Fall 2022 Research Report: sslang

Feitong Leo Qiao
`flq2101@columbia.edu`

December 2022

1 Introduction

In the Fall 2022 sslang research group, I led the Type System team, consisting of the following members: Yiming Fang, Hao Zhou, Chris Yoon. As a team, the objectives were:

- first half of the semester: become familiar with theory and implementations of the Hindley-Milner (HM) type system
- second half of the semester: become familiar with the sslang codebase and port pattern-match related code from AST to IR.

In parallel to the team objectives, my work on sslang focused on rewriting the type-check/type-inference in a constraint-based approach. In this report, I will only discuss my work on the new constraint-based type system. A full type system report will be separately written by the entire Type System team, and reports regarding the pattern-match related code will be submitted by respective individual team members.

The old type inference algorithm in sslang roughly follows the traditional syntax-directed Algorithm-J, with union-find as the underlying unification algorithm. It is the classic, textbook implementation of HM type inference, but it has several drawbacks. First, the syntax-directed property of the algorithm forces the entire type-inference pass to be dependent on the IR representation. Intricate inference logic must be written for each kind of expression/pattern nodes, and changes/additions to IR must have sound typing rules. Second, the inference algorithm is extremely delicate, making it difficult to extend upon. Ideally, we want a core type system that is robust and flexible enough to handle various extensions that we would like to support (e.g. (improved) type annotations, kind system, typeclasses, linear types, etc.)

To resolve these issues, I introduce a new constraint-based type system to sslang. A total of 3 versions had been written, each inspired by some existing work in the field.

- Version 0: inspired by the type system HM(X) and its type inference procedure, described in Pottier and Remy's *The Essence of ML Type Inference* [4]; follows the OCaml reference implementation, *mini*.

- Version 1: inspired by Pottier's *Hindley-Milner Elaboration in Applicative Style* [3]; follows the OCaml reference implementation, *inferno.*
- Version 2: inspired by the type-system implementation of the Elm programming language; similar to version 0.

In the sslang GitHub repository, Version 0 and Version 1 are in the now obsolete "#108 Constraint-based type inference" pull request; Version 2 is in the "#122 Version 2: constraint-based type inference" pull request, and it will be merged into the main branch once we resolve som remaining unit test issues. There is too much code to be put inside in report (3284 and 1874 lines added, respectively), so those who are interested please refer to the PRs themselves.

In this report, Section 2 briefly introduces the HM(X) constraint-based type system, analyzes Version 0 and discusses its lack of *type elaboration* capability. Section 3 introduces Pottier's "constraints with a value" approach to HM elaboration and discusses the trade-offs of Version 1. Section 4 introduces the finalized Version 2 and some of its details. Section 5 discusses future plans.

2 HM(X): Constraint-based ML Type System

The HM(X) is a constraint-based type system that is parametrized by model/interpretation of some constraints (e.g. the subtype $\leq$ constraint). Pottier and Remy proves that, when constraints are made up of equations (no subtyping) between free, finite terms, HM(X) is equivalent to the traditional Damas-Hindley-Milner type system (DM). It is called the *syntactic, equality model.* In this section, I will give a quick introduction of the HM(X) type system under the syntactic, equality model. Due to time constraints, I will only discuss parts of the constraint language, but not the HM(X) type system formulation nor the rigorous constraint semantics.

2.1 Constraints

The core constraint language is:

$$
\begin{array}{rlr}
\tau ::= & \alpha & \textit{type variable} \\
& \mid \tau \to \tau & \textit{arrow type} \\
& \mid ... & \textit{built-in types} \\
C ::= & \mathit{true} & \textit{truth} \\
& \mid C \wedge C & \textit{conjunction} \\
& \mid \tau = \tau & \textit{equality} \\
& \mid \exists \alpha . C & \textit{existential quantification} \\
& \mid \text{let } x = \lambda \alpha . C \text{ in } C & \textit{constraint abstraction binding} \\
& \mid x\ \tau & \textit{constraint abstraction instantiation}
\end{array}
$$

It is a more concise version than the constraints in HM(X) and comes from Pottier's later work [3]. The core idea is the same, and I find this formulation to be friendlier.

Besides the last two regarding constraint abstractions, all other constraints should be self-explanatory and have the expected semantics. The 'let' construct binds the variable x to the constraint abstraction $\lambda\alpha.C$. The instantiation construct $x\ \tau$ applies the constraint abstraction denoted by x to the type τ. The semantics can be best explained by the following substituion law:

$$\text{let } x = \lambda\alpha.C_1 \text{ in } C_2 \equiv \exists\alpha.C_1 \wedge [\lambda\alpha.C_1/x]C_2$$

In other words, the let constraint is equivalent to (1) requiring the existence of some type that satisfies C_1 and (2) substituting every reference of x in C_2 with the constraint abstraction.

