable type systems, and (ii) to understand what features such a system should support. That is, we want to understand if there is a problem, but not necessarily the scope of the problem. We begin with a discussion of our methodology, next we present several representative examples of the programming patterns we found, and finally we present a summary of our most important quantitative findings.

2.1 Methodology

We used the www.grep.app website to search for relational nullability programming patterns in open source projects hosted on GitHub. We searched for strings that are indicative of relational nullable programming patterns. For example, the strings "cannot both be null" and "cannot both be non-null" which are likely to occur in comments or error messages. We also searched for many spelling variants of such strings, including "can't both be null' and "canot both be null" (sic). We also substitute "null' for "NULL", "nil", "None", "Some", and so forth. In addition to these natural language queries, we also searched for specific program fragments. For example, `(None, None) =>` (a fragment of a pattern match on two None values). We collected the results from the queries and manually inspected the results. We included a program fragment based on these selection criteria:

- the program fragment was a relational nullability programming pattern.
- the program fragment was readable (i.e. we excluded auto-generated or bizarre code.)
- the program fragment had an invariant within the reach of static type systems.[1]
- the program fragment was a not a duplicate (i.e. due to duplicated code on GitHub.)
- we limited the study to one program fragment per GitHub repository.
- we limited the study to programming languages with static type systems.

In total we collected 108 manually inspected program fragments that satisfied the above criteria. The program fragments are available in the supplementary material. The preliminary study is only the tip of the iceberg: The www.grep.app website only covers an estimated 0,5% of all public GitHub repositories and our simple textual queries are unlikely to discover all relevant program fragments. Yet, the study is still sufficient to demonstrate that there is a use case for relational nullable type systems and to help inform the design of such type systems.

2.2 Selected Examples

We now discuss a few representative examples of the collected program fragments. Next, we present the quantitative results of the study.

pmd/pmd (Java) `pmd-core/src/main/java/net/sourceforge/pmd/util/database/DBType.java`

```
1  public DBType(String subProtocol, String subnamePrefix) throws IOException {
2      /* omitted */
3      if (null == subProtocol && null == subnamePrefix) {
4          throw new RuntimeException("subProtocol and subnamePrefix cannot both be null");
5      }
6      /* omitted */
7  }
```

[1]For example, a program fragment that performs an RPC call over the network and expects a record with two null or non-null fields is out of scope. This assumption is in line with most work on nullable type systems.

The above code fragment, from the pmd/pmd project, shows part of the DBType constructor. The constructor takes two arguments: subProtocol and subnamePrefix. If both are null the constructor throws a RuntimeException to abort execution.

CloudburstMC/Nukkit (Java) src/main/java/cn/nukkit/OfflinePlayer.java

```java
public OfflinePlayer(Server server, UUID uuid, String name) {
    /* omitted */
    if (uuid != null) {
        nbt = this.server.getOfflinePlayerData(uuid, false);
    } else if (name != null) {
        nbt = this.server.getOfflinePlayerData(name, false);
    } else {
        throw new IllegalArgumentException("Name and UUID cannot both be null");
    }
    /* omitted */
}
```

The above code fragment, from the CloudburstMC/Nukkit project, shows part of the OfflinePlayer constructor. The constructor takes three arguments: server, uuid, and name. If both uuid and name are null the constructor throws an IllegalArgumentException to abort execution.

apache/druid (Java) sql/src/main/java/org/apache/druid/sql/calcite/rel/Projection.java

```java
Projection(@Nullable final List<PostAggregator> postAggregators,
           @Nullable final List<VirtualColumn> virtualColumns,
           final RowSignature outputRowSignature) {
    if (postAggregators == null && virtualColumns == null) {
        throw new IAE("postAggregators and virtualColumns cannot both be null");
    } else if (postAggregators != null && virtualColumns != null) {
        throw new IAE("postAggregators and virtualColumns cannot both be nonnull");
    }
    /* omitted */
}
```

The above code fragment, from the apache/druid project, shows the Projection constructor which takes three arguments: postAggregators, virtualColumns, and outputRowSignature. If both postAggregators and virtualColumns are null then the constructor throws an IAE exception. Similarly, if both postAggregators and virtualColumns are non-null then the constructor throws an IAE exception. In other words, exactly one of the two arguments must be null.

openssl/openssl (C) crypto/ffc/ffc_params_generate.c

```c
int ossl_ffc_params_FIPS186_4_gen_verify(/*...*/, FFC_PARAMS *params, /*...*/) {
    /* omitted */
    /* For generation: p & q must both be NULL or NON-NULL */
    if ((params->p == NULL) != (params->q == NULL)) {
        *res = FFC_CHECK_INVALID_PQ;
        goto err;
    }
    /* omitted */
}
```

The above code fragment, from the openssl/openssl project, shows part of the ossl_ffc_params_FIPS186_4_gen_verify procedure. The procedure takes several parameters of which the params argument must be a struct with two fields: p and q. The above fragment asserts that either: (i) both p and q are NULL or (ii) both p and q are non-NULL.

torvalds/linux (C) `include/drm/drm_atomic_helper.h`

```c
drm_atomic_plane_disabling(struct drm_plane_state *old_plane_state,
                           struct drm_plane_state *new_plane_state) {
    /* omitted */
    /*
     * When disabling a plane, CRTC and FB should always be NULL together.
     * Anything else should be considered a bug in the atomic core, so we
     * gently warn about it.
     */
    WARN_ON((new_plane_state->crtc == NULL && new_plane_state->fb != NULL) ||
            (new_plane_state->crtc != NULL && new_plane_state->fb == NULL));
    return old_plane_state->crtc && !new_plane_state->crtc;
}
```

The above code fragment, from the `torvalds/linux` project, shows part of the `drm_atomic_plane_disabling` procedure. The procedure has a local variable new_plane_state which is a struct with two fields: crtc and fb. As the comment explains, if either field is `NULL` the other field must also be `NULL`. It is a bug if one field is `NULL` and the other is non-`NULL`.

rust-analyzer/rust-analyzer (Rust) `crates/ide/src/completion/complete_record.rs`

```rust
/* omitted */
match (ctx.record_pat_syntax.as_ref(), ctx.record_lit_syntax.as_ref()) {
    (None, None)          => return None,
    (Some(_), Some(_))    => unreachable!("A record cannot be both ..."),
    (Some(record_pat), _) => ctx.sema.record_pattern_missing_fields(record_pat),
    (_, Some(record_lit)) => ctx.sema.record_literal_missing_fields(record_lit),
};
```

The above code fragment, from the `rust-analyzer/rust-analyzer` project, shows a program fragment that does a pattern match on the two fields: ctx.record_pat_syntax and ctx.record_lit_syntax. If both fields are `None` the function returns `None`. If one or the other is `Some` then the pattern match generates an error message. If *both* values are `Some` then the function triggers a "panic" using the `unreachable!` macro which aborts execution. As the Rust documentation explains[2], a panic is serious and should only be used when a program reaches an unrecoverable state.

snowplow/iglu (Scala) `0-common/igluctl/src/main/scala/.../ctl/commands/S3cp.scala`

```scala
/* omitted */
val credentialsProvider = (accessKeyId, secretAccessKey, profile) match {
    case (Some(keyId), Some(secret), None) =>
        /*...*/ new BasicAWSCredentials(keyId, secret)
    case (None, None, Some(p))             =>
        /*...*/ new ProfileCredentialsProvider(p)
    case (None, None, None)                =>
        /*...*/ DefaultAWSCredentialsProviderChain.getInstance()
    case _                                 =>
        /*...*/ Error.ConfigParseError("Invalid AWS authentication method."))
}
```

The above code fragment, from the `snowplow/iglu` project, shows a pattern match on three values: accessKeyId, secretAccessKey, and profile. The pattern match returns an error value if either: (i) accessKeyId is `Some` and secretAccessKey is `None` (or vise versa), or if (ii) accessKeyId, secretAccessKey, and profile are all `Some`.

[2]https://doc.rust-lang.org/std/macro.panic.html

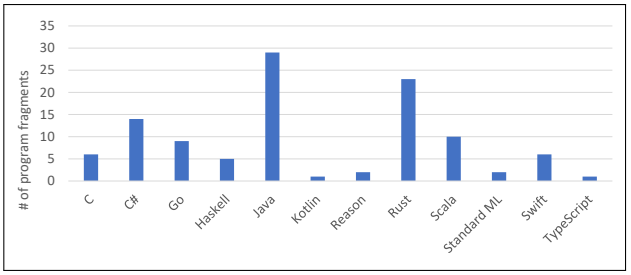

Fig. 1. Programming languages.

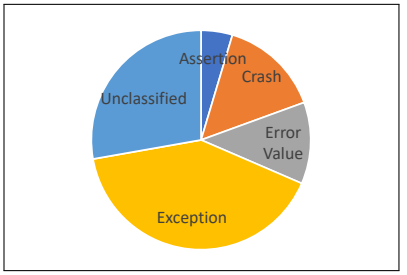

Fig. 2. Enforcement mechanism.

combust/mleap (Scala) `mleap-core/src/main/scala/ml/combust/.../feature/StandardScalerModel.scala`

```scala
case class StandardScalerModel(std: Option[Vector], mean: Option[Vector]) {
    /* omitted */
    val size = (std, mean) match {
        case (None, None)              => throw new IllegalStateException("...")
        case (Some(stdV), None)        => stdV.size
        case (None, Some(meanV))       => meanV.size
        case (Some(stdV), Some(meanV)) => stdV.size
    }
    /* omitted */
}
```

The above code fragment, from the `combust/mleap` project, shows the (inline) constructor for the StandardScalerModel class. The constructor takes two arguments: std and mean. If both are None then the constructor throws an `IllegalStateException` to abort execution.

2.3 Preliminary Results

We now present the quantitative results of the study. In total we found 108 program fragments that satisfied the selection criteria. The fragments are available in the supplementary material.

2.3.1 Programming Languages. Figure 1 shows a breakdown of the programming languages that each program fragment is written in. The figure shows that relational nullability programming patterns arise across a wide range of programming languages, including in languages that have `null` (e.g. C, C#, Go, Java), languages that have the `Option` / `Maybe` data type (e.g. Haskell, Rust), and languages that have both `null` and `Option` (e.g. Scala). This is not too surprising: Inspection of the program fragments reveals that relational nullability is a property of the problem domain.

2.3.2 Enforcement Mechanism. Figure 2 shows a breakdown of the enforcement mechanisms used in the program fragments. Given that these programming languages do not have relational nullable type systems, some other mechanism must be used to enforce such invariants. We grouped these enforcement mechanisms into five categories: Assertion, Crash, Error Value, Exception, and Unclassified. Most of the categories are self-explanatory. The Crash category covers enforcement mechanisms more severe than exceptions, such as panic! in Rust and fatalError in Swift. The Error Value covers enforcement mechanisms where the programmer returned a "monadic" error value such as Err of the Result data type. We used the Unclassified category for everything else or when we were in doubt. The figure shows that most common enforcement mechanism is to throw an exception. Together, Assertion (4%), Crash (15%), and Exception (41%) account for 60% of the enforcement mechanisms. Thus, in lieu of type system support, violations of relation nullability invariants cause run-time errors.

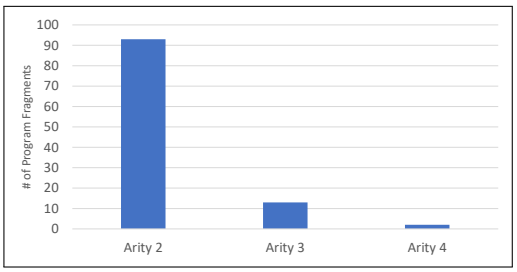

Fig. 3. Number of related expressions.

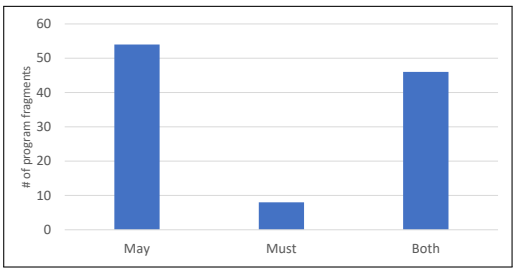

Fig. 4. Use of may and must information.

2.3.3 Related Expressions. Figure 3 shows a breakdown of the number of expressions whose nullability is inter-dependent. The figure shows that it is common to have *two* expressions whose nullity depend on each other. However, there are also cases where the nullity of *three* expressions depend on each other. We can imagine that as the number of related expressions grow the invariants may become too complex for the programmer to understand and use. However, with better type system support we believe that programmers might start to encode more complex domain invariants.

This a common theme in programming language design. The introduction of a new feature that makes it simpler and safer to use certain programming patterns may encourage the use of such patterns and lead to a new way of writing programs.

2.3.4 May and Must Nullity. Figure 4 shows a breakdown of how many of the program fragments require may and must information about the nullability or non-nullability of related expressions. A standard nullable type system can guarantee that an expression *cannot* evaluate to null, but it cannot enforce that an expression *must* evaluate to null. Thus it cannot capture a situation where if e_1 evaluates to null then e_2 must *also* evaluate to null. The figure shows that *may* knowledge is much more common than *must* knowledge. However, for approximately half of the program fragments, *both* may and must knowledge is required to capture the relational nullability invariants. This strongly suggests that a relational nullability type system must track two pieces of information: the conditions under which an expression evaluates to null and the conditions under which it evaluates to a non-null value.

