Type Checker Design and Implementation

Purpose

This document describes the design and implementation of the Swift type checker. It is intended for developers who wish to modify, extend, or improve on the type checker, or simply to understand in greater depth how the Swift type system works. Familiarity with the Swift programming language is assumed.

Approach

The Swift language and its type system incorporate a number of popular language features, including object-oriented programming via classes, function and operator overloading, subtyping, and constrained parametric polymorphism. Swift makes extensive use of type inference, allowing one to omit the types of many variables and expressions. For example:

```
func round(_ x: Double) -> Int { /* ... */ }
var pi: Double = 3.14159
var three = round(pi) // 'three' has type 'Int'

func identity<T>(_ x: T) -> T { return x }
var eFloat: Float = -identity(2.71828)  // numeric literal gets type 'Float'
```

Swift's type inference allows type information to flow in two directions. As in most mainstream languages, type information can flow from the leaves of the expression tree (e.g., the expression `pi`, which refers to a double) up to the root (the type of the variable `three`). However, Swift also allows type information to flow from the context (e.g., the fixed type of the variable `eFloat`) at the root of the expression tree down to the leaves (the type of the numeric literal 2.71828). This bi-directional type inference is common in languages that use ML-like type systems, but is not present in mainstream languages like C++, Java, C#, or Objective-C.

Swift implements bi-directional type inference using a constraint-based type checker that is reminiscent of the classical Hindley-Milner type inference algorithm. The use of a constraint system allows a straightforward, general presentation of language semantics that is decoupled from the actual implementation of the solver. It is expected that the constraints themselves will be relatively stable, while the solver will evolve over time to improve performance and diagnostics.

The Swift language contains a number of features not part of the Hindley-Milner type system, including constrained polymorphic types and function overloading, which complicate the presentation and implementation somewhat. On the other hand, Swift limits the scope of type inference to a single expression or statement, for purely practical reasons: we expect that we can provide better performance and vastly better diagnostics when the problem is limited in scope.

Type checking proceeds in three main stages:

Constraint Generation Given an input expression and (possibly) additional contextual information, generate a set of type constraints that describes the relationships among the types of the various subexpressions. The generated constraints may contain unknown types, represented by type variables, which will be determined by the solver.

Constraint Solving Solve the system of constraints by assigning concrete types to each of the type variables in the constraint system. The constraint solver should provide the most specific solution possible among different alternatives.

Solution Application Given the input expression, the set of type constraints generated from that expression, and the set of assignments of concrete types to each of the type variables, produce a well-typed expression that makes all implicit conversions (and other transformations) explicit and resolves all unknown types and overloads. This step cannot fail.

The following sections describe these three stages of type checking, as well as matters of performance and diagnostics.

Constraints

A constraint system consists of a set of type constraints. Each type constraint either places a requirement on a single type (e.g., it is an integer literal type) or relates two types (e.g., one is a subtype of the other). The types described in constraints can be any type in the Swift type system including, e.g., builtin types, tuple types, function types, enum/struct/class types, protocol types, and generic types. Additionally, a type can be a type variable `T` (which are typically numbered, `T0`, `T1`, `T2`, etc., and are introduced as needed). Type variables can be used in place of any other type, e.g., a tuple type `(T0, Int, (T0) -> Int)` involving the type variable `T0`.

There are a number of different kinds of constraints used to describe the Swift type system:

Equality An equality constraint requires two types to be identical. For example, the constraint `T0 == T1` effectively ensures that `T0` and `T1` get the same concrete type binding. There are two different flavors of equality constraints:

- Exact equality constraints, or "binding", written `T0 := X` for some type variable `T0` and type `X`, which requires that `T0` be exactly identical to `X`;
- Equality constraints, written `X == Y` for types `X` and `Y`, which require `X` and `Y` to have the same type, ignoring lvalue types in the process. For example, the constraint `T0 == X` would be satisfied by assigning `T0` the type `X` and by assigning `T0` the type `@lvalue X`.

Subtyping A subtype constraint requires the first type to be equivalent to or a subtype of the second. For example, a class type `Dog` is a subtype of a class

type `Animal` if `Dog` inherits from `Animal` either directly or indirectly. Subtyping constraints are written `X < Y`.

