15-781: Computation & Deduction Project Report: Intersection Types & Inversion of Type Preservation

Jonathan Moody
jwmoody+@cs

May 11, 2001

1 Introduction

Type preservation has been discussed in class and is a desirable property for a type assignment system to have. However, some terms can be evaluated successfully even though they do not have a type. It would would be good to know that the type system is not too conservative in assigning types. For example, in the context of Mini-ML and its typing rules:

```
eval (app (lam [x] z) (lam [y] fst z)) z  &  of z nat
```

```
but not
```

```
of (app (lam [x] z) (lam [y] fst z)) nat
```

Here, although evaluation is successful, the type system is too strict. It checks terms which have no bearing on the evaluation. We would like to know that whenever evaluation terminates, if a type can be inferred for the result, then that same type can be inferred for the original expression. This is the inverse in some sense of type preservation

$$E \hookrightarrow V \quad \& \quad \cdot \vdash V : \tau \quad \Rightarrow \quad \cdot \vdash E : \tau \tag{1}$$

If this can be proved, then we can be reassured that the type system will not disallow any well-defined programs. However, the implementation of typechecking is another matter. The proof of this property relies on knowledge of the entire evaluation proof tree, which a decideable implementation of typechecking will not have access to.

2 Intersection Types

When I expressed an interest in this problem, it was suggested that I start with a system with intersection types. A type system with intersections has the property of β-invariance according to [1] and [2].

$$M =_\beta N \quad \Rightarrow \quad \cdot \vdash M : \tau \iff \cdot \vdash N : \tau \tag{2}$$

Because of the β-invariance property, it should at least be possible to prove inverted type preservation for the pure lambda calculus fragment of Mini-ML, and possibly for more constructs. To make the system of intersection types interesting, it must be possible to assign more than one type to an expression. Therefore, I start out by building on the subtyping implementation discussed in class. To this I add the following constructors and deduction rules:

```
% New Types

omega : tp.
and   : tp -> tp -> tp.

% Omega Rule
tp_omega : of E omega.

% Intersection
tp_and_i : of E (and T1 T2)
    <- of E T1
    <- of E T2.

% not necessary with subtyping rules, but convenient

tp_and_e1 : of E T1
```

```
     <- of E (and T1 T2).

tp_and_e2 : of E T2
     <- of E (and T1 T2).

% Subtyping of intersection types
sub_and1 : sub (and T1 T2) T
    <- sub T1 T.

sub_and2 : sub (and T1 T2) T
    <- sub T2 T.

sub_andr : sub T (and T1 T2)
    <- sub T T1
    <- sub T T2.
```

One critical property of intersection types is the existence of the universal type ω. It will allow us to assign a non-meaningful type, but a type nonetheless, to any term. It turns out that non-trivial intersections are useful as well, for expressing the type of a term that occurs several times at different types.

3 Theorem: Inverse of Type Preservation

The proof and its supporting lemma were implemented in Twelf. The following is the Elf signature representing the theorem:

```
tpsi  : eval E V -> of V T -> of E T -> type.
%mode tpsi +D +P -Q.
```

The proof strategy used is induction on the derivations of evaluation and typing judgements. Most of the cases consisted of appeals to the induction hypothesis. The interesting cases were ones in which a substitution is performed.

Some cases are especially easy. The evaluation rules for values, for example, have identical expressions on the left and right hand are identical ($E \hookrightarrow V$, where $E \equiv V$). In this case, The proof of $\vdash V : \tau$ can be reused immediately to prove the consequent.

```
tpsi_z : tpsi (ev_z) P P.

tpsi_lam : tpsi (ev_lam) P P.
```

The role of ω is to provide a derivation of a type in cases where the actual type is not know or unimportant. In the case of $ev_f st$, only the first component of the pair has a known type by derivation P. For the second component one can simply provide the trivial derivation (*tp_omega*).

```
tpsi_fst : tpsi (ev_fst D) P (tp_fst Q)
    <- tpsi D (tp_pair (tp_omega) P) Q.
```