2.2 Constraint Generation

For the core ML calculus, the constraint generation is simple. For a judgement of the form $t\ :\ \tau$, the corresponding constraint detoted $[[t\ :\ \tau]]$ is recursively defined to be the following:

$$\begin{aligned}
[[x\ :\ \tau]] &= x\tau \\
[[\lambda x.u\ :\ \tau]] &= \exists\alpha_1\alpha_2.(\tau = \alpha_1 \to \alpha_2 \wedge \text{def } x = \alpha_1 \text{ in } [[u\ :\ \alpha_2]]) \\
[[t_1\ t_2\ :\ \tau]] &= \exists\alpha.([[t_1\ :\ \alpha \to \tau]] \wedge [[t_2\ :\ \alpha]]) \\
[[\text{let } x = t_1 \text{ in } t_2\ :\ \tau]] &= \text{let } x = \lambda\alpha.[[t_1\ :\ \alpha]] \text{ in } [[t_2\ :\ \tau]]
\end{aligned}$$

Here, "def $x = \tau$ in c" is syntactic sugar for "let $x = \lambda\alpha.(\alpha = \tau)$ in c".

In practice, there are a lot more constraint generation rules, because sslang IR contains a lot more structures. In particular, built-in types, algebraic datatypes, extern declarations, mutual & non-mutual recursion, pattern matching and user type annotations all require non-trivial constraint generation rules. Due to the reference feature in sslang, in order to preserve type safety, *value restriction* is also implemented on let-bindings, which essentially means that only bindings for *values* (i.e. variables, data constructors, literals and lambdas) will be generalized.

2.3 Constraint Solving

The constraint solver of Version 0 is close to the final Version 2 in theory, so I will defer the implementation details of constraint solver until Section 4. For now, it suffices to know that it uses union-find as the underlying unification algorithm and represents type unification variables as union-find points. The type inference algorithm is then simply implemented as: given a term t, create a fresh unification variable α (i.e. a union-find point), generate constraint for the judgement $t : \alpha$, solve the constraint, and finally reconstruct the type that α represents in the union-find graph after all the unifications performed during constraint solving. This type is the inferred type of the term t.

2.4 Problem: Inference vs. Elaboration

Given a term t, the type inference algorithm described above can output a boolean answer for whether t typechecks, and if the output is "True", we can retrieve the inferred type for term t. However, for many language implementations including sslang, type inference is not enough; they require a *type elaboration* procedure.

Type elaboration refers to the process of transforming an *implicitly-typed* program representation into an *explicitly-typed* program representation. An explicitly-typed program representation generally means that enough type information is contained inside the representation such that the type of any expression node can be immediately acquired and checked. Type elaboration would need to do even more when the type system becomes more complicated; for example, implicit dictionary evidences need to be elaborated to support typeclasses, and type functions and applications would need to be elaborated to elaborate to a System F representation. For our purpose in sslang, type elaboration is not extremely complicated.

In sslang, after type inference, each node of IR representation should have a `Type` field that records its inferred type. This is not possible with just type-inference, where we only get access to the top-level unification variable. In order to solve the type elaboration problem, Version 1 and Version 2 offer distinct approaches. In the following sections, we will analyze the techniques and tradeoffs.

3 HM Type Elaboration in Applicative Style

As Pottier explains in [3], The type inference problem determines whether a program is well-typed. It does not solve the elaboration problem, which is the problem of constructing an explicitly-typed program representation. If the solver returns only a Boolean answer, how does one construct a type-annotated term t'? How does one obtain the necessary type information? A "satisfiable" or "unsatisfiable" answer is not nearly enough.

A good idea is to let the solver produce not only the satisfiability answer, but also some witness W. One would then write an elaboration function, mapping term t and witness W to an explicitly-typed term t'. This approach cuts the implementation into 3 parts: constraint generation, constraint solving and elaboration. We will return to this idea in Version 2, described in Section 5.

Pottier believes that this approach is rather "unpleasant", because the type inference and elaboration process is now split into three phases and only the second phase is independent of the programming language at hand. He wanted a more generic, separated and "elegant" approach. For this reason, he formalized and implemented the idea of "constraints with a value", where the first and third phase can be expressed together, and the second phase can be abstracted away as a generic constraint-solver library.

