Type Synthesis

CIS352 — Fall 2022

Kris Micinski
Type Synthesis

Useful in practice and very interesting!
- If your program typechecks, it is “type correct”
- No need to write annotations everywhere, that’s a pain
- How “really correct” type correctness is depends on the expressivity of the type system:
  - In STLC, it just tells you you get something of the right “shape”
  - In higher-order logics, you can do full verification
Question

Last lecture: Curry-Howard Isomorphism

Curry-Howard tells us that every type system is a logic: does that mean type synthesis is a kind of proof synthesis?

Answer: kind of, the term is not being synthesized directly, but a proof of the existence of the term is—assuming the type synthesis is sound.
What is the correct type?

\[(\text{lambda} \ (f) \ (\text{lambda} \ (x) \ (\text{if} \ (\text{if-zero?} \ (f \ x)) \ 1 \ 0))))\]

Is it:

(a) \(f = \text{int-}\rightarrow\text{int}, \ x = \text{int}\)
(b) \(f = \text{bool-}\rightarrow\text{int}, \ x = \text{bool}\)
(c) \(f = (\text{int-}\rightarrow\text{int})-\rightarrow\text{int}, \ x = \text{int-}\rightarrow\text{int}\)
What is the correct type?

\[(\lambda (f) \ (\lambda (x) \ (\text{if} \ (\text{if-zero?} \ (f \ x)) \ 1 \ 0)))\]

Is it:
(a) \( f = \text{int->int}, \ x = \text{int} \)
(b) \( f = \text{bool->int}, \ x = \text{bool} \)
(c) \( f = (\text{int->int})->\text{int}, \ x = \text{int->int} \)
(d) All of the above
Type Variables

(lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))

Lesson:
We can’t pick just one type. Instead, we need to be able to instantiate f and x whenever a suitable type may be found.
For example, what if we let-bind the lambda and use it in two different ways!?

(let ([g (lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))]
  (+ ((g (lambda (x) x)) 0) ((g (lambda (x) 1)) #f))

This usage requires f = nat->nat and x = nat

This usage requires f = bool->nat and x = bool
Generalizations

(l lambda (f) (lambda (x) (if (if-zero? (f x)) 1 0)))

Instead, we can keep a generalized type by using a type variable, allowing a good type inference system to derive (for this example, using type var T):

Type of f = T -> int
Type of x = T

Notice that this system demands we must be able to compare T for equality! This is actually nontrivial when we add polymorphism, but is simple in STLC (structural equality)
Constraint-Based Typing

The crucial trick to implementing type inference is to use a constraint-based approach. In this setting, we walk over each subterm in the program and generate a constraint

Unannotated lambdas generate new type variables, which are later constrained by their usages

Later, we will solve these constraints by using a process named unification
(define (build-constraints env e)
  (match e
    ;; Literals
    [(? integer? i) (cons `(,i : int) (set))]
    [(? boolean? b) (cons `(,b : bool) (set))]
    ;; Look up a type variable in an environment
    [(? symbol? x) (cons `(,x : ,(hash-ref env x)) (set))]
    ;; Lambda w/o annotation
    [`(lambda (,x) ,e)
     ;; Generate a new type variable using gensym
     ;; gensym creates a unique symbol
     (define T1 (fresh-tyvar))
     (match (build-constraints (hash-set env x T1) e)
       [(cons `(,e+ : ,T2) S)
        (cons `((lambda (,x : ,T1) ,e+) : (,T1 -> ,T2)) S)])]
    ;; Application: constrain input matches, return output
    [`(,e1 ,e2)
     (match (build-constraints env e1)
       [(cons `(,e1+ : ,T1) C1)
        (match (build-constraints env e2)
          [(cons `(,e2+ : ,T2) C2)
           (define X (fresh-tyvar))
           (cons `(((,e1+ : ,T1) ,e2+ : ,T2)) : ,X)
           (set-union C1 C2 (set `(= ,T1 (,T2 -> ,X))))))])]
    ;; Type stipulation against t--constrain
    [`(,e : ,t)
     (match (build-constraints env e)
       [(cons `(,e+ : ,T) C)
        (define X (fresh-tyvar))
        (cons `((,e+ : ,T) : ,X) (set-add (set-add C `(= ,X ,T)) `(= ,X ,t))))]
    ;; If: the guard must evaluate to bool, branches must be
    ;; of equal type.
    [`(if ,e1 ,e2 ,e3)
     (match-define (cons `(,e1+ : ,T1) C1) (build-constraints env e1))
     (match-define (cons `(,e2+ : ,T2) C2) (build-constraints env e2))
     (match-define (cons `(,e3+ : ,T3) C3) (build-constraints env e3))
     (cons `(((if ,(e1+ : ,T1) ,(e2+ : ,T2) ,(e3+ : ,T3)) : ,T2)
              (set-union C1 C2 C3 (set `(= ,T1 bool) `(= ,T2 ,T3))))))]}
At the end of constraint-building, we have a ton of equality constraints between base types and type variables