The case for *ev_app* is representative of the more difficult cases involving substitution. After one application of the induction hypothesis we have:

$$D = \frac{\dfrac{D1}{e_1 \hookrightarrow lam x.e_1'} \quad \dfrac{D2}{e_2 \hookrightarrow v_2} \quad \dfrac{D3}{[v_2/x]e_1' \hookrightarrow v}}{e_1 e_2 \hookrightarrow v}[ev\_app]$$

$$\text{by IH on D3, P}: \quad [v_2/x]e_1' \vdash \tau : \tag{3}$$

Now it seems reasonable that there should be a "reverse" substitution lemma that would allow us to conclude that:

$$\text{by IH on D3, P}: \quad [v_2/x]e_1' \vdash \tau : \tag{4}$$
$$\text{by reverse subst? and IH on D2 P2}: \quad \vdash e_2 : \tau' \tag{5}$$
$$\text{by reverse subst? and IH on D1 P1}: \quad \vdash e_1 : \tau' \to \tau \tag{6}$$
$$\text{by ev\_app}: \quad e_1 e_2 \vdash \tau : \tag{7}$$

In fact, it is possible to prove such a lemma, and I discuss its implementation in the next section. With such a lemma defined (named 'invs') the implementation of the cases for application become:

```
tpsi_app_1 : tpsi (ev_app D3 (D2 : eval E2 V2) (D1 : eval E1 (lam [x] E1' x)))
      P (tp_app P2 P1)
      <- tpsi D3 P P3'
      <- invs ([x] E1' x) (P3' : of (E1' V2) T) Q R
```

```
      <- tpsi D2 R (P2 : of E2 T2)
      <- tpsi D1 (tp_lam Q) (P1 : of E1 (arrow T2 T)).

tpsi_app_0 : tpsi (ev_app D3 (D2 : eval E2 V2) (D1 : eval E1 (lam [x] E1')))
      P (tp_app P2 P1)
      <- tpsi D3 P P3'
      <- tpsi D2 (tp_omega) P2
      <- tpsi D1 (tp_lam ([x:exp][u:of x omega] P3')) P1.
```

The rule tpsi_app_1 represents the case in which the bound variable 'x' does appear in the function body E1', and tpsi_app_0 the case when it does not. The proof proceeds from the subgoal 1, which gives us $[V2/x]E1' \vdash T$:, to an application of the lemma 'invs'. The results Q and R represent derivations (x:exp of x T2 $\rightarrow$ of (E1' V2) T) and (of V2 T2) respectively. Q and R are then used in conjunction with two more applications of the IH for the argument evaluation D2 and function evaluation D1 components of ev_app.

Other cases, such as those for let value and let name, are similar; they involve substitution and thus require an application of the 'invs' lemma.

Unfortunately, the proof fails on the case for Mini-ML case expressions. I initially thought that derivations for both branches of the case would be necessary to construct a typing derivation. Since only one branch is evaluated, the other branch would have to be assigned type omega. But then the branches would not match in type, unless the other branch is also of the trivial type. This difficulty was surmounted by using the tp_case_zero and tp_case_pos rules, which only require type derivations for one branch.

However, the real obstacle lies in constructing a type derivation for the test expression of the case. For the ev_case_s rule, the second branch may or may not not contain any occurrences of the test expression E1'. If it does not occur, then nothing is available to help prove (s E1') has type nat, which is necessary to type the case statement. The inverted substitution lemma is no help; it will construct the trivial typing derivation tp_omega for E1'. The derivation D1 for evaluation of E1 does not yield anything useful either. Consider the following example:

```
%query 1 1
eval (case (s (pair z z)) z ([x:exp] fst x)) V.
---------- Solution 1 ----------
```

```
V = z.
D = ev_case_s (ev_fst (ev_pair ev_z ev_z)) (ev_s (ev_pair ev_z ev_z)).
-------------------------------------------
```

Evaluation works perfectly on this expression, and it can even serve as the test in our case expression, but any assumption that it is of type nat or pos is unsound. Thus the case in which E1' does occur is impossible to prove either. The type for E1' extracted from the second branch will be derivations for ((nat * nat) & (nat * nat) ...) which cannot be used to prove of (s E1') pos .

