HOT-Compilation: Typechecking F_ω

*TA: Akiva Leffert - aleffert@andrew.cmu.edu

Out: September 5, 2006
Due: September 19, 2006 (before midnight)

1 Introduction

The problem of typechecking a language is often reducible to the problem of deciding type equivalence for that language. Sometimes type equivalence is a simple syntactic notion, but for many interesting languages, type equivalence is non-trivial and requires a judgmental definition. In this assignment, you will implement a typechecker for a prototypical language with a non-trivial notion of type equivalence, System F_ω, the polymorphic λ-calculus with type operators.

2 Overview

Your task is to create a file `typecheck.sml` defining a structure `Typecheck` matching the signature `TYPECHECK` found in `typecheck-sig.sml`, reproduced here for your convenience.

```
signature TYPECHECK =
  sig
    exception Check of string
    exception Error of string

    structure Ctx : CONTEXT
                    where type con = Fomega.con
                      and type kind = Fomega.kind

    type context = Ctx.context

    val equiv_con : context -> Fomega.con -> Fomega.con -> Fomega.kind -> bool

    val check_con : context -> Fomega.con -> Fomega.kind
    val check_exp : context -> Fomega.exp -> Fomega.con
  end
```

The functions `check_con` and `check_exp` should implement kind checking and type checking, respectively, as described in Section 3. Typechecking will depend upon type equivalence, `equiv_con`, which you may implement using the weak head normalization technique from class, also described in Section 3. Signal any type errors by raising `Check` with an appropriate error message; you may use the `Error` exception for internal errors such as violated invariants or the occurrence of "impossible" conditions.

*Originally prepared by William Lovas (Fall 2005)

κ	::=	$\mathbf{T}$	kind of base types
	$\mid$	$\kappa_1 \rightarrow \kappa_2$	kind of type operators
c	::=	α	constructor variables
	$\mid$	$c_1 \rightarrow c_2$	function types
	$\mid$	$\forall\alpha{:}\kappa.c$	polymorphic types
	$\mid$	$\lambda\alpha{:}\kappa.c$	type operators
	$\mid$	$c_1 c_2$	type operator application
e	::=	x	variables
	$\mid$	$\lambda x{:}c.e$	abstraction
	$\mid$	$e_1 e_2$	appliation
	$\mid$	$\Lambda\alpha{:}\kappa.e$	polymorphic abstraction
	$\mid$	$e[c]$	polymorphic application
Γ	::=	$\cdot$	empty context
	$\mid$	Γ, α	kinding declaration
	$\mid$	$\Gamma, x{:}c$	typing declaration

Figure 1: Syntax of F_ω

p	::=	α	constructor variables
	$\mid$	pc	path application
n	::=	p	paths
	$\mid$	$c_1 \rightarrow c_2$	function types
	$\mid$	$\forall\alpha{:}\kappa.c$	polymorphic types
	$\mid$	$\lambda\alpha{:}\kappa.c$	type operators

Figure 2: Paths, p, and weak head normal forms, n, of F_ω constructors

All of the above functions take a `context`. We provide the `ContextFn` functor in the file `common/context-fn.sml` which you can instantiate to generate this type. Contexts contain both expression variable typing declarations and constructor variable kinding declarations.

The file `fomega.sml` defines the `Fomega` structure, which describes the abstract syntax of F_ω.

You may use the test harness interface in the `Top` structure (see `top-sig.sml`) to experiment with your implementation. Several interesting examples are included in the `examples.fom` file. Section 4 gives the grammar for the concrete ASCII syntax, along with some sample output.

Submit your code via AFS by copying `typecheck.sml` to the directory

```
/afs/andrew/course/15/501-819/submit/<your andrew id>/fomega/
```

Note: Your submission will be graded automatically; if you submit a `typecheck.sml` that fails to compile under SML/NJ 110.59 using this assignment's base distribution, we won't be able to grade it.

