# Completing Programs with Relational Types

**ACM Reference Format:**
. 2023. Completing Programs with Relational Types. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

### 1.1 Context

**TODO: blah blah blah** - attribute grammars/rules - bidirectional typing - horn clauses - predicate abstraction

### 1.2 Gap

**TODO: blah blah blah**

### 1.3 Innovation

This work introduces a technique for guiding the completion of untyped programs by inferring the expectation of next part of the program from the portion completed thus far. The expectation may represent the next terminal, symbol, or nonterminal necessary to construct a correct program. If the expectation is an expression nonterminal, then an expected type is also generated. The expected type represents an upper bound on the value of the next portion to be constructed. The types are expressive enough to represent relations between values rather than merely the shape of values. Thus, a *relational type* can be viewed as a structured representation of a predicate defined by horn clauses.

This work offers the first technique for inferring relations from incomplete untyped programs. The the object under analysis is a incomplete program constructed by a human or a human-like agent in concrete syntax. Thus, the analysis must be able to operate without a abstract syntax tree. Instead, we implement the analysis as *attribute rules*, interwoven with top-down parsing rules to the form an *attribute grammar*. Due to the left-recursion problem in top-down parsing, the parser consolidates information in right-associative manner. Thus, one difference of analysis in a top-down parsing stack vs an recursion over an AST is re-associating program information to the left on the fly. One example of this would be the association of a chain of record selections, e.g.

$$x.uno.dos.tres$$

$$x(.uno(.dos(.tres)))$$

$$((x.uno).dos).tres$$

The automatic analysis of incomplete programs generating expected relations during top-down parsing is novel. Our technique leverages bidirectional typing along with intersection and union types. Union types are necessary in the absence of intrinsic types or a specified universe of logical qualifiers/predicates to choose from. Our technique's combination of bidirectional typing with union and intersection can be viewed as analogous to the duality method of solving horn clauses.

We have designed a structured language of types in order to express relations and shapes succinctly. Although horn clauses would be sufficient to express the same semantics, relational types make type annotations easier to write and inferred types easier to read.

The analysis handles the semantics at two levels: typing and subtyping. Typing is used when the value of an expression is syntactically represented by an expression. When types or expressions cannot be decomposed easily, more advanced reasoning techniques are necessary, and the problem is lifted to subtyping. To handle subtyping in a rich language that includes relations, our technique encodes relational types into horn clauses and leverages the capabilities of established solvers.

**TODO: blah blah blah**

**TODO: more detail on innovation but still at a high level**

## 2 Overview

**TODO: blah blah blah**

## 3 Language

The programming language is pure and functional. Its syntax and dynamic semantics are fairly standard. The main departure from tradition is that its function and application rules subsume pattern matching. This departure enables a more direct correspondence between the structures of programs and their types, but it is not a necessary condition. The syntax is given in Fig. ??. It includes functions with pattern matching, records, a fixed point combinator, let binding, tags for discriminating cases, and application. A function consists of a sequence of paths, where each path maps a pattern to an expression. A record consists of a sequence of fields, where each field maps a unique label to an expression. The type language


*Conference'17, July 2017, Washington, DC, USA*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

includes tag types, field types, implications, unions, intersections, inductions, existentials, universals, top, and bottom. The existential type consists of multiple bound variables, a payload containing the bound variables, and a subtyping constraint over the bound variables. If the bound variables aren't indicated, then all variables in the payload are assumed to be bound variables. If the subtyping constraint isn't indicated, then it is assumed to be a tautology, such as (`unit<:unit`). The universal type consists of a bound variable, the variable's upper bound, and a payload. If the upper bound is not indicated, it is assumed to be the top type (`top`). The typing semantics rely on a typing environment for keeping track of typings of term variables. The subtyping semantics rely on subtyping environment for keeping track of and constraints on type variables. Note that the syntax of the subtyping environment allows an upper bound constraint over a type rather than merely a type variable to allow for relational constraints.

**TODO: update tag syntax: `cons;cons;e:cons//cons//T nil;():nil//unit`**

### 3.1 Types

***Quantified types.*** A universal type is a second order type with universally quantified type variables. An existential type is a second order type with existentially quantified type variables.

