Correctness proofs for Hindley-Milner type inference algorithms

Konstantine Arkoudas

March 23, 2001

Abstract

In this paper we present a Hindley-Milner type system for a simple ML-like language, and we implement the type inference algorithm as a theorem prover that does not simply produce a type τ for a given expression e and type environment A, but rather fully derives the type judgment $A \vdash e : \tau$ from the primitive inference rules of the type system. The derivation can be seen as a correctness certificate for the produced type τ. This greatly increases the credibility of the result, since it is much easier to trust the primitive inference rules of the type system rather than the inference algorithm. We use the denotational proof language [2] Athena for our implementation, giving full working code for every part of the system. This is a real-life example of a conceptually involved algorithm expressed as a DPL method and reaping the associated soundness benefits.

1.1 The type system and inference algorithm

In this section we will formulate a Hindley-Milner type system for a simple language called Core-ML, and we will present Milner's famous $\mathcal{W}$ algorithm for type inference. We leave out recursion for now since it does not introduce any new ideas to the type issue; we will add it later on.

The abstract syntax of the expressions of Core-ML is defined by the following grammar:

$$\begin{array}{rcl} e & ::= & x \mid n \mid \textbf{true} \mid \textbf{false} \mid \lambda\, x.\, e \mid e_1\, e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \\ & & \langle e_1, e_2 \rangle \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \end{array}$$

where x ranges over an unspecified set of variables and n ranges over the integers. The set of types is defined by

$$\tau ::= \textbf{int} \mid \textbf{bool} \mid \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid (\forall\, \alpha)\, \tau$$

where α ranges over an unspecified countably infinite set of *type variables*. Free and bound occurrences of a type variable in a given type are defined as usual; e.g., in the type $(\alpha_1 \times \alpha_2) \to (\forall\, \alpha_2)\, (\alpha_2 \to \alpha_2)$, α_1 occurs free while α_2 occurs both free and bound. We will write $FV(\tau)$ for the set of type variables that occur free in τ.

By a *type substitution* (or just "substitution") θ we will mean a function from the set of type variables to the set of types that is the identity almost everywhere. For distinct $\alpha_1, \ldots, \alpha_n$, we write

$$\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$$

for the substitution that maps each α_i to τ_i, and every other type variable to itself. Thus $\{\}$ denotes the identity function on the set of type variables. Further, we write $\theta[\alpha \mapsto \tau]$ for the substitution that maps α to τ and every type variable $\alpha' \neq \alpha$ to $\theta(\alpha')$. Given a substitution θ, we define a function $\overline{\theta}$ from the set of all types to itself as follows:

$$\begin{array}{rcl} \overline{\theta}(\alpha) & = & \theta(\alpha) \\ \overline{\theta}(\textbf{int}) & = & \textbf{int} \\ \overline{\theta}(\textbf{bool}) & = & \textbf{bool} \\ \overline{\theta}(\tau_1 \to \tau_2) & = & \overline{\theta}(\tau_1) \to \overline{\theta}(\tau_2) \\ \overline{\theta}(\tau_1 \times \tau_2) & = & \overline{\theta}(\tau_1) \times \overline{\theta}(\tau_2) \\ \overline{\theta}((\forall\, \alpha)\, \tau) & = & (\forall\, \alpha)\, \overline{\theta[\alpha \mapsto \alpha]}(\tau) \end{array}$$

The function $\overline{\theta}$ called the *lift* of θ.[1] Note that applying $\overline{\theta}$ to a type τ could result in variable capture; but in what follows we will make sure that this never happens.[2] Finally, given any two substitutions θ_1, θ_2 we can define a new substitution $\theta_1 \circ \theta_2$ as:

$$\theta_1 \circ \theta_2 = \lambda\, \alpha\, .\, \overline{\theta_1}(\theta_2(\alpha)).$$

We refer to $\theta_1 \circ \theta_2$ as the *composition* of θ_1 and θ_2.

A type τ_2 is an *instance* of a type τ_1 iff there is a θ such that $\overline{\theta}(\tau_1) = \tau_2$. We say that τ_1 is *more general* that τ_2. For example, the type $\mathbf{int} \to \mathbf{int}$ is an instance of $\alpha \to \alpha$ under the substitution $\{\alpha \mapsto \mathbf{int}\}$. Two types τ_1, τ_2 are *unifiable* iff there is a θ such that $\overline{\theta}(\tau_1) = \overline{\theta}(\tau_2)$. For example, the types $\alpha_1 \to (\alpha_1 \times \mathbf{int})$ and $\mathbf{bool} \to (\mathbf{bool} \times \mathbf{int})$ are unifiable via the substitution $\{\alpha_1 \mapsto \mathbf{bool}\}$.

The following algorithm U takes two quantifier-free types τ_1, τ_2 and produces a unifying substitution for them, if τ_1 and τ_2 are unifiable. If not, the algorithm raises an exception. The algorithm is written using ML-style pattern matching. Also, for a boolean-valued expression B, we write $B \Rightarrow E_1, E_2$ to mean "if B then E_1 else E_2".

$U(\alpha, \tau) = \alpha \in FV(\tau)? \Rightarrow (\alpha \neq \tau? \Rightarrow \textit{raise failure}, \{\}), \{\alpha \mapsto \tau\}$
$U(\tau, \alpha) = U(\alpha, \tau)$
$U(\tau_1 \to \tau_2, \tau_3 \to \tau_4) = U(\overline{\theta}(\tau_2), \overline{\theta}(\tau_4)) \circ \theta$, where $\theta = U(\tau_1, \tau_3)$
$U(\tau_1 \times \tau_2, \tau_3 \times \tau_4) = U(\overline{\theta}(\tau_2), \overline{\theta}(\tau_4)) \circ \theta$, where $\theta = U(\tau_1, \tau_3)$
$U(\mathbf{int}, \mathbf{int}) = U(\mathbf{bool}, \mathbf{bool}) = \{\}$
$U(\_, \_) = \textit{raise failure}.$

By an "atomic type assignment" we will mean an ordered pair (x, τ) consisting of a (Core-ML) variable x and a type τ. We will write such a pair more suggestively as $x : \tau$, and we will call x and τ the subject and value of that assignment, respectively. By a *type context* we will mean a finite list of atomic assignments:

$$\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle. \tag{1.1}$$

We will use the letter A to range over type contexts, and we write $x : \tau.A$ for the context obtained by prepending (consing) the assignment $x : \tau$ in front of A. For any given substitution θ and type context A of the form 1.1, we write $\overline{\theta}(A)$ for the context

$$\langle x_1 : \overline{\theta}(\tau_1), \ldots, x_n : \overline{\theta}(\tau_n) \rangle.$$

The expression $FV(A)$ will denote the set of type variables that occur free in the value of some assignment in A. Finally, we write $A(x) = \tau$ to signify that $x : \tau$ is the first (leftmost) assignment in A with subject x.

The inference rules of the Core-ML type system are shown in Figure 1.1. A type is called *shallow* iff it is of the form $(\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau$ for some $n \geq 0$ and quantifier-free τ. That is, a type is shallow iff it has no quantifiers or else every quantifier is up front and its scope includes everything to its right. This type system allows us to prove that certain expressions have non-shallow types. For example, we can show that the expression $\lambda\, f\, .\, \langle f\ 3, f\ \mathbf{true} \rangle$ has the non-shallow type

$$[(\forall\, \alpha)\, \alpha \to \alpha] \to [\mathbf{int} \times \mathbf{bool}]$$

as follows:

[1] If we exclude quantifications, the remaining types form a free algebra over the set of type variables, and $\overline{\theta}$ then coincides with the unique homomorphic extension of θ. Milner's inference algorithm essentially ignores quantifications and treats the set of all types as a free algebra.