2.4 Summary of Findings

We conclude with a summary of our most important observations:

- (Obs I): Programmers use programming patterns where the nullability of one expression depends on the nullability of other related expressions.
- (Obs II): Programmers use such programming patterns across a wide range of languages, including imperative, object-oriented, and functional languages.
- (Obs III): Reasoning about such patterns requires both *may* and *must* information.
- (Obs IV): Relational nullability typically depends on two or three expressions.
- (Obs V): In lieu of type system support, programmers currently rely on run-time checks to enforce relational nullability invariants.

Interestingly, in the initial design of our relational nullable type system, we only considered when an expression *may* evaluate to null, but the preliminary study demonstrated that it is equally important to also capture when an expression *must* evaluate to null. Without the study, we would have ended up with a less useful design.

3 MOTIVATION

We now give an informal introduction to the proposed calculus and type system. We work in an ML-style programming language à la Standard ML or OCaml. We begin by introducing a special constant null and a new pattern matching construct called choose. The choose construct allows us to match on the *nullity* of a sequence of expressions. As we shall see, the choose construct mimics many of the programming patterns shown in Section 2. The choose construct differs from a normal pattern match in three important ways: (i) patterns can only match on nullity (i.e. null or non-null), (ii) patterns cannot be nested, and (iii) patterns do not have to be exhaustive. The latter point is worth emphasizing: In our system, *a pattern match does not have to be exhaustive; instead the type system restricts the types of the match expressions to the defined patterns.*

3.1 Motivating Examples

Example I. The following program illustrates that we can pattern match on whether the value of a single expression is null or non-null.

```
1  let f = x -> choose x {
2      case null  => println("x is null!")
3      case w     => println("x is non-null!")
4  };
5  f(null);    // OK
6  f(1234)     // OK
```

The program binds f to a lambda expression with argument x. The lambda expression uses the choose construct to pattern match on the nullity of x. The match has two cases: one for when x is null and one for when *x* is non-null. Note that a variable pattern w *only* matches a non-null value! The program is well-typed and prints "x is null!" followed by "x is non-null!" when executed. The type of the function f is automatically inferred and in this case imposes no requirement on the nullity of its argument x. Thus both calls to f type check.

Example II. The program below is a variant of the previous program with a single pattern for when x is non-null:

```
1  let f = x -> choose x {
2      case w => println("x is non-null!")
3  };
4  f(null);    // NOT OK - type error
5  f(1234)     // OK
```

In this case, the inferred type of the function f ensures that it *cannot* be called with a null value. Thus the first call to f no longer type checks. (Recall that a variable pattern *only* matches non-values.)

Example III. Conversely, the program below contains a single pattern for when x is null:

```
1  let f = x -> choose x {
2      case null => println("x is null!")
3  };
5  f(null);    // OK
6  f(123)      // NOT OK - type error
```

Unlike most nullability systems, our system can enforce that a variable *must* be null. For a pattern match on a single expression this is may seem useless, but it becomes useful when matching on multiple expressions.

Example IV. The previous examples all match on a *single* expression. The program below matches on two variables x and y and handles the case where either (i) the values of both variables are null, or (ii) the values of both variables are non-null.

```
1  let f = (x, y) -> choose (x, y) {
2      case (null, null)   => println("both null!")
3      case (u, v)         => println("both non-null!")
4  };
5  f(null, null);  // OK
6  f(1234, 5678);  // OK
7  f(1234, null);  // NOT OK
8  f(null, 5678)   // NOT OK
```

The above program is well-typed and prints `"x and y are both null!"` followed by `"x and y are both non-null!"` when executed. The inferred type of the function `f` ensures that it is a type error to call it with a null and a non-null value (or vise versa).

Example V. If we modify the above program to:

```
let f = (x, y) -> choose (x, y) {
    case (null, v) => println("x is null and y is non-null.")
    case (u, null) => println("x is non-null and y is null.")
};
f(null, 1234);  // OK
f(null, null)   // NOT OK - type error
```

then the inferred type of the function `f` ensures that it can only be invoked with two values: exactly one of which must be null and the other must be non-null.

As these examples demonstrate, an important property of our system is that:

> Given a pattern match on the expressions $e_1, \cdots, e_n$, the choose construct permits matching on any subset of cases. If a case is omitted the type system enforces that the types of $e_1, \cdots, e_n$ cannot give rise to that case at run-time.

A corollary is that an exhaustive pattern match does not impose any requirements on the nullity of $e_1, \cdots, e_n$. Conversely, an *empty* pattern match enforces that $e_1, \cdots, e_n$ must be uninhabited.

3.2 Type System

We now informally describe how the type system works. The typing judgement of an expression is of the form $\Gamma \vdash e : \pi \ ? \ (\varphi, \psi)$, where π is a proper type (e.g. String), and φ and ψ are Boolean formulas, called the *nullity formulas* of the type. The Boolean formula φ captures when the expression *may* evaluate to null, while ψ captures when the expression *may* evaluate to a non-null value.

Consequently, the values null and non-null have the following types (for any φ and ψ):

$$\mathsf{null} : \alpha \ ? \ (\mathsf{T}, \psi) \qquad v : \pi \ ? \ (\varphi, \mathsf{T})$$

If we have an expression with a type of the form $\pi \ ? \ (\mathsf{F}, \psi)$ (for any π, ψ) then the expression *cannot* evaluate to null. Similarly, if we have an expression with a type of the form $\pi \ ? \ (\varphi, \mathsf{F})$ (for any π, φ) then the expression cannot evaluate to a non-null value. If we have an expression with a type of the form $\pi \ ? \ (\mathsf{T}, \mathsf{T})$ then we have no information; the expression could evaluate to null or to a non-null value. For example, the expression "if c then null else 123" has the type $\mathsf{Int} \ ? \ (\mathsf{T}, \mathsf{T})$ because the type system combines the information from the two branches. Finally, if an expression has a type of the form $\pi \ ? \ (\mathsf{F}, \mathsf{F})$ then it cannot evaluate to any value at all; the type is uninhabited.

Let us now consider what the type of the function on the left should be:

```
let f = (x, y) -> choose (x, y) {
    case (null, null) => 21
    case (null, v)    => 42
    case (u, v)       => 84
};
```

$$P = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} 0 & \boxdot \\ \boxdot & 1 \end{pmatrix} = \mathsf{Saturate}(P)$$

For simplicity, let us assume that the types of `x` and `y` are of the form:

$$x : \pi_1 \ ? \ (X_0, X_1) \qquad y : \pi_2 \ ? \ (Y_0, Y_1)$$

where π_1 and π_2 are proper types, and the Boolean variables (X_0, X_1) and (Y_0, Y_1) indicate the nullity formulas of `x` and `y`.

We want to translate the pattern match into a constraint on X_0, X_1, Y_0, Y_1 that captures when the pattern match is defined. The matrix P, shown above, represents the value vectors matched

by the pattern match, where 0 represents null, 1 represents a non-null value, and ⊡ represents a wildcard. Note that an expression like (null, if c then null else 123) corresponds to $(0, ⊡)$, which is not covered by a single row of P, although its concrete instances $(0, 0)$ and $(0, 1)$ are covered. To remedy this, a saturation procedure (cf. Section 4.6), combines any two rows that differ in a single entry. The result matrix Saturate(P) covers all value vectors where either x is definitely null (i.e. $\neg X_1$), or y is definitely non-null (i.e. $\neg Y_0$). Thus the exhaustiveness constraint is given by the side condition $\neg X_1 \vee \neg Y_0$.

How do we handle the side condition? At this point, we call a Boolean unification procedure on the unification problem $\neg X_1 \vee \neg Y_0 \stackrel{?}{=} \mathsf{T}$. This yields a most general unifier, for instance $[Y_0 \mapsto Y_0 \wedge \neg X_1]$. We can now give the following type to the function f:

$$f : \pi_1 \ ?\ (X_0, X_1) \to \pi_2 \ ?\ (Y_0 \wedge \neg X_1, Y_1) \to \mathsf{Int} \ ?\ (\mathsf{F}, \mathsf{T})$$

To understand why this is correct, let us consider what happens if we try to call f where the *first* argument is non-null. In this case, we know that $X_1 = \mathsf{T}$ and thus the type of the second argument simplifies to $\pi_2 \ ?\ (\mathsf{F}, Y_1)$. But now the constraint on the *second* argument ensures that it cannot be null! Indeed, the pattern match has no case for a (non-null, null) pair.

4 CALCULUS

We now present $\lambda^{\mathsf{rel}}_{\mathsf{null}}$: a lambda calculus with a relational nullable type system. We then present an extension of Algorithm W that enables Hindley-Milner style type inference. We begin with a brief introduction to Boolean unification which is the key technique that enables type inference.

4.1 Preliminaries: Boolean Unification

Boolean Formulas. Let $\mathcal{B} = \{\mathsf{T}, \mathsf{F}\}$ denote the Boolean values true and false. Let *BoolVar* denote a set of Boolean variables $(\beta \cdots)$. The set of Boolean formulas is then defined inductively as:

$$\varphi, \psi \in \mathit{Formula} = \beta \mid \mathsf{T} \mid \mathsf{F} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

A *valuation* V is a mapping from Boolean variables to $\{\mathsf{T}, \mathsf{F}\}$. Valuations are extended to evaluate formulas, by means of the standard truth tables. We write $\varphi \equiv_{\mathbb{B}} \psi$ if $V(\varphi) = V(\psi)$ for all valuations V (i.e. φ and ψ are logically equivalent). For example, we have:

$$\mathsf{T} \equiv_{\mathbb{B}} \neg\mathsf{F} \qquad x \equiv_{\mathbb{B}} \neg(\neg x) \qquad x \wedge y \equiv_{\mathbb{B}} y \wedge x \qquad x \wedge \neg x \equiv_{\mathbb{B}} \mathsf{F} \wedge z$$

Note that two formulas do not have to share the same variables to be equivalent. We write $\varphi \Rightarrow \psi$ if φ logically implies ψ. That is, for all valuations V, if $V(\varphi) = \mathsf{T}$ then $V(\psi) = \mathsf{T}$.

Substitutions. A Boolean *substitution* S is a partial mapping *BoolVar* $\hookrightarrow$ *Formula*. We write $\{\beta_i \mapsto \varphi_i\}_{i \in I}$ for the substitution S, such that $S(\beta_i) = \varphi_i$. We extend substitutions to total functions on formulas, and write φS for the result of applying S to φ. The composition $S_1 S_2$ of two substitutions S_1 and S_2 is defined such that $\varphi(S_1 S_2) = (\varphi S_1) S_2$. We write $S_1 \leq_{\mathbb{B}} S_2$ (substitution S_1 is more general than S_2) iff there exists a substitution S_3 such that for all $x \in \beta$, $x S_1 S_3 \equiv_{\mathbb{B}} x S_2$.

Definition 4.1 (Boolean Unification). Given formulas φ and ψ the *Boolean Unification Problem* $\varphi \stackrel{?}{=} \psi$ is to compute a solution, i.e., a substitution S such that $\varphi S \equiv_{\mathbb{B}} \psi S$, or to report that no solution exists. A solution S is a most general unifier (mgu), if $S \leq_{\mathbb{B}} S'$ for all other solutions S'.

Note that the most general unifier is not unique, not even modulo $\equiv_{\mathbb{B}}$, but importantly the set of all solutions can be represented by a single unifier, i.e. the Boolean unification problem is unitary [Martin and Nipkow 1989].

Theorem 4.2 (Martin and Nipkow [1989]). *Boolean Unification is decidable. Moreover, if $\varphi \stackrel{?}{=} \psi$ is solvable then there exists a most general unifier.*

The original result goes back Boole himself [Boole 1847]. The most general unifier can be computed by the *Successive Variable Elimination* (SVE) algorithm [Martin and Nipkow 1989] or Löwenheim's algorithm [Löwenheim 1908]. In this paper, we shall not concern us with the specific implementation details of these algorithms, but instead we refer the reader to [Martin and Nipkow 1989] and [Madsen and van de Pol 2020]. The existence of most general unifiers and algorithms to compute them is what enables us to extend Algorithm W and ultimately to build a type inference algorithm for the proposed relational nullable type system.
We now present a few examples of Boolean unification problems. We begin with:

$$x \wedge y \stackrel{?}{=} \mathsf{T} \quad \text{has mgu} \quad S = \{x \mapsto \mathsf{T}, y \mapsto \mathsf{T}\} \tag{Example I}$$

$$x \wedge y \stackrel{?}{=} \mathsf{F} \quad \text{has mgu} \quad S = \{x \mapsto x \wedge \neg y\} \tag{Example II}$$

The substitution $\{x \to \mathsf{F}, y \to \mathsf{T}\}$ is another unifier for Example II, but it is *not* most general. Other unifiers are $\{x \to \mathsf{T}, y \to \mathsf{F}\}$ and $\{x \to \neg y\}$, but none of these is most general, and all are instances of the most general unifier given above. Note that the unifier is equi-general to another most-general unifier $\{y \to y \wedge \neg x\}$. There can be many unifiers and several most general unifiers, but once we have *one* most general unifier, we can obtain all other unifiers by instantiating it.