4 Inverse Substitution Lemma

Both of the papers [1] and [2] describe the following lemma "algorithmically" in the course of writing out the informal proof of β-invariance. What is required is to extract the parts of the proof tree for $\vdash [V2/x]E1' : T$ that correspond to occurrences of the bound variable. The Elf signature chosen for this lemma was:

```
invs : {E1:exp -> exp} (of (E1 V2) T)
        -> ({x:exp} of x T2 -> of (E1 x) T)
        -> (of V2 T2) -> type.

%mode invs +E +P -Q -R.
```

It was convenient to generate the hypothetical component Q and the typing derivation R for V2 simultaneously. The expression E1 is necessary because the typing derivation P does not contain enough information about the term in question. Without it, one cannot distinguish the case in which the variable or "hole" in E1 is encountered (see below).

```
invs_id : invs ([x] x) P
   ([x:exp] [u:of x T2] u) P.
```

In each case, the typing derivation is traversed in parallel with the expression structure, so that when we arrive at the case invs_id, P is taken as a typing derivation for 'x'. The hypothetical component Q of the judgement is built incrementally from the result of applying the induction hypothesis.

```
invs_s_nat : invs ([x] s (E x)) (tp_s_nat P1)
      ([x:exp][u:of x _] tp_s_nat (Q x u)) R
  <- invs ([x] E x) P1 Q R.
\begin{verbatim}

The case for lambda was one of the harder ones. Of course, the other
cases which involve binding mechanisms such as case and 'letv' were similar.

\begin{verbatim}
invs_lam_1 : invs ([x] lam [y] E x y) (tp_lam P1)
    ([x:exp][u:of x _] tp_lam (Q x u)) R
    <- ({y:exp} {v:of y T1}
       invs ([x] y) v ([y:exp][w:of y omega] v) (tp_omega)
       -> invs ([x] E x y) ((P1 y v):of (E _ y) T)
  ([x:exp][u:of x _] Q x u y v) R).
```

Notice the four parameters in the pattern for Q of the last line. In addition to the parameters 'x' and 'u', there are parameters for 'y' and it type derivation 'u' because these were introduced as parameters before proving the main 'invs' (inverted substitution) goal. The base case for ($[x]y$) is introduced as a hypothesis.

Also interesting were the cases for pair and other expressions in which the variable might occur multiple times. For example:

```
invs_pair_11 : invs ([x] pair (E1 x) (E2 x))
(tp_pair P2 P1) ([x:exp] [u:of x (and T2' T2'')]
   tp_pair (Q2 x (tp_and_e2 u))
   (Q1 x (tp_and_e1 u)))
(tp_and_i R2 R1)
<- invs ([x] E1 x) P1 Q1 R1
<- invs ([x] E2 x) P2 Q2 R2.
```

The bound variable 'x' occurs in both branches of the pair, so two appeals to the induction hypothesis are necessary. This results in two possibly different type derivations R1 and R2. These two derivations are combined with tp_and_i to yield an intersection type for 'x'. Q1 and Q2 are also different; they expect derivations of the (Elf) type of R1 and R2 respectively. Thus it is necessary to "fix up" the derivations when they are combined by using the rules for 'and' elimination.

5 Conclusion

Intersection types were critical to allowing the proof of this theorem. Type omega was used judiciously in cases when the value of a particular subexpression was not relevant to the final result, and finite intersections were used to collect (possibly different) type derivations for each occurrence of the variable x in E when a substitution $[v/x]E$ occurs in the course of evaluation.

As mentioned before, it was not possible to prove this for the entire Mini-ML language. A rule for case statements cannot be constructed because one cannot assure that the test expression is well-typed and in fact a 'nat'. There was also trouble with the Twelf implementation of the rule for 'let name' (an error message about unresolved constraints), but I suspect that this could be fixed given time.

References

[1] M. Coppo, M Dezani Ciancaglini, and B Venneri. "Functional Characters of Solvable Terms" Zeitschrift fur Math, Logik, und Grundlagen d. Math. 1981 (v27) pp45-58.