3 Details

Figure 1 shows the abstract syntax of F_ω, corresponding closely to the datatypes in `fomega.sml`. Type equivalence for this language is non-trivial because the language includes a somewhat sophisticated "meta-programming" layer at the constructor level, i.e. abstraction and application for constructors. In fact, the constructor level is actually a complete copy of the simply-typed λ-calculus, "one level up".

$\boxed{\Gamma \vdash c : \kappa}$

$$\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha : \kappa} \quad \frac{\Gamma \vdash c_1 : \mathbf{T} \quad \Gamma \vdash c_2 : \mathbf{T}}{\Gamma \vdash c_1 \to c_2 : \mathbf{T}} \quad \frac{\Gamma, \alpha{:}\kappa \vdash c : \mathbf{T} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \forall \alpha{:}\kappa.c : \mathbf{T}}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c : \kappa' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda \alpha{:}\kappa.c : \kappa \to \kappa'} \quad \frac{\Gamma \vdash c_1 : \kappa \to \kappa' \quad \Gamma \vdash c_2 : \kappa}{\Gamma \vdash c_1 c_2 : \kappa'}$$

$\boxed{\Gamma \vdash e \Rightarrow c}$

$$\frac{\Gamma(x) = c}{\Gamma \vdash x \Rightarrow c} \quad \frac{\Gamma \vdash c : \mathbf{T} \quad \Gamma, x{:}c \vdash e \Rightarrow c' \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x{:}c.e \Rightarrow c \to c'}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow c_1 \quad \Gamma \vdash c_1 \Downarrow c \to c' \quad \Gamma \vdash e_2 \Rightarrow c_2 \quad \Gamma \vdash c \Leftrightarrow c_2 : \mathbf{T}}{\Gamma \vdash e_1 e_2 \Rightarrow c'}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash e \Rightarrow c \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \Lambda \alpha{:}\kappa.e \Rightarrow \forall \alpha{:}\kappa.c} \quad \frac{\Gamma \vdash e \Rightarrow c_1 \quad \Gamma \vdash c_1 \Downarrow \forall \alpha{:}\kappa.c_1' \quad \Gamma \vdash c : \kappa}{\Gamma \vdash e[c] \Rightarrow [c/\alpha]c_1'}$$

$\boxed{\Gamma \vdash c \Downarrow n}$

$$\frac{\Gamma \vdash c_1 \Downarrow \lambda \alpha{:}\kappa.c \quad \Gamma \vdash [c_2/\alpha]c \Downarrow n}{\Gamma \vdash c_1 c_2 \Downarrow n} \quad \frac{\Gamma \vdash c_1 \Downarrow p_1}{\Gamma \vdash c_1 c_2 \Downarrow p_1 c_2} \quad \frac{}{\Gamma \vdash n \Downarrow n}$$

$\boxed{\Gamma \vdash c_1 \Leftrightarrow c_2 : \kappa}$

$$\frac{\Gamma \vdash c_1 \Downarrow n_1 \quad \Gamma \vdash c_2 \Downarrow n_2 \quad \Gamma \vdash n_1 \leftrightarrow n_2 : \mathbf{T}}{\Gamma \vdash c_1 \Leftrightarrow c_2 : \mathbf{T}} \quad \frac{\Gamma, \alpha{:}\kappa \vdash c_1 \alpha \Leftrightarrow c_2 \alpha : \kappa' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash c_1 \Leftrightarrow c_2 : \kappa \to \kappa'}$$