The seemingly more complicated unification problem:

$$x \vee y \stackrel{?}{=} x \wedge y \quad \text{has mgu} \quad S = \{x \mapsto y\} \tag{Example III}$$

However, not all Boolean unification problems have solutions:

$$\mathsf{T} \stackrel{?}{=} \mathsf{F} \quad \text{cannot be unified.} \tag{Example IV}$$

$$x \stackrel{?}{=} \neg x \quad \text{cannot be unified.} \tag{Example V}$$

4.2 Syntax and Semantics

Syntax. The syntax of the $\lambda^{\mathsf{rel}}_{\mathsf{null}}$ calculus (cf. Figure 5a) includes the standard lambda calculus constructs: variables, constants, lambda abstractions, and function applications. The let-expression allows polymorphic generalization, as is standard in Hindley-Milner. We include the if-then-else expression to illustrate how the type system merges information from control-flow paths. Pairs and projections illustrate that we can handle nested data types. The null value is our raison d'être. The expression choose $\overline{e_m}\,\{P \Rightarrow \overline{e_b}\}$ enables pattern matching on a sequence of expressions. Note that we use $\overline{x}$ throughout the paper to denote finite sequences of the proper length.

$$\begin{array}{rcl}
v \in \mathit{Val} & = & c \mid \lambda x.\, e \mid (v, v) \mid \mathsf{null} \\
e \in \mathit{Exp} & = & x \mid v \mid e\ e \\
& \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
& \mid & (e, e) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \\
& \mid & \mathsf{choose}\ \overline{e_m}\ \{P \Rightarrow \overline{e_b}\} \\
P \in \mathit{Mat} & = & m \times n \text{ matrix of } \mathit{Pat} \\
p \in \mathit{Pat} & = & \boxdot \mid x \mid \mathsf{null} \\
\\
c \in \mathit{Cst} & = & \text{a set of constants} \\
x, y \in \mathit{Var} & = & \text{a set of variable symbols}
\end{array}$$

(a) Syntax of $\lambda^{\mathsf{rel}}_{\mathsf{null}}$.

$$\begin{array}{rcl}
\varphi \in \mathit{Formula} & = & \mathsf{T} \mid \mathsf{F} \mid \beta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
\pi \in \mathit{ProperType} & = & \alpha \mid \iota \mid \tau \to \tau \mid \tau \times \tau \\
\tau \in \mathit{NullableType} & = & \pi\ ?\ (\varphi, \psi) \\
\\
\sigma \in \mathit{Scheme} & = & \tau \mid \forall\alpha.\, \sigma \mid \forall\beta.\, \sigma \\
\\
\iota \in \mathit{BaseType} & = & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Nat} \mid \mathsf{String} \mid \cdots \\
\alpha \in \mathit{TypeVar} & = & \text{a set of proper type variables} \\
\beta \in \mathit{BoolVar} & = & \text{a set of Boolean variables}
\end{array}$$

(b) Types and Type Schemes of $\lambda^{\mathsf{rel}}_{\mathsf{null}}$.

Fig. 5. Syntax and Types of $\lambda^{\mathsf{rel}}_{\mathsf{null}}$.

$$(\lambda x.\, e)\, v \longrightarrow e[x \mapsto v] \quad \text{(E-App)}$$

$$\text{let } x = v \text{ in } e \longrightarrow e[x \mapsto v] \quad \text{(E-Let)}$$

$$\text{if true then } e_2 \text{ else } e_3 \longrightarrow e_2 \quad \text{(E-Ite-1)}$$

$$\text{if false then } e_2 \text{ else } e_3 \longrightarrow e_3 \quad \text{(E-Ite-2)}$$

$$\text{fst } (v_1, v_2) \longrightarrow v_1 \quad \text{(E-Fst)}$$

$$\text{snd } (v_1, v_2) \longrightarrow v_2 \quad \text{(E-Snd)}$$

$$\frac{\text{smallest } i \text{ s.t. } S \triangleq \text{matchRow}(P_i, \overline{v})}{\text{choose } \overline{v}\, \{P \implies \overline{e_b}\} \longrightarrow e_b^i S} \quad \text{(E-Choose)}$$

$$\text{matchRow}(\overline{p}, \overline{v}) = \bigcup_i \text{matchPat}(p_i, v_i) \quad \text{(M-Row)}$$

$$\text{matchPat}(\boxdot, v) = [] \quad \text{(M-Wild)}$$

$$\text{matchPat}(\text{null}, \text{null}) = [] \quad \text{(M-Null)}$$

$$\frac{v \neq \text{null}}{\text{matchPat}(x, v) = [x \mapsto v]} \quad \text{(M-Var)}$$

Fig. 6. Evaluation Rules of $\lambda^{\text{rel}}_{\text{null}}$.

The choose $\overline{e_m}\, \{P \implies \overline{e_b}\}$ expression is akin to a pattern match. The expression consists of the following components: The vector of *match expressions* $\overline{e_m}$, the *pattern matrix* P, and the vector of *expression bodies* $\overline{e_b}$. We shall write P_i for the i^{th} row of the pattern matrix. We require that the length of the match expression vector $\overline{e_m}$ matches (no pun intended) the length of each row in the pattern matrix P, i.e. $|\overline{e_m}| = |P_i|$. We also require that the number of rows equals the length of the body vector, i.e. $|P| = |\overline{e_b}|$. The entries of the pattern matrix consist of three types of patterns: A wildcard $\boxdot$ pattern, a null pattern, and a non-null pattern: variable x. This opens a local scope for x, which can only be used in the corresponding body e_b^i. We require that patterns are linear, i.e. a variable cannot occur twice in the same pattern row P_i.

A choose expression differs from a normal pattern match in three important ways: (i) a variable pattern exclusively matches non-null values, (ii) patterns cannot be nested, and (iii) patterns do not have to be exhaustive. Note that a consequence of (i) is that a wildcard pattern is *not* equivalent to a variable pattern with a fresh variable name.

Semantics. The semantics of $\lambda^{\text{rel}}_{\text{null}}$ is based on a call-by-value small-step operational semantics for the lambda calculus. Figure 6 shows the evaluation rules of $\lambda^{\text{rel}}_{\text{null}}$. The evaluation rules are standard. We write $e[x \mapsto v]$ for the capture avoiding substitution of $x \mapsto v$ into e, and extend it to simultaneous substitutions $e\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. We omit the congruence rules, but intuitively evaluation always proceeds from left to right. The only rule that warrants discussion is the evaluation rule for the choose construct.

The (E-Choose) rule states that if we have a choose expression where the match expression vector has been reduced to a value vector $\overline{v}$ then we look for the first row P_i in the pattern matrix where P_i matches $\overline{v}$. The partial function $\text{matchRow}(\overline{p}, \overline{v})$ is defined when the value vector $\overline{v}$ matches the pattern vector $\overline{p}$. Otherwise, if there is no match between $\overline{v}$ and $\overline{p}$, the function is undefined. When defined, the function returns a substitution of the variables in the pattern vector. If row i is the first row where a pattern match is found, then the choose expression reduces to the match body expression e_b^i with the substitution S applied to it. If there is no match, the choose expression is stuck. We shall show that the type system ensures that no expression ever gets stuck.

The (M-Row) rule defines the partial function $\text{matchRow}(\overline{p}, \overline{v})$ as the union of the substitutions of $\text{matchPat}(p_i, v_i)$ for each pattern and value pair. We know from our syntactic requirement that the length of $\overline{p}$ and $\overline{v}$ are the same. Moreover, we can safely compute the union of the substitutions returned by $\text{matchPat}(p_i, v_i)$ because the variables in $\overline{p}$ are assumed to be disjoint (i.e. all patterns are linear). If the partial function matchPat is undefined for any (p_i, v_i) then matchRow is undefined for the whole vector $(\overline{p}, \overline{v})$. The rules (M-Wild), (M-Null), and (M-Var) define when a pattern

p matches a value v. The rules are straightforward: A wildcard matches anything (but does not bind any variable), the null pattern matches only the null value (and does not bind any variable), and finally a variable pattern x matches any non-null value and binds x to that value. In other words, the matchPat function is undefined in two cases: (i) when the pattern is null and the value is non-null, and (ii) when the pattern is a variable x and the value is null.

4.3 How $\lambda^{\mathsf{rel}}_{\mathsf{null}}$ Programs "Get stuck"

We briefly illustrate how $\lambda^{\mathsf{rel}}_{\mathsf{null}}$ programs may get stuck during evaluation. The obvious reason is when null is applied as a function, or when null is used as a condition. Another reason why a program gets stuck is if in a choose expression none of the cases apply to the match expressions:

$$\mathsf{choose}\ \mathsf{null}\ \{\mathsf{case}\ x \Rightarrow e\} \qquad \text{(Example I)}$$

$$\mathsf{choose}\ (123, \mathsf{null})\ \{\mathsf{case}\ (\boxdot, y) \Rightarrow e\} \qquad \text{(Example II)}$$

$$\mathsf{choose}\ (\mathsf{null}, \mathsf{null})\ \{\mathsf{case}\ (\boxdot, y) \Rightarrow e_1, \mathsf{case}\ (x, \boxdot) \Rightarrow e_2\} \qquad \text{(Example III)}$$

In Example I, the variable pattern *does not* match the null value. In Example II, the value vector $(123, \mathsf{null})$ does not match the single case in the pattern match. In Example III, the value vector $(\mathsf{null}, \mathsf{null})$ does not match any of the two cases in the pattern match.

4.4 Declarative Type System

Mono Types. The types of $\lambda^{\mathsf{rel}}_{\mathsf{null}}$ are separated into *proper types* π and *nullable types* τ, cf. Fig. 5b. The proper types and nullable types are defined by mutual induction. The proper types π include types variables α, a set of base types ι (e.g. Bool, Int, etc.), function types $\tau_1 \to \tau_2$, and product types $\tau_1 \times \tau_2$. Nullable types τ are triples of the form $\pi\ ?\ (\varphi, \psi)$ where π is a proper type, and φ and ψ are two Boolean formulas that capture the nullity of the type. Intuitively, φ and ψ capture when an expression *may* evaluate to null and *may* evaluate to a non-null value. If $\varphi \equiv_{\mathbb{B}} \mathsf{F}$, the expression *cannot* evaluate to null. Similarly, if $\psi \equiv_{\mathbb{B}} \mathsf{F}$, the expression *cannot* evaluate to a non-null value.

We extend Boolean equivalence to proper and nullable types, as the smallest relation such that:

- If $\tau_1 \equiv_{\mathbb{B}} \tau_1'$ and $\tau_2 \equiv_{\mathbb{B}} \tau_2'$ then both $(\tau_1, \tau_2) \equiv_{\mathbb{B}} (\tau_1', \tau_2')$ and $\tau_1 \to \tau_2 \equiv_{\mathbb{B}} \tau_1' \to \tau_2'$
- If $\pi \equiv_{\mathbb{B}} \pi'$, $\varphi \equiv_{\mathbb{B}} \varphi'$, and $\psi \equiv_{\mathbb{B}} \psi'$ then $\pi\ ?\ (\varphi, \psi) \equiv_{\mathbb{B}} \pi'\ ?\ (\varphi', \psi')$

A *substitution* S is a mapping in $(\mathit{TypeVar} \to \mathit{ProperType}) \cup (\mathit{BoolVar} \to \mathit{Formula})$, mapping type variables to proper types and Boolean variables to formulas. We define $\tau S = \tau'$, the application of substitution S to type τ, in a standard way. In such a case, τ is said to be *more general than* τ'. We write $S_0 S_1$ for the substitution such that $\tau(S_0 S_1) = (\tau S_0) S_1$.

Type Schemes. Type schemes σ extend nullable types by quantification over proper type variables α and Boolean variables β. That is, a type scheme is of the form $\forall \overline{\gamma}.\ \tau$, where $\overline{\gamma}$ is a vector of proper type variables and Boolean variables. Figure 5b shows the types and type schemes of $\lambda^{\mathsf{rel}}_{\mathsf{null}}$.

We define $\mathsf{ftv}(\sigma)$ as the variables that occur free in σ. These include unbound proper type variables α and unbound Boolean variables β. We extend substitutions to capture-avoiding substitutions on type schemes (written σS). We write $\sigma \sqsubseteq \tau$ (type τ is an instance of type scheme σ) if $\sigma = \forall \overline{\gamma} \tau'$ and $\tau' S = \tau$ for some substitution S with $\mathrm{dom}(S) = \overline{\gamma}$.