Conversion A conversion constraint requires that the first type be convertible to the second, which includes subtyping and equality. Additionally, it allows a user-defined conversion function to be called. Conversion constraints are written `X <c Y`, read as "`X` can be converted to `Y`."

Construction A construction constraint, written `X <C Y` requires that the second type be a nominal type with a constructor that accepts a value of the first type. For example, the constraint `Int <C String` is satisfiable because `String` has a constructor that accepts an `Int`.

Member A member constraint `X[.name] == Y` specifies that the first type (`X`) have a member (or an overloaded set of members) with the given name, and that the type of that member be bound to the second type (`Y`). There are two flavors of member constraint: value member constraints, which refer to the member in an expression context, and type member constraints, which refer to the member in a type context (and therefore can only refer to types).

Conformance A conformance constraint `X conforms to Y` specifies that the first type (`X`) must conform to the protocol `Y`.

Checked cast A constraint describing a checked cast from the first type to the second, i.e., for `x as T`.

Applicable function An applicable function requires that both types are function types with the same input and output types. It is used when the function type on the left-hand side is being split into its input and output types for function application purposes. Note, that it does not require the type attributes to match.

Overload binding An overload binding constraint binds a type variable by selecting a particular choice from an overload set. Multiple overloads are represented by a disjunction constraint.

Conjunction A constraint that is the conjunction of two or more other constraints. Typically used within a disjunction.

Disjunction A constraint that is the disjunction of two or more constraints. Disjunctions are used to model different decisions that the solver could make, i.e., the sets of overloaded functions from which the solver could choose, or different potential conversions, each of which might resolve in a (different) solution.

Archetype An archetype constraint requires that the constrained type be bound to an archetype. This is a very specific kind of constraint that is only used for calls to operators in protocols.

Class A class constraint requires that the constrained type be bound to a class type.

Self object of protocol An internal-use-only constraint that describes the conformance of a `Self` type to a protocol. It is similar to a conformance constraint but "looser" because it allows a protocol type to be the self object of its own protocol (even when an existential type would not conform to its own protocol).

Dynamic lookup value A constraint that requires that the constrained type be DynamicLookup or an lvalue thereof.

Constraint Generation

The process of constraint generation produces a constraint system that relates the types of the various subexpressions within an expression. Programmatically, constraint generation walks an expression from the leaves up to the root, assigning a type (which often involves type variables) to each subexpression as it goes.

Constraint generation is driven by the syntax of the expression, and each different kind of expression—function application, member access, etc.—generates a specific set of constraints. Here, we enumerate the primary expression kinds in the language and describe the type assigned to the expression and the constraints generated from such as expression. We use `T(a)` to refer to the type assigned to the subexpression `a`. The constraints and types generated from the primary expression kinds are:

Declaration reference An expression that refers to a declaration `x` is assigned the type of a reference to `x`. For example, if `x` is declared as `var x: Int`, the expression `x` is assigned the type `@lvalue Int`. No constraints are generated.

When a name refers to a set of overloaded declarations, the selection of the appropriate declaration is handled by the solver. This particular issue is discussed in the Overloading section. Additionally, when the name refers to a generic function or a generic type, the declaration reference may introduce new type variables; see the Polymorphic Types section for more information.

Member reference A member reference expression `a.b` is assigned the type `T0` for a fresh type variable `T0`. In addition, the expression generates the value member constraint `T(a).b == T0`. Member references may end up resolving to a member of a nominal type or an element of a tuple; in the latter case, the name (`b`) may either be an identifier or a positional argument (e.g., `1`).

Note that resolution of the member constraint can refer to a set of overloaded declarations; this is described further in the Overloading section.

Unresolved member reference An unresolved member reference `.name` refers to a member of a enum type. The enum type is assumed to have a fresh variable type `T0` (since that type can only be known from context), and a value member constraint `T0.name == T1`, for fresh type variable `T1`, captures the fact that it has a member named `name` with some as-yet-unknown type `T1`. The type of the unresolved member reference is `T1`, the type of the member. When the

unresolved member reference is actually a call `.name(x)`, the function application is folded into the constraints generated by the unresolved member reference.

Note that the constraint system above actually has insufficient information to determine the type `T0` without additional contextual information. The Overloading section describes how the overload-selection mechanism is used to resolve this problem.