[2] Of course in practice this can always be ensured by alphabetically renaming τ before applying $\overline{\theta}$.

$$\frac{}{A \vdash n : \mathbf{int}}\ [Num] \qquad \frac{}{A \vdash \mathbf{true} : \mathbf{bool}}\ [True]$$

$$\frac{}{A \vdash \mathbf{false} : \mathbf{bool}}\ [False] \qquad \frac{}{\begin{array}{c} A \vdash x : \tau \\ \text{provided } A(x) = \tau \end{array}}\ [Var]$$

$$\frac{x : \tau_1 . A \vdash e : \tau_2}{A \vdash \lambda\, x\, .\, e : \tau_1 \to \tau_2}\ [Abs] \qquad \frac{A \vdash e_1 : \tau_1 \quad x : \tau_1 . A \vdash e_2 : \tau_2}{A \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}\ [Let]$$

$$\frac{A \vdash e_1 : \tau_1 \to \tau_2 \quad A \vdash e_2 : \tau_1}{A \vdash e_1\ e_2 : \tau_2}\ [App] \qquad \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2}{A \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ [Prod]$$

$$\frac{A \vdash e_1 : \mathbf{bool} \quad A \vdash e_2 : \tau \quad A \vdash e_3 : \tau}{A \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}\ [If]$$

$$\frac{A \vdash e : \tau}{\begin{array}{c} A \vdash e : (\forall\, \alpha)\, \tau \\ \text{provided } \alpha \notin FV(A) \end{array}}\ [Gen] \qquad \frac{A \vdash e : (\forall\, \alpha)\, \tau}{A \vdash e : \overline{\{\alpha \mapsto \tau'\}}(\tau)}\ [Spec]$$

Figure 1.1: The Hindley-Milner type system.

1.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash f : (\forall\, \alpha)\, \alpha \to \alpha$	$[Var]$
2.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash f : \mathbf{int} \to \mathbf{int}$	1, $[Spec]$
3.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash 3 : \mathbf{int}$	$[Num]$
4.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash f\ 3 : \mathbf{int}$	2, 3, $[App]$
5.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash f : \mathbf{bool} \to \mathbf{bool}$	1, $[Spec]$
6.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash \mathbf{true} : \mathbf{bool}$	$[True]$
7.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash f\ \mathbf{true} : \mathbf{bool}$	5, 6, $[App]$
8.	$[f : (\forall\, \alpha)\, \alpha \to \alpha] \vdash \langle f\ 3, f\ \mathbf{true} \rangle : \mathbf{int} \times \mathbf{bool}$	4, 7, $[Prod]$
9.	$[] \vdash \lambda\, f\, .\, \langle f\ 3, f\ \mathbf{true} \rangle : [(\forall\, \alpha)\, \alpha \to \alpha] \to [\mathbf{int} \times \mathbf{bool}]$	8, $[Abs]$

In fact with non-shallow types we can even type self-applications. Consider the following proof of

$$[] \vdash \lambda\, x\, .\, x\ x : [(\forall\, \alpha)\, \alpha \to \alpha] \to [(\forall\, \alpha)\, \alpha \to \alpha] :$$

1.	$[x : (\forall\, \alpha)\, \alpha \to \alpha] \vdash x : (\forall\, \alpha)\, \alpha \to \alpha$	$[Var]$
2.	$[x : (\forall\, \alpha)\, \alpha \to \alpha] \vdash x : [(\forall\, \alpha)\, \alpha \to \alpha] \to [(\forall\, \alpha)\, \alpha \to \alpha]$	1, $[Spec]$
3.	$[x : (\forall\, \alpha)\, \alpha \to \alpha] \vdash x\ x : (\forall\, \alpha)\, \alpha \to \alpha$	2, 1, $[App]$
4.	$[] \vdash \lambda\, x\, .\, x\ x : [(\forall\, \alpha)\, \alpha \to \alpha] \to [(\forall\, \alpha)\, \alpha \to \alpha]$	3, $[App]$

Milner's type inference *algorithm*, however, is only able to infer shallow types. For instance, the algorithm will fail on the expression $\lambda\, f\, .\, \langle f\ 3, f\ \mathbf{true} \rangle$. The problem is that there is no effective (algorithmic) way of deducing non-shallow types. Accordingly, Milner's algorithm is incomplete with

respect to the type system shown of Figure 1.1: even though certain judgments are provable in the system, Milner's algorithm will not derive them. Alternatively, we could syntactically restrict our set of types by weeding out non-shallow types and then slightly reformulating the rules so as to make the algorithm complete with respect to the inference system, which is the approach of Milner and Damas [4]. Our approach here is similar to Cardelli's [3].

The algorithm for type inference, $\mathcal{W}$, takes a Core-ML expression e and a context A and returns a type τ. The soundness claim made by the algorithm is this: if $\mathcal{W}(e, A) = \tau$ then $A \vdash e : \tau$. The other useful property is that $\mathcal{W}(e, A)$ is a *principal* type, i.e., more general that any other type that can be derived for e with respect to A. Formally: for all τ, if $A \vdash e : \tau$ then τ is an *instance* of $\mathcal{W}(e, A)$. This property follows from the fact that unification is used to "guess" the types of bound Core-ML variables, and unification always returns the most general possible unifying substitution. We will not prove these two claims here, but we will get the first property—soundness—for free when we come to write an Athena method for type inference.

The algorithm $\mathcal{W}$ is given below in conventional ML-like notation. It uses an auxiliary function $\mathcal{V}$ that takes an expression e and a context A and returns a pair (τ, θ) consisting of a type τ (the most general type of e w.r.t. A), and a substitution θ that is used to update our current "guesses" about the types of the bound variables of e.

$\mathcal{W}(e, A) = \tau$, where $(\tau, \theta) = \mathcal{V}(e, A)$ and
$\mathcal{V}(n, A) = (\mathbf{int}, \{\})$
$\mathcal{V}(\mathbf{true}, A) = \mathcal{V}(\mathbf{false}, A) = (\mathbf{bool}, \{\})$
$\mathcal{V}(x, A) = [A(x) = (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau]? \Rightarrow (\overline{\{\alpha_1 \mapsto \alpha'_1, \ldots, \alpha_n \mapsto \alpha'_n\}}(\tau), \{\}), \textit{raise failure}$
where $\alpha'_1, \ldots, \alpha'_n$ are fresh, $n \geq 0$
$\mathcal{V}(\lambda\, x\, .\, e, A) = (\overline{\theta}(\alpha \to \tau), \theta)$, where $(\tau, \theta) = \mathcal{V}(e, x : \alpha.A)$ and α is fresh
$\mathcal{V}(e_1\, e_2, A) = (\overline{\theta}(\alpha), \theta \circ (\theta_2 \circ \theta_1))$,
where $(\tau_1, \theta_1) = \mathcal{V}(e_1, A)$, $(\tau_2, \theta_2) = \mathcal{V}(e_2, \overline{\theta_1}(A))$, $\theta = U(\overline{\theta_2}(\tau_1), \tau_2 \to \alpha)$, and α is fresh
$\mathcal{V}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, A) = (\tau_2, \theta_2 \circ \theta_1)$,
where $(\tau_1, \theta_1) = \mathcal{V}(e_1, A)$, $(\tau_2, \theta_2) = \mathcal{V}(e_2, A')$, $A' = x : (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau_1.\overline{\theta_1}(A)$,
and $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau_1) - FV(\overline{\theta_1}(A))$
$\mathcal{V}(\langle e_1, e_2 \rangle, A) = (\overline{\theta}(\tau_1 \times \tau_2), \theta)$, where $(\tau_1, \theta_1) = \mathcal{V}(e_1, A)$, $(\tau_2, \theta_2) = \mathcal{V}(e_2, \overline{\theta_1}(A))$, $\theta = \theta_2 \circ \theta_1$
$\mathcal{V}(\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3, A) = (\overline{\theta'}(\tau_3), \theta' \circ (\theta_3 \circ (\theta_2 \circ (\theta \circ \theta_1))))$,
where $(\tau_1, \theta_1) = \mathcal{V}(e_1, A)$, $\theta = U(\tau_1, \mathbf{bool})$, $(\tau_2, \theta_2) = \mathcal{V}(e_2, \overline{\theta \circ \theta_1}(A))$,
$(\tau_3, \theta_3) = \mathcal{V}(e_3, \overline{\theta_2 \circ (\theta \circ \theta_1)}(A))$, and $\theta' = U(\overline{\theta_3}(\tau_2), \tau_3)$

