1 Input Language

In this assignment you will be analyzing programs written in a very simple language, PCF, which is essentially the Simply-Typed Lambda Calculus with conditionals, a fixpoint operator, and an extra base type (integer), with accompanying operations. Table 1 describes the syntax of the language. We assume call-by-value semantics.

<table>
<thead>
<tr>
<th>variable</th>
<th>$e_{PCF} ::= x$</th>
<th>⟨term⟩ ::= ⟨var⟩</th>
<th>$e_{PCF} ::= x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>int constant</td>
<td>⟨i⟩</td>
<td>⟨num⟩</td>
<td>⟨i⟩</td>
</tr>
<tr>
<td>abstraction</td>
<td>⟨λx. e⟩</td>
<td>⟨(lambda var term)⟩</td>
<td>⟨λx : τ, e⟩</td>
</tr>
<tr>
<td>application</td>
<td>⟨e @ e⟩</td>
<td>⟨(term) (term)⟩</td>
<td>⟨e @ e⟩</td>
</tr>
<tr>
<td>int operation</td>
<td>⟨e ⊕ e⟩</td>
<td>⟨(op) (term) (term)⟩</td>
<td>⟨e ⊕ e⟩</td>
</tr>
<tr>
<td>conditional</td>
<td>⟨if e e⟩</td>
<td>⟨(if (term) (term) (term))⟩</td>
<td>⟨if e e⟩</td>
</tr>
<tr>
<td>fixpoint</td>
<td>⟨fix e⟩</td>
<td>⟨(fix (term))⟩</td>
<td>⟨fix e⟩</td>
</tr>
<tr>
<td>lifting</td>
<td>⟨lift e⟩</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$⟨var⟩ ::= a \mid b \mid \ldots \mid z$

$⟨num⟩ ::= -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots$

$⟨op⟩ ::= + \mid * \mid < \mid \ldots$

Table 1: Term grammars for $PCF$, its ASCII representation, and $PCF$

Free variables (variables not bound by any $\lambda$-abstraction) will be considered as inputs to the program. Their value is unknown at compile time, but will be part of the environment that the program executes in. A term (program) in $PCF$ returns a single result, the value that it evaluates to.

The programs that you will be working on are provided in an ASCII encoding of the $PCF$ language, also described on Table 1. The terms in this representation are written in a LISP-like, prefix notation style, which ensures that the parsing is unambiguous and straightforward to implement (any LISP parser should work, for instance Peter Norvig’s lis.py).

2 Constraint Systems

A constraint system is a framework for expressing relations between unknown values. Such frameworks can be used to express the constraints arising during the analysis of a program. The resulting system of constraints can be fed to a decision procedure specifically designed for that constraint system, which will produce some (hopefully small) solution. That solution can then be used to reason about the properties of the original program. Constraint systems differ in their expressiveness, which affects both how accurately the constraints can model the desired properties of the program (the limitations of a less expressive system may

---

1The language used in this assignment differs slightly from the usual definition of $PCF$: We provide arithmetic operations directly in the syntax, while most other sources will only provide a successor operation, and define the rest using that.
require us to over-approximate in order to maintain soundness), and how efficiently a system of constraints
can be solved.

In this assignment, you will be using the following two constraint systems:

2.1 Conditional Unification

This system supports constraints of the following forms:

- (Structural) Equality: \( a = b \)
- Conditional (Structural) Equality: \( a \succ b \), which is equivalent to \( a = \bot \lor a = b \)

References

- Su & Aiken, “Entailment with Conditional Equality Constraints” (section 2)
- Steensgaard, “Points-to Analysis in Almost Linear Time”

2.2 Set Constraints

We will be using the LR-class of set constraints described in class, with conditional constraints.

References

- Lecture 7 slides
- Fähndrich et al., “Partial Online Cycle Elimination in Inclusion Constraint Graphs” (sections 2.1–2.4)

3 Binding-Time Analysis

The purpose of Binding-Time Analysis (BTA) is, given information about the availability of data at compile
time, to decide which parts of a program can be safely evaluated statically (i.e., which variables and expres-
sions can be “bound” to statically computed values, hence the name). Constant values in the program, and
any sub-expressions that can be safely reduced (evaluated) at compile time are bound “early”, while the rest
(i.e., those parts that depend on user input) are bound “late”.

BTA’s job is to annotate parts of the program according to their binding time. That information can
later be used by the compiler, to perform the actual (partial) evaluation. The only code that would be
emitted after such an optimization is for parts of the program that couldn’t be evaluated statically.

For example, consider the lambda term \( \lambda x . x \wedge y \wedge z \), where \( y \) and \( z \) are derived from user input. The
top-level application can be safely reduced at compile time, giving the term \( z \wedge y \). This last term cannot be
reduced any further without knowledge of the user input.

