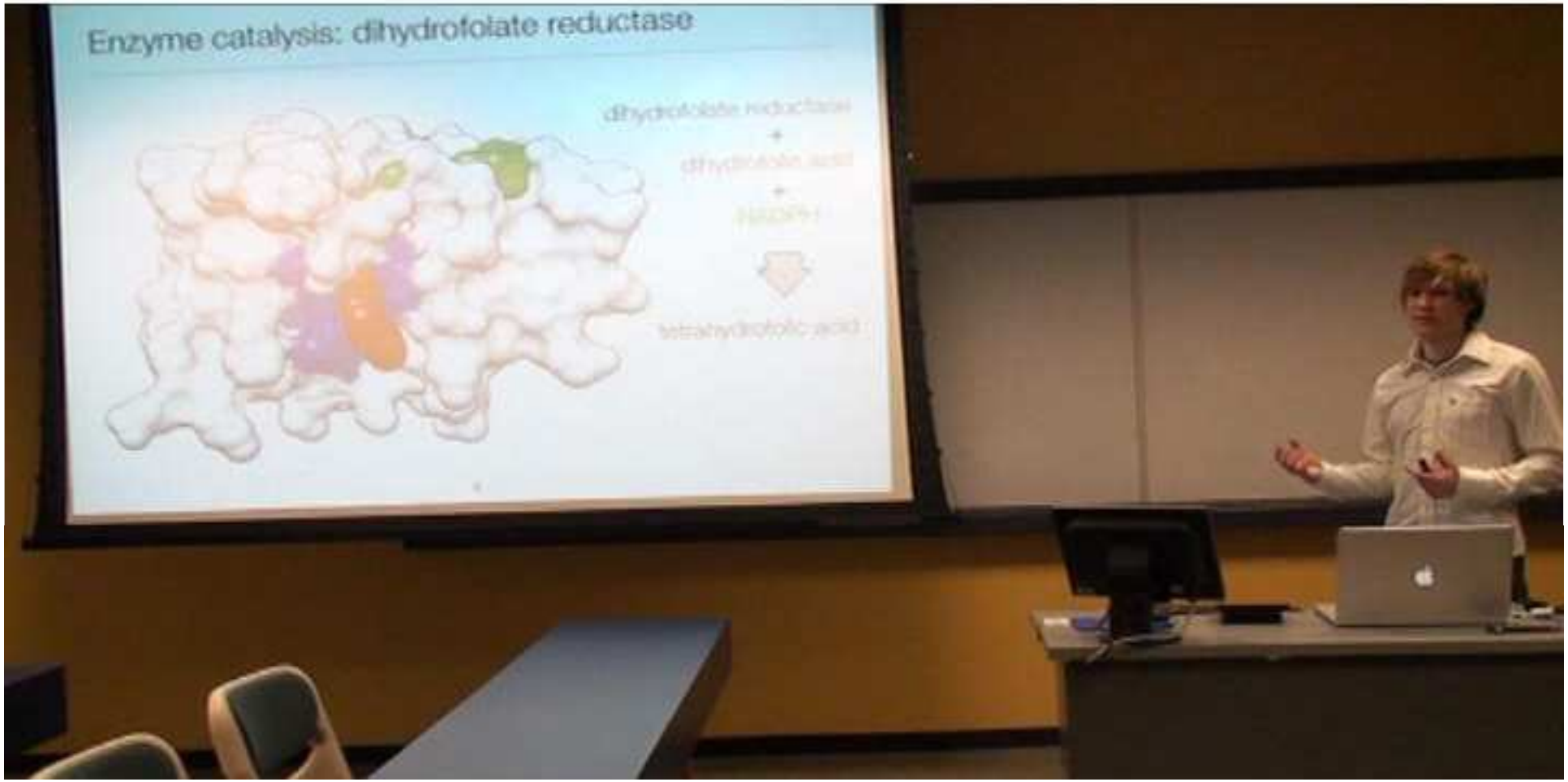
# COMP 400: Technical Communication in Computer Science