Note the difference in the handling of **let**-bound variables vs. λ-bound variables. When we encounter an abstraction $\lambda\, x\, .\, e$, we augment the current type context with an assignment $x : \alpha$, for some fresh α, and move on with the body e. Once α becomes instantiated to some type τ within e—say to **int**—it remains an instance of τ throughout. This means that the type of x cannot be freshly instantiated as needed at different places within the body of e—in Hindley-Milner parlance, x is a *non-generic* variable. Every occurence of x within e must have the same (principal) type. This is why the algorithm is unable to infer the non-shallow type of $\lambda\, f\, .\, \langle f\ 3, f\ \mathbf{true} \rangle$: first, the type of f is bound to some fresh undetermined type α; then, when we come to the body $\langle f\ 3, f\ \mathbf{true} \rangle$ and process the left element of the pair, we conclude that $\alpha = \mathbf{int} \to \alpha'$, and we record that in our context. At this point the type of f has been permanently constrained to be an instance of $\mathbf{int} \to \alpha'$. That is, any future use of f[3] will have to be reconciled with this type, *not* with a copy of the original, unconstrained α. Thus later when we come to process the right element of the pair, we get a mismatch: we are trying

[3] Strictly speaking, we should say "any future use of a free occurence of f in the scope of the λ". It is possible that f will be later bound as a **let** variable, and in that case all subsequent free occurences of f in that binding's scope would be generic.

to apply a function of type $\mathbf{int} \to \alpha'$ to a boolean, and the algorithm fails. Therefore, an application such as

$$(\lambda\, f\,.\,\langle f\ 3, f\ \mathbf{true}\rangle)\ (\lambda\, x\,.\,x) \tag{1.2}$$

would be rejected even though it is perfectly legitimate.

But now consider the expression

$$\mathbf{let}\ f = \lambda\, x\,.\,x\ \mathbf{in}\ \langle f\ 3, f\ \mathbf{true}\rangle \tag{1.3}$$

which is semantically equivalent to 1.2. Algorithm $\mathcal{W}$ *is* able to infer the correct type for this expression, even though it fails on 1.2, because it can process the identity function $f = \lambda\, x\,.\,x$ first, get a type judgment $f : \alpha \to \alpha$ for it, then *generalize* this to $f : (\forall\, \alpha)\, \alpha \to \alpha$, and then freely specialize this generic type as needed within the body $\langle f\ 3, f\ \mathbf{true}\rangle$. That is why we say that the occurrences of f within the body of 1.3 are *generic*. Note that $\mathcal{W}$ performs specialization in the clause for variables, which is

$V(x, A) = [A(x) = (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau]? \Rightarrow (\overline{\{\alpha_1 \mapsto \alpha'_1, \ldots, \alpha_n \mapsto \alpha'_n\}}(\tau), \{\}),$ *raise failure*
where $\alpha'_1, \ldots, \alpha'_n$ are fresh

Hence, if x is generic, i.e., if it was previously bound by a **let**, then its type in the current context is of the form $(\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau$, and the above clause will return a copy of τ with $\alpha_1, \ldots, \alpha_n$ replaced by fresh type variables $\alpha'_1, \ldots, \alpha'_n$.

In summary, occurrences of **let**-bound variables are generic: their types are universally generalized and then specialized as needed later on. By contrast, occurrences of λ-bound variables are non-generic: once instantiated, their types become permanently constrained and cannot be reinstantiated differently at different places.

1.2 An Athena implementation

1.2.1 Defining the logic

We begin by modelling the relevant data types:

```
>(domain Num)

New domain Num introduced.

>(use-numerals (0 ...) Num)

Numerals in the range (0 ...) can now be used as terms of Num.

>(define-numeric-operations)

The following binary functions have been introduced:
plus, minus, times, div, mod, num-equal?, less?, and greater?.
These functions can be applied to any numerals in the (0 ...) range.

>(structure Exp
   True
   False
   (NumConst Num)
```

```
   (Var Ide)
   (Abs Ide Exp)
   (App Exp Exp)
   (Prod Exp Exp)
   (Cond Exp Exp Exp)
   (Let Ide Exp Exp))

New structure Exp defined.

>(structure Type
   IntType
   BoolType
   (VarType Ide)
   (FunType Type Type)
   (ProdType Type Type)
   (GenType Ide Type))

New structure Type defined.
```

The first three declarations allow for the use of numerals `0`, `1`, ..., as terms of sort `Num`, and define some standard arithmetic operations on these numerals. The structures `Exp` and `Type` model the set of Core-ML expressions and types, respectively. Below are some auxiliary definitions and declarations that will be needed:

```
>(define (subst tv t' t)
   (match t
     ((VarType (val-of tv)) t')
     ((FunType t1 t2) (FunType (subst tv t' t1) (subst tv t' t2)))
     ((ProdType t1 t2) (ProdType (subst tv t' t1) (subst tv t' t2)))
     ((GenType (val-of tv) _) t)
     ((GenType tv' body) (GenType tv' (subst tv t' body)))
     (t t)))

Function subst defined.

>(define (remove x l)
   (match l
     ((split l1 (list-of (val-of x) l2)) (remove x (join l1 l2)))
     (_ l)))

Function remove defined.

>(define (occurs? x L)
   (match L
     ((split L1 (list-of (val-of x) L2)) true)
     (_ false)))

Function occurs? defined.

>(define (free-type-vars t)
   (match t
```

```
     ((VarType x) [x])
     ((FunType t1 t2) (join (free-type-vars t1) (free-type-vars t2)))
     ((ProdType t1 t2) (join (free-type-vars t1) (free-type-vars t2)))
     ((GenType x body) (remove x (free-type-vars body)))
     (_ []))))

Function free-type-vars defined.

>(structure (List-Of T)
   Nil
   (Cons T (List-Of T)))

New structure List-Of defined.

>(define (map f l)
   (match l
     (Nil Nil)
     ((Cons x rest) (Cons (f x) (map f rest)))))

Function map defined.

>(structure (Pair-Of S T)
   (Pair S T))

New structure Pair-Of defined.

>(define Context (List-Of (Pair-Of Ide Type)))

Defined sort abbreviation Context.

>(define (context-free-type-vars c)
   (match c
    (Nil [])
    ((Cons (Pair _ t) rest) (join (free-type-vars t)
                                  (context-free-type-vars rest)))))

Function context-free-type-vars defined.

>(define (apply-context c x)
   (match c
     ((Cons (Pair (val-of x) type) _) type)
     ((Cons _ rest) (apply-context rest x))))

Function apply-context defined.

>(declare has-type (-> (Exp Context Type) Boolean))

New symbol has-type declared.

>(define (~ b)
  (match b
```

```
    (true false)
    (false true)))

Function ~ defined.
```

The main relation of interest is `has-type`. A proposition of the form

(has-type e c t)

is intended to assert that the expression e has type t with respect to the context c.