[2] J. Roger Hindley. "Types with Intersection: An Introduction". 1992 (v4) pp470-486.

6 Appendix

The following files are available via AFS from
/afs/cs.cmu.edu/user/jwmoody/classes/ded-comp/proj

6.1 proof.elf : Implementation of Proof

```
% Lemma: Inversion of Type Derivation on Sustituted Expression
% breaks up type derivation for E1:exp -> exp into two components:
% one for the parameter or "hole" in E1, and a hypothetical
% judgement suitable for typing the rest of the expression.


invs : {E1:exp -> exp} (of (E1 V2) T)
        -> ({x:exp} of x T2 -> of (E1 x) T)
        -> (of V2 T2) -> type.

%name invs I.
%mode invs +E +P -Q -R.



% subtyping rule

invs_sub : invs ([x] E x) (tp_sub P2 P1)
    Q R
    <- invs ([x] E x) P1 Q R.


% Encountering the typing derivation for the variable in question:
% at this point, simply copy the derivation that appears in the input.

invs_id : invs ([x] x) P
   ([x:exp] [u:of x T2] u) P.



% Nats:

% x does not occur

invs_z_nat : invs ([x] z) (tp_z_nat)
      ([x:exp][u:of x omega] (tp_z_nat)) (tp_omega).
```

```
invs_z_zero : invs ([x] z) (tp_z_zero)
       ([x:exp][u:of x omega] (tp_z_zero)) (tp_omega).

% x might occur

invs_s_nat : invs ([x] s (E x)) (tp_s_nat P1)
      ([x:exp][u:of x _] tp_s_nat (Q x u)) R
  <- invs ([x] E x) P1 Q R.

invs_s_pos : invs ([x] s (E x)) (tp_s_pos P1)
      ([x:exp][u:of x _] tp_s_pos (Q x u)) R
  <- invs ([x] E x) P1 Q R.


% Pairs : 3 cases for all the possible occurrence patterns

invs_pair_11 : invs ([x] pair (E1 x) (E2 x))
(tp_pair P2 P1) ([x:exp] [u:of x (and T2' T2'')]
   tp_pair (Q2 x (tp_and_e2 u))
   (Q1 x (tp_and_e1 u)))
(tp_and_i R2 R1)
<- invs ([x] E1 x) P1 Q1 R1
<- invs ([x] E2 x) P2 Q2 R2.

invs_pair_10 : invs ([x] pair (E1 x) E2) (tp_pair P2 P1)
([x:exp][u:of x _] tp_pair P2 (Q1 x u)) R1
<- invs ([x] E1 x) P1 Q1 R1.



invs_pair_01 : invs ([x] pair E1 (E2 x)) (tp_pair P2 P1)
([x:exp][u:of x _] tp_pair (Q2 x u) P1) R2
<- invs ([x] E2 x) P2 Q2 R2.




invs_fst : invs ([x] fst (E x)) (tp_fst P1)
    ([x:exp][u:of x _] tp_fst (Q x u)) R
     <- invs ([x] E x) P1 Q R.

invs_snd : invs ([x] snd (E x)) (tp_snd P1)
    ([x:exp][u:of x _] tp_snd (Q x u)) R
     <- invs ([x] E x) P1 Q R.
```

```
% Lambda : 2 cases (one in which the parameter [y] does not occur)

invs_lam_1 : invs ([x] lam [y] E x y) (tp_lam P1)
    ([x:exp][u:of x _] tp_lam (Q x u)) R
    <- ({y:exp} {v:of y T1}
       invs ([x] y) v ([y:exp][w:of y omega] v) (tp_omega)
