Comp Sci 411 – Type Inference Study Guide

Corky Cartwright

Produced: May 5, 2021

1 Synopsis of Implicitly Polymorphic Jam

The syntax of (Implicitly) Polymorphic Jam is a restriction of the syntax of untyped Jam. Every legal Polymorphic Jam program is also a legal untyped Jam Program. But the converse is false, because there may not be a valid typing for a given untyped Jam program.

1.1 Abstract Syntax

The following grammar describes the abstract syntax of Polymorphic Jam. Each clause in the grammar corresponds directly to a node in the abstract syntax tree. The `let` construction has been limited to a single binding for the sake of notational simplicity. It is straightforward to generalize the rule to multiple bindings (with mutual recursion). Note that `let` is *recursive*.

$$
\begin{array}{rl}
M ::= & M\ \texttt{(}M \cdots M\texttt{)} \mid P\ \texttt{(}M \cdots M\texttt{)} \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \mid \texttt{let}\ x\ \texttt{:=}\ M\ \texttt{in}\ M \mid V \\
V ::= & \texttt{map}\ x \cdots x\ \texttt{to}\ M \mid x \mid n \mid \texttt{true} \mid \texttt{false} \mid \texttt{empty} \\
n ::= & \texttt{1} \mid \texttt{2} \mid \texttt{...} \\
P ::= & \texttt{cons} \mid \texttt{first} \mid \texttt{rest} \mid \texttt{empty?} \mid \texttt{cons?} \mid \texttt{+} \mid \texttt{-} \mid \texttt{/} \mid \texttt{*} \mid \texttt{=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{<-} \mid \texttt{+} \mid \texttt{-} \mid \\
 & \texttt{\~{}} \mid \texttt{ref} \mid \texttt{!} \\
x ::= & \text{variable names}
\end{array}
$$

In the preceding grammar, unary and binary operators are treated exactly like primitive functions.

Monomorphic types in the language are defined by τ, below. Polymorphic types are defined by σ. The $\rightarrow$ corresponds to a function type, whose inputs are to the left of the arrow and whose output is to the right of the arrow.

$$
\begin{array}{rl}
\sigma ::= & \forall \alpha_1 \cdots \alpha_n . \tau \\
\tau ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau_1 \times \cdots \times \tau_n \rightarrow \tau \mid \alpha \mid \mathsf{list}\,\tau \mid \mathsf{ref}\,\tau \\
\alpha ::= & \text{type variable names}
\end{array}
$$

1.2 Type Checking Rules

Each proof rule in our type system is formulated as a "natural deduction" rule as originally conceived by Gerhard Gentzen. For each construct in the language including the applications of unary and binary operators (which are treated as applications of unary and binary functions, respectively, where infix notation is viewed as syntactic sugar for explicit function applications) has an associated rule consisting of a conclusion that is a typing judgement $\Gamma \vdash M : \tau$ (where Γ is a type environment mapping program variables to types or type schemes and the outermost operation in M matches the name of the rule) and a finite set of premises which are typing judgments that can be syntactically constructed from the conclusion $\Gamma \vdash M : \tau$. The type environment in each typing judgment in the rule is a finite set of declarations of the form $x : \tau$ or $x : \sigma$. Note that the *only* place where type schemes σ may appear is within type environments; they *never* appear on the right hand side of $\vdash$ in a typing judgment.

In the following rules, the notation $\Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n]$ means $\Gamma\backslash\{x_1, \ldots, x_n\} \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ and Γ' abbreviates $\Gamma[x_1 : \tau_1', \ldots, x_n : \tau_n']$. Note that $\Gamma \setminus \{x_1, \ldots, x_n\}$ means Γ less the type assertions (if any) for $\{x_1, \ldots, x_n\}$.

