23-2 Programming Language Theory

HW5 due 12.09 Sat

22100579 Jinju Lee

1. One-page Summary

Chapter: Introduction to types (p.p 108 - 113)

Objects were not included in SMoL due to the lack of universality, the absence of standardization in their implementation details, and the ability to achieve similar functionality through desugaring. Now, the focus shifts to types in SMoL, where the term refers to static checks performed based on the program source. Types provide guarantees without running the program, addressing scenarios where execution is expensive, impossible, or unsafe. While not universally agreed upon, common ground exists for introducing types in SMoL.

Types are introduced through a type checker in a basic interpreter model. Types serve as abstractions of run-time values, consolidating similarities and distinctions. The initial interpreter includes expressions and binary operators like addition (plus). The calculation function (calc) evaluates expressions, while the type checker (tc) assesses program type correctness.

Initially, the type checker unconditionally returns #true for all programs, rendering them seemingly type-correct. To enhance the model, new types and operations, such as string concatenation (++), are introduced. The type checker is adjusted accordingly, enabling more meaningful error possibilities in type checking. The modified type checker assesses programs based on introduced types and operations, providing a more robust approach to type correctness.

The core problem is that the initial type checker lacks information about the types of sub-expressions. To address this, the type checker needs to calculate and return actual types instead of just checking correctness. The type "checker" becomes a type calculator, operating over the universe of abstracted values (types). This upgrade involves using a richer type signature (tc : (Exp -> Type)), where Type includes numT and strT. The type checker follows the SImPl schema, operates with "weak" values, and strengthens the inductive hypothesis by returning actual types instead of Booleans.

A concise notation, |- e : T, is introduced to express that expression e has type T. Axioms are presented for numeric (Num) and string (Str) expressions. For Booleans, both an explicit enumeration and a general rule (Bool) are provided. Typing rules, such as for addition, are expressed compactly in the form of antecedents and consequents. The antecedents specify conditions, and the consequents indicate the resulting type. This notation simplifies the representation of typing information compared to the corresponding code.

Chapter: Growing types: Division, Conditionals (p.p 114 - 121)

Handling division in programming involves addressing its partial nature when the denominator is zero. Three strategies are:

- Option Type: Division returns a type capturing its partiality, necessitating explicit checks for valid results.
- Restricting Arguments: Division is allowed only with non-zero second arguments, requiring proof from callers.
- Exception Handling: Division maintains the same type as other numeric operations, handling exceptional cases through exceptions or errors.

Languages often opt for the third strategy for its pragmatism, while some explore the first two options, placing the burden on programmers.

Types can be seen as a static discipline, akin to parsing, where static judgments about programs are made before runtime. This perspective views types as an extension of the idea behind parsing. In computability theory terms, parsing is usually context-free, while types often involve context-sensitive constraints. Separating these checks into distinct phases, with parsing preceding type-checking, is motivated by simplicity and reduced complexity in handling well-formed programs.

When applying type rules to a program, we recursively compose them to determine the program's type. Using conditional rules, we analyze each part of the program, marking terms that match axioms in green. The resulting tree of judgments confirms whether the program successfully type-checks. This process mirrors the execution pattern of the type-checker, emphasizing the efficiency of pattern-matching in skipping detailed passing and returning steps.

Type errors occur when constructing type judgments and the rules or axioms do not match the program structure. A complete tree, or judgment, is formed when every part aligns with established rules and axioms. Incomplete trees result from mismatches, signifying an inability to judge the initial expression.

The type rule for conditionals checks conditions and both branches, requiring them to have the same type denoted by a placeholder "U." Examples illustrate the rule's application, detecting type errors when conditions are not met. The addition of functions involves introducing lambda abstraction and function application constructs. Desugaring let into lambda is suggested as a potential extension to the type-checker.

The type-checker and evaluator follow different traversal strategies. While an evaluator runs over simple values, a type-checker traverses both branches of conditionals, even when only one branch may execute. This difference arises from the conservative nature of type-checking, as it must handle programs that might run at various times. The relationship between type-checking and testing is highlighted, with type-checking providing lightweight code coverage and testing offering deep coverage at the value level.

2. Question

Our FAE implementation does not support type checking now. So, the following code is not processed well.

```
{+ 4 {fun {x} x}}
```

(1) Define axioms and rules with antecedents and consequent for FAE that supports two numbers for arithmetic operations:

(+ and -)

a. Axioms

$|-n: Num$ //for numbers

$|- \{fun\ \{x\}\ e\}: Fun\ (Num, T)$ //for functions

(T represents the type of the expression e.)

b. Rules

$|-e1: Num\ |-e2: Num\ /\ |-\{+\ e1\ e2\}: Num$ //for addition

$|-e1: Num\ |-e2: Num\ /\ |-\{-\ e1\ e2\}: Num$ //for subtraction

(2) Deploy the judgements as in the textbook page 116 for the case:

{+ 4 {fun {x} x}}

a. rhs is number, so it becomes…

$|-\ 4: Num$

b. lhs is functions, so it becomes…

$|-\{fun\{x\}\ x\ \}: Fun$

c. function param and return type x is number, so it becomes…

$|-\ \{fun\ \{x\}\ x\ \}\ 4: Num$

d. entire expression is addition, so it becomes…

$|-\ \{+\ 4\ fun\ \{x\}\ x\ \}\ : Num$