Type Rules. Figure 7 shows the declarative type rules of $\lambda^{\mathsf{rel}}_{\mathsf{null}}$. A declarative typing judgement is of the form $\Gamma \vdash_d e : \tau$. As is standard, the context $\Gamma : \mathit{Var} \hookrightarrow \mathit{Scheme}$ is a partial function from variables to type schemes. We extend substitutions to contexts (ΓS) and define $\mathsf{ftv}(\Gamma)$ as the union of all free type variables in the range of Γ. The type judgement $\Gamma \vdash_d e : \tau$, which expands to $\Gamma \vdash_d e : \pi\ ?\ (\varphi, \psi)$, states that the expression e has proper type π under type environment Γ with nullity formulas φ and ψ. Most of the type rules are straightforward.

$$\boxed{\Gamma \vdash_d e : \tau}$$

$$\frac{\Gamma \vdash_d e : \tau_1 \quad \tau_1 \equiv_{\mathbb{B}} \tau_2}{\Gamma \vdash_d e : \tau_2} \text{ (T-Eq)} \qquad \frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_d x : \tau} \text{ (T-Var)}$$

$$\frac{\text{typeOf}(c) = \iota}{\Gamma \vdash_d c : \iota \ ?\ (\varphi, \mathsf{T})} \text{ (T-Cst)} \qquad \frac{}{\Gamma \vdash_d \text{null} : \pi \ ?\ (\mathsf{T}, \psi)} \text{ (T-Null)}$$

$$\frac{\Gamma \vdash_d e : \tau_1 \times \tau_2 \ ?\ (\mathsf{F}, \psi)}{\Gamma \vdash_d \text{fst}\, e : \tau_1} \text{ (T-Fst)} \qquad \frac{\Gamma \vdash_d e : \tau_1 \times \tau_2 \ ?\ (\mathsf{F}, \psi)}{\Gamma \vdash_d \text{snd}\, e : \tau_2} \text{ (T-Snd)}$$

$$\frac{\Gamma \vdash_d e_1 : \tau_1 \quad \Gamma \vdash_d e_2 : \tau_2}{\Gamma \vdash_d (e_1, e_2) : (\tau_1 \times \tau_2) \ ?\ (\varphi, \mathsf{T})} \text{ (T-Pair)}$$

$$\frac{\Gamma, x : \tau_1 \vdash_d e : \tau_2}{\Gamma \vdash_d \lambda x.\, e : (\tau_1 \to \tau_2) \ ?\ (\varphi, \mathsf{T})} \text{ (T-Abs)}$$

$$\frac{\Gamma \vdash_d e_1 : (\tau_1 \to \tau_2) \ ?\ (\mathsf{F}, \psi) \quad \Gamma \vdash_d e_2 : \tau_1}{\Gamma \vdash_d e_1\, e_2 : \tau_2} \text{ (T-App)}$$

$$\frac{\Gamma \vdash_d e_1 : \tau_1 \quad \Gamma, x : \text{gen}(\Gamma, \tau_1) \vdash_d e_2 : \tau_2}{\Gamma \vdash_d \text{let}\, x = e_1 \,\text{in}\, e_2 : \tau_2} \text{ (T-Let)}$$

$$\frac{\Gamma \vdash_d e_1 : \text{Bool} \ ?\ (\mathsf{F}, \psi) \quad \Gamma \vdash_d e_2 : \tau \quad \Gamma \vdash_d e_3 : \tau}{\Gamma \vdash_d \text{if}\, e_1 \,\text{then}\, e_2 \,\text{else}\, e_3 : \tau} \text{ (T-Ite)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_d e^j_m : \pi_j \ ?\ (\varphi_j, \psi_j) \qquad (\forall j) \\ \Gamma, \text{ext}(\Gamma, P_i) \vdash_d e^i_b : \tau \qquad (\forall i) \\ P \text{ is exhaustive for } (\overline{\varphi}, \overline{\psi})\end{array}}{\Gamma \vdash_d \text{choose}\, \overline{e_m}\, \{P \Rightarrow \overline{e_b}\} : \tau} \text{ (T-Choose)}$$

$$\text{gen}(\Gamma, \tau) = \forall \overline{\gamma}.\, \tau \text{ where } \overline{\gamma} = \text{ftv}(\tau) \setminus \text{ftv}(\Gamma)$$
$$\text{ext}(\Gamma, P_i) = \{x_j : \pi_j \ ?\ (\chi_{i,j}, \mathsf{T}) \mid P_{i,j} = x_j\}$$

Fig. 7. Declarative Type Rules for $\lambda^{\text{rel}}_{\text{null}}$. A typing judgement is of the form $\Gamma \vdash_d e : \tau$ where τ is nullable type. The expanded form is $\Gamma \vdash_d e : \pi \ ?\ (\varphi, \psi)$ where π is a proper type and φ and ψ are Boolean formulas.

The (T-Eq) rule states that if an expression e can be typed as τ_1, that type can be replaced by any equivalent type $\tau_2 \equiv_{\mathbb{B}} \tau_1$. The (T-Var) rule is the standard Hindley-Milner instantiation rule. It states that if the assumption $x : \sigma$ is in the context, then we can *instantiate* σ to a specific type τ, and conclude $x : \tau$. The (T-Cst) rule states that constants can be assigned a base type ι, according to the typeOf function. The nullity (φ, T) encodes that a constant is not null: the constant *may evaluate* to non-null. The polymorphic φ indicates it may or may not evaluate to null. This is always sound, and the flexibility in choosing φ allows the constant to occur in a branch together with null-values. The (T-Null) rule states that the polymorphic type of null is any proper type π with nullity formulas (T, ψ): It may evaluate to null (T), but it can occur in a branch together with non-null values (ψ). The (T-Pair), (T-Fst), and (T-Snd) rules type pairs and projections. The (T-Pair) rule *states* that a pair is not null (φ, T). The projection rules (T-Fst), (T-Snd) *require* that a pair cannot evaluate to null (F, φ). The (T-Abs) and (T-App) rules are straightforward. The (T-Abs) rule states that an abstraction is non-null. The (T-App) rule requires that the applied lambda abstraction must be non-null. The (T-Let) rule is the standard Hindley-Milner generalization rule. The rule states that if we can type e_1 as τ_1 under the environment Γ then we may *generalize* the type τ_1 to a type scheme σ, and type e_2 under an extended environment with $x : \sigma$. The (T-Ite) rule requires that the condition is a non-null Boolean and that the two branches have the same type.

4.5 Type Checking and Pattern Matching

We now present the type rule (T-Choose). In Section 4.7, we will discuss a strict extension of the system with rule (T-Choose-★), which allows *relational nullable polymorphism*.

Given the expression choose $\overline{e_m}\ \{P \Rightarrow \overline{e_b}\}$, where $\overline{e_m}$ is the vector of match expressions, P is the pattern matrix, and $\overline{e_b}$ is the vector of body expressions, (T-Choose) imposes three requirements:

- We require that for each column j, the match expression can be typed as in $\Gamma \vdash_d e_m^j : \tau_j\ ?\ (\varphi_j, \psi_j)$ — all match expressions e_m^j can have different types τ_j and nullity formulas (φ_j, ψ_j).
- We also require that the body expression of each row i can be typed in an extended environment, as in $\Gamma \cup \Gamma_i \vdash_d e_b^i : \tau$. Note that a choice is a similar dataflow merge point as an if-then-else expression. Here we require that all possible body expressions have *the same type* τ. This restriction will be lifted later in (T-Choose-★). The body e_b^i may use variables that are introduced (bound) in the pattern. So the extended context Γ_i extracts all variables from the non-null patterns in row P_i, i.e. those for which $P_{i,j}$ is a variable x_j. Their proper type should be the same as the corresponding match expression $e_m^j : \pi_j$. We add the nullity assumption $(\chi_{i,j}, \mathsf{T})$, which captures that row i is only chosen when e_m^j is non-null (since $P_{i,j}$ requested a non-null value for x_j).
- Finally, we require that the pattern matrix P must be exhaustive for the vectors $\overline{\varphi}$ and $\overline{\psi}$.

Pattern Matrix Exhaustiveness. Intuitively, P is exhaustive, if it has a matching row for each possible value of the match expressions $\overline{e_m}$. We only have to consider their nullity: 0 (for the null value) and 1 (for any non-null value). We first define the possible nullity values of an expression $e : \pi\ ?\ (b, c)$, for Boolean values $b, c \in \mathcal{B}$. Recall that b indicates that e may be null and c indicates that e may be a non-null value. For a vector $\overline{e} : \overline{\pi}\ ?\ (\overline{b}, \overline{c})$, the possible nullity vectors are all combinations of the nullity values of its elements:

$$\mathsf{values}(\mathsf{T},\mathsf{T}) = \{0,1\} \qquad \mathsf{values}(\mathsf{T},\mathsf{F}) = \{0\} \qquad \mathsf{values}(\mathsf{F},\mathsf{T}) = \{1\} \qquad \mathsf{values}(\mathsf{F},\mathsf{F}) = \emptyset$$

$$\mathsf{valuesRow}(\overline{b}, \overline{c}) = \mathsf{values}(b_1, c_1) \times \cdots \times \mathsf{values}(b_n, c_n)$$

In particular, note that if one of the match expressions e_i has nullity (F,F), then the whole vector is uninhabited ($\emptyset$). Next, we define which nullity values are covered by a single pattern, and by a row of patterns:

$$\mathsf{covers}(\mathsf{null}) = \{0\} \qquad \mathsf{covers}(x) = \{1\} \qquad \mathsf{covers}(\boxdot) = \{0,1\}$$

$$\mathsf{coversRow}(\overline{p}) = \mathsf{covers}(p_1) \times \cdots \times \mathsf{covers}(p_n)$$

We can now finally state when a pattern matrix is exhaustive:

Definition 4.3 (Exhaustiveness). A pattern matrix P is exhaustive for the nullity formulas $(\overline{\varphi}, \overline{\psi})$, if for all valuations V, and for all $\overline{a} \in \mathsf{valuesRow}(V(\overline{\varphi}), V(\overline{\psi}))$, there exists a row P_i in P, such that $\overline{a} \in \mathsf{coversRow}(P_i)$.

Example 4.4. Let $e_1 : \pi\ ?\ (\mathsf{F}, \beta)$ and let $e_2 : \pi\ ?\ (\mathsf{T},\mathsf{T})$. We check exhaustiveness of the expression:

$$\mathsf{choose}\ (e_1, e_2)\ \{\mathsf{case}\ (\boxdot, x) \Rightarrow \cdots, \mathsf{case}\ (y, \mathsf{null}) \Rightarrow \cdots\}$$

Note that e_1 must be a non-null value at runtime and that e_2 will be either null or a non-null value. For $\beta = \mathsf{F}$, valuesRow is empty. For $\beta = \mathsf{T}$, $\mathsf{valuesRow}((\mathsf{F}, \beta), (\mathsf{T},\mathsf{T})) = \{1\} \times \{0,1\} = \{(1,0),(1,1)\}$. Pattern row $(\boxdot, x)$ covers instances $\{(0,1),(1,1)\}$ and pattern row (y, null) covers the instance $\{(1,0)\}$. So the value vector $(1,0)$ is covered by the second row, and the value vector $(1,1)$ is covered by the first row. Thus the pattern matrix is exhaustive for these match expressions.

Soundness. We now match the nullity formulas (φ, ψ) derived by the type system with the actual nullity of the values. The following "Canonical Forms Lemma" is essentially proved by inspecting the last applied derivation rule.

Lemma 4.5. *Let v be a typeable value with $\Gamma \vdash_D v : \pi \ ? \ (\varphi, \psi)$. Then:*
(1) If $v = \mathit{null}$ then $\varphi \equiv_{\mathbb{B}} T$.
(2) If $v \neq \mathit{null}$, then $\psi \equiv_{\mathbb{B}} T$.

We can now prove that, when we get to a well-typed choose expression at runtime, the exhaustive pattern matrix will always have a matching row, so the choose cannot "get stuck":

Lemma 4.6. *Let $\overline{v}$, $\overline{\varphi}$ and $\overline{\psi}$ be given. Assume $\Gamma \vdash_D v_j : \pi_j \ ? \ (\varphi_j, \psi_j)$ (for each j). Assume that pattern matrix P is exhaustive for $(\overline{\varphi}, \overline{\psi})$. Then matchRow$(P_i, \overline{v})$ must be defined for some row i in P.*

Proof. Assume $\Gamma \vdash_D v_j : \pi_j \ ? \ (\varphi_j, \psi_j)$. Define $a_j := 0$, if $v_j = \mathsf{null}$ and $a_j := 1$, otherwise. We claim (*) that $\overline{a} \in \mathsf{ValuesRow}(V(\overline{\varphi}), V(\overline{\psi}))$ for all valuations V. Then, since P is exhaustive, we obtain that there is a row i such that $\overline{a} \in \mathsf{coversRow}(P_i)$, i.e. for each j, $a_j \in \mathsf{covers}(P_{i,j})$. By comparing the definition of covers and matchPat, we see that this implies that $\mathsf{matchPat}(P_{i,j}, v_j)$ is defined for each j. But then also $\mathsf{matchRow}(P_i, \overline{v})$ is defined.