Here is how we model the inference rules in Figure 1.1:

```
(primitive-method (pm-true c) (has-type True c BoolType))

(primitive-method (pm-false c) (has-type False c BoolType))

(primitive-method (pm-num c n) (has-type (NumConst n) c IntType))

(primitive-method (pm-var c v t)
  (check ((equal? (apply-context c v) t) (has-type (Var v) c t))))

(primitive-method (pm-abs c x body t1 t2)
  (check ((holds? (has-type body (Cons (Pair x t1) c) t2))
                  (has-type (Abs x body) c (FunType t1 t2)))))

(primitive-method (pm-app c e1 e2 t1 t2)
  (check ((& (holds? (has-type e1 c (FunType t1 t2)))
             (holds? (has-type e2 c t1)))
                     (has-type (App e1 e2) c t2))))

(primitive-method (pm-let c x e body t1 t2)
  (check ((& (holds? (has-type e c t1))
             (holds? (has-type body (Cons (Pair x t1) c) t2)))
                     (has-type (Let x e body) c t2))))

(primitive-method (pm-prod c e1 e2 t1 t2)
  (check ((& (holds? (has-type e1 c t1))
             (holds? (has-type e2 c t2)))
                     (has-type (Prod e1 e2) c (ProdType t1 t2)))))

(primitive-method (pm-cond c e1 e2 e3 t)
  (check ((& (holds? (has-type e1 c BoolType))
             (holds? (has-type e2 c t))
             (holds? (has-type e3 c t)))
                     (has-type (Cond e1 e2 e3) c t))))

(primitive-method (pm-gen c e t x)
  (check ((& (holds? (has-type e c t))
             (~ (occurs? x (context-free-type-vars c))))
             (has-type e c (GenType x t)))))
```

```
(primitive-method (pm-spec c e x t t')
  (check ((holds? (has-type e c (GenType x t)))
                  (has-type e c (subst x t' t)))))
```

That's it! Now here is a proof of $[] \vdash \lambda\, x\, .\, x : \alpha \to \alpha$:

```
>(dbegin
   (!pm-var (Cons (Pair 'x (VarType 'a)) Nil)  'x (VarType 'a))
   (!pm-abs Nil 'x (Var 'x) (VarType 'a) (VarType 'a)))

Theorem: (has-type (Abs 'x
                        (Var 'x))
                   Nil
                   (FunType (VarType 'a)
                            (VarType 'a)))
```

And here is the proof of

$$\lambda\, f\, .\, \langle f\ 3, f\ \mathbf{true}\rangle : [(\forall\, \alpha)\, \alpha \to \alpha] \to [\mathbf{int} \times \mathbf{bool}]$$

that was given in page 2:

```
>(dlet ((gtype (GenType 'a (FunType (VarType 'a) (VarType 'a))))
        (c (Cons (Pair 'f gtype) Nil))
        (L1 (!pm-var c 'f gtype))
        (L2 (!pm-spec c (Var 'f) 'a
                        (FunType (VarType 'a) (VarType 'a)) IntType))
        (L3 (!pm-num c 3))
        (L4 (!pm-app c (Var 'f) (NumConst 3) IntType IntType))
        (L5 (!pm-spec c (Var 'f) 'a
                        (FunType (VarType 'a) (VarType 'a)) BoolType))
        (L6 (!pm-true c))
        (L7 (!pm-app c (Var 'f) True BoolType BoolType))
        (L8 (!pm-prod c (App (Var 'f) (NumConst 3)) (App (Var 'f) True)
                        IntType BoolType))
        (prod-type (ProdType IntType BoolType)))
   (!pm-abs Nil 'f (Prod (App (Var 'f) (NumConst 3))
                         (App (Var 'f) True)) gtype prod-type))

Theorem: (has-type (Abs 'f
                        (Prod (App (Var 'f)
                                   (NumConst 3))
                              (App (Var 'f)
                                   True)))
                   Nil
                   (FunType (GenType 'a
                                     (FunType (VarType 'a)
                                              (VarType 'a)))
                            (ProdType IntType BoolType)))
```

Note that lines 1—8 in the proof of page 2 correspond exactly to "lemmas" `L1`—`L8` in the above Athena proof. Line 9, the final inference, corresponds to the body of the above `dlet`.

1.2.2 Defining algorithm $\mathcal{W}$

First we deal with the generation of fresh type variables:

```
>(define (digit->char n)
   (match n
    (0 `0) (1 `1) (2 `2) (3 `3) (4 `4)
    (5 `5) (6 `6) (7 `7) (8 `8) (9 `9)))

Function digit->char defined.

>(define (num->string n)
   (letrec ((loop
              (function (n digits)
                (check
                  ((less? n 10) (add (digit->char n) digits))
                  (else (loop (div n 10)
                              (add (digit->char (mod n 10)) digits)))))))
     (loop n [])))

Function num->string defined.

>(define tvar-counter (cell 0))

Cell tvar-counter defined.

>(define (inc c)
   (begin (set! c (plus (ref c) 1))
          (ref c)))

Function inc defined.

>(define (fresh-type-name)
   (string->id (join "T" (num->string (inc tvar-counter)))))

Function fresh-type-name defined.
```

Next we represent substitutions:

```
>(define empty-sub (function (x) (VarType x)))

Function empty-sub defined.

>(define (uni-sub x t)
   (function (y)
     (check ((equal? x y) t)
            (else (VarType y)))))

Function uni-sub defined.

>(define (extend-sub sub x t)
   (function (x') (check
                    ((equal? x x') t)
```

```
                    (else (sub x')))))

Function extend-sub defined.

>(define (lift sub)
  (function (t)
    (match t
      ((VarType x) (sub x))
      ((FunType t1 t2) (FunType ((lift sub) t1) ((lift sub) t2)))
      ((ProdType t1 t2) (ProdType ((lift sub) t1) ((lift sub) t2)))
      ((GenType x t')
         (GenType x ((lift (extend-sub sub x (VarType x))) t')))
      ((Pair v t) (Pair v ((lift sub) t)))
      (_ t))))

Function lift defined.

>(define (compose theta sigma)
   (function (x) ((lift theta) (sigma x))))

Function compose defined.

>(define (error str)
  (begin (print (join "\n" str "\n"))
         (halt)))

Function error defined.

>(define (unify t1 t2)
   (match [t1 t2]
     ([(VarType x) _] (check
                        ((occurs? x (free-type-vars t2))
                           (check ((equal? t1 t2) empty-sub)
                                  (else (error "Circularity detected."))))
                        (else (uni-sub x t2))))
     ([t1 (VarType _)] (unify t2 t1))
     ([(FunType s1 s2) (FunType s1' s2')]
         (let ((sub (unify s1 s1')))
           (compose (unify ((lift sub) s2) ((lift sub) s2')) sub)))
     ([(ProdType s1 s2) (ProdType s1' s2')]
         (let ((sub (unify s1 s1')))
           (compose (unify ((lift sub) s2) ((lift sub) s2')) sub)))
     ([BoolType BoolType] empty-sub)
     ([IntType IntType] empty-sub)
     (_ (error "Unresolvable function symbols."))))

Function unify defined.
```

And some auxiliary functions that we will need:

```
>(define (fresh-instance t)
  (letrec ((loop (function (t fresh-types)
```

```
                    (match t
                      ((GenType tv body)
                         (let ((fresh (fresh-type-name)))
                           (loop (replace tv (VarType fresh) body)
                                 (add fresh fresh-types))))
                      (_ [t (rev fresh-types)])))))
    (loop t [])))

Function fresh-instance defined.

>(define (filter L f)
  (match L
    ([] [])
    ((list-of x rest) (check ((f x) (add x (filter rest f)))
                             (else (filter rest f))))))

Function filter defined.

>(define (gen t vars)
   (match vars
     ([] t)
     ((list-of tv rest) (gen (GenType tv t) rest))))

Function gen defined.

>(define (generalize t c)
   (let ((cvars (context-free-type-vars c))
         (uvars (filter (free-type-vars t)
                        (function (tv) (~ (occurs? tv cvars))))))
     (gen t uvars)))

Function generalize defined.

>(define (infer e c)
   (match e
      (True [BoolType empty-sub])
      (False [BoolType empty-sub])
      ((NumConst n) [IntType empty-sub])
      ((Var x) [(head (fresh-instance (apply-context c x))) empty-sub])
      ((Prod e1 e2)
         (match (infer e1 c)
           ([t1 sub1]
              (match (infer e2 (map (lift sub1) c))
                ([t2 sub2] (let ((sub (compose sub2 sub1)))
                             [((lift sub) (ProdType t1 t2)) sub]))))))
      ((Abs x body)
         (let ((param-type (VarType (fresh-type-name))))
           (match (infer body (Cons (Pair x param-type) c))
              ([body-type sub]
                 [((lift sub) (FunType param-type body-type)) sub]))))
      ((App e1 e2)
```

```
         (match (infer e1 c)
           ([t1 sub1]
              (match (infer e2 (map (lift sub1) c))
                ([t2 sub2]
                   (let ((new-tvar (VarType (fresh-type-name)))
                         (sub3 (unify ((lift sub2) t1)
                                      (FunType t2 new-tvar))))
                     [((lift sub3) new-tvar)
                      (compose sub3 (compose sub2 sub1))]))))))
      ((Let x e body)
         (match (infer e c)
           ([t sub]
              (let ((c' (map (lift sub) c)))
                (match (infer body (Cons (Pair x (generalize t c')) c'))
                  ([t-b sub-b] [t-b (compose sub-b sub)]))))))))
```

```
Function infer defined.
```