Function application A function application `a(b)` generates two constraints. First, the applicable function constraint `T0 -> T1 ==Fn T(a)` (for fresh type variables `T0` and `T1`) captures the rvalue-to-lvalue conversion applied on the function (`a`) and decomposes the function type into its argument and result types. Second, the conversion constraint `T(b) <c T0` captures the requirement that the actual argument type (`b`) be convertible to the argument type of the function. Finally, the expression is given the type `T1`, i.e., the result type of the function.

Construction A type construction `A(b)`, where `A` refers to a type, generates a construction constraint `T(b) <C A`, which requires that `A` have a constructor that accepts `b`. The type of the expression is `A`.

Note that construction and function application use the same syntax. Here, the constraint generator performs a shallow analysis of the type of the "function" argument (`A` or `a`, in the exposition above); if it obviously has metatype type, the expression is considered a coercion/construction rather than a function application. This particular area of the language needs more work.

Subscripting A subscript operation `a[b]` is similar to function application. A value member constraint `T(a).subscript == T0 -> T1` treats the subscript as a function from the key type to the value type, represented by fresh type variables `T0` and `T1`, respectively. The constraint `T(b) <c T0` requires the key argument to be convertible to the key type, and the type of the subscript operation is `T1`.

Literals A literal expression, such as `17`, `1.5`, or `"Hello, world!"`, is assigned a fresh type variable `T0`. Additionally, a literal constraint is placed on that type variable depending on the kind of literal, e.g., "`T0` is an integer literal."

Closures A closure is assigned a function type based on the parameters and return type. When a parameter has no specified type or is positional (`$1`, `$2`, etc.), it is assigned a fresh type variable to capture the type. Similarly, if the return type is omitted, it is assigned a fresh type variable.

When the body of the closure is a single expression, that expression participates in the type checking of its enclosing expression directly. Otherwise, the body of the closure is separately type-checked once the type checking of its context has computed a complete function type.

Array allocation An array allocation `new A[s]` is assigned the type `A[]`. The type checker (separately) checks that `T(s)` is an array bound type.

Address of An address-of expression `&a` always returns an `@inout` type. Therefore, it is assigned the type `@inout T0` for a fresh type variable `T0`. The subtyping constraint `@inout T0 < @lvalue T(a)` captures the requirement that input expression be an lvalue of some type.

Ternary operator A ternary operator `x ? y : z` generates a number of constraints. The type `T(x)` must conform to the `LogicValue` protocol to determine which branch is taken. Then, a new type variable `T0` is introduced to capture the result type, and the constraints `T(y) <c T0` and `T(z) <c T0` capture the need for both branches of the ternary operator to convert to a common type.

There are a number of other expression kinds within the language; see the constraint generator for their mapping to constraints.

Overloading Overloading is the process of giving multiple, different definitions to the same name. For example, we might overload a `negate` function to work on both `Int` and `Double` types, e.g.::

func negate(_ x: Int) -> Int { return -x } func negate(_ x: Double) -> Double { return -x }

Given that there are two definitions of `negate`, what is the type of the declaration reference expression `negate`? If one selects the first overload, the type is `(Int) -> Int`; for the second overload, the type is `(Double) -> Double`. However, constraint generation needs to assign some specific type to the expression, so that its parent expressions can refer to that type.

Overloading in the type checker is modeled by introducing a fresh type variable (call it `T0`) for the type of the reference to an overloaded declaration. Then, a disjunction constraint is introduced, in which each term binds that type variable (via an exact equality constraint) to the type produced by one of the overloads in the overload set. In our negate example, the disjunction is `T0 := (Int) -> Int or T0 := (Double) -> Double`. The constraint solver, discussed in the later section on Constraint Solving, explores both possible bindings, and the overloaded reference resolves to whichever binding results in a solution that satisfies all constraints. It is possible that both overloads will result in a solution, in which case the solutions will be ranked based on the rules discussed in the section Comparing Solutions.