       -> invs ([x] E x y) ((P1 y v):of (E _ y) T)
  ([x:exp][u:of x _] Q x u y v) R).

invs_lam_0 : invs ([x] lam [y] E x) (tp_lam P1)
    ([x:exp][u:of x _] tp_lam (Q x u)) R
      <- ({y:exp} {v:of y T1}
    invs ([x] E x) ((P1 y v):of (E _) T)
    ([x:exp][u:of x _] Q x u y v) R).


% Application : 3 cases for all the possible occurrence patterns

invs_app_11 : invs ([x] app (E1 x) (E2 x)) (tp_app P2 P1)
       ([x:exp][u:of x (and T' T'')] tp_app
  (Q2 x (tp_and_e1 u)) (Q1 x (tp_and_e2 u)))
       (tp_and_i R1 R2)
       <- invs ([x] E1 x) P1 Q1 R1
       <- invs ([x] E2 x) P2 Q2 R2.

invs_app_10 : invs ([x] app (E1 x) E2) (tp_app P2 P1)
       ([x:exp][u:of x _] tp_app P2 (Q1 x u)) R1
       <- invs ([x] E1 x) P1 Q1 R1.

invs_app_01 : invs ([x] app E1 (E2 x)) (tp_app P2 P1)
       ([x:exp][u:of x _] tp_app (Q2 x u) P1) R2
       <- invs ([x] E2 x) P2 Q2 R2.


% Case : 7 cases for all the possible occurrence patterns

invs_case_nat_111 : invs ([x] case (E1 x) (E2 x) ([y] E3 x y)) (tp_case_nat P3 P2 P1)
     ([x:exp][u:of x (and (and T' T'') T''')]
tp_case_nat (Q3 x (tp_and_e2 u))
            (Q2 x (tp_and_e2 (tp_and_e1 u)))
            (Q1 x (tp_and_e1 (tp_and_e1 u))))
     (tp_and_i R3 (tp_and_i R2 R1))
     <- invs ([x] E1 x) P1 Q1 R1
     <- invs ([x] E2 x) P2 Q2 R2
```

```
     <- ({y:exp} {v:of y nat}
   invs ([x] y) v ([y:exp][w:of y omega] v) (tp_omega)
   -> invs ([x] E3 x y) ((P3 y v):of (E3 _ y) T)
      ([x:exp][u:of x _] Q3 x u y v) R3).

invs_case_nat_100 : invs ([x] case (E1 x) E2 ([y] E3 y)) (tp_case_nat P3 P2 P1)
     ([x:exp][u:of x _] tp_case_nat P3 P2 (Q1 x u))
     R1
     <- invs ([x] E1 x) P1 Q1 R1.

invs_case_nat_010 : invs ([x] case E1 (E2 x) ([y] E3)) (tp_case_nat P3 P2 P1)
     ([x:exp][u:of x _] tp_case_nat P3 (Q2 x u) P1)
     R2
     <- invs ([x] E2 x) P2 Q2 R2.

invs_case_nat_110 : invs ([x] case (E1 x) (E2 x) ([y] E3 y)) (tp_case_nat P3 P2 P1)
     ([x:exp][u:of x (and T' T'')]
tp_case_nat P3
                                    (Q2 x (tp_and_e1 u))
            (Q1 x (tp_and_e2 u)))
     (tp_and_i R1 R2)
     <- invs ([x] E1 x) P1 Q1 R1
     <- invs ([x] E2 x) P2 Q2 R2.


invs_case_nat_001 : invs ([x] case E1 E2 ([y] E3 x y)) (tp_case_nat P3 P2 P1)
     ([x:exp][u:of x _] tp_case_nat (Q3 x u) P2 P1)
     R3
     <- ({y:exp} {v:of y nat}
   invs ([x] y) v ([y:exp][w:of y omega] v) (tp_omega)
   -> invs ([x] E3 x y) ((P3 y v):of (E3 _ y) T)