$$\frac{\Gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash M : \tau}{\Gamma \vdash \texttt{map } x_1 \ldots x_n \texttt{ to } M : \tau_1 \times \cdots \times \tau_n \to \tau}[\textbf{abs}]$$

$$\frac{\Gamma \vdash M : \tau_1 \times \cdots \times \tau_n \to \tau \quad \Gamma \vdash M_1 : \tau_1 \quad \cdots \quad \Gamma \vdash M_n : \tau_n}{\Gamma \vdash M \ \texttt{(}M_1 \cdots M_n\texttt{)} : \tau}[\textbf{app}]$$

$$\frac{\Gamma \vdash M_1 : \textsf{bool} \quad \Gamma \vdash M_2 : \tau \quad \Gamma \vdash M_3 : \tau}{\Gamma \vdash \texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 : \tau}[\textbf{if}]$$

Note that there are two rules for `let` expressions. The [**letmono**] rule corresponds to the **let** rule of Typed Jam; it places no restriction on the form of the right-hand side M_1 of the `let` binding. The [**letpoly**] rule generalizes the free type variables (not occurring in the type environment Γ) in the type inferred for the right-hand-side of a `let` binding – provided that the right-hand-side M_1 is a *syntactic* value: a *constant* like `empty` or `cons`, a `map` expression, or a variable. Syntactic values are expressions whose evaluation is trivial, excluding evaluations that allocate storage.

$$\Gamma[x : \tau] \vdash x : \tau$$

$$\frac{\Gamma' \vdash M_1 : \tau_1' \quad \ldots \quad \Gamma' \vdash M_n : \tau_n' \quad \Gamma' \vdash M : \tau}{\Gamma \vdash \texttt{let } x_1 \texttt{ := } M_1\texttt{; } \ldots\texttt{; } x_n \texttt{ := } M_n\texttt{; in } M : \tau}[\textbf{letmono}]$$

$$\frac{\Gamma' \vdash M_1 : \tau_1' \quad \ldots \quad \Gamma' \vdash M_n : \tau_n' \quad \Gamma[x_1 : C_{M_1}(\tau_1', \Gamma), \ldots, x_n : C_{M_n}(\tau_n', \Gamma)] \vdash M : \tau}{\Gamma \vdash \texttt{let } x_1 \texttt{ := } M_1\texttt{; } \ldots\texttt{; } x_n \texttt{ := } M_n\texttt{; in } M : \tau}[\textbf{letpoly}]$$

$$\Gamma[x : \forall\alpha_1, \ldots, \alpha_n.\, \tau] \vdash x : O(\forall\alpha_1, \ldots, \alpha_n.\, \tau,\ \tau_1, \ldots, \tau_n)$$

The functions $O(\cdot, \cdot)$ and $C.(\cdot, \cdot)$ are the keys to polymorphism. Here is how $C.(\cdot, \cdot)$ is defined:

$$C_V(\tau, \Gamma) := \forall\{\mathrm{FTV}(\tau) - \mathrm{FTV}(\Gamma)\}.\, \tau$$

$$C_N(\tau, \Gamma) := \tau$$

where V is a syntactic value, N is an expression that is not a syntactic value, and $\mathrm{FTV}(\alpha)$ means the "free type variables in the expression (or type environment) α".

When closing over a type, you must find all of the free variables in τ that are not free in any of the types in the environment Γ. Then, build a polymorphic type by quantifying τ over all of those type variables.

To open a polymorphic type

$$\forall\alpha_1, \ldots, \alpha_n.\, \tau,$$

substitute *any* type terms $\tau_1, \ldots, \tau_n$ for the quantified type variables $\alpha_1, \ldots, \alpha_n$:

$$O(\forall\alpha_1, \ldots, \alpha_n.\, \tau,\ \tau_1, \ldots, \tau_n) = \tau_{[\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]}$$

which creates a monomorphic type from a polymorphic type. For example,

$$O(\forall\alpha.\, \alpha \to \alpha, \tau) = \tau \to \tau$$

1.3 Types of Primitives

The following table gives types for all of the primitive constants, functions, and operators. The symbol n stands for any integer constant. Programs are type checked starting with a primitive type environment Γ_0 consisting of this table.