Now we prove the claim (*) that $\overline{a} \in \mathsf{ValuesRow}(V(\overline{\varphi}), V(\overline{\psi}))$, i.e. $a_j \in \mathsf{Values}(V(\varphi_j), V(\psi_j))$, for each j. If $a_j = 0$ then $v_j = \mathsf{null}$ and by Lemma 4.5, $\varphi_j \equiv_{\mathbb{B}} \mathsf{T}$, so $V(\varphi_j) = \mathsf{T}$. Similarly, if $a_j = 1$ then $v_j \neq \mathsf{null}$ and by Lemma 4.5, $\psi_j \equiv_{\mathbb{B}} \mathsf{T}$, so $V(\psi_j) = \mathsf{T}$. In both cases, $a_j \in \mathsf{Values}(V(\varphi_j), V(\psi_j))$ by the definition of Values. □

We are now in the position to state soundness of our declarative type system. The following two theorems guarantee that well-typed expressions cannot "get stuck":

Theorem 4.7 (Progress). *Suppose e is a closed, well-typed expression (that is, $\vdash_d e : \tau$ for some τ). Then either e is a value or else there is some expression e' such that $e \longrightarrow e'$.*

Theorem 4.8 (Preservation). *If $\Gamma \vdash_d e : \tau$ and $e \longrightarrow e'$ then $\Gamma \vdash_d e' : \tau$.*

4.6 Type Inference with Algorithm W and Boolean Unification

We now turn to type inference. Figure 8 presents an extension to the classical Algorithm W. It derives judgements of the form $\Gamma \vdash_w e : \tau; S$. The judgement should be read as: Given as input a context Γ and an expression e, compute as output its most general type τ and a substitution to the type variables, such that $\Gamma S \vdash_d e : \tau$. If e is not typeable in an instance of Γ, then the algorithm should fail. This happens for instance when some unification problem fails.

The rules are syntax directed: for each expression it is clear which rule to use. The antecedents should be executed in sequential order (top to bottom). There are three types of antecedents:

- Recursive calls to the algorithm $\vdash_w$ and repeated recursive calls $\vdash_w^*$. For instance, (W-Abs) has a simple recursive call, and (W-Pair) has a repeated recursive call. We write $\Gamma \vdash_w^* \overline{e_1, e_2} : \overline{\tau_1, \tau_2}; S$ as an abbreviation of the two calls $\Gamma \vdash_w e_1 : \tau_1'; S_1$ and $\Gamma S_1 \vdash_w e_2 : \tau_2'; S_2$. The returned types are $\overline{(\tau_1, \tau_2)} := \overline{(\tau_1' S_2, \tau_2')}$ and the substitution is $S := S_1 S_2$. This can be generalized to three calls (W-Ite) or even an arbitrary number of calls (W-Choose), resembling a fold-operation.
- The creation of fresh type variables (proper type variables and nullity variables). This is presented as a non-functional operation, but we implemented a functional version, where fresh variables are derived in a canonical manner from the call graph of Algorithm W.
- Unification problems of the form $S = (t_1 \stackrel{?}{=} t_2)$. This denotes that substitution S is the most general unifier of t_1 and t_2. If the terms cannot be unified, Algorithm W fails. Type unification proceeds by structural matching with occurs check, as in Martelli-Montanari [Martelli and

$$\boxed{\Gamma \vdash_w e : \tau; S}$$

$$\frac{\alpha, \beta \text{ fresh}}{\Gamma \vdash_w \mathsf{null} : \alpha \ ?\ (\mathsf{T}, \beta); id} \quad \text{(W-Null)}$$

$$\frac{\mathsf{typeOf}(c) = \iota \quad \beta \text{ fresh}}{\Gamma \vdash_w c : \iota \ ?\ (\beta, \mathsf{T}); id} \quad \text{(W-Cst)}$$

$$\frac{(x, \sigma) \in \Gamma \quad \tau = \mathsf{inst}(\sigma)}{\Gamma \vdash_w x : \tau; id} \quad \text{(W-Var)}$$

$$\frac{\Gamma \vdash_w^* \overline{e_1, e_2} : \overline{\tau_1, \tau_2}; S \quad \beta \text{ fresh}}{\Gamma \vdash_w (e_1, e_2) : \tau_1 * \tau_2 \ ?\ (\beta, \mathsf{T}); S} \quad \text{(W-Pair)}$$

$$\frac{\Gamma \vdash_w^* \overline{e_1, e_2} : \overline{\tau_1, \tau_2}; S_0 \quad \alpha, \beta, \gamma, \beta_1 \text{ fresh} \quad S_1 = (\tau_1 \stackrel{?}{=} (\tau_2 \to (\alpha \ ?\ (\beta, \gamma)) \ ?\ (\mathsf{F}, \beta_1)))}{\Gamma \vdash_w e_1 e_2 : (\alpha \ ?\ (\beta, \gamma))S_1; S_0 S_1} \quad \text{(W-App)}$$

$$\frac{\alpha, \beta, \gamma, \beta_1 \text{ fresh} \quad \Gamma, x : (\alpha \ ?\ (\beta, \gamma)) \vdash_w e : \tau; S}{\Gamma \vdash_w \lambda x.e : (\alpha \ ?\ (\beta, \gamma))S \to \tau \ ?\ (\beta_1, \mathsf{T}); S} \quad \text{(W-Abs)}$$

$$\frac{\Gamma \vdash_w e : \tau; S_1 \quad \alpha_1, \beta_1, \gamma_1, \alpha_2, \beta_2, \gamma_2, \beta \text{ fresh} \quad S_2 = (\tau \stackrel{?}{=} ((\alpha_1 \ ?\ (\beta_1, \gamma_1)) * (\alpha_2 \ ?\ (\beta_2, \gamma_2)) \ ?\ (\mathsf{F}, \beta)))}{\Gamma \vdash_w Fst(e) : (\alpha_1 \ ?\ (\beta_1, \gamma_1))S_2; S_1 S_2 \quad \Gamma \vdash_w Snd(e) : (\alpha_2 \ ?\ (\beta_2, \gamma_2))S_2; S_1 S_2} \quad \text{(W-Fst / W-Snd)}$$

$$\frac{\Gamma \vdash_w^* \overline{e_1, e_2, e_3} : \overline{\tau_1, \tau_2, \tau_3}; S_0 \quad \beta \text{ fresh} \quad S_1 = (\tau_1 \stackrel{?}{=} \mathsf{Bool} \ ?\ (\mathsf{F}, \beta)) \quad S_2 = (\tau_2 S_1 \stackrel{?}{=} \tau_3 S_1)}{\Gamma \vdash_w \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau_2 S_1 S_2; S_0 S_1 S_2} \quad \text{(W-Ite)}$$

$$\frac{\Gamma \vdash_w e_1 : \tau_1; S_1 \quad \sigma = \mathsf{gen}(\Gamma S_1, \tau_1) \quad \Gamma S_1, x : \sigma \vdash_w e_2 : \tau_2; S_2}{\Gamma \vdash_w \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2; S_1 S_2} \quad \text{(W-Let)}$$

$$\frac{\Gamma \vdash_w^* \overline{e_m} : \overline{\pi \ ?\ (\varphi, \psi)}; S_0 \quad \overline{\beta_{i,j}} \text{ fresh variables} \quad \Gamma_{ext,i} = \{x_j : \pi_j \ ?\ (\beta_{i,j}, \mathsf{T}) \mid P_{i,j} = x_j\} \quad (\Gamma \cup \overline{\Gamma_{ext}})S_0 \vdash_w^* \overline{e_b} : \overline{\tau}; S_1 \quad S_2 = \{\tau_i \stackrel{?}{=} \tau_{i+1} \mid 0 \le i < |\overline{e_b}|\} \quad S_3 = (\mathsf{MatrixFormula}(P, \vec{\varphi}, \vec{\psi})S_1 S_2 \stackrel{?}{=} \mathsf{T})}{\Gamma \vdash_w \mathsf{choose}\ \overline{e_m}\ \{P \implies \overline{e_b}\} : \tau_0 S_2 S_3; S_0 S_1 S_2 S_3} \quad \text{(W-Choose)}$$

$$\mathsf{inst}(\forall \overline{\gamma}.\tau) = \tau[\overline{\gamma} := \overline{\beta}] \text{ for fresh variables } \overline{\beta} \qquad \mathsf{gen}(\Gamma, \tau) = \forall \overline{\gamma}.\tau, \text{ where } \overline{\gamma} = \mathsf{ftv}(\tau) \setminus \mathsf{ftv}(\Gamma)$$

Fig. 8. Type Rules of Algorithm W for $\lambda^{\mathsf{rel}}_{\mathsf{null}}$.

Montanari 1982], until two Boolean formulas must be unified. For Boolean unification (Sec. 4.1) we use successive variable elimination [Martin and Nipkow 1989]. We refer to [Madsen and van de Pol 2020] for a detailed description of the combination.

All rules, except (W-Choose), extend the standard rules of Algorithm W by propagating the nullity information. Note how "arbitrary formulas" (φ, ψ) in the conclusions of the T-rules are implemented by fresh type variables in the corresponding W-rules (e.g. in the constructors (W-Null), (W-Cst), (W-Pair) and (W-Abs)). These variables can become instantiated by further calls to Algorithm W. On the other hand, checks for nullity in the T-rules are implemented by unifying formulas with F in the corresponding W-rules (e.g. in the destructors (W-App), (W-Fst), (W-Snd), (W-Ite)). When the T-rules require equal types, the W-rules implement this by unification (e.g. in (W-App), (W-ite) and (W-Choose)). Hence, Algorithm W does not need the equivalent of the (T-Eq)-rule. The (W-Let) rule implements the standard generalization of types.

There is a small implementation detail: for generalization, the formulas in the context must be simplified to eliminate non-essential variables.[3]

[3]For instance, $\beta \vee \neg\beta$ should be simplified to T, to avoid blocking the generalization of β, ultimately leading to a less general type. This can be achieved by representing Boolean formulas in canonical form, such as BDDs or Boolean Rings.

The rule (W-Choose) requires extra explanation. First, note the definition of the extended contexts $\Gamma_{ext,i}$. Here $(\beta_{i,j}, \mathsf{T})$ encodes that a variable x_j in a pattern P_i matches a non-null value. The polymorphic fresh variable $\beta_{i,j}$ can become instantiated when unifying it with types from other branches. By a slight abuse of notation, the following $\vdash_w^*$ uses different contexts $\Gamma \cup \Gamma_{ext,i}$ in each iteration. Recall that substitution S_0 is accumulated and threaded through this computation, resulting in the final S_1.

Next, we explain how we can infer the proper nullity information. A crucial step is to define a Matrix Formula that expresses the exhaustiveness of the pattern matrix P. The strategy to generate this matrix formula is (1) to create an "abstract" pattern matrix from P; (2) to saturate (and minimize) the pattern matrix; and (3) to transform the saturated pattern matrix to a Boolean formula.

The abstract pattern matrix $Q = \mathsf{Abstract}(P)$ consists of values in $\{0, 1, \boxdot\}$, and is obtained from the pattern matrix P by replacing all its pattern variables by a 1. Given the typed match expressions $e_m^j : \pi_j \ ? \ (\varphi_j, \psi_j)$ and an abstract pattern matrix Q, we define a Boolean formula that expresses that $\overline{e_m}$ is guaranteed to match some row i of Q. In words, expressions matching a 1 cannot be null ($\neg\varphi_j$) and expressions matching a 0 cannot be non-null ($\neg\psi_j$). We ensure that we don't impose unnecessary restrictions for uninhabited types (F, F):

$$\mathsf{Inhabited}(\overline{\varphi}, \overline{\psi}) = \bigwedge_j (\varphi_j \vee \psi_j)$$

$$\mathsf{matchesRow}(Q_i, \overline{\varphi}, \overline{\psi}) = \bigwedge_j \{\neg\varphi_j \mid Q_{ij} = 1\} \wedge \bigwedge_j \{\neg\psi_j \mid Q_{ij} = 0\}$$

$$\mathsf{matchesMatrix}(Q, \overline{\varphi}, \overline{\psi}) = \bigvee_i \{\mathsf{matchesRow}(Q_i, \overline{\varphi}, \overline{\psi})\}$$

$$\mathsf{MatrixFormula}(P, \overline{\varphi}, \overline{\psi}) = \neg\mathsf{Inhabited}(\overline{\varphi}, \overline{\psi}) \vee \mathsf{matchesMatrix}(\mathsf{Saturate}(\mathsf{Abstract}(P)), \overline{\varphi}, \overline{\psi})$$

The main reason why saturation must be applied, is that matchesMatrix requires that the match expressions are covered by a single row in the matrix. However, this is too strong, since an exhaustive matrix only requires that *each instance* of these expressions is covered by *some row*. By combining information from multiple rows into a single row, the saturated matrix has the property that it covers the match expressions by a single row. Before defining the saturation procedure in detail, we motivate and explain it with an example.