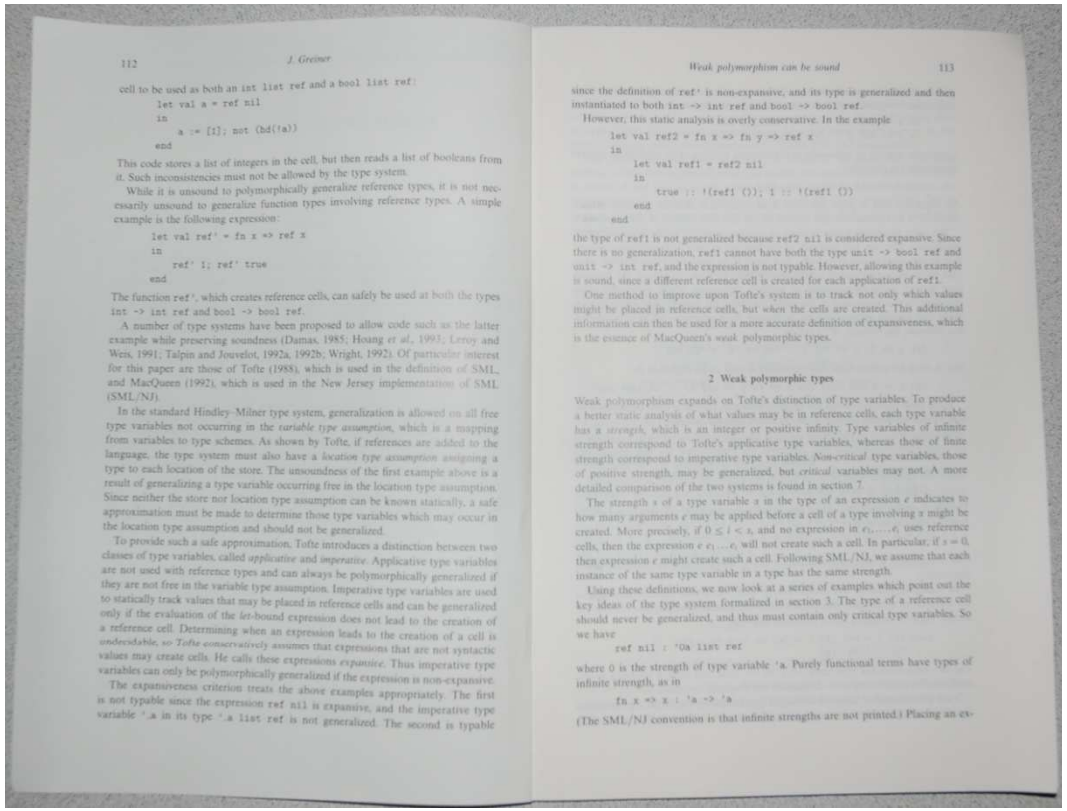
John Greiner

“An engineer who can’t communicate works for one who can.”

“If you can’t tell them what you are doing, they’ll hire someone else who can.”




Jan 13 Jan 20 Jan 27 Feb 3 Feb 10



112 J. Greiner

cell to be used as both an `int list ref` and a `bool list ref`:

```
let val a = ref nil
in
    a := [1]; not (hd(!a))
end
```

This code stores a list of integers in the cell, but then reads a list of booleans from it. Such inconsistencies must not be allowed by the type system.

While it is unsound to polymorphically generalize reference types, it is not necessarily unsound to generalize function types involving reference types. A simple example is the following expression:

```
let val ref' = fn x => ref x
in
    ref' 1; ref' true
end
```

The function `ref'`, which creates reference cells, can safely be used at both the types `int -> int ref` and `bool -> bool ref`.

A number of type systems have been proposed to allow code such as the latter example while preserving soundness (Damas, 1985; Hoang *et al.*, 1993; Leroy and Weis, 1991; Talpin and Jouvelot, 1992a, 1992b; Wright, 1992). Of particular interest for this paper are those of Tofte (1988), which is used in the definition of SML, and MacQueen (1992), which is used in the New Jersey implementation of SML (SML/NJ).

In the standard Hindley–Milner type system, generalization is allowed on all free type variables not occurring in the *variable type assumption*, which is a mapping from variables to type schemes. As shown by Tofte, if references are added to the language, the type system must also have a *location type assumption* assigning a type to each location of the store. The unsoundness of the first example above is a result of generalizing a type variable occurring free in the location type assumption. Since neither the store nor location type assumption can be known statically, a safe approximation must be made to determine those type variables which may occur in the location type assumption and should not be generalized.

To provide such a safe approximation, Tofte introduces a distinction between two classes of type variables, called *applicative* and *imperative*. Applicative type variables are not used with reference types and can always be polymorphically generalized if they are not free in the variable type assumption. Imperative type variables are used to statically track values that may be placed in reference cells and can be generalized only if the evaluation of the *let*-bound expression does not lead to the creation of a reference cell. Determining when an expression leads to the creation of a cell is *undecidable, so Tofte conservatively assumes* that expressions that are not syntactic values may create cells. He calls these expressions *expansive*. Thus imperative type variables can only be polymorphically generalized if the expression is non-expansive.

