CPSC 539 Final Project

Brooke Dai, 2022W2

Code repo: https://github.com/brookedai/CPSC539-type-system

1. Overview

The three topics that I chose for my type system are:

1. Simple types with structure data
2. Hindley-Milner Type Inference
3. Refinement types

However, though I was able to add some scaffolding to support refinement types, unfortunately I was unable to finish implementing refinement type checking to this system. Therefore, the main feature of this type system is type inference, which can eliminate type-mismatch bugs without the user having to explicitly annotate types in the language. In this write-up, I will mostly focus on the Hindley-Milner type inference implementation; however, I will still briefly go over any refinement type scaffolding that has been added to stay faithful to what is present in the implementation.

2. Language

The language contains the main features of STLC with boolean and integer types, as well as conditional branching, inequalities, and simple arithmetic operations. The surface syntax of the language shown in Fig 2.1 is essentially the result of mapping the mathematical notations to Racket symbolic expressions, which is helpful for parsing.

t ::=		**terms:**
	x	variable
	$(lambda\ x\ T\ t)$	abstraction
	$(app\ t\ t)$	application
	$(succ\ t)$	successor
	$(pred\ t)$	predecessor
	$(iop\ t\ t)$	inequality
	$(aop\ t\ t)$	binary arithmetic
	$(if\ t\ then\ t\ else\ t)$	conditional
v ::=		**values:**
	$true \mid false$	boolean constants
	n	integer constants
	$(lambda\ x\ T\ t)$	abstraction value
iop ::=		**inequality operations:**
	$< \mid <= \mid = \mid >= \mid > \mid != $	
aop ::=		**arithmetic operators:**
	$+ \mid - \mid * \mid /$	
n ::=		**integer numbers:**
	$0, -1, 1, ...$	

Fig 2.1 SLTLC surface syntax

The values in this language are the boolean values of $true$ and $false$, integers, and functions (lambda abstractions). Variables are bound to values in functions. Applications apply functions to values. *succ* adds 1 to a given integer, and *pred* subtracts 1 from a given integer.

Operators in SLTLC are written in prefix notation. Inequalities compare two terms and gives a *Bool* result; for example, $(> 1\ 2)$ evaluates to $false$ since 1 is not greater than 2. Binary arithmetic operators take two terms and gives an Int result; for example, $(+ 1\ 2)$ evaluates to 3.

Conditionals check a boolean condition and evaluates to the *then* branch if the condition is $true$, and the *else* branch if the condition is $false$. The *then* and *else* branches must be of the same type.

$T ::=$		**types:**
	$auto$	infer type
	$(B\ p)$	refinement type
	$(T \to T)$	function type
$B ::=$		**base types:**
	$Bool$	boolean
	Int	integer
	X	type variable

Fig 2.2 SLTLC refinement and base types

$p ::=$		**refinement predicates:**
	q	predicate
	κ	liquid type variable
$q ::=$		**predicates:**
	x	variable
	$true \mid false$	boolean constants
	$(iop\ expr\ expr)$	inequalities
	$(lop\ q\ q)$	logic operations
$iop ::=$		**inequality operations:**
	$< \mid <= \mid = \mid >= \mid > \mid != $	
$lop ::=$		**logic operations:**
	$and \mid or$	
$expr ::=$		
	$(laop\ expr\ expr)$	linear arithmetic
	n	integer numbers
	x	variable
$laop ::=$		**linear arithmetic operators:**
	$+ \mid -$	
$n ::=$		**integer numbers:**
	$0, -1, 1, ...$	

Fig 2.3 SLTLC Refinement logic

Every type in SLTLC is a refinement type as specified in Fig 2.2, i.e. it has a base type B and a refinement p (or, in the case of functions, both the argument and the return type are refinement types). The user can also put *auto* as the type, which will tell the type checker to automatically infer the type.

The base types are $Bool$ and Int, and type variables represent unknown base types.

Refinements are drawn from QF-LIA (quantifier-free linear integer arithmetic) (3) and are specified in Fig 2.3.