Example 4.9. Consider the program fragment

$$\mathsf{choose}\ (\mathsf{null}, 5, \mathsf{if}\ b\ \mathsf{then}\ 17\ \mathsf{else}\ \mathsf{null})\ \{\mathsf{case}\ (\mathsf{null}, x, y) \Rightarrow \cdots, \mathsf{case}\ (\boxdot, z, \mathsf{null}) \Rightarrow \cdots\}$$

The match expressions $(\mathsf{null}, 5, \mathsf{if}\ b\ \mathsf{then}\ 17\ \mathsf{else}\ \mathsf{null})$ have most general nullity information (T, β_1), (β_2, T) and (T, T), i.e. their nullity vectors are $\overline{\varphi} = (\mathsf{T}, \beta_2, \mathsf{T})$ and $\overline{\psi} = (\beta_1, \mathsf{T}, \mathsf{T})$. Note that in this case, $\mathsf{Inhabited}(\overline{\varphi}, \overline{\psi}) \equiv_{\mathbb{B}} \mathsf{T}$. The corresponding nullity vectors are $\{0\} \times \{1\} \times \{0, 1\} = \{(0, 1, 0), (0, 1, 1)\}$.

We match $(\overline{\varphi}, \overline{\psi})$ against the abstract pattern matrix of the two cases, which is Q^1 on the left:

$$Q^1 = \begin{pmatrix} 0 & 1 & 1 \\ \boxdot & 1 & 0 \end{pmatrix} \quad \Rightarrow \quad Q^2 = \begin{pmatrix} 0 & 1 & 1 \\ \boxdot & 1 & 0 \\ 0 & 1 & \boxdot \end{pmatrix} \quad \Rightarrow \quad Q^3 = \begin{pmatrix} \boxdot & 1 & 0 \\ 0 & 1 & \boxdot \end{pmatrix}$$

Note that the nullity vectors are not covered by a single row, although value $(0, 1, 0)$ is covered by the second row and $(0, 1, 1)$ is covered by the first row. We saturate this matrix by adding a third row, which combines the "shared information" of the original rows (see middle matrix Q^2). Note that both nullity vectors are now covered by the new row in the saturated matrix. Finally, we can simplify the pattern matrix, since the first row is subsumed by the new third row (Q_1^2 covers a subset

of the nullity vectors covered by Q_3^2). This results in the matrix Q^3. Simplification is not obligatory, but it helps in reducing the final formula. For this example, the generated Matrix Formula is:

$$\mathsf{matchesRow}(Q_1^3, \overline{\varphi}, \overline{\psi}) \vee \mathsf{matchesRow}(Q_2^3, \overline{\varphi}, \overline{\psi}) = (\neg\beta_2 \wedge \neg\mathsf{T}) \vee (\neg\beta_1 \wedge \neg\beta_2) \equiv_{\mathbb{B}} \neg\beta_1 \wedge \neg\beta_2$$

Algorithm W unifies $\neg\beta_1 \wedge \neg\beta_2 \stackrel{?}{=} \mathsf{T}$, yielding the most general substitution $\{\beta_1 \mapsto \mathsf{F}, \beta_2 \mapsto \mathsf{F}\}$. □

We now define the saturation and simplification procedure more formally. Define a partial order on abstract patterns $\{0, 1, \boxdot\}$ by the following laws:

$$0 \leq \boxdot \qquad 1 \leq \boxdot \qquad x \leq x$$

The corresponding greatest lowerbound on elements is defined by the following laws:

$$x \sqcap \boxdot = x \qquad \boxdot \sqcap x = x \qquad x \sqcap x = x$$

Note that $0 \sqcap 1$ and $1 \sqcap 0$ are undefined, since 0 and 1 have no common lowerbound in this partial order. Next, extend this partial order and the partial greatest lowerbound function in a pointwise fashion to vectors; in particular, we will apply them to rows in the pattern matrix.

Define the partial operation $\oplus$ to combine compatible rows as follows:

$$(\overline{x_0}, 1, \overline{x_1}) \oplus (\overline{y_0}, 0, \overline{y_1}) = (\overline{z_0}, \boxdot, \overline{z_1}), \text{ iff } \overline{z_0} := \overline{x_0} \sqcap \overline{y_0} \text{ and } \overline{z_1} := \overline{x_1} \sqcap \overline{y_1} \text{ are defined.}$$

Example 4.10. In Example 4.9, $Q_1^2 \leq Q_3^2$ since in the third column, $1 \leq \boxdot$. The third row in Q^2 is explained by $(0, 1, 1) \oplus (\boxdot, 1, 0) = (0, 1, \boxdot)$, with $\overline{x_0} = (0, 1)$, $\overline{y_0} = (\boxdot, 1)$, and $\overline{z_0} = (0, 1) \sqcap (\boxdot, 1) = (0, 1)$. In this example, the rightmost vectors $\overline{x_1} = \overline{y_1} = \overline{z_1} = ()$. □

Definition 4.11 (Saturated and Minimal). An abstract pattern matrix Q is *saturated*, if for every two rows Q_i and Q_j, if $Q_i \oplus Q_j$ is defined, then there is a row Q_k with $Q_i \oplus Q_j \leq Q_k$.
An abstract pattern matrix Q is *minimal*, if for no two different rows Q_i and Q_j, $Q_i \leq Q_j$.

Lemma 4.12. *For every abstract pattern matrix Q, there exists a unique (modulo order of rows) saturated and minimal matrix Saturate(Q).*

Proof. We first saturate the matrix, by repeatedly applying saturation steps: While Q_i and Q_j are rows in the matrix and $Q_i \oplus Q_j$ is defined but not a row in the matrix, then add the row $Q_i \oplus Q_j$. Since we only add rows, and the number of possible different rows is finite, this process terminates with a unique result (modulo the order of rows) and yields a saturated matrix.

Next, we simplify the matrix by minimization steps: While Q_i and Q_j are different rows with $Q_i \leq Q_j$, remove row Q_i. Since we only remove rows, this process terminates with a unique result: the set of minimal rows, hence it yields a minimal matrix. Note that every minimization step keeps the matrix saturated (since Q_j still covers everything covered by Q_i).

So the result is uniquely defined (modulo row order), minimal, and saturated. □

In Example 4.9, $Q^1 \Rightarrow Q^2$ was a saturation step and $Q^2 \Rightarrow Q^3$ was a minimization step.[4]

Lemma 4.13. *Let Q be an abstract pattern matrix.*

(1) Saturate(Q) covers the same nullity vectors as Q;
(2) If Q is saturated and covers all nullity vectors in valuesRow$(\overline{b}, \overline{c})$, then these are already covered by a single row Q_i;
(3) "P is exhaustive for $\overline{\varphi}, \overline{\psi}$" $\equiv_{\mathbb{B}}$ MatrixFormula$(P, \overline{\varphi}, \overline{\psi})$.

Proof. (1) In saturation, we add a row $Q_i \oplus Q_j$, but all its instances are already covered by either Q_i or by Q_j. In minimization, we remove row Q_i, but all its instances are also covered by Q_j.

[4] In the implementation, we minimize the matrix initially and after each saturation step, and we only add $Q_i \oplus Q_j$ if it is not yet covered by any Q_k, in order to avoid adding rows that would be removed later.

(2) Let $\overline{a_1}, \overline{a_2} \in \mathsf{ValuesRow}(\overline{b}, \overline{c})$. Let $A_i := \{a_1^i, a_2^i\}$, then for each i, $A_i \subseteq \mathsf{Values}(b_i, c_i)$, by definition of ValuesRow. By assumption, all elements in $A_1 \times \cdots \times A_n$ are covered by Q. Define $a_3^i = a_1^i$, if $a_1^i = a_2^i$, and $a_3^i = \boxdot$, otherwise. Since Q is saturated, $\overline{a_3}$ is covered by some row k of Q. But then $\overline{a_1}$ and $\overline{a_2}$ are covered by the same row k in Q.

(3) By Lemma 4.12, we can compute $Q = \mathsf{Saturate}(\mathsf{Abstract}(P))$.
$\Rightarrow$: Let P be exhaustive. There are two cases. If $(\overline{\varphi}, \overline{\psi})$ contains a (F, F) entry for all valuations, then $\neg\mathsf{Inhabited}(\overline{\varphi}, \overline{\psi})$ holds, and the Matrix Formula is true.
Otherwise, there is a valuation V such that $\mathsf{valuesRow}(V(\overline{\varphi}), V(\overline{\psi})) \neq \emptyset$. Since P is exhaustive, all $\overline{a}$ in its nullity vector are covered by P. By (1), Q still covers the same $\overline{a}$. By (2), they are even covered by a single row Q_k. Hence $\mathsf{matchesRow}(Q_k, V(\overline{\varphi}), V(\overline{\psi}))$ holds, and the Matrix Formula is true.
$\Leftarrow$: Assume $V(\mathsf{MatrixFormula}(P, \overline{\varphi}, \overline{\psi})) = \mathsf{T}$ for arbitrary V. We must show that P is exhaustive for $(V(\overline{\varphi}), V(\overline{\psi}))$. Let $\overline{a} \in \mathsf{valuesRow}(V(\overline{\varphi}), V(\overline{\psi}))$ be given, then $\mathsf{Inhabited}(V(\overline{\varphi}), V(\overline{\psi}))$ is true, so $\mathsf{matchesMatrix}(Q, V(\overline{\varphi}), V(\overline{\psi}))$ holds. So for some row k, the disjunct $\mathsf{matchesRow}(Q_k, V(\overline{\varphi}), V(\overline{\psi}))$ holds. This means that Q_k covers $\overline{a}$, hence Q covers $\overline{a}$, hence P covers $\overline{a}$ by (1). Hence P is exhaustive.

□

Example 4.14. In Example 4.4, we obtained the pattern matrix $P = ((\boxdot, 1), (1, 0))$. We saturate it by adding the generalized row $(1, \boxdot)$. Now we remove the redundant row $(1, 0)$. So the minimal saturated pattern matrix is $((\boxdot, 1), (1, \boxdot))$. Recall that the nullity of the matched terms was (F, β) and (T, T). So the Matrix Formula in this case is: $(\neg\mathsf{T} \vee \neg\mathsf{F}) \equiv_{\mathbb{B}} \mathsf{T}$. Indeed, the matrix is exhaustive, since the given expressions are covered by the second row of the *minimal saturated* matrix.

Lemma 4.13(3) is the key to relate the declarative system with Algorithm W. We formulate the correctness of Algorithm W with null and pattern matching as follows:

Theorem 4.15 (Soundness of W). *If* $\Gamma \vdash_w e : \tau; S$ *then* $\Gamma S \vdash_d e : \tau$.

Theorem 4.16 (Completeness of W). *If* $\Gamma S \vdash_d e : \tau$ *then for some* τ' *and* S', $\Gamma \vdash_w e : \tau'; S'$ *and there exists* S_0, τ_0 *such that* $\Gamma S' S_0 = \Gamma S$ *and* $gen(\Gamma S', \tau')S_0 \sqsubseteq \tau_0 \equiv_{\mathbb{B}} \tau$.

Corollary 4.17 (Principal Types). *The calculus* λ^{rel}_{null} *enjoys the principal type property.*

4.7 Extension: Polymorphic Relational Nullability with the Choose-⋆ Rule

We now discuss an extension of the type system that enables *relational polymorphic nullability*. Consider the following two program fragments:

```
choose x {
    case null => "Hello World"
    case w    => "Goodbye World"
}
```

```
choose x {
    case null => null
    case w    => w + 42
}
```

The left program fragment can be given the non-nullable type String ? (F, T), because both branches return a non-null string. On the other hand, the program fragment on the right *cannot* be given the non-nullable type Int ? (F, T) — *not even when* x *would be a constant value.* The reason is that the (T-Choose) rule, like (T-Ite), requires that the types of all branches must be the same, which would be Int ? (T, T) in this case. Thus, even if we know that x cannot possibly be null, this information is not preserved in the type of a choose expression. In our extension, the nullities of the branches will be combined with $\vee$. Moreover, the type of each branch will depend on its pattern. Consequently, the program fragment on the right gets type Int ? (β, γ), whenever x : Int ? (β, γ). This is achieved by the (Choose-⋆) rule, shown in Figure 9. A similar extension can be made to the (T-Ite) rule.