The expansiveness criterion treats the above examples appropriately. The first is not typable since the expression `ref nil` is expansive, and the imperative type variable `'_a` in its type `'_a list ref` is not generalized. The second is typable



since the definition of `ref'` is non-expansive, and its type is generalized and then instantiated to both `int -> int ref` and `bool -> bool ref`.

However, this static analysis is overly conservative. In the example

```
let val ref2 = fn x => fn y => ref x
in
    let val ref1 = ref2 nil
    in
        true :: !(ref1 ()); 1 :: !(ref1 ())
    end
end
```

the type of `ref1` is not generalized because `ref2 nil` is considered expansive. Since there is no generalization, `ref1` cannot have both the type `unit -> bool ref` and `unit -> int ref`, and the expression is not typable. However, allowing this example is sound, since a different reference cell is created for each application of `ref1`.

One method to improve upon Tofte's system is to track not only which values might be placed in reference cells, but when the cells are created. This additional information can then be used for a more accurate definition of expansiveness, which is the essence of MacQueen's weak polymorphic types.

## 2 Weak polymorphic types

Weak polymorphism expands on Tofte's distinction of type variables. To produce a better static analysis of what values may be in reference cells, each type variable has a *strength*, which is an integer or positive infinity. Type variables of infinite strength correspond to Tofte's applicative type variables, whereas those of finite strength correspond to imperative type variables. *Non-critical* type variables, those of positive strength, may be generalized, but *critical* variables may not. A more detailed comparison of the two systems is found in section 7.

The strength $s$ of a type variable $\alpha$ in the type of an expression $e$ indicates to how many arguments $e$ may be applied before a cell of a type involving $\alpha$ might be created. More precisely, if $0 \le i < s$, and no expression in $e_1, \ldots, e_i$ uses reference cells, then the expression $e\ e_1 \ldots e_i$ will not create such a cell. In particular, if $s = 0$, then expression $e$ might create such a cell. Following SML/NJ, we assume that each instance of the same type variable in a type has the same strength.

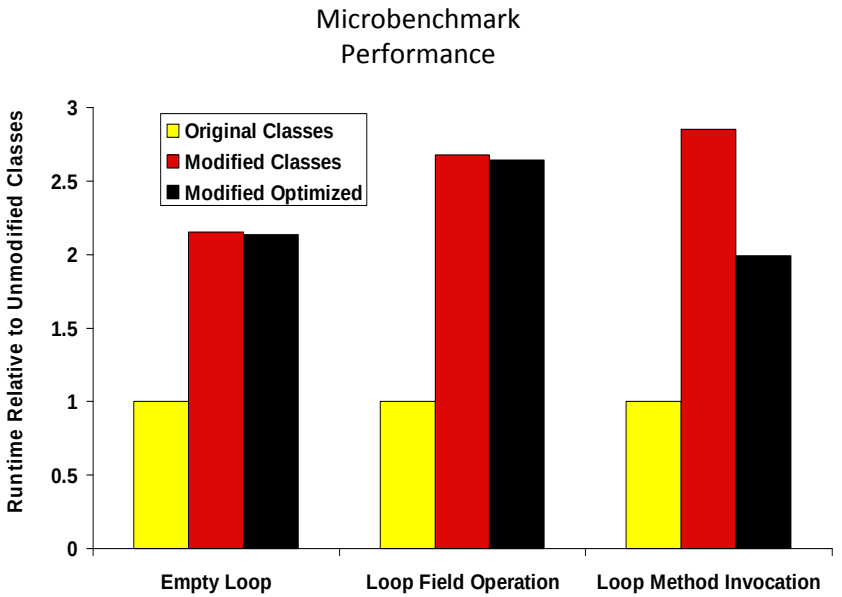
Using these definitions, we now look at a series of examples which point out the key ideas of the type system formalized in section 3. The type of a reference cell should never be generalized, and thus must contain only critical type variables. So we have

```
ref nil : '0a list ref
```