3.1 Constraints with a Value

The syntax of constraints-with-a-value is:

$$\begin{array}{lr}
C ::= \mathit{true} & \mathit{truth} \\
\quad\mid C \wedge C & \mathit{conjunction} \\
\quad\mid \tau = \tau & \mathit{equality} \\
\quad\mid \exists \alpha . C & \mathit{existential\ quantification} \\
\quad\mid \text{let } x = \lambda \alpha . C \text{ in } C & \mathit{constraint\ abstraction\ binding} \\
\quad\mid x\ \tau & \mathit{constraint\ abstraction\ instantiation} \\
\quad\mid \mathit{map}\ f\ C & \mathit{map\ constraint}
\end{array}$$

The only addition to the constraint language is the "$\mathit{map}\ f\ C$", where f is a meta-language function (e.g. a Haskell function). The semantics of this constraint is that it is satisfied when C is satisfied, and if the constraint C produces some value V, then $\mathit{map}\ f\ C$ produces the value fV.

The other constraints hold the same logical meaning as before, and also acquires a new meaning as producers of meta-language values. When satisfied, the values that they produce are:

- true: a unit value
- $C_1 \wedge C_2$: the pair (V_1, V_2), where V_1, V_2 are values produced by C_1, C_2 respectively
- $\tau = \tau$: a unit value
- $\exists \alpha . C$: the pair (T, V), where T is the witness/value/decoded type that must be assigned to the type variable α in order to satisfy C, and V is the value produced by C
- let $x = \lambda \alpha . C_1$ in C_2: a tuple of three values. First, the canonical form (i.e. decoded type scheme) of the constraint abstraction $\lambda \alpha . C_1$. Second, a value of the form $\Lambda \overrightarrow{\alpha} . V_1$ where V_1 is the value produced by C_1. Third, the value V_2 produced by C_2.
- $x\ \tau$: a vector $\overrightarrow{T}$ of decoded types. These are witnesses: they indicate how the type scheme associated with x must be instantiated in order to obtain the type τ.

3.2 Elaboration using constraint-with-a-value

With the constraint-with-a-value language, Pottier's key idea is to, while generating constraints as before, encode local elaboration logic as functions f and *map* the f's onto the values of the constraints. As such, the constraint generation logic and elaboration logic are merged into a single, big constraint, which is then solved by the constraint solver. In addition to a satisfiability answer, the constraint solver would output the value produced by the constraint, which would be the fully elaborated program representation.

Since the constraint ADT provides a *map*, Pottier proves that it is indeed a functor. Furthermore, it turns out that the constraint ADT is an applicative. This is why Pottier calls it type elaboration in applicative style.

I will not go further into the details of the implementations of such type elaboration procedure, because it is rather complicated. But here is a code snippet of the constraint generation + elaboration logic for a simple variable-expression:

```
hastype :: Expr Annotations -> Variable -> SolverM s (Co (Expr Type
    ))
hastype (Var x _) w =
inst x w <$$> \((gs, t), witnesses) -> Var x (subTVars gs witnesses
     t)
...
```

Note that `c <$$> f` is an infix version of the *map* f c constraint, `(gs, t)` is the type scheme $\forall\overrightarrow{gs}.t$, and `witnesses` is how `gs` are instantiated one by one.

Here, `hastype` takes an implicitly-typed `Expr Annotations` and a type variable that the expression should typecheck with and monadically returns a constraint with an explicitly-typed `Expr Type`. For the case where the expression is a `Var x`, it instantiates x with w (i.e. the x w constraint), and maps the type scheme and witnesses to an explicitly-typed IR expression.

Hopefully this clearly illustrate how the type elaboration works on a high-level. I will not go into further details.

3.3 Problem: Engineering Complexity of constraint-with-a-value

The theory of constraint-with-a-value brought me a lot of excitement in the beginning. The idea of combining constraint generation and elaboration logic seemed promissing and profound. In reality, after implementing it, I was not very happy about its level of engineering complexity. The constraint solver had to be written in a generic manner, making it difficult to understand (especially in Haskell). The result of combining constraint generation and elaboration was not the promised clarity but a rather disastrous mess.

For these reasons, I abandoned this constraint-with-a-value idea, and began Version 2 using the simpler 3-phase approach.

4 Constraint-based Type System in sslang

The final iteration of the constraint-based type system in sslang is Version 2. The theory remain close to the HM(X) described in *The Essence of ML Type Inference*, and the engineering is inspired by the open-source Elm compiler.

