Generating subtypes

A/J/J

Abstract

Notes about how to generates subtypes

1 Introduction

This note aims to describe how to generate subtypes of Julia method signature for the purpose of testing type stability.

The definition of Julia is imprecise, so generating the exact set of subtypes of a signature may not be possible. We choose to generate a superset of that set of types, and rely on the Julia type checker to filter out extraneous types.

We strive to limit the size of the superset for perfomance reasons, the space of types is large and checking any type is costly.

2 A specification of the generator

The Julia type language is rich and its semantics is complex. Previous work has shown that the subtyping relation is not decidable.

Moreover, even the Julia language specification admits to being at best heuristic (See the discussion of Diagonal types).

Our approach is to start with a specification for a generator that is simple but ignores some important features of the language. As a specification, we aim for readability rather than as a guide towards a practical implementation.

We will then discuss extensions and implemtation issues.

2.1 Core Julia

Julia has an idiosyncratic syntax and terminology, this section presents the type language using more traditional terminology. Metavariables t, v and d range over, respectively, types, type instances, and type declarations. The notation t^* stands for zero or more types t and $t_1 \dots t_n$ for the same. The type grammar is given in Fig. 1. For the remainder, we shall assume that types are well-formed (as defined in the text).

A type declaration d such as $\exists X^t.\, T(X) <: T'(X)$ introduces a type constructor $T(X)$ parameterized by an existentially quantified type variable X upper bounded by type t. Furthermore, this type is declared to be a subtype of $T'(X)$. The scope of the type variable extends to the parent type. Type constructors without parameters can be written without parentheses, i.e. $T() \equiv T$. The declaration of type $Any()$ is assumed. In general a declarations may introduce multiple variables and provide any type to parent. For instance, $\exists X^{Any}.\, \exists Y^{Any}.\, T(X,Y) <: T'(X, Any)$ introduce the type constructor $T(X,Y)$ as a subtype of T', both variables are bounded by Any and the parent type is instantiated with $T'(X, Any)$. A type declaration is well-formed if it has no free variable, all of its constituent types are well-formed, and the instantiation of the parent type constructor respects that types' bounds. In Julia terms, type declarations cover all of abstract, primitive and composite type declarations. Concrete types are types that have no subtypes. For

$$
\begin{array}{lll}
v ::= & T(t^*) & \text{value} \\
& \langle t^* \rangle & \\
t ::= & v & \text{type} \\
& X & \\
& \cup(t^*) & \\
& \exists X^t.\, t & \\
d ::= & l <: T(t^*) & \text{declaration} \\
l ::= & T(X^*) & \\
& \exists X^t.\, l &
\end{array}
$$

Fig. 1: Grammar

our purposes, we ignore the `mutable` qualifier as it does not affect subtyping —mutable types are limtied to concrete types— and size modifiers for primitive types.

A type, t, can be an instantiation of a type constructor, a tuple of types $\langle t^* \rangle$, a variable X, the union of types $\cup(t^*)$ and a variable introduction $\exists X^t.\, T(t^*)$. In terms of subtyping, Julia limits parametric types to be invariant in their arguments, so that `Ref{Int}<:Ref{Real}` does not hold even if `Int` is a subtype of `Real`. On the other hand, tuples are co-variant, and thus `Tuple{Int}<:Tuple{Real}` holds. Julia supports lower bounds for existential, but they are rarely used. Not supporting them, entails the generation of more subtypes than necessary. Well-formed types have correct instantiations of parametric types, no undefined type constructor and, at the top-level, no free variables.

An instance v is the type of a run-time value in Julia, these can be either type constructors or tuples. Note that the inner types can be parametric.

2.2 A Core Generator

A generator is a relation $t \rightsquigarrow S$ that, for a set of type declarations d^*, generates all subtypes of a type t where S is a $\{v_1 \ldots v_n\}$ of value types. The relation is specified in Fig. **??**. The rules are non-deterministic, each application of a generator returns *all* derivable types. Each of the rules is described next.

Rule GenSet extends the generator to sets of values by union of the pointwise application of $\cdot \rightsquigarrow \cdot$. As mentioned above, any given v can derive multiple sets of values, we assume that $v \rightsquigarrow S$ returns the union of all such sets.

Rule GenUnion generates the subtypes of a union as the union of the pointwise sets for each element of the union. Unions are not part of the set of a run-time values.

Rule GenTuple deals with tuples. First, subtypes are generated pointwise for every element t_i of the tuple type. The result is obtained as the combination of every possible tuple that could be obtained from the values in each S_i, this combination is written $[\![\langle S_1 \ldots S_2 \rangle]\!]$, note that we take liberty with the syntax by creating a tuple of sets, the role $[\![\cdot]\!]$ is to shake this out into a set of well-formed tuples.

Rule GenExists introduces a type variable X in some type t', written $\exists X^t.\, t'$ where X has an upper bound of t. The rule start by generating the set S' of subtypes of the bound t. Then for each indivudal value v in S', substitute that value for X in t' and generates all subtypes of the result. The notation $[\![t'[X/S']]\!]$ denotes the set of types obtained by replacing X in t' with each of the values in S'.