`true`	bool
`false`	bool
n	int
`empty`	$\forall\alpha.\,\mathsf{list}\,\alpha$
`cons`	$\forall\alpha.\,\alpha \times \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha$
`first`	$\forall\alpha.\,\mathsf{list}\,\alpha \to \alpha$
`rest`	$\forall\alpha.\,\mathsf{list}\,\alpha \to \mathsf{list}\,\alpha$
`cons?`	$\forall\alpha.\,\mathsf{list}\,\alpha \to \mathsf{bool}$
`empty?`	$\forall\alpha.\,\mathsf{list}\,\alpha \to \mathsf{bool}$
`=`	$\forall\alpha.\,\alpha \times \alpha \to \mathsf{bool}$
`!=`	$\forall\alpha.\,\alpha \times \alpha \to \mathsf{bool}$

`+`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
`-`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
`*`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
`/`	$\mathsf{int} \times \mathsf{int} \to \mathsf{int}$
`<`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
`>`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
`<=`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
`>=`	$\mathsf{int} \times \mathsf{int} \to \mathsf{bool}$
(unary) `-`	$\mathsf{int} \to \mathsf{int}$
(unary) `+`	$\mathsf{int} \to \mathsf{int}$
(unary) `~`	$\mathsf{bool} \to \mathsf{bool}$
`<-`	$\forall\alpha.\,\mathsf{ref}\,\alpha \times \alpha \to \mathsf{unit}$
`ref`	$\forall\alpha.\,\alpha \to \mathsf{ref}\,\alpha$
`!`	$\forall\alpha.\,\mathsf{ref}\,\alpha \to \alpha$

1.4 Typed Jam

The Typed Jam language used in Assignment 5 (absent the explicit type information embedded in program text) can be formalized as a subset of Polymorphic Jam. For the purposes of these exercises, Typed Jam is simply Polymorphic Jam less the **letpoly** inference rule which prevents it from inferring polymorphic types for program-defined functions.

2 Exercises

Task 1: Prove the following type judgements for Typed Jam or explain why they are not provable:

1. Γ_0 `|- (map x to x(10))(map x to x) :  int`
2. Γ_0 `|- let fact := map n to if n=0 then 1 else n*(fact(n-1));`
   `in fact(10)+fact(0) :  int`
3. Γ_0 `|- (map x to 1 + (1/x))(0) :  int`
4. Γ_0 `|- (map x to x) (map y to y) :  (int -> int)`
5. Γ_0 `|- let id := map x to x; in id(id) :  (int -> int)`

Task 2: Are the following Polymorphic Jam programs typable? Justify your answer either by giving a proof tree (constructed using the inference rules for PolyJam) or by showing a conflict in the type constraints generated by matching the inference rules against the program text.

1.
```
let listMap := map f,l to
                 if empty?(l) then empty
                 else cons(f(first(l)), listMap(f, rest(l)))
in listMap(first,empty);
```

2.
```
let length := map l to if empty?(l) then 0
                       else 1 + length(rest(l));
         l := cons(cons(1,empty),cons(cons(2,cons(3,empty)),empty));
in length(l)+length(first(l))
```

Task 3: Give a simple example of an untyped Jam expression that is not typable in Typed Jam but is typable in Polymorphic Jam.

3 Solutions to Selected Exercises

Task 1 : The first four expressions are typable in Typed Jam, but the fifth is not.

1. *Tree 1*:

$$\dfrac{\dfrac{\Gamma_0[\mathtt{f}:\mathsf{int}\to\mathsf{int}]\vdash \mathtt{10}:\mathsf{int} \qquad \Gamma_0[\mathtt{f}:\mathsf{int}\to\mathsf{int}]\vdash \mathtt{f}:\mathsf{int}\to\mathsf{int}}{\Gamma_0[\mathtt{f}:\mathsf{int}\to\mathsf{int}]\vdash \mathtt{f(10)}:\mathsf{int}}[\textbf{app}]}{\Gamma_0\vdash \mathtt{map\ f\ to\ f(10)}:(\mathsf{int}\to\mathsf{int})\to\mathsf{int}}[\textbf{abs}]$$