All related code is located in `IR.Constraint` module. To use it in the compiler pipeline, the `Typechecking` submodule exposes a single function `typecheckProgram :: Program Annotations -> Pass (Program Type)`. The whole pass is separated into 3 phases: constraint generation, constraint solving and elaboration. The submodules that contains definitions used across all phases are:

- `Canonical`: define the canonical type system constructs (e.g. `Type`, `Annotation`, `Scheme`, etc.); some are synonyms to definitions in the `IR.Type` module
- `Type`: define the constraint-based type system constructs (e.g. `Constraint`, `Type`, `Descriptor`, `Rank`, etc.); these are the core data structures for the whole pass
- `Monad`: define the typechecking monad `TC a = StateT TCState (ExceptT Error IO) a`; notice that the innermost wrapped monad is `IO`, which is entered with `unsafePerformIO`; we use `IO` because we need mutable references like `IORef` for union-find
- `Error`: define the errors that could be thrown from the pass (e.g. infinite type, unification error, etc.)
- `UnionFind`: the crucial internal data structure used across the entire pass; many definitions in the `Type` module rely on this

The submodules for the constraint generation are:

- `Constrain`: exposes a function `run :: Program Annotations -> TC (Constraint, Program Variable)`, which performs constraint generation and inserts witnesses (i.e. Variable) into the program
- `Constrain.Program`: used by the `Constrain` module; exposes a function `constrain :: Program (Annotations, Variable) -> TC Constraint`, which generates constraint for a program
- `Constrain.Expression`: used by `Constrain.Program`; generates constraints for expressions
- `Constrain.Pattern`: used by `Constrain.Expression`; generates constraints for patterns
- `Constrain.Annotations`: used by `Constrain.Expression`; generates constraints for annotations

The submodules for the constraint solving are:

- `Solve`: exposes a function `run :: Constraint -> TC ()`, which performs unifications and solves the constraint; it throws error if it encounters errors (e.g. infinite type, unification error, etc.); a notable feature is that it does not throw on the first error but instead collects as many errors as it can and throw them together in a list; generalization technique is due to Kuan and Macqueen [1]
- `Instantiate`: contains type scheme instantiation logic
- `Unify`: a crucial module containing type unification logic; implemented in continuation-passing style (CPS); exposes the function `unify :: Variable -> Variable -> TC Answer`, where `Answer` is `Ok` or `Err Error.Type Error.Type`.
- `Occurs`: provides the function `occurs :: Variable -> TC Bool` that performs an occurs-check on the unification variable (i.e. checking whether the unification variable is an infinite type)

The submodules for the elaboration are:

- `Elaborate`: exposts a function `run :: Program Variable -> TC (Program Type)` , which transforms the witnesses (i.e. unification variables) to concrete, canonical types

5 Future Work

The switch to a constraint-based type system has already been beneficial for us. It provided a much more solid ground for typechecking features like user type annotations and patterns. In fact, it has detected a few incorrect unit test cases and revealed the incorrectness of how we had typechecked patterns in the old type inference algorithm. This has given us much more confidence in the type safety of sslang as a ML variant.

The next step is to go beyond ML and explore other useful type features. The one at the top of the list is type-class support. In previous semesters, I have explored the ML-style typechecking for typeclasses; now, I am starting to study the open-source Sixten + Sixty compilers on GitHub to understand how to implement type-classes in an elaboration approach. The inference techniques are from Peyton Jones et al. [2]. Another direction is linear type system. The 1st chapter of ATTAPL book and the Rust compiler are all great resources for inspirations.

A final remark is that I believe the sslang IR should be slightly reworked. As an explicitly-typed IR, I believe System-F is a more appropriate representation than the current sslang IR (i.e. having a type associated with each node). System-F is robust enough to remain well-typed up to reduction/rewrite, as we have noticed when discussing about the static inlining optimization. Also, System-F is a standard core language. It has a rich theoretical repertoir and is used by compilers like GHC, Sixten and Sixty. It also has a simple typechecking algorithm that one can run after each compiler pass. As a result, I will explore the idea of using System-F as IR.

References

[1] George Kuan and David MacQueen. Efficient type inference using ranked type variables. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, page 3–14, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *J. Funct. Program.*, 17(1):1–82, jan 2007.

[3] François Pottier. Hindley-milner elaboration in applicative style: Functional pearl. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, page 203–212, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Francois Pottier and Didier Rémy. *The Essence of ML Type Inference*, pages 389–489. 01 2005.