Overloading can be introduced both by expressions that refer to sets of overloaded declarations and by member constraints that end up resolving to a set of overloaded declarations. One particularly interesting case is the unresolved member reference, e.g., `.name`. As noted in the prior section, this generates the constraint `T0.name == T1`, where `T0` is a fresh type variable that will be bound to the enum type and `T1` is a fresh type variable that will be bound to the type of the selected member. The issue noted in the prior section is that this constraint does not give the solver enough information to determine `T0` without guesswork. However, we note that the type of an enum member actually has a

regular structure. For example, consider the `Optional` type::

```
enum Optional<T> {
  case none
  case some(T)
}
```

The type of `Optional<T>.none` is `Optional<T>`, while the type of `Optional<T>.some` is `(T) -> Optional<T>`. In fact, the type of an enum element can have one of two forms: it can be `T0`, for an enum element that has no extra data, or it can be `T2 -> T0`, where `T2` is the data associated with the enum element. For the latter case, the actual arguments are parsed as part of the unresolved member reference, so that a function application constraint describes their conversion to the input type `T2`.

Polymorphic Types The Swift language includes generics, a system of constrained parameter polymorphism that enables polymorphic types and functions. For example, one can implement a `min` function as, e.g.,::

```
func min<T : Comparable>(x: T, y: T) -> T {
  if y < x { return y }
  return x
}
```

Here, `T` is a generic parameter that can be replaced with any concrete type, so long as that type conforms to the protocol `Comparable`. The type of `min` is (internally) written as `<T : Comparable> (x: T, y: T) -> T`, which can be read as "for all `T`, where `T` conforms to `Comparable`, the type of the function is `(x: T, y: T) -> T`." Different uses of the `min` function may have different bindings for the generic parameter `T`.

When the constraint generator encounters a reference to a generic function, it immediately replaces each of the generic parameters within the function type with a fresh type variable, introduces constraints on that type variable to match the constraints listed in the generic function, and produces a monomorphic function type based on the newly-generated type variables. For example, the first occurrence of the declaration reference expression `min` would result in a type `(x : T0, y : T0) -> T0`, where `T0` is a fresh type variable, as well as the subtype constraint `T0 < Comparable`, which expresses protocol conformance. The next occurrence of the declaration reference expression `min` would produce the type `(x : T1, y : T1) -> T1`, where `T1` is a fresh type variable (and therefore distinct from `T0`), and so on. This replacement process is referred to as "opening" the generic function type, and is a fairly simple (but effective) way to model the use of polymorphic functions within the constraint system without complicating the solver. Note that this immediate opening of generic function types is only valid because Swift does not support first-class polymorphic functions, e.g., one cannot declare a variable of type `<T> T -> T`.

Uses of generic types are also immediately opened by the constraint solver. For example, consider the following generic dictionary type::

```
class Dictionary<Key : Hashable, Value> {
  // ...
}
```

When the constraint solver encounters the expression `Dictionary()`, it opens up the type `Dictionary`—which has not been provided with any specific generic arguments—to the type `Dictionary<T0, T1>`, for fresh type variables `T0` and `T1`, and introduces the constraint `T0 conforms to Hashable`. This allows the actual key and value types of the dictionary to be determined by the context of the expression. As noted above for first-class polymorphic functions, this immediate opening is valid because an unbound generic type, i.e., one that does not have specified generic arguments, cannot be used except where the generic arguments can be inferred.

Constraint Solving

The primary purpose of the constraint solver is to take a given set of constraints and determine the most specific type binding for each of the type variables in the constraint system. As part of this determination, the constraint solver also resolves overloaded declaration references by selecting one of the overloads.

Solving the constraint systems generated by the Swift language can, in the worst case, require exponential time. Even the classic Hindley-Milner type inference algorithm requires exponential time, and the Swift type system introduces additional complications, especially overload resolution. However, the problem size for any particular expression is still fairly small, and the constraint solver can employ a number of tricks to improve performance. The Performance_ section describes some tricks that have been implemented or are planned, and it is expected that the solver will be extended with additional tricks going forward.

This section will focus on the basic ideas behind the design of the solver, as well as the type rules that it applies.

Simplification The constraint generation process introduces a number of constraints that can be immediately solved, either directly (because the solution is obvious and trivial) or by breaking the constraint down into a number of smaller constraints. This process, referred to as *simplification*, canonicalizes a constraint system for later stages of constraint solving. It is also re-invoked each time the constraint solver makes a guess (at resolving an overload or binding a type variable, for example), because each such guess often leads to other simplifications. When all type variables and overloads have been resolved, simplification terminates the constraint solving process either by detecting a trivial constraint that is not satisfied (hence, this is not a proper solution) or by reducing the set of constraints down to only simple constraints that are trivially satisfied.