   Tree 2:

$$\dfrac{Tree\ 1 \qquad \dfrac{\Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{x}:\mathsf{int}}{\Gamma_0\vdash \mathtt{map\ x\ to\ x}:\mathsf{int}\to\mathsf{int}}[\textbf{abs}]}{\Gamma_0\vdash \mathtt{(map\ f\ to\ f(10))(map\ x\ to\ x)}:\mathsf{int}}[\textbf{app}]$$

2. Type Inference Proof Omitted.

3. *Tree 1*:

$$\dfrac{\Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{/}:\mathsf{int}\times\mathsf{int}\to\mathsf{int} \qquad \Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{1}:\mathsf{int} \qquad \Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{x}:\mathsf{int}}{\Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{1/x}:\mathsf{int}]}[\textbf{app}]$$

   Tree 2:

$$\dfrac{\dfrac{\Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{+}:\mathsf{int}\times\mathsf{int}\to\mathsf{int} \qquad \Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{1}:\mathsf{int} \qquad Tree\ 1}{\Gamma_0[\mathtt{x}:\mathsf{int}]\vdash \mathtt{(1\ +\ (1/x))}:\mathsf{int}}[\textbf{app}]}{\Gamma_0\vdash \mathtt{(map\ x\ to\ 1\ +\ (1/x))}:\mathsf{int}\to\mathsf{int}}[\textbf{abs}]$$

   Tree 3:

$$\dfrac{Tree\ 2 \quad \Gamma_0\vdash \mathtt{0}:\mathsf{int}}{\Gamma_0\vdash \mathtt{(map\ x\ to\ 1\ +\ (1\ /x))(0)}:\mathsf{int}}[\textbf{app}]$$

4. *Tree 1*:

$$\dfrac{\Gamma_0\,\mathtt{[x}:\mathsf{int}\to\mathsf{int}\mathtt{]}\vdash \mathtt{x}:\mathsf{int}\to\mathsf{int}}{\Gamma_0\vdash \mathtt{(map\ x\ to\ x)}:(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})}[\textbf{abs}]$$

   Tree 2:

$$\dfrac{\Gamma_0\,\mathtt{[y}:\mathsf{int}\mathtt{]}\vdash \mathtt{y}:\mathsf{int}}{\Gamma_0\vdash \mathtt{(map\ y\ to\ y)}:\mathsf{int}\to\mathsf{int}}[\textbf{abs}]$$

   Tree 3:

$$\dfrac{Tree\ 1 \quad Tree\ 2}{\Gamma_0\vdash \mathtt{(map\ x\ to\ x)(map\ y\ to\ y)}:\mathsf{int}\to\mathsf{int}}[\textbf{app}]$$

5. This example is almost identical to the previous one, but the identity function `id` is defined only once in a let binding and then applied to itself. Since Typed Jam does not support polymorphism, we can only assign one typing to `id`. But we needed two different typings for the identity in the preceding example, so we cannot type this program.

Task 2: Both programs are typable in Polymorphic Jam. In fact, the first program is typable in Typed Jam because the `listMap` function is only applied to one type of list. Hence the **letmono** rule can be used to type the `let` expression in this program instead of the more general **letpoly** rule.

1. The type inference proof is omitted. It is technically straightforward, but the body of the `let` is unusual because it uses the constant `empty` which has polymorphic type $\forall\alpha.\,\mathsf{list}\,\alpha$ in the base type environment Γ_0. In type inference proofs, all occurrences of `empty` have ambiguous type $\mathsf{list}\,\alpha$ where α is a fresh type variable not in the type environment. In this sample program, the constant function `first` is passed as the function argument to `listMap`. Since `first` requires a list as input, the type of `listMap` in the body of the `let` requires a $\mathsf{list}\,\mathsf{list}\,\beta$ for some β as its second input and the occurrence of `first` passed as the first input has type (by unification constraints) $\mathsf{list}\,\beta \to \beta$, forcing `empty` to have type $\mathsf{list}\,\mathsf{list}\,\beta$.

