Symbolent Calculus of Symbolic Type Inference

by Sven Nilsen, 2018

In this paper I introduce a calculus notation called "Symbolent Calculus" for Symbolic Type Inference. This notation generalizes Sequent Calculus for indexed contexts using context types. Context types can be used directly in path semantics for assigning self-references to the symbolic expression of the sub-type. With other words, a context type depends on the expression of an object instead of the object.

Assume you have a function that does type inference:

infer : term → res[term]

For simplicity, let all numbers be natural numbers.

A closure returning `3` has the type `\() → nat`:

infer("\() = 3") = ok("\() → nat")

A symbolic type inference function returns a program that returns the type:

symbolic_infer : term → res[term]

symbolic_infer("\() = 3") = ok("\() → typeof(3)")

The advantage of symbolic type inference is that parts of the type checking inference can be undefined.

For example, one can use an uninterpreted function `f` to define some sub-type `[f] a`:

symbolic_infer("\(x : [f] a) = x") = ok("\([f] a) → [f] a")

This is possible because the function without the input constraint is identical to `id`:

(\(x) = x) <=> id
∃id{T} <=> T

An even more lazy symbolic inference is the following:

symbolic_infer("\(x : [f] a) = x") = ok("\([f] a) → ∃(\(x) = x){[f] a}")

Then you have the following proof:

∃(\(x) = x){[f] a}
∃id{[f] a} (\(x) = x) <=> id
[f] a ∃id{T} <=> T

An automated theorem prover should be able to prove the following:

$\because$ ((\(x) = x) <=> id) => (∃(\(x) = x){[f] a} <=> ∃id{[f] a})
$\because$ (∃id{T} <=> T) => (∃id{[f] a} <=> [f] a)
$\because$ $(A_0 \Rightarrow (B_0 \Leftrightarrow B_1)) \wedge (A_1 \Rightarrow (B_1 \Leftrightarrow B_2)) \Rightarrow ((A_0 \wedge A_1) \Rightarrow (B_0 \Leftrightarrow B_2))$
$\therefore$ ((\(x) = x) <=> id) ∧ (∃id{T} <=> T)) => (∃(\(x) = x){[f] a} <=> [f] a)

In path semantics, symbolic type inference is required, because not everything that can be said about a term is preserved by reducing a type.

For example, `3` can not be reduced to `nat`, because it would erase information
about the sub-type `(= 3)`. Instead, the expression `typeof(3)` is used. If there are no other parts of the proof which depends on knowing what might be said about `3`, then it can be reduced to `nat`:

not_needed(3) => (typeof(3) = nat)

For example, some very simple sub-type encodes a lot of information:

[f] a
∀f ∧ [f{∀f}] (a : ∀(∃f{∀f}) ∧ ∃f{∀f})

When adding more type information, the rule gets more complex:

nat ∧ [f] a
nat ∧ ∀f ∧ [f{nat ∧ ∀f}] (a : ∀(∃f{nat ∧ ∀f}) ∧ ∃f{nat ∧ ∀f})

To generalize this, one can use a context type:

$\Gamma_0 \wedge$ [f] a
$\Gamma_0 \wedge$ ∀f ∧ [f{$\Gamma_0 \wedge$ ∀f}] (a : ∀(∃f{$\Gamma_0 \wedge$ ∀f}) ∧ ∃f{$\Gamma_0 \wedge$ ∀f})
$\Gamma_1 \wedge$ [f{Γ_1}] (a : ∀(∃f{Γ_1}) ∧ ∃f{Γ_1}) $\quad\quad$ $\Gamma_1 := \Gamma_0 \wedge$ ∀f
Γ_2 $\quad\quad$ $\Gamma_2 := \Gamma_1 \wedge$ [f{Γ_1}] (a : ∀(∃f{Γ_1}) ∧ ∃f{Γ_1})

All the other information about the sub-type is assigned a context type `$\Gamma_0$`. When some rule lifts new information to the top level, this becomes part of a new context level `$\Gamma_1$`. When all lifting is done, the whole expression is assigned a context type `$\Gamma_2$`. With other words, the context type has 3 levels.

Context types are motivated by the need to preserve information, reason consistently about rules in path semantics and to provide clearity in a system that mixes types and sub-types.

The general rule is that if some term is erased in the expression, then the other terms do not loose information. It means that each term contains all the information that they need. The terminology for this is “the expression is saturated”. The saturation required depends on the context type of expression.

A context type is similar to that of a context in Sequent Calculus. However, Sequent Calculus does not distinguish between context levels. Neither is it possible to express that a context type is a sub-type of types that constrains their syntactical form. Therefore, a more general calculus of types is required, called “Symbolent Calculus”. The name implies it is meant to be used for symbolic type inference. The rest of the paper introduces Symbolent Calculus and some of its properties.

The symbol `Γ` is used to refer to the context type. A context type is written with the notation:

$$\Gamma^{a}_{b\,c\,d}$$

`a` is number of references (zero permits erasing when no longer needed)
`b` is level
`c` is term index in `∧` expression
`d` is graph node index to type judgement `x : T` in type checker

A reference at level `n` here means the same as counting the context types of level `n-1`.

