Assignment 4: Type Checking Issues, Dataflow Analysis, and Register Allocation

15-411: Compiler Design
David McWherter (cache@cs) and Noam Zeilberger (noam@cs)

Due: Thursday, October 25, 2007 (1:30 pm)

Problem 1 — Overloading and coercions [20 points]

Imagine (don't worry, this is only in our imagination!) that we decide to extend L3 with a type **float** of floating-point numbers. We would like to support them through a combination of operator overloading and implicit coercions, making the following changes from the definition given in the handout for Lab 3:

- We add floating-point constants.
- We leave the typing rules for function calls, returns, and assignments unchanged, but add subtyping axioms **int** $\leq$ **float** and **float** $\leq$ **int**. In any of these situations, there can now be run-time conversions from **int** to **float** or vice versa.
- Every arithmetic operator (other than %) can take either **int**s or **float**s as arguments, as can every comparison operator. If both arguments are **int**s, the semantics are of the corresponding integer operation, while if they are both **float**s, the semantics are of the corresponding floating-point operation. If one argument is a **float** and one argument an **int**, the latter is first coerced to a **float** before performing the floating-point operation.

Question (a) [3 points] Letting $\oplus$ stand for a generic arithmetic operator (other than %), and $\bigcirc\!\!\!\!\sim$ for a generic comparison operator, write typing rules for the expressions $e_1 \oplus e_2$ and $e_1 \bigcirc\!\!\!\!\sim e_2$.

Suppose our compiler's front-end originally contained these four type-checking routines (among others):

```
chk_conv : A.tp -> A.tp -> unit               (* "chk_conv tp1 tp2" checks tp1 <= tp2   *)
syn_exp : Sym.table -> A.exp -> A.tp          (* "syn_exp env e" synthesizes type for e *)
chk_exp : Sym.table -> A.exp -> A.tp -> unit  (* "chk_exp env e tp" checks e : tp       *)
chk_stm : Sym.table -> A.stm -> unit          (* "chk_stm env s" checks s valid         *)
```

They all raise an exception upon failure, and are implemented as follows:

```
fun chk_conv (A.Pointer(A.Void)) (A.Pointer(tp)) = ()
  | chk_conv tp1 tp2 = chk_equal tp1 tp2      (* checks syntactic equality tp1 = tp2 *)

fun syn_exp env (A.OpExp(A.PLUS,[e1,e2])) =
  (case (syn_exp env e1, syn_exp env e2)
     of (A.Int, A.Int) => A.Int
      | _ => raise ErrorMsg.Error "both operands of + must be int")
  | syn_exp (* ... OTHER CASES ELIDED ... *)

fun chk_exp env e tp = chk_conv (syn_exp env e) tp
```

```
fun chk_stm env (A.Assign(v,e)) = chk_exp env e (syn_exp env v)
  | chk_stm (* ... OTHER CASES ELIDED ... *)
```

Now, in order to add support for **float**s, we will augment the type-checking phase so that it simultaneously performs translation into intermediate code—deciding between overloaded operators and making implicit coercions explicit. To begin, we change the specification of `chk_conv` to take an extra IR tree as argument, and return an IR tree with any necessary coercions:

```
chk_conv : A.tp -> A.tp -> T.exp -> T.exp
```

Question (b) [4 points] Rewrite `chk_conv` to fit this specification. You can assume routines `itof : T.exp -> T.exp` and `ftoi : T.exp -> T.exp` that encode **int**-to-**float** and **float**-to-**int** conversions, respectively.

Likewise, we modify the specification of the other type-checking routines to return intermediate code:

```
syn_exp : Sym.table -> A.exp -> A.tp * T.exp
chk_exp : Sym.table -> A.exp -> A.tp -> T.exp
chk_stm : Sym.table -> A.stm -> T.stm
```

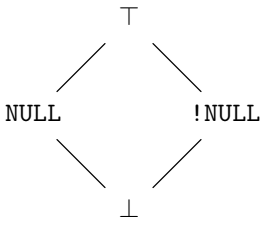
Question (c) [10 points] Rewrite these three routines to meet the specification and support floating-point arithmetic. For `syn_exp`, you only have to consider the `A.PLUS` case (as above), and likewise for `chk_stm` you only have to consider the `A.Assign` case. You can assume the following:

- `A.FLOAT : A.tp` stands for the type **float**.
- `T.BINOP : T.binop * T.exp * T.exp -> T.exp` builds an IR tree for an arithmetic operation.
- `T.IPLUS`, `T.FPLUS : T.binop` stand for integer and floating-point addition, respectively.
- `T.MOVE : T.exp * T.exp -> T.stm` builds an assignment (where 1st arg = src, 2nd arg = dst).