The simplification process breaks down constraints into simpler constraints, and each different kind of constraint is handled by different rules based on the Swift type system. The constraints fall into five categories: relational constraints, member constraints, type properties, conjunctions, and disjunctions. Only the first three kinds of constraints have interesting simplification rules, and are discussed in the following sections.

Relational Constraints Relational constraints describe a relationship between two types. This category covers the equality, subtyping, and conversion constraints, and provides the most common simplifications. The simplification of relationship constraints proceeds by comparing the structure of the two types and applying the typing rules of the Swift language to generate additional constraints. For example, if the constraint is a conversion constraint::

```
A -> B <c C -> D
```

then both types are function types, and we can break down this constraint into two smaller constraints `C < A` and `B < D` by applying the conversion rule for function types. Similarly, one can destroy all of the various type constructors—tuple types, generic type specializations, lvalue types, etc.—to produce simpler requirements, based on the type rules of the language 1.

Relational constraints involving a type variable on one or both sides generally cannot be solved directly. Rather, these constraints inform the solving process later by providing possible type bindings, described in the Type Variable Bindings section. The exception is an equality constraint between two type variables, e.g., `T0 == T1`. These constraints are simplified by unifying the equivalence classes of `T0` and `T1` (using a basic union-find algorithm), such that the solver need only determine a binding for one of the type variables (and the other gets the same binding).

Member Constraints Member constraints specify that a certain type has a member of a given name and provide a binding for the type of that member. A member constraint `A.member == B` can be simplified when the type of `A` is determined to be a nominal or tuple type, in which case name lookup can resolve the member name to an actual declaration. That declaration has some type `C`, so the member constraint is simplified to the exact equality constraint `B := C`.

The member name may refer to a set of overloaded declarations. In this case, the type `C` is a fresh type variable (call it `T0`). A disjunction constraint is introduced, each term of which new overload set binds a different declaration's type to `T0`, as described in the section on Overloading_.

The kind of member constraint—type or value—also affects the declaration type `C`. A type constraint can only refer to member types, and `C` will be the declared type of the named member. A value constraint, on the other hand, can refer to either a type or a value, and `C` is the type of a reference to that entity. For a reference to a type, `C` will be a metatype of the declared type.

Strategies The basic approach to constraint solving is to simplify the constraints until they can no longer be simplified, then produce (and check) educated guesses about which declaration from an overload set should be selected or what concrete type should be bound to a given type variable. Each guess is tested as an assumption, possibly with other guesses, until the solver either arrives at a solution or concludes that the guess was incorrect.

Within the implementation, each guess is modeled as an assumption within a new solver scope. The solver scope inherits all of the constraints, overload selections, and type variable bindings of its parent solver scope, then adds one more guess. As such, the solution space explored by the solver can be viewed as a tree, where the top-most node is the constraint system generated directly from the expression. The leaves of the tree are either solutions to the type-checking problem (where all constraints have been simplified away) or represent sets of assumptions that do not lead to a solution.

The following sections describe the techniques used by the solver to produce derived constraint systems that explore the solution space.

Overload Selection Overload selection is the simplest way to make an assumption. For an overload set that introduced a disjunction constraint `T0 := A1 or T0 := A2 or ... or T0 := AN` into the constraint system, each term in the disjunction will be visited separately. Each solver state binds the type variable `T0` and explores whether the selected overload leads to a suitable solution.

Type Variable Bindings A second way in which the solver makes assumptions is to guess at the concrete type to which a given type variable should be bound. That type binding is then introduced in a new, derived constraint system to determine if the binding is feasible.

The solver does not conjure concrete type bindings from nothing, nor does it perform an exhaustive search. Rather, it uses the constraints placed on that type variable to produce potential candidate types. There are several strategies employed by the solver.

Meets and Joins

A given type variable `T0` often has relational constraints placed on it that relate it to concrete types, e.g., `T0 <c Int` or `Float <c T0`. In these cases, we can use the concrete types as a starting point to make educated guesses for the type `T0`.