The notation permits omitting indices to be made generic or filled in automatically:

Γ^{a}	Some context type with `a` number of references
Γ_{b}	Some context type with level `b`
Γ^{0}_{000}	The first context type in some type checker
Γ	Some generic context

Here are some examples:

$nat : \Gamma^{0}$	The type `nat` has some context type with zero references
$\exists f\{T\} : \Gamma^{1}$	The existential path of `f` constrained to `T` has some context type with one reference (because of `T`)
$\exists f : \Gamma^{0}$	The existential path of `f` with no constraints has some context type with zero references.
$\Gamma \vdash A$	Some generic context can prove `A`
$\Gamma_{2} \vdash A$	Some generic context with level 2 can prove `A`

One reason context types are useful is that they can refer to the complement of any sub-type within a sub-type context. By omitting the indices, one can say that there is some complement of the sub-type without referring to a specific complement in particular.

$A \wedge B \wedge C : \Gamma_{2}$	This is the whole sub-type expression
$B \wedge C : \Gamma_{10}$	By taking out `A`, one gets `B ∧ C`
$A \wedge C : \Gamma_{11}$	By taking out `B`, one gets `A ∧ C`
$A \wedge B : \Gamma_{12}$	By taking out `C`, one gets `A ∧ B`

The context type `$\Gamma_{1}$` refers to some part of the sub-type expression that you get by taking out something. Therefore, when somebody writes:

$$\Gamma_{1} \wedge A$$

One can infer that what we are talking about is:

$$\Gamma_{10} \wedge A$$

The context level `1` includes the sub-types that are lifted up from the inference rules:

$$\Gamma_1 \wedge [f\{T\}]\ a \quad <=> \quad \Gamma_1 \wedge T \wedge [f\{T\}]\ a$$

The context level `0` excludes the sub-types that are lifted up from the inference rules.
However, if the context can already prove the lifted sub-type, then it is included.

$$(\Gamma_0 : (\neg\vdash T)) \wedge [f\{T\}]\ a \quad <\neg=> \quad \Gamma_1 \wedge T \wedge [f\{T\}]\ a$$

$$(\Gamma_0 : (\vdash T)) \wedge [f\{T\}]\ a \quad <=> \quad \Gamma_1 \wedge T \wedge [f\{T\}]\ a$$

This is interpreting the sub-type naively, which is different from the definitional sub-type interpretation.

Now, an inference rules which lifts sub-types is a function that has the type:

$x : \{a, c, d\} \times \Gamma^{a}_{0cd} \rightarrow \Gamma^{a}_{1cd}$	Showing all inferred indices from implied arguments
$x : \Gamma_0 \rightarrow \Gamma_1$	Short version
$x(A : \Gamma_{01}) \wedge B : \Gamma_2$	Applying `x` to `A` to include lifted sub-types from `B`
$(A' : \Gamma_{11}) \wedge B : \Gamma_2$	`A'` has context type `$\Gamma_{11}$`

One thing worth noticing: The inference rule operates on the complement of the term, to pass around information to other terms that can be inferred by looking at the term. This happens for every argument, meaning that there are multiple ways of writing the same inferred expression:

$$(A' : \Gamma_{11}) \wedge B : \Gamma_2 \quad <=> \quad (B' : \Gamma_{10}) \wedge A : \Gamma_2$$

In practice, a monotonic type checker infers what it can from each term, then it checks whether anything new can be inferred from adding the information that it it has learn. This is repeated until nothing new is learned. The word “monotonic” means that no new information invalidates proofs that already have been used.

Computationally, it is trivial to go from context level `1` to `2`.
The hard problem is getting from `0` to `1`.

To do type inference efficiently, one must collect everything known about a term in one place:

$$((a : T_0) \wedge (a : T_1)) => (a : T_0 \wedge T_1) \quad : T_0 \times T_1 \rightarrow \Gamma_0 \rightarrow \Gamma_1$$

This is in itself an inference rule. Notice that this information is sent to both terms, now equivalent.

As a final example, one can compute `symbolic_infer(“[f] b”)` and if `f` is total, one can simplify:

$\because$	$\Gamma_0 \wedge \forall f \wedge [f\{\Gamma_0 \wedge \forall f\}]\ (b : \forall(\exists f\{\Gamma_0 \wedge \forall f\}) \wedge \exists f\{\Gamma_0 \wedge \forall f\}) : f \times b \rightarrow \Gamma_0 \rightarrow \Gamma_2$		
$\because$	$f : (A \rightarrow B) \wedge [\forall]\ true_1$		`f` is total
$\because$	$\forall(\exists f\{T\}) => B$	$: T \times (f : A \rightarrow B) \rightarrow \Gamma_0 \rightarrow \Gamma_1$	
$\therefore$	$\Gamma_0 \wedge [f\{\Gamma_0\}]\ (b : B \wedge \exists f\{\Gamma_0\})$	$: f \times b \rightarrow \Gamma_0 \rightarrow \Gamma_2$	New rule for total functions

Proved using Symbolent Calculus, one can see an iteration on total functions has no lifted sub-types.