2. Let Γ_1 abbreviate $\Gamma_0[\texttt{length} : \mathsf{list}\,\alpha \to \mathsf{int}, \texttt{l} : \mathsf{list}\,\mathsf{list}\,\mathsf{int}]$;
let Γ_2 abbreviate $\Gamma_0[\texttt{length} : \forall\alpha.\,(\mathsf{list}\,\alpha \to \mathsf{int}), \texttt{l} : \mathsf{list}\,\mathsf{list}\,\mathsf{int}]$;
and let Γ_3 abbreviate $\Gamma_1[\texttt{l} : \mathsf{list}\,\alpha]$. *Tree 1*:

$$\cfrac{\cfrac{\Gamma_3 \vdash \texttt{rest} : \mathsf{list}\,\alpha \to \mathsf{list}\,\alpha \qquad \Gamma_3 \vdash \texttt{l} : \mathsf{list}\,\alpha}{\Gamma_3 \vdash \texttt{rest(l)} : \mathsf{list}\,\alpha}[\mathbf{app}] \qquad \Gamma_3 \vdash \texttt{length} : \mathsf{list}\,\alpha \to \mathsf{int}}{\Gamma_3 \vdash \texttt{length(rest(l))} : \mathsf{int}}[\mathbf{app}]$$

Tree 2:

$$\cfrac{\Gamma_3 \vdash \texttt{+} : \mathsf{int} \times \mathsf{int} \to \mathsf{int} \qquad \Gamma_3 \vdash \texttt{1} : \mathsf{int} \qquad \textit{Tree } 1}{\Gamma_3 \vdash \texttt{1+length(rest(l))} : \mathsf{int}}[\mathbf{app}]$$

Tree 3:

$$\cfrac{\cfrac{\cfrac{\Gamma_3 \vdash \texttt{empty?} : \mathsf{list}\,\alpha \to \mathsf{bool} \qquad \Gamma_3 \vdash \texttt{l} : \mathsf{list}\,\alpha}{\Gamma_3 \vdash \texttt{empty?(l)} : \mathsf{bool}}[\mathbf{app}] \qquad \Gamma_3 \vdash \texttt{0} : \mathsf{int} \qquad \textit{Tree } 2}{\Gamma_3 \vdash \texttt{if empty?(l) then 0 else 1+length(rest(l))} : \mathsf{int}}[\mathbf{if}]}{\Gamma_1 \vdash \texttt{map l to if empty?(l) then 0 else 1+length(rest(l))} : \mathsf{list}\,\alpha \to \mathsf{int}}[\mathbf{abs}]$$

Tree 4:

$$\cfrac{\Gamma_1 \vdash \texttt{cons} : \mathsf{int} \times \mathsf{list}\,\mathsf{int} \to \mathsf{list}\,\mathsf{int} \qquad \Gamma_1 \vdash \texttt{1} : \mathsf{int} \qquad \Gamma_1 \vdash \texttt{empty} : \mathsf{list}\,\mathsf{int}}{\Gamma_1 \vdash \texttt{cons(1,empty)} : \mathsf{list}\,\mathsf{int}}[\mathbf{app}]$$

Tree 5:

$$\cfrac{\Gamma_1 \vdash \texttt{cons} : \mathsf{int} \times \mathsf{list}\,\mathsf{int} \to \mathsf{list}\,\mathsf{int} \qquad \Gamma_1 \vdash \texttt{3} : \mathsf{int} \qquad \Gamma_1 \vdash \texttt{empty} : \mathsf{list}\,\mathsf{int}}{\Gamma_1 \vdash \texttt{cons(3,empty)} : \mathsf{list}\,\mathsf{int}}[\mathbf{app}]$$