4.7.1 Typing. In order to infer a more precise, relational nullable type, we now extend the declarative type system with (T-Choose-⋆) and Algorithm W with (W-Choose-⋆), see Figure 9. The old rules (T-Choose) and (W-Choose) are now superfluous.

$$\frac{\begin{array}{c}\Gamma \vdash_d e_m^j : \pi_j \ ?\ (\varphi_j, \psi_j) \qquad (\forall j) \\ \Gamma_{ext,i} := \{x_j : \pi_j \ ?\ (\chi_{i,j}, \mathsf{T}) \mid P_{i,j} = x_j\} \\ \Gamma \cup \Gamma_{ext,i} \vdash_d e_b^i : \pi' \ ?\ (\varphi_i', \psi_i') \qquad (\forall i) \\ P \text{ is exhaustive for } (\overline{\varphi}, \overline{\psi}) \\ \varphi_r = \bigvee_i (\mathsf{Line}(P_i, \overline{\varphi}, \overline{\psi}) \wedge \varphi_i') \\ \psi_r = \bigvee_i (\mathsf{Line}(P_i, \overline{\varphi}, \overline{\psi}) \wedge \psi_i')\end{array}}{\Gamma \vdash_d \mathsf{choose}\ \overline{e_m}\ \{P \Rightarrow \overline{e_b}\} : \pi' \ ?\ (\varphi_r, \psi_r)} \text{(T-Choose-}\star\text{)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_w^* \overline{e_m} : \overline{\pi \ ?\ (\varphi, \psi)}; S_0 \\ \overline{\beta_{i,j}} \text{ fresh variables} \\ \Gamma_{ext,i} := \{x_j : \pi_j \ ?\ (\beta_{i,j}, \mathsf{T}) \mid P_{i,j} = x_j\} \\ (\Gamma \cup \overline{\Gamma_{ext}}) S_0 \vdash_w^* \overline{e_b} : \pi' \ ?\ (\varphi', \psi'); S_1 \\ S_2 = \{\pi_i' \stackrel{?}{=} \pi_{i+1}' \mid 0 \le i < |\overline{e_b}|\} \\ S_3 = (\mathsf{MatrixFormula}(P, \overline{\varphi}, \overline{\psi}) S_1 S_2 \stackrel{?}{=} \mathsf{T}) \\ \varphi_r = \bigvee_i (\mathsf{Line}(P_i, \overline{\varphi}, \overline{\psi}) S_1 \wedge \varphi_i') \\ \psi_r = \bigvee_i (\mathsf{Line}(P_i, \overline{\varphi}, \overline{\psi}) S_1 \wedge \psi_i')\end{array}}{\begin{array}{l}\Gamma \vdash_w \mathsf{choose}\ \overline{e_m}\ \{P \Rightarrow \overline{e_b}\} : \\ \qquad (\pi_0' \ ?\ (\varphi_r, \psi_r)) S_2 S_3; S_0 S_1 S_2 S_3\end{array}} \text{(W-Choose-}\star\text{)}$$

Fig. 9. Extension of Declarative Type System and Algorithm W with Choose-★

In both cases, we still require that the proper types π' of all the bodies are equal. However, we now allow different nullity formulas for the various branches. The nullity formulas of the whole choose-construct will be the disjunction of the formulas of the branches. In order to specify *relational nullable types*, we strengthen the result types of the branch i with the information that the corresponding pattern row has matched. This information is encoded as the formula $\mathsf{Line}(P_i, \overline{\varphi}, \overline{\psi})$. A row matches, if each single entry j matches an expression with nullity formula (φ_j, ψ_j), which is encoded in the formula $\mathsf{mayCover}(P_{i,j}, \varphi_j, \psi_j)$. Pattern null can only apply if the expression may be null, i.e., φ holds. Pattern x can only apply if the expression may be non-null, i.e. ψ holds. Finally, pattern $\boxdot$ can only apply if the expression is either null or non-null, i.e. $\varphi \vee \psi$ holds.

$$\mathsf{mayCover}(\mathsf{null}, \varphi, \psi) = \varphi \qquad \mathsf{mayCover}(x, \varphi, \psi) = \psi \qquad \mathsf{mayCover}(\boxdot, \varphi, \psi) = \varphi \vee \psi$$

Row i may apply, if all its entries may cover the corresponding match value:

$$\mathsf{Line}(P_i, \overline{\varphi}, \overline{\psi}) = \bigwedge_j (\mathsf{mayCover}(P_{i,j}, \varphi_j, \psi_j))$$

4.7.2 Soundness. We would like to state soundness, i.e. the progress and preservation theorems, for the declarative type system extended with the (T-Choose-★) rule. Progress is straightforward, but it turns out that preservation does not hold. To understand why, consider the following program fragment which is typeable with (T-Choose-★) but not with (T-Choose):

```
1  let k = x -> y -> x;
2  let u = choose* (if (false) null else 123) {
3      case null => null
4      case x    => 456 : Int ? (false, true) // type ascription
5  }
6  k(u)
```

Here the proper type of $k(u)$ is $\alpha \to \mathsf{Int32}\ ?\ (\mathsf{T}, \mathsf{T})$ for some α. This type arises because the type of the if-then-else expression admits both null and non-null values and, consequently, the type system cannot exclude either branch of the choose expression and thus u may be null or non-null.

Let us now perform two reduction steps: First, the if-then-else expression is reduced to the value 123. Next, the choose expression is reduced to the value 456 (which is given the explicit type Int ? (F, T)) with a type ascription[5]. After these two reduction steps, the proper type of $k(u)$ is $\alpha \to \mathsf{Int32}\ ?\ (\mathsf{F}, \mathsf{T})$. But the two proper types: $\alpha \to \mathsf{Int32}\ ?\ (\mathsf{T}, \mathsf{T})$ and $\alpha \to \mathsf{Int32}\ ?\ (\mathsf{F}, \mathsf{T})$ are

[5]The use of a type ascription is immaterial, the exact same typing can be obtained with an even more elaborate program.

not equal — not even modulo Boolean equivalence. Consequently, in the declarative type system extended with the (T-Choose-★) rule, preservation does not hold in its original formulation.

However, all is not lost. We conjecture that preservation still holds if the type system is extended with a notion of "Boolean sub-typing". Informally, $\tau <: \tau'$ if τ is structurally equivalent to τ' and whenever there are two Boolean formulas φ and φ' then $\varphi \Rightarrow \varphi'$ (or $\varphi' \Rightarrow \varphi$ in contravariant positions). With this notion, we believe:

Conjecture 4.18 (Preservation-★). *If* $\Gamma \vdash_d e : \tau$ *and* $e \longrightarrow e'$ *then* $\Gamma \vdash_d e' : \tau'$ *for some* $\tau' <: \tau$.

However, Boolean sub-typing, with the corresponding extension of principal type inference, deserves its own separate treatment, which is left as future work.

5 IMPLEMENTATION

We have implemented the $\lambda^{\text{rel}}_{\text{null}}$ calculus in two systems: As a minimal proof-of-concept calculus and as an extension of the Flix programming language.

5.1 The Flix Programming Language

Flix is a functional, imperative, and logic programming language that supports algebraic data types, pattern matching, parametric polymorphism, type classes, currying, higher-order functions, polymorphic effects, extensible records, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination [Madsen and Lhoták 2018, 2020; Madsen and van de Pol 2020; Madsen et al. 2016]. The Flix compiler project, including the standard library and tests, is approximately 135,000 lines of Flix and Scala code.

5.2 Minimal Proof-of-Concept Implementation

We have implemented a minimal proof-of-concept version of the $\lambda^{\text{rel}}_{\text{null}}$ calculus as a Flix program. The proof-of-concept includes a complete implementation of the type inference system based on Algorithm W. The implementation closely follows the formal type inference rules (Section 4.6).

5.3 Flix Extension with Relational Nullable Types

Flix, like Haskell, OCaml, and Rust does not have a null value, but instead has the Option data type. However, as discussed in Section 2, relational nullable types are also relevant for programming languages with Option types. For Flix, we introduce a new data type called Choice[t, a, p] which is parameterized by a type t, and two booleans a and p. The Choice data type has two constructors:

$$\texttt{Absent} : \texttt{Choice}[\texttt{t}, \texttt{true}, \texttt{p}] \qquad \texttt{Present} : \texttt{Choice}[\texttt{t}, \texttt{a}, \texttt{true}]$$

Note how the type of Absent and Present closely mirrors that of null and non-null values.

To work with Choice values, we introduce *two* pattern matching constructs: choose and choose* corresponding to the type rules (T-Choose) and (T-Choose-★). A choose expression destructs a sequence of choice values into a value of any type, whereas a choose* expression destructs a sequence of choice values into a value that must be of the Choice type. Other than these two changes, everything else follows the formal system.

In terms of implementation effort, the extension required less than 3,000 lines of code, mostly related to computing saturated pattern match matrices and type inference for the choose and choose* expressions. We also extended the standard library with several combinator functions.

Flix, with our extension, is open source, ready for use, and freely available at:

https://flix.dev/ and https://github.com/flix/flix/

5.3.1 Higher-Order Combinators. The extended Flix type system naturally supports combinator functions such as filter, map, flatMap, and flatten.[6]

For example, the definition of map is straightforward:

```
def map(f: s -> t, c: Choice[s, a, p]): Choice[t, a, p] = choose* c {
    case Absent     => Absent
    case Present(x) => Present(f(x))
}
```

The definition of flatMap is also straightforward, but gives rise to a more interesting type:

```
def flatMap(f: s -> Choice[t, a2, p2], c: Choice[s, a1, p1]):
            Choice[t, a1 or (p1 and a2), p1 and p2] = choose* c {
    case Absent     => Absent
    case Present(v) => f(v)
}
```

Let us break it down. Let us consider when the result can be Absent. This can happen for two reasons: If the argument c is already Absent (i.e. a_1 holds) or if f returns Absent (i.e. a_2 holds) *and* the argument c was Present (i.e. p_1 holds). This gives us the constraint $a_1 \vee (p_1 \wedge a_2)$. Now, let us consider when the result can be Present. This can only happen if the argument c is already Present (i.e. p_1 holds) *and* f returns a Present value (i.e. p_2 holds). This gives us the constraint $p_1 \wedge p_2$.

The definition of filter is also straightforward:

```
def filter(f: t -> Bool, c: Choice[t, a, p]): Choice[t, a or p, p] = choose* c {
    case Absent     => Absent
    case Present(v) => if (f(v)) Present(v) else Absent
}
```

Let us break it down again. First, observe that we have no information about whether f returns true or false, consequently in the Present case the value can be Absent or Present. Let us now consider when the result can be Absent. This can happen for two reasons: If the argument c is already Absent (i.e. a holds) or if c is Present (i.e. p holds). This gives us the constraint $a \vee p$. Let us consider when the result can be Present. This can only happen if c is Present (i.e. if p holds).

We can define a withDefault function that, given two arguments, returns the first argument if it is Present and otherwise returns the second argument:

```
def withDefault(c1: Choice[s, a1, p1], c2: Choice[s, a2, p2]):
                Choice[s, a1 and a2, p1 or (a1 and p2)] = choose* c1 {
        case Absent     => c2
        case Present(v) => Present(v)
    }
```

Let us break it down again. Let us consider when the result can be Absent. This can only happen if c_1 is Absent (i.e. a_1 holds) and if c_2 is also Absent (i.e. a_2 holds). Thus the result can be Absent when $a_1 \wedge a_2$ holds. Let us now consider when the result can be Present. This can happen if c_1 is Present (i.e. p_1 holds) or if c_1 is Absent and c_2 is Present (i.e. a_1 holds and p_2 holds). Thus the result can be Present when $p_1 \vee (a_1 \wedge p_2)$ holds.

We close with an interesting, if perhaps useless, combinator:

```
def invert(c: Choice[s, a, p], v: s): Choice[s, p, a] = choose* c {
    case Absent     => Present(v)
    case Present(_) => Absent
}
```

For example, `invert(invert(invert(Absent, 1), 2), 3) == 3`. To the best of our knowledge, no existing nullable type system can express such a combinator.

[6]For the sake of exposition, we have slightly simplified some of the type schemes. The type schemes are still correct, but they are less general than the most general type scheme permitted by the relational nullable type system.

5.3.2 Type Errors. An important practical concern is how to report type errors to the programmer. In particular, Hindley-Milner style type inference is known to produce type errors that are difficult for programmers to understand [Chitil 2001]. The use of Boolean formulas and unification only exacerbates this problem.

In the current Flix implementation, we report a type error when two Boolean formulas φ and ψ fail to unify. We have found that such type errors are difficult to untangle. In future work, we would like to explore a better approach: in case of a unification failure, we want to construct a sequence of nullable or non-nullable values that are not matched by any row in the pattern match. We think that such type errors could be more helpful than ordinary unification errors.

5.4 Evaluation: Recasting the Program Fragments in Flix

We have used the Flix implementation to experiment with reformulations of the program fragments from the preliminary study (Section 2). We believe that all program fragments can be suitably reformulated in Flix using the relational nullable type system. To illustrate, we show three examples from the study recast in Flix:

pmd/pmd (Recast in Flix)

```
let mkDbType = (subProtocol, subnamePrefix) ->
    choose (subProtocol, subnamePrefix) {
        case (Absent, Present(prefix))         => /* omitted */
        case (Present(proto), Absent)          => /* omitted */
        case (Present(proto), Present(prefix)) => /* omitted */
    }
```

which captures that subProtocol and subnamePrefix cannot both be Absent.

openssl/openssl (Recast in Flix)

```
let ossl_ffc_params_FIPS186_4_gen_verify = params ->
    choose (params.p, params.q) {
        case (Absent, Absent)           => /* omitted */
        case (Present(p1), Present(p2)) => /* omitted */
    }
```

which captures that the p and q fields of the record params must both be Absent or Present.

snowplow/iglu (Recast in Flix)

```
let credentialsProvider = (accessKeyId, secretAccessKey, profile) ->
    choose (accessKeyId, secretAccessKey, profile) {
        case (Present(k), Present(s), Absent) => BasicAWSCredentials(k, s)
        case (Absent, Absent, Present(p))     => ProfileCredentialsProvider(p)
        case (Absent, Absent, Absent)         => DefaultAWSCredentialsProvider()
    }
```

which captures that either both accessKeyId and secretAccessKey must be provided, or profile must be provided, or none of the arguments must be provided.