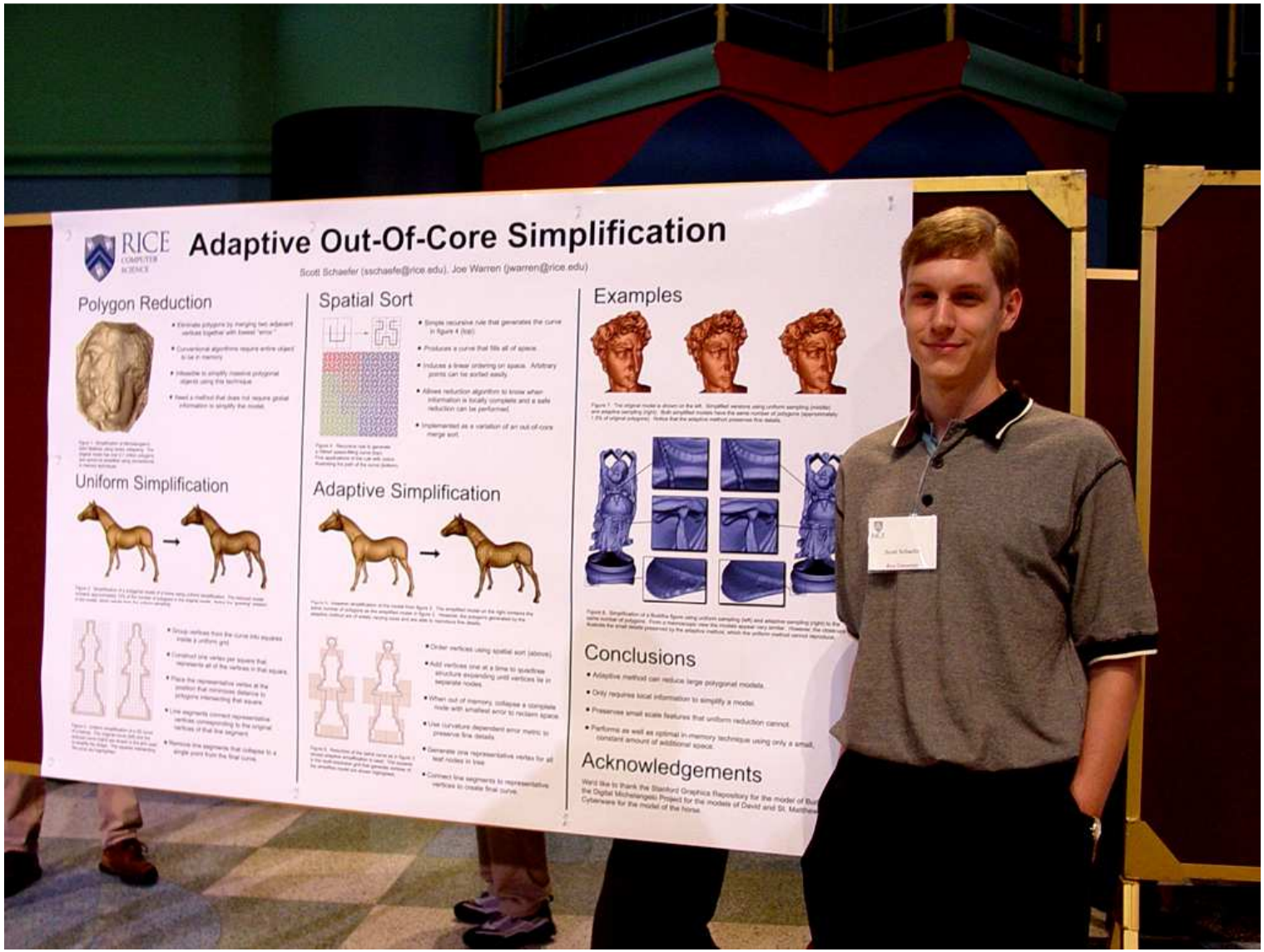
where 0 is the strength of type variable `'a`. Purely functional terms have types of infinite strength, as in

```
fn x => x : 'a -> 'a
```

(The SML/NJ convention is that infinite strengths are not printed.) Placing an ex-




Feb 17 Feb 24 Mar 10




Mar 17 Mar 24 Mar 31 Apr 7 Apr 15-16

# Critique  this experiment

Apr 21

# Types of Technical Communication

- Reports
- Proposals
- Instructions
- Procedures
- Manuals
- Brochures
- Memos
- Emails
- Letters

Strategies for Technical Communication in the Workplace, Gurak & Lannon

# Expectations

~~Sit and listen~~

Do and discuss

Read and prepare

http://www.clear.rice.edu/comp400/

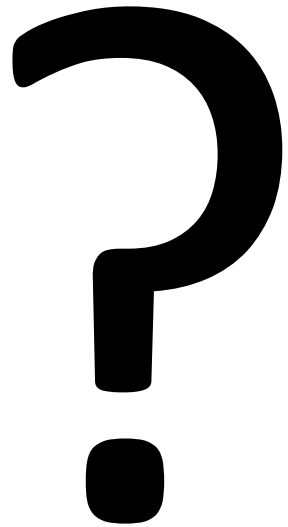
# Grading

In-class activities

Participation

Homework/projects

~~Exams~~



# Activity

Introduction

Background

Goals