The fundamental requirement of a correct BTA is soundness: the analysis should only assign “early”
binding to a term (program fragment) if that term will reduce in the same way on any execution of the
program (regardless of the program’s input). The opposite is not necessarily true, i.e. it’s acceptable (and
usually unavoidable) to miss some optimization opportunities, and defer some computation to run time, even
if it could have been carried out safely at compile time.

A partially evaluating compiler for \( PCF \) requires three kinds of annotations:

- binding-time annotations on bound variables
- binding-time annotations on every instance of a language operator (we use an overbar for early binding
  and an underline for late binding)
- special “lift” operations added as needed, to mark the points where a statically computed value needs
to be inserted into a dynamic context (i.e., emitted as code)
The set of BT-annotated PCF-terms forms the language PCF, described in Table 1.

Not all PCF-terms are correct w.r.t. to the soundness requirements of BTA. These requirements can be encoded on top of the base language using a custom type system, outlined in Figure 1. Note that this type system is separate from the regular type system of the Simply-Typed Lambda Calculus (you can assume that regular type checking/inference has already been performed, and we are operating on well-typed PCF-terms).

\[ \tau ::= S \mid D \mid \tau \rightarrow \tau \]

Figure 1: Binding-time type grammar for PCF

The meaning of the types in this type system is as follows:

- **S**: statically computable values of the base type (integer), e.g. 1, \((\lambda x. x + 3) \@ 12\).
- **D**: value of any type, that cannot be evaluated at compile time, e.g. \(y, f \@ 3\), where \(y\) and \(f\) are derived from program input. Note that we don’t care about the structure of such a “dynamic” type, i.e. if it’s a base value or a function, only that it can’t be evaluated statically.
- **\(\tau \rightarrow \tau\)**: function that can be applied to an argument at compile time, e.g. \(\lambda x. x + 3\). Note that applying such a function statically does not necessarily produce a static value, e.g. in the case of \((\lambda x. x \@ y) \@ z\), where \((\lambda x. x \@ y)\) can be typed as \(D \rightarrow D\): we can apply it to \(z\), thus producing \(z \@ y\), which is of dynamic type \(D\).

The type-checking rules in Table 2 outline an algorithm to verify the consistency of the BT-annotations on a PCF-term: Any term for which we can build a type derivation tree starting with an empty context \((\Gamma = \emptyset)\) can be safely evaluated according to its BT-annotations. Note that a term which doesn’t typecheck may still be safe for partial evaluation—this is a side-effect of the limited expressiveness of the type system (which also ensures that type checking is tractable). Also note that the only expressions that can be lifted are those of base type (i.e., not abstraction values).

The actual analysis involves translating these type checking rules into type inference rules, which allow us to reduce the problem of assigning BT-annotations to solving a system of constraints. A BTA analysis is designed in such a way that any solution of the constraints describes a placement of annotations that makes the term typable according to the rules above. To make the most of the partial evaluator, the solution to those constraints should minimize the number of dynamic annotations.

In this assignment, you will be implementing Henglein’s Binding-Time Analysis (described in the publication listed below), which is based on conditional unification. Note the our setting differs slightly from the assumptions made in Henglein’s paper: we consider all free variables to be of dynamic type.

**References**

- Henglein, “Efficient Type Inference for Higher-Order Binding-Time Analysis”

4 Control Flow Analysis

The goal of Control Flow Analysis (CFA) is to produce a sound approximation of a program’s call graph, i.e. decide for every invocation statement in the program, which functions can potentially be called when that statement executes. The soundness requirement means that the analysis can never miss a call target if it could ever appear in an execution, but is allowed to include targets that cannot actually arise. This problem becomes especially challenging for a higher-order language like PCF, where functions are first-class values.

In this assignment, you will be implementing a CFA algorithm for PCF, using set constraints, similar to the example discussed in class. In addition to the cases covered in lecture, you will need to handle conditionals and the fixpoint operator (integer constants and operations don’t play a role in CFA), as explained below. You can also assume that no \(\lambda\)-abstractions are passed as inputs to the program.
The lambdas that can reach the result of an “if” condition are exactly those to which either of the two branches can evaluate: $L(e_2) \cup L(e_3) \subseteq L(\text{if } e_1 e_2 e_3)$.

The “fix” operator is the only construct in PCF capable of introducing recursion. It operates by taking a function value and calling it with a specially constructed term, so that if the body of the function ever uses its argument, the function itself gets called recursively to provide it. The evaluation rule involving “fix” looks like this:

$$\text{fix } \lambda x. e \rightsquigarrow [x \mapsto \text{fix } \lambda x. e]e$$

where $[a \mapsto b]c$ denotes the syntactic replacement of every occurrence of $a$ inside of $c$ with $b$.

For this part of the assignment, we will assume that the only place where a fixpoint operator can appear is around a syntactic lambda, i.e. we will never have to handle code like this: $\text{fix } (\text{<expr> <expr>})$ (which is acceptable in the general case). When processing an instance of “fix” like $\text{fix } \lambda x. e$, we propagate any lambda reaching the body $e$ of the wrapped function to every instance of its bound variable (to emulate the function calling itself), i.e. $L(e) \subseteq L(\text{fix } \lambda x. e)$.

