Class Information

- Project 3:
  - Download the newest updates (May 1, 2:07pm).
  - Performance requirements are listed on next page.
  - Sakai submission site is open now.
  - **There is no late submission!**
- Today is the last day of recitations (section 3); next Monday is the last day of office hours.
- Final exam on Thursday, May 8, at noon: Rooms TIL 254 and TIL 258. What room you should report to will be posted later on class website.
  - closed book, closed notes, cumulative (over 50% after midterm material)
  - no electronics, backpacks etc.
  - Sorry, no bathroom break.
- One hour review session on Tuesday, May 6, at noon in our regular classroom (TIL 254). Will return HW# 8 then.
- Will bring unclaimed homeworks and midterm exams to review session.

Project 3 - Grading

The project will be graded on correctness and performance.

Performance: You should achieve similar timings (and corresponding speed-ups) with your parallel versions as listed here.

null.cs.rutgers.edu: 8 core, 3.4 GHz machine with 16 GB memory

Sequential version: 13 - 14 seconds
T2 Single Loop: 8 - 8.5 seconds
T4 Single Loop: 5 - 5.5 seconds

atlas.cs.rutgers.edu: 12 core 2.1 GHz machine with 32 GB memory

Sequential version: 25 - 27 seconds
T2 Single Loop: 16 - 21 seconds
T4 Single Loop: 11 - 16 seconds

For the `fastest` versions, we do not give you the performance numbers. We want to keep it exciting!!!

Type Errors and Type Systems

Scott, Chapter 7.2; ALSU Chapter 6.5

Type Error: Applying a function of type $S \rightarrow T$ to an argument not of type S

Goal: No type error remains undetected, i.e., type errors are detected before they actually occur.

How to achieve this goal?

Each language construct (operator, expression, statement, ...) has a type.

basic types: integer, real, character, symbol, void ...

constructed types: lists, pointers, arrays, records, sets, functions

A type system is a collection of *rules* for assigning *type expressions* to operators, expressions, ... in the program. Type systems are language dependent.

A type checker implements the type system, i.e., deduces type expressions for program constructs based on the type inference rules of the type system. The type checker "computes" or "reconstructs" type expressions.

Type expressions

1. A basic type is a type expression. A special basic type, *typeError* will signal an error. A basic type *void* denotes an untyped statement.

2. Since type expressions may be named, a type name is a type expression. (e.g.: `typedef struct foo bar;`)

3. Type expressions may contain variables whose values are type expressions (e.g.: useful for languages without type declarations, or polymorphism).

4. A *type constructor* applied to type expressions is a type expression. Examples:

   (a) arrays

   (b) cartesian products

   (c) records

   (d) pointers

   (e) functions

Example type rules

- If both operands of the arithmetic operators of addition, subtraction, and multiplication are of type integer, then the result is of type integer (Pascal definition).

  Rule for $+$ (analogue rules for $-$ and $*$):

  $$\frac{E \vdash e_1 : integer \quad E \vdash e_2 : integer}{E \vdash (e_1 + e_2) : integer}$$

  where E is a *type environment* that maps constants and variables to their types.

  In combination with the following axiom in the type system for constants c: $\{c : \alpha\} \vdash c : \alpha$we can now infer, that $(2 + 3)$ is of type integer:

  $$\frac{E \vdash 2 : integer \quad E \vdash 3 : integer}{E \vdash (2 + 3) : integer}$$

  where $E = \{2 : integer, 3 : integer\}$.

  In general, type deduction proofs work bottom up.

Example type rules

α is a type variable, which is a placeholder for other type expressions.

- The result of the unary & operator is a pointer to the object referred to by the operand. If the operand is of type "foo", then the type of the result is a "pointer to foo". (C and C++ definition)

$$\frac{E \vdash e : \alpha}{E \vdash \&e : \mathit{pointer}(\alpha)}$$

- Two expressions can only be compared if they have the same types. The result is of type boolean.

$$\frac{E \vdash e_1 : \alpha \quad E \vdash e_2 : \alpha}{E \vdash (e_1 = e_2) : \mathit{boolean}}$$

Type variables

Type expressions may contain variables (*type variables*) whose values are type expressions.

Type variables are used for implicitly typed languages or languages with polymorphic types.

Programming languages can be

- explicitly typed — every object is declared with its type (**type checking**)
- implicitly typed — type of object is derived from its use (**type reconstruction**)
- monomorphic — every function or data type has a unique, single type
- polymorphic — allows functions or data types to have more than one type (e.g.: *list* in Scheme and & in C)

Type variables — polymorphism

- Polymorphic cons:

$$\frac{E \vdash e_1 : \alpha \quad E \vdash e_2 : list(\alpha)}{E \vdash cons(e_1, e_2) : list(\alpha)}$$

  cons has the type expression
  $\forall \alpha.(\alpha \times list(\alpha)) \rightarrow list(\alpha)$

- Polymorphic '():

$$E \vdash \ '() : list(\alpha)$$

  '() has the type expression $\forall \alpha. list(\alpha)$

Questions:

- Are "&" and "=" monomorphic or polymorphic functions?
- What is the type of cons(1,'())?
- What is the type of cons('a,1)?

Type variables — implicitly typed

Recall:

$$\frac{E \vdash e_1 : integer \quad E \vdash e_2 : integer}{E \vdash (e_1 + e_2) : integer}$$

where E is a type environment. In other words, "+" has the type expression $(integer \times integer) \rightarrow integer$.

What are the types of the variables a and b in the following program:

```
read(a);
read(b);
...   a + b ...;
```

Here is an idea: Guess the types of variables you don't know about and use *unification* to match guesses with rules.

`read(a)`	$\{$a:$\alpha\}$
`read(b)`	$\{$a:α, b:$\beta\}$
`a + b`	unify(α, integer)
	unify(β, integer)
	apply type rule; result integer

Unification

`unify` generates a mapping `U` from type variables to type expressions such that two type expressions become syntactically identical.

Example:

- Two type expressions:

  $type\_expr_1 = \alpha \rightarrow \beta$

  $type\_expr_2 = (\beta \times \beta) \rightarrow integer$

- Mapping U = unify($type\_expr_1$, $type\_expr_2$) =

  $\{ (\alpha, (integer \times integer)), (\beta, \text{integer}) \}$.

- U($type\_expr_1$) = U($type\_expr_2$) =

  $(integer \times integer) \rightarrow integer$

Next Lecture

Next time:

- Review session for final exam.