\[
tv0 = \text{int} \\
ty1 = tv0 \rightarrow tv2 \\
tv2 = tv3 \\
tv3 = tv4 \\
\]

(lambda (x : ty1) \ldots)

In this example, what is \(ty1\)?

Answer: think about constraints and equalities: \(ty1\) must be \(\text{int}\rightarrow\text{int}\)
;; within the constraint constr, substitute S for T
(define (ty-subst ty X T)
  (match ty
    [(? ty-var? Y) #:when (equal? X Y) T]
    [(? ty-var? Y) Y]
    ['bool 'bool]
    ['int 'int]
    ['(,T0 -> ,T1) `(,(ty-subst T0 X T) -> ,(ty-subst T1 X T))]))

(define (unify constraints)
  ;; Substitute into a constraint
  (define (constr-subst constr S T)
    (match constr
      [`(=` ,C0 ,C1) `(=` ,(ty-subst C0 S T) ,(ty-subst C1 S T))])
  ;; Is t an arrow type?
  (define (arrow? t)
    (match t
      [`(,_ -> ,_) #t]
      [_ #f]))
  ;; Walk over constraints one at a time
  (define (for-each constraints)
    (match constraints
      [(] (hash)
        [`(= ,S ,T) . ,rest] (cond [(equal? S T) (for-each rest)]
          [(and (ty-var? S) (not (set-member? (free-type-vars T) S)))
            (hash-set (unify (map (lambda (constr) (constr-subst constr S T)) rest)) S T)]
          [(and (ty-var? T) (not (set-member? (free-type-vars S) T)))
            (hash-set (unify (map (lambda (constr) (constr-subst constr T S)) rest)) T S)]
          [(and (arrow? S) (arrow? T))
            (match-define `(,S1 -> ,S2) S)
            (match-define `(,T1 -> ,T2) T)
            (unify (cons `=` ,S1 ,T1) (cons `=` ,S2 ,T2) rest))]
          [else (error "type failure")])))

Unification
Why Type Theory?

Why is type synthesis / checking useful?

- Can write **fully-verified** programs.
  - Cons: type systems are esoteric, complicated, academic, etc…
  - Popular languages (Swift, Rust, etc…) are *tending towards more elaborate type systems as they evolve*

- Type synthesis offers me “proofs for free:”
  - “If my program type checks it works” — **not** true in C/C++/…

- Less **mental burden**, like CoPilot (etc… tools), type systems can integrate into IDEs to use synthesis information in guiding programming
  - In some ways, this reflects the logical statements underlying the type system’s design (Curry Howard)
Our group: build the world’s fastest fixpoint solvers

Slog: data-parallel deductive logic (Horn-SAT)
- Scaled control-flow analysis up to 1000 cores of the Theta supercomputer

Ascent:
- Programming with lattices, parallelization of declarative analytics on large unified-memory machines
- Macro-embedded language in Rust
- Parallelization using Rayon
Fast Horn-SAT

Lots of applications use restricted finite-domain propositional logic:
- Transitive closure, triangle counting, k-clique, …

\[
((\text{path } x \ y) \leftarrow (\text{edge } x \ y)) \ ;; \ \text{Initial step}
\]
\[
((\text{path } x \ z) \leftarrow (\text{path } x \ y) \ (\text{edge } y \ z)) \ ;; \ # \ \text{Inductive rule}
\]

- We can do programs like these at the highest scale currently known
  - Transitive closure: thousands of cores on Theta, graphs w/ billions of edges

Also forms the basis for program analysis: on-the-fly reachability of a program’s control-flow graph:
- Scalable
# Fast transitive closure at scale