Function `infer` is the function $\mathcal{V}$ that was given earlier. The case of conditionals (**if**s) is left as an exercise. Let us define a "global type context" and a function for printing out types in a more readable form:

```
>(define global-type-context
   (Cons (Pair 'succ (FunType IntType IntType))
         (Cons (Pair '< (FunType (ProdType IntType IntType) BoolType))
               Nil)))
```

```
Term global-type-context defined.
```

```
>(define (type->string t)
   (match t
     ((VarType id) (add `' (id->string id)))
     ((FunType t1 t2) (join "(" (type->string t1)
                            " --> " (type->string t2) ")"))
     ((ProdType t1 t2) (join "(" (type->string t1)
                             " * " (type->string t2) ")"))
     (IntType "int")
     (BoolType "bool")
     ((GenType x t) (join "(ALL " (add `' (id->string id))
                          " " (type->string t) ")"))))
```

```
Function type->string defined.
```

We can finally define $\mathcal{W}$ as follows:

```
>(define (W e)
   (print (join "\n" (type->string (head (infer e global-type-context)))
                "\n")))
```

```
Function W defined.
```

We can now test `W` on various expressions:

```
>(define let-exp1
```

```
   (Let 'id (Abs 'x (Var 'x)) (App (Var 'id) (NumConst 876))))

Term let-exp1 defined.

>(define let-exp2 (Let 'id (Abs 'x (Var 'x))
                           (Prod (App (Var 'id) (NumConst 3))
                                 (App (Var 'id) (Var 'true)))))

Term let-exp2 defined.

>(define prod-exp (Prod let-exp2 let-exp2))

Term prod-exp defined.

>(define prod-fun-exp (Abs 'f (App (Var 'f) prod-exp)))

Term prod-fun-exp defined.

>(W let-exp1)

int

Unit: ()

>(W let-exp2)

bool

Unit: ()

>(W prod-exp)

((int * bool) * (int * bool))

Unit: ()

>(W prod-fun-exp)

((((int * bool) * (int * bool)) --> 'T67) --> 'T67)

Unit: ()
```

Note that `W` does not *prove* anything. It essentially "guesses" the most general possible (shallow) type for a given expression by using unification. The algorithm is correct—sound with respect to the inference system—but this is far from obvious. We can prove its soundness, but the proof is neither easy to discover nor particularly easy to follow. Another approach is to express a type inference algorithm as an Athena method. This way every time we get a result, we can be assured that the result is provable from the rules of the type system (our primitive methods). In fact, we can request and obtain that proof explicitly. That proof can be regarded as a correctness *certificate* for the result [1].

1.3 Automatic construction of type derivations

How can we write such a method? First, let us call an expression well-typed with respect to a context A iff we can prove $A \vdash e : \tau$ for some τ. Also, let us take principality for granted here, i.e., let us assume that if an expression e is well-typed at all, then it has a most general type. In other words, for every e and context A, there exists a unique[4] shallow type τ_0 such that:

1. $A \vdash e : \tau_0$; and
2. for all τ, if $A \vdash e : \tau$ then τ is an instance of τ_0.

Now, how can we design a proof recipe that takes an arbitrary expression e and context A, and assuming that e is well-typed in A, *derives* the judgment $A \vdash e : \tau_0$, where τ_0 is the most general type of e in A?

A bit of experimentation will show that the main problem lies in the handling of the λ-bound variables. There is no way to know in what capacity such a variable is used in the body of an abstraction, since there are no explicit type declarations.

To get around this difficulty, let us reduce our problem to an easier problem, which we will then tackle separately. Imagine that instead of implicitly typed expressions, we were instead dealing with expressions that were decorated with explicit type information. In particular, consider the following expression grammar:

$$\begin{array}{rcl} e & ::= & x \mid n \mid \textbf{true} \mid \textbf{false} \mid \lambda\, x : \tau \,.\, e \mid e_1\, e_2 \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \mid \\ & & \langle e_1, e_2 \rangle \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \end{array}$$

where every λ-bound parameter has a type attached to it. Then, given such an e and a context A, we can try to prove a judgment of the form $A \vdash e : \tau$ as follows:

- If e is a variable x and $A(x) = \tau$, then use $[Var]$ to derive $A \vdash e : \tau$.
- If e is a number n or one of the two boolean constants, use one of the three relevant axioms.
- If e is an abstraction $\lambda\, x : \tau \,.\, e'$, use the algorithm recursively on e' and the context $x : \tau.A$. That will presumably derive a judgment of the form $x : \tau.A \vdash e' : \tau'$. Then, using this as a premise, invoke $[Abs]$ to infer $A \vdash \lambda\, x : \tau \,.\, e : \tau \to \tau'$.
- If e is an application $e_1\, e_2$, then use the algorithm recursively on e_1 and A, and then on e_2 and A. If those recursive calls derive judgments of the form $A \vdash e_1 : \tau_1 \to \tau_2$ and $A \vdash e_2 : \tau_1$, then use these as premises for the rule $[App]$ to derive $A \vdash e_1\, e_2 : \tau_2$.
- If e is of the form $\textbf{let}\ x = e_1\ \textbf{in}\ e_2$, then use the algorithm recursively on e_1 and A to derive a judgment of the form $A \vdash e_1 : \tau_1$. Then repeatedly use $[Gen]$ on every type variable in $FV(\tau_1) - FV(A)$, eventually deriving a judgment of the form $A \vdash e_1 : (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau_1$. Finally, use the algorithm recursively on e_2 and A', where $A' = x : (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau_1.A$. If and when this establishes a judgment of the form $A' \vdash e_2 : \tau_2$, use $[Let]$ to infer $A \vdash e : \tau_2$.

The cases of products and conditionals are straightforward.

This is a *recursive proof method* that proceeds by induction on the structure of the given expression. It is essentially a recipe that tells us how to put together a type derivation for any given e and A.

[4] Unique up to alphabetic conversion.

Note that the type of the derived judgment might not be the most general type that we could obtain if we stripped away the types. This is because the types given for the λ-bound parameters might be overly constraining. For instance, given the expression $\lambda\, x : \mathbf{int}\,.\, x$ and the empty type context, our proof method will derive the judgment

$$[] \vdash \lambda\, x : \mathbf{int}\,.\, x : \mathbf{int} \to \mathbf{int}$$

even though if we stripped the type tag **int**, we could derive $[] \vdash \lambda\, x\,.\, x : (\forall\, \alpha)\, \alpha \to \alpha$. Thus our method derives the most general possible typing judgment *modulo* the given type information for the λ-bound variables. So the key observation now becomes that if we could somehow be given the most general possible types for the λ-bound variables, then our method would indeed derive the most general possible type for the entire given expression. For instance, given the most generally explicitly typed expression $\lambda\, x : \alpha\,.\, x$, our method would derive the judgment $[] \vdash \lambda\, x : \alpha\,.\, x : \alpha \to \alpha$, which we could then generalize to

$$[] \vdash \lambda\, x\,.\, x : (\forall\, \alpha)\, \alpha \to \alpha.$$

It turns out that we can easily obtain the most general types of the λ-bound variables of a given expression by using the substitution returned by $\widehat{\mathcal{V}}$, a slight modification of algorithm $\mathcal{V}$.

So our strategy now becomes: given some undecorated expression e and context A, use $\widehat{\mathcal{V}}$ to obtain a decorated expression e', where the explicitly given type of every λ-bound variable in e' is the most general possible. Then use the above proof method on e' and A to derive the desired judgment.

