Completing Programs with Relational Types

ACM Reference Format:
. 2024. Completing Programs with Relational Types. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Context

Much research has been devoted to automatically verifying the correctness of programs with simple types. One of the main tricks for handling the challenge of correctness has been to design languages to make verifying correctness easier. Languages, such as Java and ML, are *intrinsically typed*, requiring nearly all terms to be associated with some type specified by the user. The clever design of ML allows annotations to be fairly sparse by having types specified at constructor definitions and relying on type inference elsewhere.

Despite the advantages of intrinsically typed languages, untyped or *extrinsically typed* languages, such as Javascript/-Typescript and Python, have surged in popularity. Untyped languages place less initial burden on the programmer to define the upper bounds of various parts of their program and allow reusing code in more flexible ways.

The more expressive the static properties the stronger the notion of correctness. Simple types found in ML or Java cannot express relations between values. To this end, researchers have extended the simple types found in Java and ML into *refinement types*, *predicate subtyping*, and *dependent types*. Correctness of a program can also be lifted into a purely logical form, such as *horn clauses*.

Refinement types offer greater precision than simple types, but still rely on intrinsic type specifications. Dependent types can express detailed relations, but may require users to provide proofs along with detailed annotations. Predicate subtyping offers some of the expressivity of dependent types, but with the automatic subtyping of refinement types. All of these techniques are based on intrinsic typing and therefore require users to provide additional specifications beyond the runtime behavior of their programs.

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

Another challenge is guiding the construction of correct programs. Subparts of a program can be guided locally with *bidirectional typing*, which decomposes types and propagates expected types down the syntax tree to guide subparts of the program. Another issue in incomplete programs are the scenarios where the abstract syntax tree cannot be built, due to an incomplete expression represented concretely. In such a case, the analysis can be combined with a top-down parser, forming an *attribute grammar*.

1.2 Gap

To the best of our knowledge, the problem of guiding the completion of incomplete and untyped programs with relational information has not yet been tackled.

NSG [] offers a technique for guiding program completion by extracting information from incomplete Java programs to guide the next portion. The analysis for extracting information takes the form of *attribute rules* mixed in with parsing rules to form an *attribute grammar*. However, the information extracted is quite simple and directly available in the syntax of programs due to the intrinsic typing and required type annotations of Java. There is no guidance based on relations.

Synquid [] tackles the problem of synthesis from a formal specification, rather than completion from the left. Since Synquid can construct its program from scratch, its analysis works directly on an AST and it can avoid using a complex attribute grammar.

Synquid's guidance can express relations but it requires an intrinsically typed language with a universe of logical qualifiers provided, which it leverages to perform a combination of *Hindley-Milner type inference* and *predicate abstraction*.

1.3 Innovation

This work introduces the first technique for guiding the completion of untyped programs with relational information. It is novel in inferring relations without intrinsic types or declared universes of relational qualifiers in conjunction with performing such an analysis during top-down parsing.

The analysis produces a description of the expectation for the next part of the program. The expectation may represent a terminal, a symbol, or a 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 possible values 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.

In addition to intersection types, seen in refinement type systems, our system also requires union types due to the absence of intrinsic types or a specified universe of logical qualifiers/predicates to choose from. That is, we must be able to start with a strong type and weaken as more use cases are uncovered in the program. In contrast, intrinsic typing explicitly declares a reasonable weakest type, which a refinement type system can strengthen via intersections (or conjunctions).

Additionally, our technique leverages bidirectional typing. The inheriting attribute rules include the downward propagation of types. The synthesizing attribute rules include the upward propagation of types.

For introduction rules, our technique propagates types both downward and upward. The original bidirectional typing and Synquid's variation *roundtrip typing* only propagate downward for introduction rules. Our problem requires the additional upward propagation because the specification may need to be extracted from the program, whereas previous problems assumed a provided specification (e.g. for a function definition). For elimination rules, our technique propagates types both upward and downward. The original bidirectional typing only propagates upward for elimination rules, but Synquid's variation *roundtrip typing* introduced propagating in both directions for elimination rules. Consider application of a function to an argument. Using downward propagation, type propagation can guide the construction of the applicator's function body, rather than waiting for the body's full construction and only checking the correctness of the body once the result of the application is used somewhere else.

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.

Decomposing types into its constituent parts may be necessary when propagating types upward from elimination rules or when propagating types downward over introduction rules. These type propagation rules call the solver to find a decomposition that satisfies a subtyping. Thus, solving subtyping provides a generic interface that type propagation can use for type decomposition without having to specify the type syntax restrictions directly. In contrast, Synquid's typing rules specify the syntactic forms of types directly. Additionally, Synquid is more limited in the syntactic forms of types that it allows. For example, our system allows the type of a function to be constructed as an intersection of implications, while Synquid requires the function's type to be a single implication (i.e. arrow) type. Our system must be more flexible than Synquid due to the specification being extracted from the program, rather than specified by the user (as is the case in Synquid).

The object under analysis is a incomplete program in concrete syntax constructed by a human or a human-like agent. Thus, the analysis must be able to operate without an abstract syntax tree. Instead, we implement the analysis as *attribute rules*, interwoven with top-down parsing rules to form an *attribute grammar*.

Using attribute rules during parsing adds many complications that would not be present for an analysis over an AST. The analysis must be separated into rules for *inheriting attributes* and rules for *synthesizing attributes*. Additionally, due to the left-recursion problem in top-down parsing, the parser consolidates information in a right-associative manner. Thus, one difference of an analysis over a top-down parsing stack as opposed to recursion over an AST is the re-associating of program information from right to the left on the fly. Consider a chain of record projections:

$$x.uno.dos.tres$$

is parsed as

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

but represents the expression

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

Propagating types downward over a top-down parse-tree results in a varying interpretations of the downwardly propagated type used for local guidance. For syntax that represents complete expressions, e.g.

$$x$$

, the inherited type represents the prescriptive type for that expression. However, for syntax that cannot represent a complete expression, the inherited type must represent something else. For instance, in the parse tree for a chain of projections, a syntactic subtree of projections, such as

$$.dos.tres$$

does not constitute a complete expression. There is no prescriptive type for this syntax. Thus, the inherited type represents the type that

$$.dos$$

is expected to project from (or cut into), e.g. the descriptive type of

$$x.uno$$

.

Our technique's combination of bidirectional typing with union and intersection can be viewed as analogous to the *duality* method of solving horn clauses.

2 Overview

TODO: blah blah blah

3 Language

The language consists of pure expressions and optional type annotations. 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.

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: encoding to/decoding from horn clauses

5 Experiments

TODO: develop 12 tree/list experiments

6 Related work

TODO: blah blah blah