$\boxed{\Gamma \vdash c_1 \leftrightarrow c_2 : \kappa}$

$$\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha \leftrightarrow \alpha : \kappa} \quad \frac{\Gamma \vdash c_1 \Leftrightarrow c_2 : \mathbf{T} \quad \Gamma \vdash c_1' \Leftrightarrow c_2' : \mathbf{T}}{\Gamma \vdash c_1 \to c_1' \leftrightarrow c_2 \to c_2' : \mathbf{T}} \quad \frac{\Gamma, \alpha{:}\kappa \vdash c_1 \Leftrightarrow c_2 : \mathbf{T} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \forall \alpha{:}\kappa.c_1 \leftrightarrow \forall \alpha{:}\kappa.c_2 : \mathbf{T}}$$

$$\frac{\Gamma \vdash p_1 \leftrightarrow p_2 : \kappa \to \kappa' \quad \Gamma \vdash c_1 \Leftrightarrow c_2 : \kappa}{\Gamma \vdash p_1 c_1 \leftrightarrow p_2 c_2 : k'}$$

Figure 3: Algorithmic static semantics of F_ω

Judgment	Inputs	Outputs	Interpretation
$\Gamma \vdash c : \kappa$	Γ, c	κ	Under context Γ, constructor c has kind κ
$\Gamma \vdash e \Rightarrow c$	Γ, e	c	Under context Γ, expression e has type c
$\Gamma \vdash c \Downarrow n$	c	n	Under context Γ c has weak head normal form n
$\Gamma \vdash c_1 \Leftrightarrow c_2 : \kappa$	Γ, c_1, c_2, κ		Under Γ, c_1 and c_2 are algorithmically equivalent at kind κ
$\Gamma \vdash n_1 \leftrightarrow n_2 : \kappa$	Γ, c_1, c_2	κ	Under Γ, c_1 and c_2 are structurally equivalent at kind κ

Table 1: Summary of algorithmic static semantics for F_ω

The necessity of some sort of non-trivial notion of type equivalence is easily seen through an example. Suppose we had base types like int; we would want to be able to typecheck a term like "$(\lambda x{:}((\lambda\alpha{:}\mathbf{T}.\alpha)\mathsf{int}).x)5$". Naively applying the usual syntax-directed typing rules, we will be unable to show that this term is well-typed, since the domain type of the function, $(\lambda\alpha{:}\mathbf{T}.\alpha)\mathsf{int}$, does not syntactically match the argument's type, int. Intuitively, we know this term should typecheck, because in some sense, "$(\lambda\alpha{:}\mathbf{T}.\alpha)\mathsf{int}$" and "$\mathsf{int}$" are equivalent types according to our usual interpretations of λ-calculus application. The declarative typing rules from class, reproduced for your reference in Appendix A, make this intuition precise by way of a constructor equivalence judgment, $\Gamma \vdash c_1 \equiv c_2 : \kappa$, and a type conversion rule utilizing that judgment.

To implement a typechecker for this language, we need syntax-directed algorithmic typing rules. Figure 3 shows the algorithmic typing rules given in class. There are five judgments in total, summarized in Table 1. (Aside: the kinding relation in F_ω is simple enough that the declarative definition doubles as the algorithmic definition. This won't necessarily be the case in every language.)

To help understand how these rules work together, consider the type synthesis rule for application as an example:

$$\frac{\Gamma \vdash e_1 \Rightarrow c_1 \quad \Gamma \vdash c_1 \Downarrow c \to c' \quad \Gamma \vdash e_2 \Rightarrow c_2 \quad \Gamma \vdash c \Leftrightarrow c_2 : \mathbf{T}}{\Gamma \vdash e_1 e_2 \Rightarrow c'}$$

When computing a type for $e_1 e_2$, we first synthesize a type c_1 for e_1. Then we must check that this type has the form $c \to c'$ for some c and c'. Next, we synthesize a type c_2 for e_2. Finally, we must ensure that c_2 is in fact equivalent to c, the domain type of e_1. This sort of procedure motivates a judgment for finding the "true" top-level form of a type constructor, weak head normalization, written $\Gamma \vdash c \Downarrow n$, and a judgment for algorithmically determining if two constructors are equivalent at a particular kind, written $\Gamma \vdash c_1 \Leftrightarrow c_2 : \kappa$.