      ([x:exp][u:of x _] Q3 x u y v) R3).

invs_case_nat_101 : invs ([x] case (E1 x) E2 ([y] E3 x y)) (tp_case_nat P3 P2 P1)
     ([x:exp][u:of x (and T' T'')]
tp_case_nat (Q3 x (tp_and_e1 u))
            P2
            (Q1 x (tp_and_e2 u)))
     (tp_and_i R1 R3)
     <- invs ([x] E1 x) P1 Q1 R1
     <- ({y:exp} {v:of y nat}
   invs ([x] y) v ([y:exp][w:of y omega] v) (tp_omega)
   -> invs ([x] E3 x y) ((P3 y v):of (E3 _ y) T)
      ([x:exp][u:of x _] Q3 x u y v) R3).
```

```
invs_case_nat_011 : invs ([x] case E1 (E2 x) ([y] E3 x y)) (tp_case_nat P3 P2 P1)
     ([x:exp][u:of x (and T' T'')]
 tp_case_nat (Q3 x (tp_and_e1 u))
             (Q2 x (tp_and_e2 u))
             P1)
     (tp_and_i R2 R3)
     <- invs ([x] E2 x) P2 Q2 R2
     <- ({y:exp} {v:of y nat}
   invs ([x] y) v ([y:exp][w:of y omega] v) (tp_omega)
   -> invs ([x] E3 x y) ((P3 y v):of (E3 _ y) T)
      ([x:exp][u:of x _] Q3 x u y v) R3).


% Let Value and Let Name

% not implemented, but should be similar to lam


% Recursion

invs_case_fix : invs ([x] fix [y] E x y) (tp_fix P1)
 ([x:exp][u: of x T2] tp_fix (Q x u)) R
 <- ({y:exp} {v:of y T1}
       invs ([x] y) v ([y:exp][w:of y omega] v) (tp_omega)
       -> invs ([x] E x y) ((P1 y v):of (E _ y) T1)
  ([x:exp][u:of x T2] Q x u y v) R).


%terminates {E P} (invs E P _ _).




% Inverted Type Preservation:
% if  :- V : T  &  eval E V  then it should be possible to construct
% a typing dervation :- E : T , for the same type T.

tpsi  : eval E V -> of V T -> of E T -> type.
%mode tpsi +D +P -Q.


% cases by typing derivation
% If attempting to prove 'of E omega', simply return trivial derivation.

tpsi_omega : tpsi D (tp_omega) (tp_omega).
```

```
% subtyping rule

tpsi_sub : tpsi D (tp_sub P2 P1) (tp_sub P2 Q)
    <- tpsi D P1 Q.


% cases by evaluation (and sometimes typing derivation)

% Nats:

tpsi_z : tpsi (ev_z) P P.

tpsi_s_nat : tpsi (ev_s D1) (tp_s_nat P1) (tp_s_nat Q)
      <- tpsi D1 P1 Q.

tpsi_s_pos : tpsi (ev_s D1) (tp_s_pos P1) (tp_s_pos Q)
      <- tpsi D1 P1 Q.



% Case :

tpsi_case_z: tpsi (ev_case_z (D2 : eval E2 V2) (D1 : eval E1 z)) P
      (tp_case_zero Q2 Q1)
      <- tpsi D2 P Q2
      <- tpsi D1 (tp_z_zero) Q1.


%{

% for the case with occurrences in the second branch, I check
% to see that the type of [x] is a subtype of nat. However, this
% will create coverage problems because it is possible to evaluate
% a case statement with a non well-typed test expression.