Rule GenDeclare deals with subtypes of $T(t_1^*)$. It non-deterministically picks one of the declarations with T as a supertype. $T(t_1^*)$ is an instantiation of the parent type T, with no free variables. The rule identifies a direct subtype $l <: T(t_2^*)$ where t_2^* is partial instantiation which may have some free variables corresponding to some of the introductions occurring in l. To fix the mind consider the following example:

$Int <: Any$
$\exists X^{Any}.\, \exists Y^{Any}.\, Dict(X, Y) <: Any$
$\exists X^{Any}.\, IntDict(X) <: Dict(X, Int)$

$$\text{[GenSet]}\ \frac{v_1 \rightsquigarrow S_1 \ \ldots\ v_n \rightsquigarrow S_n}{\{v_1 \ldots v_n\} \rightsquigarrow S_1 \cup \cdots \cup S_n} \qquad \text{[GenUnion]}\ \frac{t_1 \rightsquigarrow S_1 \ \ldots\ t_n \rightsquigarrow S_n}{\cup(t_1 \ldots t_n) \rightsquigarrow S_1 \cup \cdots \cup S_n} \qquad \text{[GenTuple]}\ \frac{t_1 \rightsquigarrow S_1 \ \ldots\ t_n \rightsquigarrow S_n}{\langle t_1 \ldots t_2 \rangle \rightsquigarrow [\![\langle S_1 \ldots S_n \rangle]\!]}$$

$$\text{[GenExists]}\ \frac{t \rightsquigarrow S' \qquad [\![t'[X/S']]\!] \rightsquigarrow S}{\exists X^t.\, t' \rightsquigarrow S} \qquad \text{[GenDeclare]}\ \frac{l <: T(t_2^*) \qquad match(l, T(t_1^*), T(t_2^*)) = t \qquad t \rightsquigarrow S}{T(t_1^*) \rightsquigarrow S \cup T(t_1^*)}$$

Fig. 2: Generation rules

$$\frac{X \notin t' \qquad match(t_1, t', v) = t_2}{match(\exists X^t.\, t_1, t', v) = \exists X^t.\, t_2} \qquad \frac{unify(X, t', v) = t_2 \qquad match(t_1[X/t_2], t', v) = t_3}{match(\exists X^t.\, t_1, t', v) = t_3}$$

$$\frac{t' \cong v'}{match(v, t', v') = v}$$

$$unify(X, X, t) = t \qquad \frac{\exists i \in [1,n].\, unify(X, t_i, t_i') = t}{unify(X, T(t_1 \ldots t_n), T(t_1' \ldots t_n')) = t} \qquad \frac{\exists i \in [1,n].\, unify(X, t_i, t_i') = t}{unify(X, \langle t_1 \ldots t_n \rangle, \langle t_1' \ldots t_n' \rangle) = t}$$

$$\frac{\exists i \in [1,n].\, unify(X, t_i, t_i') = t}{unify(X, \cup(t_1 \ldots t_n), \cup(t_1' \ldots t_n')) = t} \qquad \frac{X \neq Y \qquad \exists i \in [1,2].\, unify(X, t_i, t_i') = t}{unify(X, \exists Y^{t_1}.\, t_2, \exists Y^{t_1'}.\, t_2') = t}$$

$$X \cong t \qquad \frac{t_1 \cong t_1' \qquad t_n \cong t_n'}{\exists X^{t_1}.\, t_2 \cong \exists X^{t_1'}.\, t_2'} \qquad \frac{t_1 \cong t_1' \ \ldots\ t_n \cong t_n'}{\cup(t_1 \ldots t_n) \cong \cup(t_1' \ldots t_n')} \qquad \frac{t_1 \cong t_1' \ \ldots\ t_n \cong t_n'}{\langle t_1 \ldots t_n \rangle \cong \langle t_1' \ldots t_n' \rangle}$$

$$\frac{t_1 \cong t_1' \ \ldots\ t_n \cong t_n'}{T(t_1 \ldots t_n) \cong T(t_1' \ldots t_n')}$$

Fig. 3: Auxiliary definitions

The query $Dict(Int, Int) \rightsquigarrow S$ returns $S = \{Dict(Int, Int), IntDict(Int)\}$ as the only subtype of $Dict(Int, Int)$ is $IntDict(Int)$.

On the other hand $Dict(Int, Any) \rightsquigarrow S$ returns only the singleton $Dict(Int, Any)$. This is because parametric types are invariant and thus $Dict(Int, Int) \not<: Dict(Int, Any)$.

To return to GenDeclare, if there is a candidate subtype, $match(l, T(t_1^*), T(t_2^*)) = t$ will strip the declaration of variables that are bound and ensure that the parent instance matches the one in the declaration.

In the auxiliaries, $match(l, t, v) = t'$ takes three arguments, the first is a declaration with possibly some existentials, the second is the instantiation of the parent type, the third is the declaration of the parent type, and it returns a new type for l with all the bound variables stripped off. Match also ensures that t and v are similar.

Unify finds a binding for a variable.

$\cong$ ensures that two terms have the same structure module variables occuring on the left hand side.

$X \notin t$ indicates that variable X does not occur free in type t, $t[X/t']$ denotes the substitution of t' for X in type t.