To implement the algorithmic equivalence judgment $\Gamma \vdash c_1 \Leftrightarrow c_2 : \kappa$, we dispatch on the kind κ at which c_1 and c_2 are to be compared. Note that κ is an "input" to this judgment. If κ is $\mathbf{T}$, we check that c_1 and c_2 have structurally equivalent weak head normal forms. The structural equivalence judgment $\Gamma \vdash c_1 \leftrightarrow c_2 : \kappa$ checks that c_1 and c_2 have the same form and then recursively applies algorithmic equivalence to their subcomponents. Note that κ is an "output" of structural equivalence.

To algorithmically compare two constructors at higher kind $\kappa \to \kappa'$, we conduct an "experiment" by making up a variable α of the domain kind κ and comparing the applications $c_1\alpha$ and $c_2\alpha$ at the smaller kind κ'. In this way, we eventually drive algorithmic equivalence down to base kind $\mathbf{T}$, at which point any β-redices created will be normalized away by algorithmic equivalence. (This rule may seem a bit strange, but it is one way of algorithmically handling η-equivalence, and it will generalize nicely when we cover the calculus with singleton kinds.)

You are encouraged to use the rules in Figure 3 to implement your typechecker, but any implementation that is sound and complete with respect to the declarative rules in Figure 5 will be accepted. Whatever strategy you choose, be careful to avoid capture when implementing substitution over constructors!

4 Examples

κ ::= `Type`
| κ_1 `=>` κ_2

c ::= `X`
| c_1 `->` c_2
| `All X::`κ`.`c
| `lambda X::`κ`.`c
| c_1 c_2

e ::= `x`
| `lambda x:`c`.`e
| e_1 e_2
| `Lambda X::`κ`.`e
| e `[`c`]`

Figure 4: ASCII syntax for F_ω

The ASCII syntax we'll use is described in Figure 4. Concretely, we adhere to all the usual conventions: arrows in types and kinds associate to the right, application in expressions and constructors associates to the left, and the scope of a bound variable extends as far to the right as possible. Identifiers beginning with a capital letter represent constructor variables, while identifiers beginning with a lowercase letter represent expression variables.

The following interactions with the SML/NJ top-level will give you some simple examples of how your checker should work. Note that your checker need not produce this output exactly, but it should produce "equivalent" output with respect to the constructor equivalence rules in Figure 5. More sample inputs may be found in the file `examples.fom` distributed with this project's base.

```
- Top.typecheck_string "Lambda X::Type. lambda x:X. x";
|- Lambda X::Type. lambda x:X. x
 : All X::Type. X -> X
val it = () : unit

- Top.typecheck_string "Lambda Int::Type. lambda y:Int. (lambda x : ((lambda X::
Type. X) Int) . x) y";
|- Lambda Int::Type. lambda y:Int. (lambda x:(lambda X::Type. X) Int. x) y
 : All Int::Type. Int -> (lambda X::Type. X) Int
val it = () : unit

- Top.typecheck_string "Lambda T::Type => Type. lambda f : (All X::Type. X -> T
X). Lambda A::Type. lambda x:A. f [A] x";
|- Lambda T::Type => Type.
   lambda f:(All X::Type. X -> T X). Lambda A::Type. lambda x:A. f [A] x
 : All T::Type => Type. (All X::Type. X -> T X) -> All A::Type. A -> T A
val it = () : unit

(* alpha *)
- Top.equiv_strings "lambda X::Type. X" "lambda Y::Type. Y";
val it = true : bool

(* beta *)
- Top.equiv_strings
=         "All X::Type. X -> X"
=         "(lambda X::Type. X) (All X::Type. X -> X)";
val it = true : bool

(* eta *)
- Top.equiv_strings
=         "lambda F :: Type => Type. F"
=         "lambda G :: Type => Type. lambda X::Type. G X";
val it = true : bool

- Top.equiv_strings
=         "lambda F::Type => Type. lambda G::Type => Type. lambda X::Type. F X"
=         "lambda F::Type => Type. lambda G::Type => Type. lambda X::Type. G X";
val it = false : bool

- Top.equiv_strings
=         "lambda C::Type. (lambda X::Type. lambda C::Type. X) C"
=         "lambda C::Type. lambda C::Type. C";
val it = false : bool
```

A Declarative static semantics

For reference, this Appendix contains the declarative judgments defining the static semantics of F_ω.