$\Gamma ::=$		**contexts:**
	$\emptyset$	empty context
	$\Gamma, x : T$	term variable binding
$C ::=$		**constraint set:**
	$\emptyset$	empty constraint set
	$C \cup \{T_1 = T_2\}$	type constraint equation
$\mathcal{X} ::=$		**type variable set:**
	$\emptyset$	empty type variable set
	$\mathcal{X} \cup \{X\}$	type variable set

Fig 2.4 Contexts used during typechecking

During type checking, the type checker will keep track of several contexts. Γ is a mapping from variables to their type. C is a set of constraints of the form $\{T_1 = T_2\}$, where T_1 and T_2 are refinement types. $\mathcal{X}$ keeps track of all type variables introduced in subderivations.

3. Typing Rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\emptyset} \{\}} \quad \text{(CT-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2 \mid_X C}{\Gamma \vdash (lambda\ x\ T_1\ t_2) : T_1 \to T_2 \mid_{\mathcal{X}} C} \quad \text{(CT-ABS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2 \\ \mathcal{X}_1 \cap \mathcal{X}_2 = \mathcal{X}_1 \cap FV(T_2) = \mathcal{X}_2 \cap FV(T_1) = \emptyset \\ X \notin \mathcal{X}_1, \mathcal{X}_2, T_1, T_2, C_1, C_2, \Gamma, t_1, \text{or } t_2 \\ C' = C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}\end{array}}{\Gamma \vdash (app\ t_1\ t_2) : X \mid_{\mathcal{X}_1 \cup \mathcal{X}_2 \cup \{\mathcal{X}\}} C'} \quad \text{(CT-APP)}$$

$$\frac{\Gamma \vdash t_1 : T \mid_{\mathcal{X}} C \qquad C' = C \cup \{T = Int\}}{\Gamma \vdash (succ\ t_1) : Int \mid_{\mathcal{X}} C'} \quad \text{(CT-SUCC)}$$

$$\frac{\Gamma \vdash t_1 : T \mid_{\mathcal{X}} C \qquad C' = C \cup \{T = Int\}}{\Gamma \vdash (pred\ t_1) : Int \mid_{\mathcal{X}} C'} \quad \text{(CT-PRED)}$$

$$\frac{\begin{array}{c}iop \in \{<, <=, =, >=, >, ! =\} \\ \Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2 \\ \mathcal{X}_1, \mathcal{X}_2 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup \{T_1 = Int, T_2 = Int\}\end{array}}{\Gamma \vdash (iop\ t_1\ t_2) : Bool \mid_{\mathcal{X}_1 \cup \mathcal{X}_2} C'} \quad \text{(CT-IOP)}$$

$$\frac{\begin{array}{c}aop \in \{+, -, *, /\} \\ \Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \qquad \Gamma \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2 \\ \mathcal{X}_1, \mathcal{X}_2 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup \{T_1 = Int, T_2 = Int\}\end{array}}{\Gamma \vdash (aop\ t_1\ t_2) : Int \mid_{\mathcal{X}_1 \cup \mathcal{X}_2} C'} \quad \text{(CT-AOP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash t_1 : T_1 \mid_{\mathcal{X}_1} C_1 \\ \Gamma \vdash t_2 : T_2 \mid_{\mathcal{X}_2} C_2 \qquad \Gamma \vdash t_3 : T_3 \mid_{\mathcal{X}_3} C_3 \\ \mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \text{ nonoverlapping} \\ C' = C_1 \cup C_2 \cup C_3 \cup \{T_1 = Bool, T_2 = T_3\}\end{array}}{\Gamma \vdash (if\ t_1\ then\ t_2\ else\ t_3) : T_2 \mid_{X_1 \cup X_2 \cup X_3} C'} \quad \text{(CT-IF)}$$

$$\frac{n \in \mathbb{I}}{\Gamma \vdash n : Int \mid_{\emptyset} \{\}} \quad \text{(CT-INT)}$$

$$\Gamma \vdash true : Bool \mid_{\emptyset} \{\} \quad \text{(CT-TRUE)}$$

$$\Gamma \vdash false : Bool \mid_{\emptyset} \{\} \quad \text{(CT-FALSE)}$$

Fig 3.1 Constraint Typing Rules

The constraint typing rules for Hindley-Milner generally follow those described in TAPL Chapter 22 (1). Refinement types have been abbreviated to their base types; i.e.