To determine an appropriate guess, the relational constraints placed on the type variable are categorized. Given a relational constraint of the form `T0 <? A` (where `<?` is one of `<`, `<t`, or `<c`), where `A` is some concrete type, `A` is said to be "above" `T0`. Similarly, given a constraint of the form `B <? T0` for a concrete type `B`, `B` is said to be "below" `T0`. The above/below terminologies comes from a visualization of the lattice of types formed by the conversion relationship, e.g., there is an edge `A -> B` in the latter if `A` is convertible to `B`. `B` would therefore

be higher in the lattice than `A`, and the topmost element of the lattice is the element to which all types can be converted, `Any` (often called "top").

The concrete types "above" and "below" a given type variable provide bounds on the possible concrete types that can be assigned to that type variable. The solver computes [2] the join of the types "below" the type variable, i.e., the most specific (lowest) type to which all of the types "below" can be converted, and uses that join as a starting guess.

Supertype Fallback

The join of the "below" types computed as a starting point may be too specific, due to constraints that involve the type variable but weren't simple enough to consider as part of the join. To cope with such cases, if no solution can be found with the join of the "below" types, the solver creates a new set of derived constraint systems with weaker assumptions, corresponding to each of the types that the join is directly convertible to. For example, if the join was some class `Derived`, the supertype fallback would then try the class `Base` from which `Derived` directly inherits. This fallback process continues until the types produced are no longer convertible to the meet of types "above" the type variable, i.e., the least specific (highest) type from which all of the types "above" the type variable can be converted 3.

Default Literal Types

If a type variable is bound by a conformance constraint to one of the literal protocols, "`T0` conforms to `ExpressibleByIntegerLiteral`", then the constraint solver will guess that the type variable can be bound to the default literal type for that protocol. For example, `T0` would get the default integer literal type `Int`, allowing one to type-check expressions with too little type information to determine the types of these literals, e.g., `-1`.

Comparing Solutions The solver explores a potentially large solution space, and it is possible that it will find multiple solutions to the constraint system as given. Such cases are not necessarily ambiguities, because the solver can then compare the solutions to determine whether one of the solutions is better than all of the others. To do so, it computes a "score" for each solution based on a number of factors:

- How many user-defined conversions were applied.
- How many non-trivial function conversions were applied.
- How many literals were given "non-default" types.

Solutions with smaller scores are considered better solutions. When two solutions have the same score, the type variables and overload choices of the two systems are compared to produce a relative score:

- If the two solutions have selected different type variable bindings for a type variable where a "more specific" type variable is a better match, and

one of the type variable bindings is a subtype of the other, the solution with the subtype earns +1.

- If an overload set has different selected overloads in the two solutions, the overloads are compared. If the type of the overload picked in one solution is a subtype of the type of the overload picked in the other solution, then first solution earns +1.

The solution with the greater relative score is considered to be better than the other solution.

Solution Application

Once the solver has produced a solution to the constraint system, that solution must be applied to the original expression to produce a fully type-checked expression that makes all implicit conversions and resolved overloads explicit. This application process walks the expression tree from the leaves to the root, rewriting each expression node based on the kind of expression:

Declaration references Declaration references are rewritten with the precise type of the declaration as referenced. For overloaded declaration references, the `Overload*Expr` node is replaced with a simple declaration reference expression. For references to polymorphic functions or members of generic types, a `SpecializeExpr` node is introduced to provide substitutions for all of the generic parameters.

Member references References to members are similar to declaration references. However, they have the added constraint that the base expression needs to be a reference. Therefore, an rvalue of non-reference type will be materialized to produce the necessary reference.

Literals Literals are converted to the appropriate literal type, which typically involves introducing calls to the witnesses for the appropriate literal protocols.

Closures Since the closure has acquired a complete function type, the body of the closure is type-checked with that complete function type.

The solution application step cannot fail for any type checking rule modeled by the constraint system. However, there are some failures that are intentionally left to the solution application phase, such as a postfix '!' applied to a non-optional type.