All three examples illustrate the need for relational nullability and the need for may and must information, i.e., whether an expression may or must evaluate to null and whether it may or must evaluate to a non-null value.

6 RELATED WORK

6.1 Boolean Unification

Early work on Boolean unification and the successive variable elimination algorithm goes back to George Boole himself [Boole 1847]. Later work include that of Rudeanu [1974] and Buttner and Simonis [1987]. An accessible introduction to Boolean unification is provided by Martin and Nipkow [1989]. Boudet et al. [1989] study unification in Boolean rings and in combination with other theories. A study of the computational complexity of Boolean unification is provided by Baader [1998]. An alternative algorithm for Boolean unification is proposed by Löwenheim [1908].

6.2 Type Systems and Type Inference

The Damas–Hindley–Milner type system was first described by Hindley [1969] and Milner [1978]. Later, Damas [1984] studied the formal foundations of the system. The Damas–Hindley–Milner type system has been used as the theoretical foundation for several real-world functional programming languages, including Haskell, OCaml, and Standard ML. Many extensions have been proposed, notably type classes [Wadler and Blott 1989], qualified-types [Jones 2003], and region-based memory management [Tofte and Talpin 1997].

In the early years, an important question was how to prove correctness of Damas–Hindley–Milner-style type systems and similar systems. In an influential work, Wright and Felleisen [1994] describes a "syntactic approach" laying the foundations for the modern formulations of soundness in terms of the progress and preservation theorems.

6.3 Type Inference with Boolean Unification

In [Madsen and van de Pol 2020], we propose a polymorphic type and effect system based on Hindley-Milner and Boolean unification. In that system, every expression is associated with a Boolean formula that captures when the expression is *pure* (i.e. has no side-effect). The type and effect system supports effect polymorphism: The purity of a higher-order function may depend on the purity of its function arguments. The authors implement the type and effect system in the Flix programming language. Experimental results on the Flix standard library and a number of Flix applications suggest that the performance cost of Boolean unification during type and effect inference is acceptable.

While our work in [Madsen and van de Pol 2020] and this paper are both based on the Hindley-Milner type system extended with Boolean constraints, there are significant differences: First, the subject areas are very different: effect polymorphism vs. relational nullability. In [Madsen and van de Pol 2020], the goal is to capture when an expression is pure, whereas here our goal is to capture the nullability of an expression in relation to other related expressions and to express the exhaustiveness condition of the choose construct in the type system. Second, in [Madsen and van de Pol 2020], the Boolean formulas have no impact on the progress theorem (ill-effected expressions cannot get stuck). Third, our current type system relies on both may and must information.

6.4 Nullable Type Systems

There is a rich research literature on nullable type systems. We aim to provide a broad outline. Most of the work below is orthogonal to the idea of relational nullability.

6.4.1 Studies. Chalin and James [2007] conduct at study of five large open source projects. The study shows that on average 3/4 of all references could be annotated as non-null. Consequently, the authors proposed that non-nullable types should be the default. In relation to our work, null or non-null annotations are never required since our type system supports type inference; nullness is simply inferred from the source code.

6.4.2 Retrofitted Nullable Type Systems. Fähndrich and Leino [2003] propose a sound way to retrofit C# and Java with nullable type systems. Nieto et al. [2020b] present an extension of the Scala type system that makes nullable types explicit. The type system has been implemented in the Dotty compiler for the Scala 3.0 language. Our "retrofit" of Flix was straightforward since we simply introduced a new data type called Choice. An interesting question for future work would be to explore how to retrofit relational nullable type system onto other programming languages.

6.4.3 Gradual Nullable Type Systems. Brotherston et al. [2017] present Granullar, a gradual pluggable type system for Java. The type system ensures that `NullPointerExceptions` cannot occur within checked code; only on the boundary between checked and unchecked code. Nieto et al. [2020a] present a blame calculus for a gradually typed language with null.

6.4.4 Object Initialization. Fähndrich and Xia [2007] present a type system with *delayed types* that allows reasoning about when an object becomes fully initialized. Qi and Myers [2009] present a type system based on type state that prevents reading from uninitialized fields. The calculus side-steps the issue of null by relying on a flow-sensitive type system to track the set of uninitialized fields. Summers and Müller [2011] presents a type system to track object initialization. The type system ensures that objects under construction, whose invariants may not yet be fully satisfied, cannot escape except under controlled circumstances. While beyond the scope of this paper, relational nullability seems related to the problem of object initialization in the sense that whether one field is initialized may depend on whether another field is initialized. It would be interesting to explore this connection in future work.

6.4.5 Static Analysis. Spoto [2008] presents a static analysis to detect `NullPointerExceptions` in Java bytecode. The static analysis uses abstract domains implemented efficiently using binary decision diagrams (BDDs). While their work and our work both use Boolean formulas, the two approaches are very different. For future work, it would be interesting to explore whether the idea of relational nullability can be formulated as a (relational) abstract domain. Male et al. [2008] present an extension of the JVM bytecode verifier with support for nullable types. Hubert et al. [2008] present a constraint-based static analysis to infer non-null annotations. Banerjee et al. [2019] presents NullAway, a tool to find `NullPointerExceptions` . NullAway, unlike other similar tools, does not aim for soundness, but rather for a reduction of spurious warnings reported by the tool. This reduces the annotation burden for the programmer. Our type system supports type inference, so annotations are never required.

6.4.6 Unsoundness. Amin and Tate [2016] demonstrate unsoundness of Java's and Scala's type systems. The unsoundness manifests itself as a combination of wildcards, sub-typing, and nulls. Intriguingly, each individual feature is believed to be sound, but it is their combination that breaks soundness of the overall type systems.

6.5 More Powerful Type Systems

6.5.1 Logical Types. Tobin-Hochstadt and Felleisen [2010] present a type system with logic predicates for a typed Scheme. The key idea is to use control flow predicates as propositional logic formulas as part of the type of an expression. For example, in the Scheme expression `(if (number? x) e1 e2)`, the e_1 expression is typed with the knowledge that x satisfies the `number?` predicate, whereas e_2 is typed with the knowledge that x does not satisfy the predicate. Concretely, in their system, the typing judgement is of the form $\Gamma \vdash \varphi_+ \mid \varphi_- \; ; \; o$ which states that if e evaluates to a "truthy" value then the φ_+ formula holds (and otherwise the φ_- formula holds).

The work of Tobin-Hochstadt and Felleisen is closely related to ours, but with several differences. First, we introduce the notion of *relational nullability* and we compute a precise exhaustiveness

condition. Second, our type system supports parametric polymorphism whereas their type system supports sub-typing (neither type system supports both). Third, our type system supports full type inference (i.e. if a program is typeable w.r.t. the declarative system then Algorithm W will provide the typing), whereas their type system supports a limited form of local type inference [Pierce and Turner 2000].

6.5.2 Refinement Types. Our relational nullable type system can be seen as a special case of a refinement type system [Rondon et al. 2008; Vazou et al. 2014], which has typing judgements of the form $\Gamma \vdash e : \{\nu : \tau \mid \varphi\}$. Here τ is the type of the expression e subject to the logic formula φ, which may refer to the value of e as ν. For example, the refinement type $\{\nu : \mathsf{Int} \mid 1 \leq \nu \leq 99\}$ captures all integers between 1 and 99 (inclusively). With this in mind, we can view the relational nullable type $\pi \ ? \ (\varphi, \psi)$ as the refinement type $\{\nu : \pi \mid v = \mathsf{null} \Rightarrow \varphi \wedge v \neq \mathsf{null} \Rightarrow \psi\}$.

Refinement type systems are often very powerful depending on the types of formulas that are admitted. If such formulas contain undecidable fragments of logic then type checking may even be undecidable [Rondon et al. 2008; Vazou et al. 2014]. In practice, refinement type systems often rely on SMT solvers to implement type checking. In comparison, our proposed type system is in a sweet-spot: type checking and even type inference is decidable due to existence of most general unifiers for Boolean formulas.

In summary, the main novelty of our work is the support for *relational nullable types* that are expressed as *Boolean constraints* which can be *fully inferred* with a *novel extension of Algorithm W*.

7 CONCLUSION

We have presented a simple, practical, and expressive relational nullable type system. The type system extends the Hindley-Milner type system with Boolean constraints, supports parametric polymorphism, and has principal types modulo Boolean equivalence. An important property of the type system is that it supports full type inference. The key insight is that the exhaustiveness condition of a relational pattern match can be translated into a Boolean formula which can be inferred because Boolean formulas have most general unifiers. Thus we can infer types with an extension of Algorithm W.

We have conducted a preliminary study on the use of relational nullability in open source projects. Three observations from the study were: (i) programmers use programming patterns where the nullability of one expression depends on the nullability of other related expressions, (ii) reasoning about such patterns requires both *may* and *must* information, and (iii) in lieu of type system support, programmers rely on run-time checks to enforce relational nullability invariants. The presented relational nullable type system regains type safety for such programming patterns while supporting full type inference.

REFERENCES

Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems are Unsound: the Existential Crisis of Null Pointers. *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2016).

Franz Baader. 1998. On the Complexity of Boolean Unification. *Inform. Process. Lett.* (1998).

Subarno Banerjee, Lazaro Clapp, and Manu Sridharan. 2019. Nullaway: Practical Type-based Null Safety for Java. In *Proc. Joint Symposium on European Software Engineering and the Foundations of Software Engineering (ESEC/FSE)*.

George Boole. 1847. *The mathematical analysis of logic.*

Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauß. 1989. Unification in Boolean Rings and Abelian Groups. *Journal of Symbolic Computation* (1989).

Dan Brotherston, Werner Dietl, and Ondřej Lhoták. 2017. Granullar: Gradual Nullable Types for Java. In *Proc. International Conference on Compiler Construction (CC)*.

Wolfram Buttner and Helmut Simonis. 1987. Embedding Boolean Expressions into Logic Programming. *Journal of Symbolic Computation* (1987).

Patrice Chalin and Perry R James. 2007. Non-Null References by Default in Java: Alleviating the Nullity Annotation Burden. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*.

Olaf Chitil. 2001. Compositional explanation of types and algorithmic debugging of type errors. In *Proc. International Conference on Functional Programming (ICFP)*.

Luis Damas. 1984. *Type Assignment in Programming Languages*. Ph. D. Dissertation. The University of Edinburgh.

Manuel Fähndrich and K Rustan M Leino. 2003. Declaring and Checking Non-Null Types in an Object-Oriented Language. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Manuel Fähndrich and Songtao Xia. 2007. Establishing Object Invariants with Delayed Types. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Roger Hindley. 1969. The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society (AMS)* (1969).

Laurent Hubert, Thomas Jensen, and David Pichardie. 2008. Semantic Foundations and Inference of Non-Null Annotations. In *International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*.

Mark P Jones. 2003. *Qualified Types: Theory and Practice*. Cambridge University Press.

Leopold Löwenheim. 1908. *Über das Auflösungsproblem im logischen Klassenkalkul*.

Magnus Madsen and Ondřej Lhoták. 2018. Safe and Sound Program Analysis with Flix. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*.

Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the Masses: Programming with first-class Datalog Constraints. *Proc. of the ACM on Programming Languages* 4, OOPSLA (2020).

Magnus Madsen and Jaco van de Pol. 2020. Polymorphic Types and Effects with Boolean Unification. *Proc. of the ACM on Programming Languages* 4, OOPSLA (2020).

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proc. Programming Language Design and Implementation (PLDI)*.

Chris Male, David J Pearce, Alex Potanin, and Constantine Dymnikov. 2008. Java Bytecode Verification for NonNull Types. In *Proc. International Conference on Compiler Construction (CC)*.

Alberto Martelli and Ugo Montanari. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1982).

Urusula Martin and Tobias Nipkow. 1989. Boolean Unification - The Story So Far. *Journal of Symbolic Computation* (1989).

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* (1978).

Abel Nieto, Marianna Rapoport, Gregor Richards, and Ondřej Lhoták. 2020a. Blame for Null. In *Proc. European Conference on Object-Oriented Programming (ECOOP 2020)*.

Abel Nieto, Yaoyu Zhao, Ondřej Lhoták, Angela Chang, and Justin Pu. 2020b. Scala with Explicit Nulls. In *Proc. European Conference on Object-Oriented Programming (ECOOP 2020)*.

Benjamin C Pierce and David N Turner. 2000. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2000).

Xin Qi and Andrew C Myers. 2009. Masked Types for Sound Object Initialization. In *Proc. Principles of Programming Languages (POPL)*.

Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proc. Programming Language Design and Implementation (PLDI)*.

Sergiu Rudeanu. 1974. *Boolean Functions and Equations*.

Fausto Spoto. 2008. Nullness Analysis in Boolean Form. In *Proc. International Conference on Software Engineering and Formal Methods (SEFM)*.

Alexander J Summers and Peter Müller. 2011. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical types for untyped languages. In *Proc. International Conference on Functional Programming (ICFP)*.

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based Memory Management. *Information and Computation* (1997).

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proc. International Conference on Functional Programming (ICFP)*.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proc. Symposium on Principles of Programming Languages (POPL)*.

Andrew K Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* (1994).