% This case is hopeless in general.

tpsi_case_s_1 : tpsi (ev_case_s (D3 : eval (([x:exp] E3 x) V1') V)
(D1 : eval E1 (s V1'))) P
(tp_case_pos ([x:exp][u:of x nat] (Q3 x (??? u))) P1)
 <- tpsi D3 P P'
 <- invs ([x] E3 x) P' Q3 (R : of V1' T2)
%  <- sub T2 nat
 <- ({P1':sub T2 nat}
       tpsi D1 (tp_s_pos (tp_sub P1' R)) P1).
```

```
% for the case when there are no occurrences of E1' in the second
% branch, I have to make the arbitrary assumption (of E1' nat)

tpsi_case_s_0 : tpsi (ev_case_s (D3 : eval (([x:exp] E3) E1') V)
(D1 : eval E1 (s E1'))) P
 (tp_case_pos ([x:exp][u:of x nat] P') Q1)
 <- tpsi D3 P P'
 <- ({u:of E1' nat} tpsi D1 (tp_s_pos u) Q1). % CHEATING


}%


% Pairs:

tpsi_pair : tpsi (ev_pair D2 D1) (tp_pair P2 P1) (tp_pair Q2 Q1)
     <- tpsi D1 P1 Q1
     <- tpsi D2 P2 Q2.

tpsi_fst : tpsi (ev_fst D) P (tp_fst Q)
    <- tpsi D (tp_pair (tp_omega) P) Q.

tpsi_snd : tpsi (ev_snd D) P (tp_snd Q)
    <- tpsi D (tp_pair P (tp_omega)) Q.


% Functions:

tpsi_lam : tpsi (ev_lam) P P.


tpsi_app_1 : tpsi (ev_app D3 (D2 : eval E2 V2) (D1 : eval E1 (lam [x] E1' x)))
      P (tp_app P2 P1)
      <- tpsi D3 P P3'
      <- invs ([x] E1' x) (P3' : of (E1' V2) T) Q R
      <- tpsi D2 R (P2 : of E2 T2)
      <- tpsi D1 (tp_lam Q) (P1 : of E1 (arrow T2 T)).

tpsi_app_0 : tpsi (ev_app D3 (D2 : eval E2 V2) (D1 : eval E1 (lam [x] E1')))
      P (tp_app P2 P1)
      <- tpsi D3 P P3'
      <- tpsi D2 (tp_omega) P2
      <- tpsi D1 (tp_lam ([x:exp][u:of x omega] P3')) P1.
```

```
% Let expressions

tpsi_letv_1 : tpsi (ev_letv (D2 : eval (([x:exp] E2' x) _) _) D1) P (tp_letv  Q P1)
       <- tpsi D2 P P2'
       <- invs ([x] E2' x) P2' Q R
       <- tpsi D1 R P1.


tpsi_letv_0 : tpsi (ev_letv (D2 : eval (([x:exp] E2') _) _) D1) P
       (tp_letv ([x:exp][u:of x omega] P2) P1)
       <- tpsi D2 P P2
       <- tpsi D1 (tp_omega) P1.

% Error message: Typing Ambiguous, unresolved constraints

%{
Typing ambiguous -- unresolved constraints
`E2 `E1 = E3 `E1;
`E2 `E1 = E3 `E1.

tpsi_letn_1 : tpsi (ev_letn (D1 : eval (([x:exp] E2 x) E1) V)) P
       (tp_letn P1' R)
     <- tpsi D1 P (P1' : of (E2 E1) T)
     <- invs ([x] E2 x) P1' Q R.

tpsi_letn_0 : tpsi (ev_letn (D1 : eval (([x:exp] E2) _) V)) P
       (tp_letn P1' tp_omega)
     <- tpsi D1 P P1'.
}%


% Recursion:

tpsi_fix : tpsi (ev_fix (D1 : eval (([x:exp] E x) _) _)) P (tp_fix Q)
    <- tpsi D1 P P1'
    <- invs ([x] E x) P1' Q R.


%terminates {D P} (tpsi D P _).
```