Locators During constraint generation and solving, numerous constraints are created, broken apart, and solved. During constraint application as well as during diagnostics emission, it is important to track the relationship between the constraints and the actual AST nodes from which they originally came. For example, consider the following type checking problem::

```
struct X {
  // user-defined conversions
```

```
  func [conversion] __conversion () -> String { /* ... */ }
  func [conversion] __conversion () -> Int { /* ... */ }
}

func f(_ i : Int, s : String) { }

var x : X
f(10.5, x)
```

This constraint system generates the constraints "`T(f)` ==Fn `T0 -> T1`" (for fresh variables `T0` and `T1`), "`(T2, X) <c T0`" (for fresh variable `T2`) and "`T2` conforms to `ExpressibleByFloatLiteral`". As part of the solution, after `T0` is replaced with `(i : Int, s : String)`, the second of these constraints is broken down into "`T2 <c Int`" and "`X <c String`". These two constraints are interesting for different reasons: the first will fail, because `Int` does not conform to `ExpressibleByFloatLiteral`. The second will succeed by selecting one of the (overloaded) conversion functions.

In both of these cases, we need to map the actual constraint of interest back to the expressions they refer to. In the first case, we want to report not only that the failure occurred because `Int` is not `ExpressibleByFloatLiteral`, but we also want to point out where the `Int` type actually came from, i.e., in the parameter. In the second case, we want to determine which of the overloaded conversion functions was selected to perform the conversion, so that conversion function can be called by constraint application if all else succeeds.

Locators address both issues by tracking the location and derivation of constraints. Each locator is anchored at a specific AST node (expression, pattern, declaration etc.) i.e., the function application `f(10.5, x)`, and contains a path of zero or more derivation steps from that anchor. For example, the "`T(f)` ==Fn `T0 -> T1`" constraint has a locator that is anchored at the function application and a path with the "apply function" derivation step, meaning that this is the function being applied. Similarly, the "`(T2, X) <c T0` constraint has a locator anchored at the function application and a path with the"apply argument" derivation step, meaning that this is the argument to the function.

When constraints are simplified, the resulting constraints have locators with longer paths. For example, when a conversion constraint between two tuples is simplified conversion constraints between the corresponding tuple elements, the resulting locators refer to specific elements. For example, the `T2 <c Int` constraint will be anchored at the function application (still), and have two derivation steps in its path: the "apply function" derivation step from its parent constraint followed by the "tuple element 0" constraint that refers to this specific tuple element. Similarly, the `X <c String` constraint will have the same locator, but with "tuple element 1" rather than "tuple element 0". The `ConstraintLocator` type in the constraint solver has a number of different derivation step kinds (called "path elements" in the source) that describe the various ways in which larger constraints can be broken down into smaller ones.

Overload Choices Whenever the solver creates a new overload set, that overload set is associated with a particular locator. Continuing the example from the parent section, the solver will create an overload set containing the two user-defined conversions. This overload set is created while simplifying the constraint `X <c String`, so it uses the locator from that constraint extended by a "conversion member" derivation step. The complete locator for this overload set is, therefore::

```
function application -> apply argument -> tuple element #1 -> conversion member
```

When the solver selects a particular overload from the overload set, it records the selected overload based on the locator of the overload set. When it comes time to perform constraint application, the locator is recreated based on context (as the bottom-up traversal walks the expressions to rewrite them with their final types) and used to find the appropriate conversion to call. The same mechanism is used to select the appropriate overload when an expression refers directly to an overloaded function. Additionally, when comparing two solutions to the same constraint system, overload sets present in both solutions can be found by comparing the locators for each of the overload choices made in each solution. Naturally, all of these operations require locators to be unique, which occurs in the constraint system itself.

Simplifying Locators Locators provide the derivation of location information that follows the path of the solver, and can be used to query and recover the important decisions made by the solver. However, the locators determined by the solver may not directly refer to the most specific AST node for the purposes of identifying the corresponding source location. For example, the failed constraint "`Int` conforms to `ExpressibleByFloatLiteral`" can most specifically by centered on the floating-point literal `10.5`, but its locator is::

```
function application -> apply argument -> tuple element #0
```

The process of locator simplification maps a locator to its most specific AST node. Essentially, it starts at the anchor of the locator (in this case, the application `f(10.5, x)`) and then walks the path, matching derivation steps to subexpressions. The "function application" derivation step extracts the argument (`(10.5, x)`). Then, the "tuple element #0" derivation extracts the tuple element 0 subexpression, `10.5`, at which point we have traversed the entire path and now have the most specific expression for source-location purposes.

