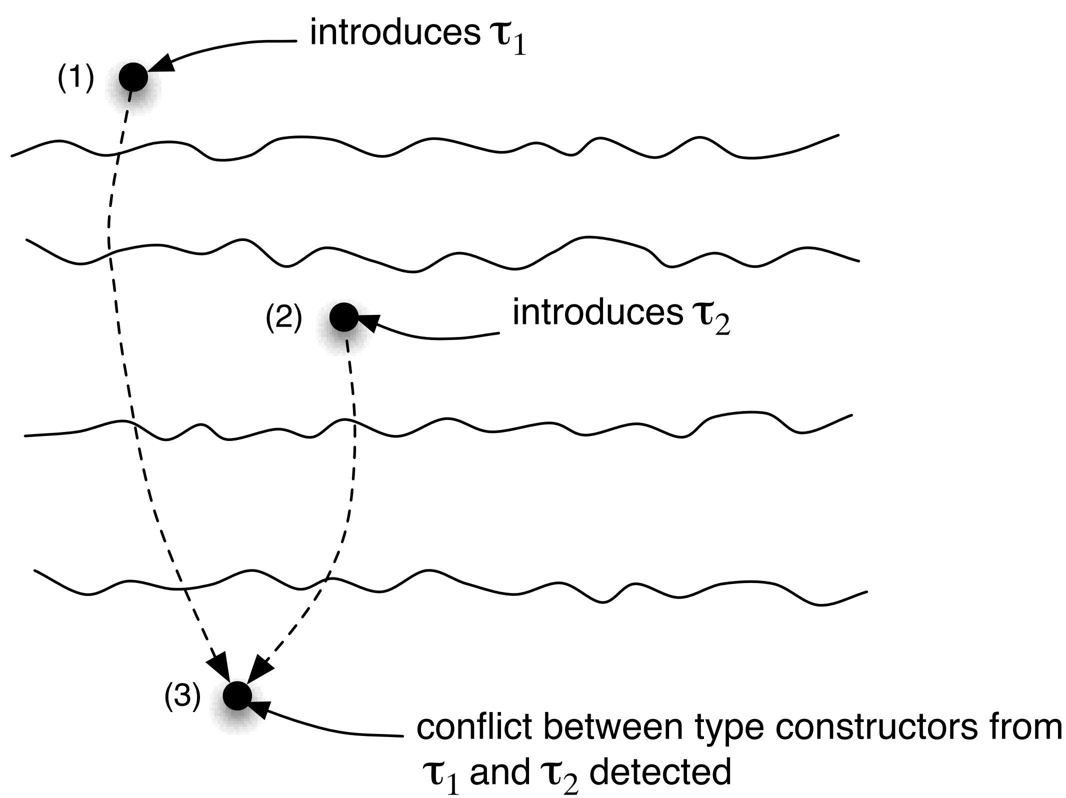
Culprits: A Simple Approach to Better Type Error Messages

David MacQueen
University of Chicago
IFIP WG 2.8 Meeting, 2003

The Problem

Type errors in ML (Haskell) can be difficult to interpret.

Long chains of unification can propagate types over long distances:



Typical error message describes the expression (3) and types of its subexpressions (e.g. function and argument).

Example

```
1    fun f x =
2        let val y = nil :: x
3            fun g(u :: _) = u + 1
4            fun h(v :: t) = g(rev t)
5         in h x
6    end
```

```
Error [line 5]: function and argument disagree
   Expression: h x
   Function type: int list -> int
   Argument type: 'X list list
```

Origins and Paths

Two type constructors conflict:

`int` from `int list -> int`

`list` from `X list list -> int`

1. Where did these constructors originate?

2. How did they come together? (propagation paths)

Some History

Many papers have addressed this general problem, starting with Wand [FPCA 198?].

A couple of recent examples:

Discriminative sum types locate the source of type errors
Matthias Neubauer, Peter Thiemann [ICFP 2003]

Type error slicing in implicitly typed higher-order languages
Christian Haack (DePaul Univ) [MPLS 2003]

Common problem is that they provide too much information and often involve complex algorithms, substantial overhead during type checking, or multi-pass type checking.

Analysis of example

```
1    fun f x =
2        let val y = nil :: x
3            fun g(u :: _) = u + 1
4            fun h(v :: t) = g(rev t)
5         in h x
6    end
```

Origins:

`list`	<--	`nil`	[line 2]
`int`	<--	`+`	[line 3]

Propagation:

`list : nil -> :: -> x`
`int  : + -> u -> g -> rev -> t -> h`

We'll call the occurrences of `nil` and `+` the *culprits*.

Claim

The most valuable information is the location of the culprits.

The propagation paths can be long, but in practice are usually obvious (or even unnecessary).

Culprit Identification Algorithm

1. Mark each type expression with the *location* of the source construct that introduces it.

```
nil : (X list)^nil
+   : (int * int -> int)^+
```

2. During unification, propagate the location annotations downward. In effect, we lazily transform

   `(int * int -> int)`$^{+}$ to `int`$^{+}$ `*`$^{+}$ `int`$^{+}$ `->`$^{+}$ `int`$^{+}$

3. If unification fails with conflicting constructors, the constructors have location annotations that identify their origins, which become the culprits.

Improved Error Message

```
Error [line 5]: function and argument disagree
   Expression: h x
   Function type: int[1] list -> int
   Argument type: 'X list[2] list
   Culprits: [1] fun g(u::_) = u + 1     [line 3]
             [2] val y = nil :: x        [line 2]
```

Circularity Errors

```
1     fun f x = x x
```

$x : (Y \rightarrow Z)^{x}$

$x : Y^{x}$

```
Error [line 1]: type circularity in function app
  Expression: x x
  Operator type: (Y -> Z)[1]
  Argument type: Y[2]
  Culprits: [1] fun f x = x x     [line 1]
            [2] fun f x = x x     [line 1]
```

Implementation

First implemented with Laurent Thiery around 1994, using Centaur based SML environment to display locations of error detection and culprits.

Reimplemented in 2003 with vanilla text user presentation.

Types:

```
datatype ty
  = VARty of tyvar
  | CONty of tycon * ty list
  | POLYty of {sign: polysign, tyfun: tyfun}
  | ...
  | MARKty of ty * SourceMap.region
```

Unify:

```
val unifyTy : Types.ty * SourceMap.region *
              Types.ty * SourceMap.region
              -> unit
```

Conclusion

Preliminary experience shows that adding culprits to error messages is a major help. In a large majority of cases where a type error message is puzzling, adding the culprits makes the source of the error obvious. Furthermore, the added mechanism to support this is quite simple and light-weight.

Claim is that adding propagation paths yields a much smaller improvement and is probably not worth the additional complexity -- *except perhaps for training novice programmers.*