6.2 test.elf : Examples

```
% self application

self = (lam [x] (app x x)).

_ : of self (arrow (and T1 (arrow T1 T2)) T2)
     = tp_lam ([x:exp] [u:of x (and T1 (arrow T1 T2))]
 (tp_app (tp_and_e1 u) (tp_and_e2 u))).


% typing rules not executeable

% %query 1 1
% D : of self (arrow (and T1 (arrow T1 T2)) T2).



% testing inverted subsitution lemma

%query 1 1
Z0 : eval (app (lam [x] (pair (s z) (lam [y] x))) z) V.

Z1 : of (pair (s z) (lam ([x:exp] z))) (cross nat (arrow T nat)) =
    tp_pair (tp_lam ([y:exp][u:of y T] (tp_z_nat))) (tp_s_nat (tp_z_nat)).

%query 1 1
Z2 : invs ([x] (pair (s z) (lam [y] x)) ) Z1 Q R.




% testing tpsi (converse of type preservation)

% tpsi  : eval E V -> of V T -> of E T -> type.
% %mode tpsi +D +P -Q.

% app without occurrence

%query 1 1
D : eval (app (lam [k] z) (s (s z))) V.

P : (of z nat) = (tp_z_nat).

D = ev_app ev_z (ev_s (ev_s ev_z)) ev_lam.
```

```
%query 2 *
I : tpsi D P P'.



% app with occurrence in body

%query 1 1
D : eval (app (lam [x] (s x)) z) V.

P : (of (s z) nat) = tp_s_nat tp_z_nat.

D = ev_app (ev_s ev_z) ev_z ev_lam.

%query 5 *
I : tpsi D P P'.



%{

% simple case statement

%query 1 1
D : eval (case (s z) (fst z) ([y] (pair y (s y)))) V.

P : (of (pair z (s z)) (cross nat nat)) = tp_pair (tp_s_nat tp_z_nat) tp_z_nat.

D = ev_case_s (ev_pair (ev_s ev_z) ev_z) (ev_s ev_z).

%query 1 1
I : tpsi D P P'.
}%

%{
% case with one branch not taken

%query 1 1
D : eval (case (app (lam [x] (s x)) z) (fst z) ([y] (pair y (app (lam [k] z) (s y))))) V.

P : (of (pair z z) (cross nat nat)) = tp_pair tp_z_nat tp_z_nat.

D = ev_case_s (ev_pair (ev_app ev_z (ev_s ev_z) ev_lam) ev_z)
      (ev_app (ev_s ev_z) ev_z ev_lam).
```

```
%query 1 1
I : tpsi D P P'.


% double check type derivation

_ : of (case (app (lam [x] (s x)) z) (fst z) ([y] (pair y (app (lam [k] z) (s y))))) (cross nat
   tp_case_pos
      ([x:exp] [u:of x nat]
          tp_pair
             (tp_app tp_omega (tp_lam ([x1:exp] [u1:of x1 omega] tp_z_nat))) u)
      (tp_app tp_z_nat (tp_lam ([x:exp] [u:of x nat] tp_s_pos u))).





% same case statement except y does not occur in branch of case

%query 1 1
D : eval (case (app (lam [x] (s x)) z) (fst z) ([y] (pair z (app (lam [k] z) (s z))))) V.

P : (of (pair z z) (cross nat nat)) = tp_pair tp_z_nat tp_z_nat.

D = ev_case_s (ev_pair (ev_app ev_z (ev_s ev_z) ev_lam) ev_z)
     (ev_app (ev_s ev_z) ev_z ev_lam).

%query 1 1
I : tpsi D P P'.

}%



% non well typed "nat".

%query 1 1
D : eval (s (s (pair z (s (s z))))) V.


% non well-typed case statement

%query 1 1
D : eval (case (s (pair z z)) z ([x] fst x)) V.
```

```
% polymorphism

%query 1 1
D : eval (letv (lam [x] x) ([f:exp] (pair (app f z) (app f (pair z z))))) V.

D =
   ev_letv
      (ev_pair (ev_app (ev_pair ev_z ev_z) (ev_pair ev_z ev_z) ev_lam)
          (ev_app ev_z ev_z ev_lam)) ev_lam.

P : (of (pair z (pair z z)) (cross nat (cross nat nat)))
     = tp_pair (tp_pair tp_z_nat tp_z_nat) tp_z_nat.


%query 1 1
I : tpsi D P P'.

_ : of (letv (lam [x] x) ([f:exp] (pair (app f z) (app f (pair z z))))) (cross nat (cross nat na
     = tp_letv
      ([x:exp]
          [u:of x
                (and (and omega (arrow nat nat))
                    (and (and omega omega)
                        (arrow (cross nat nat) (cross nat nat))))]
          tp_pair (tp_app (tp_pair tp_z_nat tp_z_nat) (tp_and_e2 (tp_and_e2 u)))
             (tp_app tp_z_nat (tp_and_e2 (tp_and_e1 u))))
      (tp_and_i
          (tp_and_i (tp_lam ([x:exp] [u:of x (cross nat nat)] u))
              (tp_and_i tp_omega tp_omega))
          (tp_and_i (tp_lam ([x:exp] [u:of x nat] u)) tp_omega)).
```