Simplification does not always exhaust the complete path. For example, consider a slight modification to our example, so that the argument to `f` is provided by another call, we get a different result entirely::

```
func f(_ i : Int, s : String) { }
func g() -> (f : Float, x : X) { }

f(g())
```

Here, the failing constraint is `Float <c Int`, with the same locator::

```
function application -> apply argument -> tuple element #0
```

When we simplify this locator, we start with `f(g())`. The "apply argument" derivation step takes us to the argument expression `g()`. Here, however, there is no subexpression for the first tuple element of `g()`, because it's simple part of the tuple returned from `g`. At this point, simplification ceases, and creates the simplified locator::

```
function application of g -> tuple element #0
```

Performance

The performance of the type checker is dependent on a number of factors, but the chief concerns are the size of the solution space (which is exponential in the worst case) and the effectiveness of the solver in exploring that solution space. This section describes some of the techniques used to improve solver performance, many of which can doubtless be improved.

Constraint Graph The constraint graph describes the relationships among type variables in the constraint system. Each vertex in the constraint graph corresponds to a single type variable. The edges of the graph correspond to constraints in the constraint system, relating sets of type variables together. Technically, this makes the constraint graph a *multigraph*, although the internal representation is more akin to a graph with multiple kinds of edges: each vertex (node) tracks the set of constraints that mention the given type variable as well as the set of type variables that are adjacent to this type variable. A vertex also includes information about the equivalence class corresponding to a given type variable (when type variables have been merged) or the binding of a type variable to a specific type.

The constraint graph is critical to a number of solver optimizations. For example, it is used to compute the connected components within the constraint graph, so that each connected component can be solved independently. The partial results from all of the connected components are then combined into a complete solution. Additionally, the constraint graph is used to direct simplification, described below.

Simplification Worklist When the solver has attempted a type variable binding, that binding often leads to additional simplifications in the constraint system. The solver will query the constraint graph to determine which constraints mention the type variable and will place those constraints onto the simplification worklist. If those constraints can be simplified further, it may lead to additional type variable bindings, which in turn adds more constraints to the worklist. Once the worklist is exhausted, simplification has completed. The use of the worklist eliminates the need to reprocess constraints that could not have changed because the type variables they mention have not changed.

Solver Scopes The solver proceeds through the solution space in a depth-first manner. Whenever the solver is about to make a guess—such as a speculative type variable binding or the selection of a term from a disjunction—it introduces a new solver scope to capture the results of that assumption. Subsequent solver scopes are nested as the solver builds up a set of assumptions, eventually leading to either a solution or an error. When a solution is found, the stack of solver scopes contains all of the assumptions needed to produce that solution, and is saved in a separate solution data structure.

The solver scopes themselves are designed to be fairly cheap to create and destroy. To support this, all of the major data structures used by the constraint solver have reversible operations, allowing the solver to easily backtrack. For example, the addition of a constraint to the constraint graph can be reversed by removing that same constraint. The constraint graph tracks all such additions in a stack: pushing a new solver scope stores a marker to the current top of the stack, and popping that solver scope reverses all of the operations on that stack until it hits the marker.

Online Scoring As the solver evaluates potential solutions, it keeps track of the score of the current solution and of the best complete solution found thus far. If the score of the current solution is ever greater than that of the best complete solution, it abandons the current solution and backtracks to continue its search.

The solver makes some attempt at evaluating cheaper solutions before more expensive solutions. For example, it will prefer to try normal conversions before user-defined conversions, prefer the "default" literal types over other literal types, and prefer cheaper conversions to more expensive conversions. However, some of the rules are fairly ad hoc, and could benefit from more study.

Arena Memory Management Each constraint system introduces its own memory allocation arena, making allocations cheap and deallocation essentially free. The allocation arena extends all the way into the AST context, so that types composed of type variables (e.g., `T0 -> T1`) will be allocated within the constraint system's arena rather than the permanent arena. Most data structures involved in constraint solving use this same arena.

Diagnostics

Swift 5.2 introduced a new diagnostic framework, which is described in detail in this blog post.

Footnotes

not specifically been documented outside of the source code. The constraints-based type checker contains a function `matchTypes` that documents and implements each of these rules. A future revision of this document will provide a more readily-accessible version.

[2]: More accurately, as of this writing, “will compute”. The solver doesn't current compute meets and joins properly. Rather, it arbitrarily picks one of the constraints “below” to start with.

meets and joins, so the solver continues until it runs out of supertypes to enumerate.