the refinement type ($Int\ true$) is written as Int in the constraint typing rules above. When constraints are collected, the keyword *auto* will have already been replaced by a type variable and liquid type variable pair during the parsing stage. During H-M inference, any inferred types will be given the refinement *true*, and any refinements that were specified by the user are left untouched and carried through the algorithm.

4. Implementation

The typechecker for SLTLC is implemented in the following stages:

1. Parsing (surface syntax → AST)
2. H-M Constraint Generation (AST → initial type, constraints)
3. H-M Unification (constraints → type mapping)
4. Substitution (initial type, type mapping → type)

If the SLTLC program successfully typechecks, the typechecker will return the type of the program. Otherwise, an error will occur.

A complete typechecker would also have the following stages, but unfortunately they are not present in the current implementation:

5. Liquid Type Constraint Generation
6. Liquid Type Constraint Solving

Notes: See main.rkt:main for examples of typechecking programs. All H-M stages (stages 2-4) are combined in `sltlc-typechecker.rkt:typecheck`.

4.1 Parsing

Code: `stages/sltlc-parser.rkt:parse`.

This stage parses surface syntax into an AST representation of the program. This is done by matching the term to the corresponding rule in the grammar, and recursively calling the parse function on the subterms of the term. Whenever there is an *auto* type, a refinement type pair ($T\ K$) is generated, where X is a fresh type variable and K is a fresh liquid type variable. Fresh type variable names are generated by using a counter with a T prefix, i.e. $T1$, $T2$, ... Fresh liquid type variable names are similarly generated with a K prefix.

4.2 H-M Constraint Generation

Code: `stages/sltlc-typechecker.rkt:hm-gather-constraints`.

Given the AST generated from the parsing stage, this stage generates the set of type constraints for the program. This is done by traversing the AST through recursive calls and following the constraint typing rules in Fig 3.1 at each level of recursion. When a new type variable is introduced, its name is generated as described in section 4.1 using

a X prefix. The type of the overall program is also updated with any type variables generated from this stage; call this the *initial type*, or the type of the program without any type variable substitutions.

Note that the set of type variables $\mathcal{X}$ is included in the implementation to stay faithful to the typing rules, but is never really used since all generated type variables are unique.

4.3 H-M Unification

Code: `stages/sltlc-typechecker.rkt:unify`.

A solution to the constraints is generated through the unification algorithm described in TAPL Chapter 22 (1). In the implementation, there are a few extra cases to account for the case where a type variable is compared to a function type. By the end of the stage, we will have a mapping from each type variable to some type.

4.4 Substitution

Code: `stages/sltlc-typechecker.rkt:hm-substitute-inferred-types`.

The mapping produced from section 4.3 has an issue: the same type variable can appear as both a key and a value. If we performed substitution using this mapping directly, we would run into issues where types that should be the same would instead be represented by different type variables. An example of such a mapping might look like this (type variables shown only for clarity):

$$\begin{aligned}(T5 &\to X1)\\(T7 &\to X2)\\(X1 &\to T9)\\(X2 &\to T9)\end{aligned} \tag{4.4.1}$$

To solve this issue, the mapping is first applied to all its values. Note that this works because SLTLC does not support recursion. The result of this step for the previous example would look like:

$$\begin{aligned}(T5 &\to T9)\\(T7 &\to T9)\\(X1 &\to T9)\\(X2 &\to T9)\end{aligned} \tag{4.4.2}$$