References
- Lecture 7 slides

5 Instructions

1. Build a parser for the ASCII representation of PCF, described in Section [1].

2. For each of these two analyses:
   - Henglein’s conditional unification-based Binding-Time Analysis
   - Set constraint-based Control Flow Analysis (as described in class)
do the following:

(a) Write a constraint extractor, i.e. code that walks the program AST and collects the relevant constraints for each syntactic construct.

(b) Implement a solver for the corresponding constraint system.

(c) Run your system on each of the examples in Section 6. You can assume that these programs are valid according to the regular typing rules of the simply-typed lambda calculus (although the analyses don’t actually require this). All λ-abstractions in the programs should be distinct, so you don’t need to do any renaming.

(d) Print out your results according to the output format described in Section 7.

3. Write a report where you comment on how the analysis performed on each of the examples, and analyze any sources of imprecision you encountered. Feel free to include any other items you’d like to report (e.g., implementation difficulties, general observations on constraint-based program analysis).

4. Email your report and the output files from step 2d to cs357-aut1314-staff@stanford.edu.

6 Input Programs

These examples are also available on the class website in text format.

fib.lam: Fibonacci numbers

```latex
(((fix (lambda f (lambda n
   (if (< n 3)
      1
      (+ (f (- n 1)) (f (- n 2)))))
   a)
```

poly.lam: Polymorphic behavior

Example of a function being applied on both static and dynamic data.

```latex
(((lambda f (+ (f 1) (f a))) (lambda x (+ x 1)))
```

powbound.lam: Power-of-2 upper bound

Returns the first power of 2 that is greater than the input a.

```latex
(((fix (lambda g (lambda x (lambda y
   (if (<= x y) y ((g x) (* y 2)))))
   a) 1)
```

prime100.lam: 100-th prime number

Calculates the 100-th prime number.

```latex
(((fix
   (lambda p (lambda n (lambda a
      ((lambda i (if (= n 100) a ((p (+ n 1)) (i (+ a 1))))
       (fix (lambda j (lambda x
          ((lambda f (if (= ((f x) 2) x) x (j (+ x 1))))
           (fix (lambda h (lambda y (lambda z
              (if (= (% y z) 0) y ((h y) (+ z 1)))))))))))
   1) 2)
```
diamond.lam: Function reachable on 2 paths
(((lambda i ((lambda f ((lambda g (+ ((i f) x) ((i g) 0)))
          (lambda b (+ b 2)))))
     (lambda a (+ a 1))))
  (lambda y y))

path.lam: Path-sensitive behavior
(((if z (lambda x (if (= x 1) (lambda a a) (lambda b b))
          (lambda y (if (= y 0) (lambda c c) (lambda d d)))) (if z 1 0)) 42)

parity.lam: Parity calculation
Code ported from Assignment 1.
(((fix (lambda f (lambda p (lambda v
      (if v ((f (- 1 p)) (& v (- v 1))) p)
   ))))))
  0) w)

7 Output Format
To present the results of each analysis on some input program, say foo.lam, you will need to create one output file for each analysis. The output files should have the same base name as the input program, but an extension specific to the analysis. For foo.lam, you would need to generate foo.bta and foo.cfa.

Each output file should contain one sub-term of the full program per line, followed by the facts that the analysis has produced regarding that sub-term, separated by whitespace. To avoid ambiguity in cases where a sub-term appears multiple times inside a program, you should print the terms according to an in-order, left-to-right traversal of the AST. The terms should be printed using the same representation as the input programs, and should not span multiple lines.

To make the instructions clear, we will demonstrate what the output files should look like for an input program foo.lam, which contains the following code:

((lambda x (+ x 4)) z)

7.1 Binding-Time Analysis
The output file should have the extension “.bta”. For every sub-term, print the word lift iff it should be wrapped with a “lift” operator. Additionally, if the term is not a variable or constant, print the word early if it should be evaluated at compile time (i.e., it represents the early-binding variant, denoted by an overbar in Table 2, or the word late if it should be evaluated at runtime (i.e., it represents the late-binding variant, denoted by an underline in Table 2).

For foo.lam, we would produce the file foo.bta, with the following contents:

x
4 lift
(+ x 4) late
(lambda x (+ x 4)) early
z
  ((lambda x (+ x 4)) z) early
7.2 Control Flow Analysis

The output file should have the extension “.cfa”. For every sub-term, print all the lambdas that flow to it. To refer to some lambda \( \lambda x. e \), you will use its unique bound variable name, \( x \).

For \texttt{foo.lam}, we would produce the file \texttt{foo.cfa}, with the following contents:

\[
x
4
(+ x 4)
(lamda x (+ x 4)) x
z
((lamda x (+ x 4)) z)
\]