$\boxed{\Gamma \vdash c : \kappa}$

$$\frac{\Gamma(\alpha) = \kappa}{\Gamma \vdash \alpha : \kappa} \quad \frac{\Gamma \vdash c_1 : \mathbf{T} \quad \Gamma \vdash c_2 : \mathbf{T}}{\Gamma \vdash c_1 \to c_2 : \mathbf{T}} \quad \frac{\Gamma, \alpha{:}\kappa \vdash c : \mathbf{T} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \forall \alpha{:}k.c : \mathbf{T}}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c : \kappa' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda \alpha{:}k.c : k \to k'} \quad \frac{\Gamma \vdash c_1 : \kappa \to \kappa' \quad \Gamma \vdash c_2 : \kappa}{\Gamma \vdash c_1 c_2 : \kappa'}$$

$\boxed{\Gamma \vdash e : c}$

$$\frac{\Gamma(x) = c}{\Gamma \vdash x : c} \quad \frac{\Gamma \vdash c : \mathbf{T} \quad \Gamma, x{:}c \vdash e : c' \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x{:}c.e : c \to c'} \quad \frac{\Gamma \vdash e_1 : c \to c' \quad \Gamma \vdash e_2 : c}{\Gamma \vdash e_1 e_2 : c'}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash e : c \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \Lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.c} \quad \frac{\Gamma \vdash e : \forall \alpha{:}\kappa.c \quad \Gamma \vdash c : \kappa}{\Gamma \vdash e[c] : [c/\alpha]c'} \quad \frac{\Gamma \vdash e : c \quad \Gamma \vdash c \equiv c' : \mathbf{T}}{\Gamma \vdash e : c'}$$

$\boxed{\Gamma \vdash c_1 \equiv c_2 : \kappa}$

$$\frac{\Gamma \vdash c : \kappa}{\Gamma \vdash c \equiv c : \kappa} \quad \frac{\Gamma \vdash c_2 \equiv c_1 : \kappa}{\Gamma \vdash c_1 \equiv c_2 : \kappa} \quad \frac{\Gamma \vdash c_1 \equiv c_2 : \kappa \quad \Gamma \vdash c_2 \equiv c_3 : \kappa}{\Gamma \vdash c_1 \equiv c_3 : \kappa}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2 : \mathbf{T} \quad \Gamma \vdash c_1' \equiv c_2' : \mathbf{T}}{\Gamma \vdash c_1 \to c_1' \equiv c_2 \to c_2' : \mathbf{T}} \quad \frac{\Gamma, \alpha{:}\kappa \vdash c_1 \equiv c_2 : \mathbf{T} \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \forall \alpha{:}\kappa.c_1 \equiv \forall \alpha{:}\kappa.c_2 : \mathbf{T}}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c_1 \equiv c_2 : \kappa' \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda \alpha{:}\kappa.c_1 \equiv \lambda \alpha{:}\kappa.c_2 : k \to k'} \quad \frac{\Gamma \vdash c_1 \equiv c_2 : k \to k' \quad \Gamma \vdash c_1' \equiv c_2' : \kappa}{\Gamma \vdash c_1 c_1' \equiv c_2 c_2' : \kappa'}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash c' : \kappa' \quad \Gamma \vdash c : \kappa \quad \alpha \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda \alpha{:}\kappa.c')c \equiv [c/\alpha]c' : \kappa'} \quad \frac{\Gamma \vdash c : \kappa \to \kappa' \quad \Gamma \notin FV(c)}{\Gamma \vdash \lambda \alpha{:}\kappa.c\alpha \equiv c : \kappa \to \kappa'}$$

Figure 5: Declarative static semantics of F_ω