Tree 6:

$$\cfrac{\Gamma_1 \vdash \texttt{cons} : \mathsf{int} \times \mathsf{list}\,\mathsf{int} \to \mathsf{list}\,\mathsf{int} \qquad \Gamma_1 \vdash \texttt{2} : \mathsf{int} \qquad \textit{Tree } 5}{\Gamma_1 \vdash \texttt{cons(2,cons(3,empty))} : \mathsf{list}\,\mathsf{int}}[\mathbf{app}]$$

Tree 7:

$$\cfrac{\Gamma_1 \vdash \texttt{cons} : \mathsf{list}\,\mathsf{int} \times \mathsf{list}\,\mathsf{list}\,\mathsf{int} \to \mathsf{list}\,\mathsf{list}\,\mathsf{int} \qquad \textit{Tree } 6 \quad \Gamma_1 \vdash \texttt{empty} : \mathsf{list}\,\mathsf{list}\,\mathsf{int}}{\Gamma_1 \vdash \texttt{cons(cons(2,cons(3,empty)),empty)} : \mathsf{list}\,\mathsf{list}\,\mathsf{int}}[\mathbf{app}]$$

Tree 8:

$$\cfrac{\Gamma_1 \vdash \texttt{cons} : \mathsf{list}\,\mathsf{int} \times \mathsf{list}\,\mathsf{list}\,\mathsf{int} \to \mathsf{list}\,\mathsf{list}\,\mathsf{int} \qquad \textit{Tree } 4 \quad \textit{Tree } 7}{\Gamma_1 \vdash \texttt{cons(cons(1,empty), cons(cons(2,cons(3,empty)), empty))} : \mathsf{list}\,\mathsf{list}\,\mathsf{int}}[\mathbf{app}]$$

Tree 9:

$$\dfrac{\Gamma_2 \vdash \texttt{length} : \mathsf{list\,int} \to \mathsf{int} \qquad \dfrac{\Gamma_2 \vdash \texttt{first} : \mathsf{list\,list\,int} \to \mathsf{list\,int} \qquad \Gamma_2 \vdash \texttt{l} : \mathsf{list\,list\,int}}{\Gamma_2 \vdash \texttt{first(l)} : \mathsf{list\,int}}[\textbf{app}]}{\Gamma_2 \vdash \texttt{length(first(l))} : \mathsf{int}}[\textbf{app}]$$

Tree 10

$$\dfrac{\Gamma_2 \vdash \texttt{+} : \mathsf{int} \times \mathsf{int} \to \mathsf{int} \qquad \dfrac{\Gamma_2 \vdash \texttt{length} : \mathsf{list\,list\,int} \to \mathsf{int} \qquad \Gamma_2 \vdash \texttt{l} : \mathsf{list\,list\,int}}{\Gamma_2 \vdash \texttt{length(l)} : \mathsf{int}}[\textbf{app}] \; \mathit{Tree}\ 9}{\Gamma_2 \vdash \texttt{length(l)+length(first(l))} : \mathsf{int}}[\textbf{app}]$$

Tree 11

$$\dfrac{\mathit{Tree}\ 3 \quad \mathit{Tree}\ 8 \quad \mathit{Tree}\ 10}{\Gamma_0 \vdash \begin{array}{l}\texttt{let length := map l to} \\ \qquad\qquad\qquad \texttt{if empty?(l) then 0} \\ \qquad\qquad\qquad \texttt{else 1 + length(rest(l))} \\ \qquad\quad \texttt{l := cons(cons(1,empty),cons(cons(2,cons(3,empty)),empty))} \\ \texttt{in length(l)+length(first(l)):int}\end{array}}[\textbf{letpoly}]$$

Task 3: The second program in the preceding section is an example. The following is a shorter (but not necessarily simpler) example:

```
let id := map x to x;
in (id(id))(0)
```

The program is not typable in Typed Jam because the function `id` is applied to an argument of type $\mathsf{int} \to \mathsf{int}$ and again (since `id(id)` is `id`) to the an argument of type int. Hence it must have type $(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$ and type $(\mathsf{int} \to \mathsf{int})$ which cannot be unified.