Question (d) [3 points] So far we have ignored the complex assignment operators `+=`, `*=`, etc. As we know, translating $v \oplus= e$ to $v = v \oplus e$ is invalid in L3, because the lvalue v could contain side-effects. Can you think of another reason why we might not want to perform this translation, even if v has no side-effects?

Problem 2 — Dataflow analysis [20 points]

As discussed in class, constant propagation can be performed by first computing reaching definitions. However, it is also possible to perform constant propagation directly as a forward dataflow analysis. In this problem, we will explore a sort of constant propagation for pointer values, keeping track of the value $V[x]$ of each pointer variable x inside a function as a point on the following lattice:



where
- $V[x] = \bot$ means "x has never been assigned to."
- $V[x] =$ `NULL` means "x may have been assigned `NULL` (but no other values)."
- $V[x] =$ `!NULL` means "x may have been assigned a non-`NULL` value (but not `NULL`)."
- $V[x] = \top$ means "x may have been assigned any value."

This sort of analysis is sometimes called *abstract interpretation.* As it is a forward analysis, information along the edges coming into a location ℓ is combined to compute the value table at ℓ. If $v_1, \ldots, v_n$ are the values of $V[x]$ at ℓ's predecessors, then at ℓ we have $V[x] = v_1 \sqcup \cdots \sqcup v_n$, where $\sqcup$ is the lattice join[1] operation. Information is propagated forward as follows:

- After an assignment x `= NULL`, we set $V[x] =$ `NULL`.
- After an assignment x `= new(`τ`)`, we set $V[x] =$ `!NULL`.
- After an assignment $x = y$, we set $V[x] = V[y]$.
- After any other assignment $x = e$, we set $V[x] = \top$.

We initialize the value table at the function entry point (see question (a) below), and then iterate the analysis to determine V at every node. Once a fixed point is reached, we can use the information to transform the code:

- If $V[x] =$ `NULL` at ℓ, we can replace any (non-lvalue) occurrences of x by `NULL`.
- We can evaluate pointer comparisons if both sides are either constant or variables known to be `NULL` or `!NULL`. For example, if $V[x] =$ `NULL` and $V[y] =$ `!NULL` at ℓ, we replace the expression $x == y$ by 0.

Questions:

(a) At the function entry point, locally-declared variables x are initialized with $V[x] = \bot$. How should we initialize function parameters?

(b) Suppose that after running the analysis, $V[x] = \bot$ at location ℓ. What does that mean about any use of x at ℓ?

Consider the following function definition:

```
      int f() {
        var y,z : int*;
1:      z = NULL;
2:      y = new(int);
3:      if (y == z) {
4:        y = NULL;
5:      }
6:      return (y == z);
      }
```

(c) Compute $V[y]$ and $V[z]$ at each line of `f`. What are $V[y]$ and $V[z]$ at the return statement?

(d) Using these values, perform the constant propagation/folding transformations described above. Potentially, could the program be further simplified?

[1] Defined by `NULL` $\sqcup$ `!NULL` $= \top$, $v \sqcup \top = \top$, $v \sqcup \bot = v$, $v \sqcup v = v$, and $v_1 \sqcup v_2 = v_2 \sqcup v_1$.

Problem 3 — Chordiality [20 points]

Code Segment 1:

```
a = 1;
b = 2;
d = a+b;
a = b+d;
c = a+1;
e = c+a;
d = e+c;
d = e+d;
return d;
```

Code Segment 2:

```
a = 1;
b = 2;
d = a+b;
a2 = b+d;
c = a2+1;
e = c+a2;
d2 = e+c;
d3 = e+d2;
return d3;
```

(a) Draw the interference graphs for both code segments. Are they chordal?

(b) Run the chordal graph coloring algorithm on both code segments. Write down the maximum cardinality search order, the final coloring, and the number of colors/registers needed for each.

While applying the maximum cardinality heuristic, please use the convention of alphabetic order to break ties (e.g., if `a` and `d` have the same cardinality, choose `a`.

(c) Code segment 2 has the property that all variables are assigned to only once. Argue informally why straight line code with this property will always have a chordal interference graph. (Hint: You should probably be talking about the uses and definitions of variables, and their liveness.)