There is one more wrinkle to be removed before this strategy can work. Consider an expression such as

$$\mathbf{let}\ I = \lambda\, x : \alpha\,.\, x\ \mathbf{in}\ \langle I\, 3, I\, \mathbf{true}\rangle.$$

Our proof method will not work here because we will be processing the body of this **let** in a context that (correctly) assigns I the type $(\forall\, \alpha)\, \alpha$; when we come to process the application $I\, 3$, and try to look up I, we will obtain $(\forall\, \alpha)\, \alpha$, which does not match **int**, the type of 3. The whole method will thus fail.

The problem is that although we properly generalize the types of **let**-bound variables, we do not perform the corresponding specializations anywhere. We could tweak our proof method to remedy this, but that would be clumsy: we would have to account for this whenever we apply an inference rule that requires two types to be identical, as in the application rule, which requires the type of the left expression to be $\tau_1 \to \tau_2$ and the type of the right expression to be identical to τ_1; in the conditional rule, which requires the types of the two branching expressions to be identical; and so on.

A more elegant—and efficient—solution is to imagine that we are working with even more decorated expressions. Suppose that every occurrence of a variable x had information attached to it that told us the most general possible type of that particular use (occurrence) of x. In other words, imagine that expressions are generated by the following abstract grammar:

$$\begin{array}{rcl} E & ::= & x : \tau \mid n \mid \mathbf{true} \mid \mathbf{false} \mid \lambda\, x : \tau\,.\, E \mid E_1\, E_2 \mid \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 \mid \\ & & \langle E_1, E_2\rangle \mid \mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3 \end{array} \tag{1.4}$$

Suppose, for example, that the above **let** expression was instead written as follows:

$$\mathbf{let}\ I = \lambda\, x : \alpha\,.\, x\ \mathbf{in}\ \langle I : \mathbf{int} \to \mathbf{int}\ 3, I : \mathbf{bool} \to \mathbf{bool}\ \mathbf{true}\rangle.$$

Then all we have to do when we process the application $I : \mathbf{int} \to \mathbf{int}\ 3$ is make sure that the given type $\mathbf{int} \to \mathbf{int}$ is an *instance* of the type of I in the current context, namely $(\forall\, \alpha)\, \alpha \to \alpha$. We can prove this by using [*Spec*].

Accordingly, we modify our proof method to handle variables as follows:

If e is a variable $x : \tau'$ and $A(x) = (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau$, for $n \geq 0$, then if τ' is an instance of τ under some substitution θ, derive the judgment $A \vdash x : \tau'$ by n applications of [*Spec*], using the substitution $\{\alpha_i \mapsto \overline{\theta}(\alpha_i)\}$ each time. (1.5)

The final question to be answered is: where will the most general types of the individual variable occurrences come from? The answer again is: from the substitution returned by $\widehat{\mathcal{V}}$. Specifically, let us refer to the expressions generated by 1.4 as *decorated*; we will use the letter E to range over decorated expressions, and e to range over regular, undecorated expressions. We now define an algorithm $\mathcal{T}$ that takes an expression e and a context A and returns a decorated version E of e, where the explicit type of every variable occurrence in E is the most general possible. This is achieved by instrumenting $\mathcal{V}$ into an algorithm $\widehat{\mathcal{V}}$ such that, instead of returning a pair (τ, θ), $\widehat{\mathcal{V}}(e, A)$ returns a *triple* (τ, θ, E) consisting of τ and θ as before, and a decorated version E of e. The decorated expressions returned by $\widehat{\mathcal{V}}$ will have general, indeterminate types. The desired decorated expression with the correct most general types will be obtained at the end by applying the final substitution to the final decorated expression returned by $\widehat{\mathcal{V}}$. Specifically, for any substitution θ and decorated expression E, we define $\widehat{\theta}(E)$ as the decorated expression obtained from E by replacing every explicit type τ by $\overline{\theta}(\tau)$. More precisely:

$$\widehat{\theta}(x : \tau) = x : \overline{\theta}(\tau) \tag{1.6}$$

$$\widehat{\theta}(n) = n \tag{1.7}$$

$$\widehat{\theta}(\mathbf{true}) = \mathbf{true} \tag{1.8}$$

$$\widehat{\theta}(\mathbf{false}) = \mathbf{false} \tag{1.9}$$

$$\widehat{\theta}(\lambda\, x : \tau\, .\, E) = \lambda\, x : \overline{\theta}(\tau)\, .\, \widehat{\theta}(E) \tag{1.10}$$

$$\widehat{\theta}(E_1\ E_2) = \widehat{\theta}(E_1)\, \widehat{\theta}(E_2) \tag{1.11}$$

$$\widehat{\theta}(\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2) = \mathbf{let}\ x = \widehat{\theta}(E_1)\ \mathbf{in}\ \widehat{\theta}(E_2) \tag{1.12}$$

$$\widehat{\theta}(\langle E_1, E_2 \rangle) = \langle \widehat{\theta}(E_1), \widehat{\theta}(E_2) \rangle \tag{1.13}$$

$$\widehat{\theta}(\mathbf{if}\ E_1\ \mathbf{then}\ E_2\ \mathbf{else}\ E_3) = \mathbf{if}\ \widehat{\theta}(E_1)\ \mathbf{then}\ \widehat{\theta}(E_2)\ \mathbf{else}\ \widehat{\theta}(E_3) \tag{1.14}$$

We can now define $\mathcal{T}$ as follows:

$\mathcal{T}(e, A) = \overline{\theta}(E)$, where $(\tau, \theta, E) = \widehat{\mathcal{V}}(e, A)$, and
$\widehat{\mathcal{V}}(n, A) = (\mathbf{int}, \{\}, n)$
$\widehat{\mathcal{V}}(\mathbf{true}, A) = (\mathbf{bool}, \{\}, \mathbf{true})$
$\widehat{\mathcal{V}}(\mathbf{false}, A) = (\mathbf{bool}, \{\}, \mathbf{false})$
$\widehat{\mathcal{V}}(x, A) = [A(x) = (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau]? \Rightarrow (\tau', \{\}, x : \tau')$, *raise failure*
where $\tau' = \overline{\{\alpha_1 \mapsto \alpha'_1, \ldots, \alpha_n \mapsto \alpha'_n\}}(\tau)$ and $\alpha'_1, \ldots, \alpha'_n$ are fresh, $n \geq 0$
$\widehat{\mathcal{V}}(\lambda\, x\, .\, e, A) = (\overline{\theta}(\alpha \to \tau), \theta, \lambda\, x : \alpha\, .\, E)$, where $(\tau, \theta, E) = \widehat{\mathcal{V}}(e, x : \alpha . A)$ and α is fresh
$\widehat{\mathcal{V}}(e_1\ e_2, A) = (\overline{\theta}(\alpha), \theta \circ (\theta_2 \circ \theta_1), E_1\ E_2)$,
where $(\tau_1, \theta_1, E_1) = \widehat{\mathcal{V}}(e_1, A)$, $(\tau_2, \theta_2, E_2) = \widehat{\mathcal{V}}(e_2, \overline{\theta_1}(A))$, $\theta = U(\overline{\theta_2}(\tau_1), \tau_2 \to \alpha)$, α fresh
$\widehat{\mathcal{V}}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2, A) = (\tau_2, \theta_2 \circ \theta_1, \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2)$,
where $(\tau_1, \theta_1, E_1) = \widehat{\mathcal{V}}(e_1, A)$, $(\tau_2, \theta_2, E_2) = \widehat{\mathcal{V}}(e_2, A')$, $A' = x : (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau_1 . \overline{\theta_1}(A)$,
and $\{\alpha_1, \ldots, \alpha_n\} = FV(\tau_1) - FV(\overline{\theta_1}(A))$
$\widehat{\mathcal{V}}(\langle e_1, e_2 \rangle, A) = (\overline{\theta}(\tau_1 \times \tau_2), \theta, \langle E_1, E_2 \rangle)$,
where $(\tau_1, \theta_1, E_1) = \widehat{\mathcal{V}}(e_1, A)$, $(\tau_2, \theta_2, E_2) = \widehat{\mathcal{V}}(e_2, \overline{\theta_1}(A))$, $\theta = \theta_2 \circ \theta_1$