System F-style. Parameterization of types indexed by types (i.e. second order).

**TODO: mention somwhere that the second order quantification serves two distinct purposes; 1. polymorphism as in System-F. 2. refinement as in first-order quantification of liquid types. Relational types is able to leverage second-order quantification for refinement, eschewing the first-order quantification used in other systems.**

***Combination types.*** One of the advantages of untyped programs is that they may be written in a flexible manner. Subtyping is necessary safely reflect the flexibility of compositions in programs, without too many false failures. Another main advantage of untyped programs is that users don't have to provide type specifications. Thus, a general way of constructing types from compositions encountered in the the program is necessary. Some compositions indicate that a type should strengthen, and some compositions indicate that a type should weaken. To this end, the type language uses intersection and union combinators, whose semantics are degenerate versions of those in set-theory.

For instance, when inferring the type of a function, the system's goal is to infer the weakest valid parameter type and the strongest valid return type for a function definition. It strengthens the parameter type with intersection and weakens the return type with union according to the function body, to arrive at a valid type for the function.

By contrast, the liquid type language relies on the less flexible tagged unions of ML datatypes, which is sufficient in its setting since those types are specified by the user. Likewise, it does not rely on union to weaken to a valid return type. Instead, it weakens to the strongest valid return type by dropping conjunctions from the return type's qualifiers until a valid return type is found.

***Inductive types.*** Similar to ML datatypes.

***Qualified types.*** In addition to expressing the shapes of terms, the system should be able express relations between terms, such as "a list has the length of some natural number". Rather than using a distinct syntax for relational predicates, the type language treats relations as just another type thereby reusing machinery already available for types, such as existential types, union types, and inductive types. Since parametric types are second order, constraining relations requires subtyping. Thus, parametric types are extended with constraints in the form of subtyping.

### 3.2 Expressions

**TODO: blah blah blah**

### 3.3 Typing

The typing is given in Fig. ??. Most of the rules are fairly standard. The rule for function typing is a bit special in that it treats a function as a sequence of paths whose type is an intersection of implications, rather than having a separate pattern matching rule. Likewise, the type of a record is an intersection of field types. The let-binding rule allows for prenex polymorphism by generalizing via subtyping.

### 3.4 Subtyping

The subtyping is given in Fig. ??. Some of the rules are fairly standard, including implication, the union rules, and intersection rules. Note that in addition to left and right rules, union and intersection each have rules for interacting with implication's antecedent and consequent, respectively. The constraint rule checks that a subtyping relation exists as a constraint in the subtyping environment. The right induction rule is standard and simply unrolls the induction. The left induction rule relies on the induction principle to construct an inductive constraint hypothesis. The field and tag rules simply check that the labels match and subtyping holds for their constituent types. The existential rules are quite special, as they involve a subtyping constraint as part of a second-order comprehension. The left existential rule checks that subtyping holds for all variations of the payload where the subtyping constraint holds. The right existential rule checks that subtyping holds for some variation of the payload where the constraint holds. The left universal rule checks that subtyping holds for some variation of the payload consistent with the variable's upper bound. The right

universal rule checks that subtyping holds for all variations of the payload consistent with the variable's upper bound.

# 4 Analysis

The analysis consists of two main parts. The top level is type inference, which corresponds to typing and generates a type for an expression. When type inference encounters constraints that its types must adhere to, it calls unification to solve these constraints. Note that since the types are expressive enough to represent constraints, an alternative approach of generating constraints and solving them in separate stages could also be designed using the same structures. Additional structures for the analysis are given in Fig. **??**. Inference generates a solution set $T$, which contains triples, each with a type variable set, a subtyping environment, and a type. Unification generates a solution set $C$, which contains subtyping environments.

## 4.1 Type Inference

**TODO: blah blah blah**

## 4.2 Subtype Solving

**TODO: type unification TODO: econding to/decoding from horn clauses**

# 5 Experiments

**TODO: develop 12 tree/list experiments**

# 6 Related work

**TODO: blah blah blah**