<table>
<thead>
<tr>
<th>Name</th>
<th>Graph Properties</th>
<th>Time (s) at Process Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Edges</td>
<td></td>
</tr>
<tr>
<td>FB-MEDIA</td>
<td>206k</td>
<td>96,652,228</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RING10000</td>
<td>10k</td>
<td>100,020,001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUITESPARSE</td>
<td>412k</td>
<td>3,354,219,810</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data-Parallel Structured Deduction

New language extending Datalog to S-expressions:
- All structures are deduplicated via a hash value
- Distributed through a cluster via that hash
- Data efficiently indexed to support operations on structures

Allows rich programming in a style that looks much closer to natural deduction

All programs compile to data-parallel relational algebra kernels implemented via all-to-all communication on top of MPI

Allows scaling structured logic programming up to hundreds/thousands of threads
Examples in Slog

```plaintext
; ref
(interp ?(clo (ref x) env)
  {interp {env-map env x}})
; lam
(interp ?(clo (lam x Eb) env)
  (clo (lam x Eb) env))
; app
[((interp !(clo Ef env)
  (clo (lam x Eb) env')))
 (= env'' (ext-env env' x (clo Ea env')))
 (interp !(clo Eb env'') v)
 -->
 (interp ?(clo (app Ef Ea) env) v)]

; ref
(interp ?(clo (ref x) env)
  {env-map env x})
; lam
(interp ?(clo (lam x Eb) env)
  (clo (lam x Eb) env))
; app
[((interp !(clo Ef env)
  (clo (lam x Eb) env')))
 (interp !(clo Ea env) Eav)
 (interp !(clo Eb (ext-env env' x Eav)) v)
 -->
 (interp ?(clo (app Ef Ea) env) v)]
```

Fig. 5. Two CE (closure-creating) interpreters in SLog; for CBN eval. (left) and CBV eval. (right).
Control-Flow Analysis of the \(\lambda\)-calculus in Slog

```scheme
;; Eval states
[(eval (ref x) k c)
  -->
  (ret {store (addr x c)} k)]
[(eval (lam x body) k c)
  -->
  (ret (clo (lam x body) c) k)]
[(eval (app ef ea) k c)
  -->
  (eval ef (ar-k ea (app ef ea) c k) c)]

;; Ret states
[(ret vf (ar-k ea call c k))
  -->
  (eval ea (fn-k vf call c k) c)]
[(ret va (fn-k vf call c k))
  -->
  (apply call vf va k c)]
[(ret v (kaddr e c))
  (store (kaddr e c) k)
  -->
  (ret v k)]

;; Apply states
[(apply call (clo (lam x Eb) _) va k c)
  -->
  (eval Eb (kaddr Eb c') c')
  (store (kaddr Eb c') k)
  (store (addr x c') va)
  (= c' {tick !(do-tick call c)}))]

;; Propagate free vars
[(free y (lam x body))
  (apply call (clo (lam x body) clam) _ _ c)
  -->
  (store (addr y {tick !(do-tick call c)})
    {store (addr y clam)})]
```
## CFA of $\lambda$-calculus vs. Souffle

<table>
<thead>
<tr>
<th>Term Sz.</th>
<th>Iters</th>
<th>Cf. Pts</th>
<th>Sto. Sz.</th>
<th>8 Processes</th>
<th>64 Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Slog</td>
<td>Soufflé</td>
</tr>
<tr>
<td>8</td>
<td>1,193</td>
<td>98,114</td>
<td>23,413</td>
<td>00:01</td>
<td>01:07</td>
</tr>
<tr>
<td>9</td>
<td>1,312</td>
<td>371,010</td>
<td>79,861</td>
<td>00:02</td>
<td>14:47</td>
</tr>
<tr>
<td>10</td>
<td>1,431</td>
<td>1,441,090</td>
<td>291,317</td>
<td>00:06</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1,550</td>
<td>5,678,402</td>
<td>1,107,957</td>
<td>00:27</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1,669</td>
<td>22,541,634</td>
<td>4,315,125</td>
<td>02:14</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1,788</td>
<td>89,822,530</td>
<td>17,022,965</td>
<td>12:17</td>
<td></td>
</tr>
</tbody>
</table>

### 3-k-CFA
Strong Scaling on Theta
coursefeedback.syr.edu