$\widehat{\mathcal{V}}(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3, A) = (\overline{\theta'}(\tau_3), \theta' \circ (\theta_3 \circ (\theta_2 \circ (\theta \circ \theta_1))), \textbf{if } E_1 \textbf{ then } E_2 \textbf{ else } E_3)$,
where $(\tau_1, \theta_1, E_1) = \widehat{\mathcal{V}}(e_1, A)$, $\theta = U(\tau_1, \textbf{bool})$, $(\tau_2, \theta_2, E_2) = \widehat{\mathcal{V}}(e_2, \overline{\theta \circ \theta_1}(A))$,
$(\tau_3, \theta_3, E_3) = \widehat{\mathcal{V}}(e_3, \overline{\theta_2 \circ (\theta \circ \theta_1)}(A))$, and $\theta' = U(\overline{\theta_3}(\tau_2), \tau_3)$

Our final algorithm for automatic proof construction, given an undecorated e and a context A, is simply to use the recursive proof method we gave earlier (with the modified variable clause 1.5) on $\mathcal{T}(e, A)$ and A. In the following section we will implement this as an Athena proof method.

1.4 Remaining Athena implementation

We begin by representing decorated expressions:

```
>(structure DecExp
   DTrue
   DFalse
   (DVar Ide Type)
   (DNumConst Num)
   (DAbs Ide Type DecExp)
   (DApp DecExp DecExp)
   (DProd DecExp DecExp)
   (DCond DecExp DecExp DecExp)
   (DLet Ide DecExp DecExp))

New structure DecExp defined.
```

The following function computes $\widehat{\theta}(E)$, as prescribed by equations 1.6–1.14:

```
>(define (apply-sub-to-dec-exp theta E)
   (letrec ((app
              (function (E)
                (match E
                 ((DVar x t) (DVar x ((lift theta) t)))
                 ((DAbs x t E1)  (DAbs x ((lift theta) t) (app E1)))
                 ((DApp E1 E2)  (DApp (app E1) (app E2)))
                 ((DProd E1 E2) (DProd (app E1) (app E2)))
                 ((DCond E1 E2 E3) (DCond (app E1) (app E2) (app E3)))
                 ((DLet x E1 E2)  (DLet x (app E1) (app E2)))
                 (_ E)))))
     (app E)))

Function apply-sub-to-dec-exp defined.
```

We are now ready to define $\mathcal{T}$; its listing appears in Figure 1.2. With $\mathcal{T}$ defined, we can go on to our main prover, the method `prove`. We will need three auxiliary methods `spec-list`, `gen-list`, and `spec-var-type`. Schematically, the first two represent the following pair of inverse rules, respectively:

$$\frac{A \vdash e : (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau}{A \vdash e : \overline{\{\alpha_n \mapsto \tau_n\}}(\cdots \overline{\{\alpha_1 \mapsto \tau_1\}}(\tau) \cdots)}$$

and

$$\frac{A \vdash e : \tau}{\begin{array}{c} A \vdash e : (\forall\, \alpha_1) \cdots (\forall\, \alpha_n)\, \tau \\ \text{where } \{\alpha_1, \ldots, \alpha_n\} = FV(\tau) - FV(A) \end{array}}$$

```
(define (T e c)
  (letrec ((V (function (e c)
                (match e
                  ((Var x)
                     (let ((fin (head (fresh-instance (apply-context c x)))))
                       [fin empty-sub (DVar x fin)]))
                  ((NumConst n) [IntType empty-sub (DNumConst n)])
                  (True [BoolType empty-sub DTrue])
                  (False [BoolType empty-sub DFalse])
                  ((Prod e1 e2)
                     (match (V e1 c)
                       ([t1 sub1 E1]
                          (match (V e2 (map (lift sub1) c))
                            ([t2 sub2 E2]
                               (let ((sub (compose sub2 sub1)))
                                 [((lift sub) (ProdType t1 t2)) sub
                                  (DProd E1 E2)])))))
                  ((Cond e1 e2 e3)
                     (match (V e1 c)
                       ([t1 sub1 E1]
                         (let ((usub (unify t1 BoolType))
                               (sub (compose usub sub1)))
                           (match (V e2 (map (lift sub) c))
                             ([t2 sub2 E2]
                                (let ((sub' (compose sub2 sub)))
                                  (match (V e3 (map (lift sub') c))
                                    ([t3 sub3 E3]
                                       (let ((sub'' (unify ((lift sub3) t2) t3)))
                                         [((lift sub'') t3)
                                          (compose sub'' (compose sub3 sub'))
                                          (DCond E1 E2 E3)])))))))))
                  ((Abs x e1)
                     (let ((param-type (VarType (fresh-type-name))))
                       (match (V e1 (Cons (pair x param-type) c))
                         ([body-type sub E1]
                            [((lift sub) (FunType param-type body-type)) sub
                             (DAbs x param-type E1)]))))
                  ((App e1 e2)
                     (match (V e1 c)
                       ([t1 sub1 E1]
                          (match (V e2 (map (lift sub1) c))
                            ([t2 sub2 E2]
                               (let ((new-tvar (VarType (fresh-type-name)))
                                     (sub3 (unify ((lift sub2) t1)
                                                  (FunType t2 new-tvar))))
                                 [((lift sub3) new-tvar)
                                  (compose sub3 (compose sub2 sub1))
                                  (DApp E1 E2)])))))
                  ((Let x e1 e2)
                     (match (V e1 c)
                       ([t sub E1]
                          (let ((c' (map (lift sub) c)))
                            (match (V e2 (Cons (pair x (generalize t c')) c'))
                              ([t-b sub-b E2]
                                 [t-b (compose sub-b sub) (DLet x E1 E2)])))))))))
     (match (V e c)
       ([_ sub E] (apply-sub-to-dec-exp sub E)))))
```

Figure 1.2: The definition of the algorithm $\mathcal{T}$.

and where $n \geq 0$ in both rules. The first one successively specializes each α_i with some arbitrary type τ_i, while the second one generalizes. These are *derived* inference rules: any application of such a rule can be mechanically replaced by a proper proof that uses only primitive inference rules. Such derived rules can be readily expressed in Athena:

```
>(define (judgment-type P)
   (match P
     ((has-type _ _ t) t)))

Function judgment-type defined.

>(define (spec-list e t c types)
   (dmatch [t types]
     ([(GenType x tb) (list-of t1 rest-types)]
         (dlet ((S (!pm-spec c e x tb t1)))
           (!spec-list e (judgment-type S) c rest-types)))
     (_ (!claim (has-type e c t)))))

Method spec-list defined.

>(define (gen-list S)
   (dletrec ((gen (method (c e t tvars)
                    (dmatch tvars
                      ([] (!claim (has-type e c t)))
                      ((list-of tv rest)
                         (dlet ((S (!pm-gen c e t tv)))
                           (!gen c e (judgment-type S) rest)))))))
     (dmatch S
       ((has-type e c t) (!gen c e t (list-diff (free-type-vars t)
                                                (context-free-type-vars c)))))))

Method gen-list defined.
```

The method `spec-var-type` does all the work described in the variable clause 1.5. Note that unification is used to determine whether the given type is an instance of the generalized type. This is not necessary; simple matching would suffice and would be more efficient. We leave this optimization as an exercise.

```
>(define (spec-var-type x c gt t)
  (dmatch (fresh-instance gt)
    ([gt-inst fresh-types]
       (dlet ((sub (unify gt-inst t)))
         (!spec-list (Var x) gt c (map-list sub fresh-types))))))

Method spec-var-type defined.
```

Finally, the method `prove` is shown in Figure 1.3.