After applying the mapping to itself, we can apply it to the program's initial type. For example, if we had the following initial type (refinements omitted for clarity):

```
(fun
 (fun (type-var 'T5)
      (type-var 'T7))
 (fun (type-var 'T9)
      (type-var 'X2)))
```

then the result of applying the previous mapping would look like:

```
(fun
  (fun (type-var 'T9)
       (type-var 'T9))
  (fun (type-var 'T9)
       (type-var 'T9)))
```

5. Example programs

5.1 Double

The following is the *double* function, a function that takes a function and a value and applies the function twice to the value, in SLTLC:

```
'(lambda f (auto -> auto)
   (lambda x auto
     (app f (app f x))))
```

The result of running this program through the type checker is:

```
(t:fun
 (t:fun
  (lt:ref-type (t:type-var 'T37) (lt:ref-type-var 'K34))
  (lt:ref-type (t:type-var 'T37) (lt:ref-type-var 'K36)))
 (t:fun
  (lt:ref-type (t:type-var 'T37) (lt:ref-type-var 'K38))
  (lt:ref-type (t:type-var 'T37) #t)))
```

Ignoring the refinements, this result corresponds to the standard textbook-typing of *double*, $(\alpha \to \alpha) \to \alpha \to \alpha$, where α = `T37`.

Small implementation note: the *t* prefix indicates parts of types that are unrelated to refinement types (from `models/sltlc-types.rkt`), and the *lt* prefix indicates types related to refinement types (from `models/sltlc-liquid-types.rkt`).

5.2 Type mismatch

An example of where the type system would catch a type mismatch is the following function, which applies a function that tries to increment its argument to a boolean value:

```
'(app (lambda x auto (succ x)) true)
```

The result of running this program through the type checker is an error, which happens during the unification step:

```
hm-unify-cs: incompatible types: #<t:bool:> #<t:int:>
  context...:
   /path/to/CPSC539-type-system/stages/sltlc-typechecker.rkt:216:2:
    ↪  hm-unify-cs
   [repeats 1 more time]
   /path/to/CPSC539-type-system/stages/sltlc-typechecker.rkt:257:0:
    ↪  typecheck
   /path/to/CPSC539-type-system/main.rkt:100:0
   body of "/path/to/CPSC539-type-system/main.rkt"
```

6. Future Work

There are many things that could be added to improve the usefulness of this type system. Some are listed here:

1. **Redesign refinement predicates**: As noticed in class, the current system tries to refine program variables directly instead of using a value variable ν as described in the Refinement Types paper (2). This is problematic in situations where there is no program variable to use (eg. for the return type of a function). The refinement grammar should be redesigned to include value variables so that type refinements are decoupled from program variables.

2. **Finish support for refinement types**: Since the refinements in the types currently are not being typechecked or inferred at all, they are more of an overhead than a help. However, adding proper support for refinement types would allow the type system to check for stronger properties in the program and provide more specific type information to the user.

3. **Add ascription**: When I was designing SLTLC, I tried to reconcile the concepts of type inference (which aims to reduce user annotation for types) and type refinement (needs user to specify the refinements that they want for their types) and ended up with the rather awkward compromise of only including type annotation for function arguments in the surface syntax. A major feature that can bridge these concepts is type ascription, which would give the user the choice of where to specify types.

4. **Add more features to SLTLC**: SLTLC seems to be a bit boring as a language right now and not very expressive. Adding features like recursion, sum/product types, etc. would make it more interesting.

7. References

[1] Benjamin C. Pierce. 2002. Types and Programming Languages (1st. ed.). The MIT Press.

[2] Jhala, R., & Vazou, N. (2020). Refinement Types: A Tutorial. ArXiv:2010.07763 [Cs]. https://arxiv.org/abs/2010.07763

[3] SMT-LIB The Satisfiability Modulo Theories Library. (n.d.). Smtlib.cs.uiowa.edu. Retrieved April 26, 2023, from https://smtlib.cs.uiowa.edu/logics-all.shtml#QF_LIA