Let us define a top-level type context giving the types of some standard identifiers:

```
>(define global-types
   [['succ (FunType IntType IntType)]
    ['-    (FunType (ProdType IntType IntType) IntType)]
    ['+    (FunType (ProdType IntType IntType) IntType)]
    ['*    (FunType (ProdType IntType IntType) IntType)]
    ['=    (GenType 'T (FunType (ProdType (VarType 'T)
                                          (VarType 'T))
                                BoolType))]])
```

```
(define (prove E c)
  (dmatch E
    ((DVar x t)
       (dmatch (!pm-var c x (apply-context c x))
         ((has-type _ _ gt) (!spec-var-type x c gt t))))
    ((DNumConst n) (!pm-num c n))
    (DTrue (!pm-true c))
    (DFalse (!pm-false c))
    ((DProd E1 E2)
       (dmatch (!prove E1 c)
         ((has-type e1 c t1)
            (dmatch (!prove E2 c)
              ((has-type e2 c t2) (!pm-prod c e1 e2 t1 t2))))))
    ((DAbs x xt E)
       (dlet ((body-theorem (!prove E (Cons (pair x xt) c))))
         (dmatch body-theorem
           ((has-type e _ body-type) (!pm-abs c x e xt body-type)))))
    ((DApp E1 E2)
       (dmatch (!prove E1 c)
         ((has-type e1 _ (FunType t1 t2))
            (dmatch (!prove E2 c)
              ((has-type e2 _ (val-of t1))
                 (!pm-app c e1 e2 t1 t2))))))
    ((DCond E1 E2 E3)
       (dmatch (!prove E1 c)
         ((has-type e1 _ BoolType)
            (dmatch (!prove E2 c)
              ((has-type e2 _ t)
                 (dmatch (!prove E3 c)
                   ((has-type e3 _ (val-of t))
                      (!pm-cond c e1 e2 e3 t))))))))
    ((DLet x E1 E2)
       (dmatch (!prove E1 c)
         ((bind J (has-type e1 _ t1))
            (dmatch (!gen-list J)
              ((has-type e1 _ gen-t1)
                 (dmatch (!prove E2 (Cons (pair x gen-t1) c))
                   ((has-type e2 _ t2) (!pm-let c x e1 e2 gen-t1 t2)))))))))
```

Figure 1.3: The definition of `prove`.

```
List global-types defined.

>(define (make-type-env type-table)
   (match type-table
     ([] Nil)
     ((list-of [x t] rest) (Cons (Pair x t) (make-type-env rest)))))

Function make-type-env defined.

>(define global-type-env (make-type-env global-types))

Term global-type-env defined.
```

The last thing that remains to be done is to generalize the final type produced by `prove`. We thus set:

```
>(define (infer-type e)
   (!gen-list (!prove (T e global-type-env) global-type-env)))

Method infer-type defined.
```

For experimentation purposes we now define two functions that read an expression, either in abstract syntax form or as a string, produce a theorem stating the most general type of the given expression with respect to the global type context, and then print out that type:

```
>(define (get-type-string s)
   (print (join "\n" (type->string (judgment-type (!infer-type (parse-exp s)))) "\n")))

Function get-type-string defined.

>(define (get-type-exp e)
   (print (join "\n" (type->string (judgment-type (!infer-type e))) "\n")))

Function get-type-exp defined.

>(get-type-string "(let (id (fun (x) x)) [(id 3) (id true)])")


int * bool

Unit: ()
```

1.5 Recursion

Recursion does not present any difficulties from a type-theoretic viewpoint. Let us extend the language of expressions with a recursion operator **fix**:

$$e ::= x \mid n \mid \mathbf{true} \mid \mathbf{false} \mid \lambda\, x\,.\, e \mid e_1\, e_2 \mid \mathbf{fix}\, x\,.\, e \mid \cdots$$

The idea is that **fix** acts as a fixed-point operator. Recall that every recursively defined function can be regarded as the least fixed point of an associated functional. For instance, in the recursive definition

$$\mathit{fact} = \lambda\, n\,.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathit{fact}(n-1)$$

we can view *fact* as the least fixed point of the functional

$$\lambda\, \mathit{fact}\,.\, \lambda\, n\,.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathit{fact}(n-1). \tag{1.15}$$

In the untyped λ-calculus there exists a fixed-point operator Y that will find the least fixed point of any functional F like 1.15, so that $F(Y(F)) = Y(F)$. So, for example, we may view *fact* as $Y(F)$, where F is the functional 1.15. The operator **fix** plays the role of Y in Core-ML: we may think of $\mathbf{fix}\, x\,.\, e$ as $Y(\lambda\, x\,.\, e)$. From an evaluation viewpoint, $\mathbf{fix}\, x\,.\, e$ produces e with every free occurrence of x replaced by $\mathbf{fix}\, x\,.\, e$. So *fact* in our setting can be expressed as

$$\mathbf{fix}\, \mathit{fact}\,.\, \lambda\, n\,.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathit{fact}(n-1). \tag{1.16}$$

The typing rule for **fix** is:

$$\frac{x : \tau . A \vdash e : \tau}{A \vdash \mathbf{fix}\, x\,.\, e : \tau} \quad [\mathit{Fix}]$$

The idea is that the bound variable x and the body e must have the same type τ. (Of course discovering *what* that type τ might be is an issue of type inference. Like the previous inference rules, $[\mathit{Fix}]$ assumes

that the appropriate type is somehow "given".) For instance, let A be a type context in which the identifiers $*$ and $-$ have the type $(\mathbf{int} \times \mathbf{int}) \to \mathbf{int}$, while $=$ has the type $(\forall\, \alpha)\,(\alpha \times \alpha) \to \mathbf{bool}$.[5] Then it is clear that in the augmented context $\mathbf{fact} : \mathbf{int} \to \mathbf{int}.A$ we can readily prove the type of the expression

$$\lambda\, n\,.\,\mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathit{fact}(n-1)$$

to be $\mathbf{int} \to \mathbf{int}$. Thus, from [*Fix*], we can now show that the type of 1.16 is also $\mathbf{int} \to \mathbf{int}$.

It is now straightforward to extend our implementation to include **fix** expressions. We first add one extra clause to each of the structures `Exp` and `DecExp`:

```
(structure Exp
...
  (Fix Ide Exp)
...
)
```

and

```
(structure DecExp
...
  (DFix Ide DecExp)
...
)
```

Now the two algorithms `apply-sub-to-dec-exp` and `T` and the method `prove` have one more case to consider. The respective additions are as follows:

```
(define (apply-sub-to-dec-exp theta E)
...
               (match E
                   ...
                 ((DFix x t E1) (DFix x ((lift theta) t) (app E1)))
                   ...) ...)

(define (T e c)
  (letrec ((V (function (e c)
                (match e
                  ...
                  ((Fix x e1)
                     (let ((param-type (VarType (fresh-type-name))))
                       (match (V e1 (Cons (pair x param-type) c))
                         ([body-type sub E1]
                            (let ((usub (unify body-type ((lift sub) param-type)))
                                  (final-sub (compose usub sub)))
                              [((lift final-sub) param-type) final-sub
                               (DFix x param-type E1)]))))
                  ...)...)...)


(define (prove E c)
  (dmatch E
```

[5] Assume that we have written $e_1 * e_2$ as infix syntax sugar for the unary application $* \langle e_1, e_2 \rangle$, and likewise for $e_1 = e_2$ and $e_1 - e_2$.

```
((DVar x t)

...

((DFix x xt E)
   (dlet ((body-theorem (!prove E (Cons (pair x xt) c))))
     (dmatch body-theorem
       ((has-type e _ body-type) (!pm-fix c x e body-type)))))

...))...)
```

Bibliography

[1] K. Arkoudas. Certified Computation. Forthcoming publication, available in the interim from `www.ai.mit.edu/people/koud/cc.ps`.

[2] K. Arkoudas. Denotational Proof Languages. PhD thesis, MIT, 2000.

[3] L. Cardelli. Basic polymorphic type checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[4] L. Damas and R. Milner. Principal type schemes for functional programs. In *Ninth Annual Symposium of Principles of Programming Languages*, pages 207–212, 1982.