Optimising Compilers 2012–2013
Exercise Sheet 4

The purpose of this exercise sheet is to practise *inference-based analysis*, *effect systems*, *points-to analysis*, *decompilation* and *instruction scheduling*.

1 Warm Up

- Explain what is meant by the "phase order problem" in compilation.
- Discuss, with an example, how optimisation phases can conflict and adversely affect each other?
- How can SSA form be used to help produce better results in both compilation and decompilation?

2 Inference-based analysis

In this exercise sheet, we review *inference-based analysis* (Lecture 12) and, more specifically, *effect systems* (Lecture 13). An inference-based analysis is specified using judgements of the following form:

$$\Gamma \vdash e : \phi$$

In this judgement, e is an expression, Γ specifies the assumptions about the free-variable context and ϕ is a program property (that tells us something useful about the expression e).

(*a*) For more information about *inference-based analysis*, review the slides from Lecture 12 and read Section 16 of the course notes.

2.1 Matrix processing

In this example, we look at a matrix processing language. Our language supports the following basic operations:

$$e \quad ::= \quad A \mid e + e \mid e - e \mid e \times e \mid e^T$$

The construct A accesses a matrix-valued variable; operators $+$ and $-$ represent point-wise plus and minus on matrices; $\times$ is used for matrix multiplication and e^T calculates a transposed matrix.

We want to use inference-based analysis to check that programmers do not attempt to apply operations on matrices of incompatible sizes. For example, if A is a matrix of size 4×2 and B is of a size 2×3 then the expression $A + B$ is invalid (because $+$ and $-$ requires the matrices

to be of the same size), but $A \times B$ is a valid expression and yields a matrix of size 4×3. More formally, the following judgement holds:

$$A : 4 \times 2,\ B : 2 \times 3 \ \vdash\ A \times B : 4 \times 3$$

The context Γ consists of matrix variables with their respective sizes and the property ϕ that we obtain from our analysis is the size of the matrix produced by an expression of our language.

A rule that describes variable access looks as follows:

$$(var)\ \frac{}{\Gamma,\ A : m \times n \ \vdash\ A : m \times n}$$

(b) Write inference rules for the remaining operations of the language (binary operators $+$, $-$, $\times$ and unary operation A^T)

(c) Aside from making sure that calculation with matrices does not apply some operation on matrices of incompatible sizes, it can also guide optimisations in the compiler or interpreter. Can you suggest some optimisation that could be done using the matrix size information? Hint: Matrix multiplication is associative, but ...

3 Type and effect systems

In the previous example, we did not need to distinguish *types* of expressions, because everything was a matrix. Moreover, the language did not have any side-effects such as I/O, mutable state or communication.

A particular case of *inference-based analysis* that provides information about types and side-effects of expressions is called *type and effect system*.

(a) For more information about *type and effect systems*, review the slides from Lecture 13 and read Section 17 of the course notes.

3.1 Tracking side-effects

Type and effect systems can be used to track side-effects of a computation including communication (channels from which the computation reads and to which it writes) and memory operations (whether it reads or writes or what global stores it uses).

(b) Answer the questions in past paper *2008 Paper 9 Question 16*, which discusses the use of type and effect systems for communication via channels.

(c) Answer the questions in past paper *2002 Paper 8 Question 7*, which considers tracking of reads and writes to ML-style mutable references.

The type and effect system in (f) annotates judgements and function types with effects $F \subseteq \{A, R, W\}$. This tells us whether an expression allocates, reads or writes *any* reference cells. In larger programs, we would like to be more precise – in particular *what* reference cells does the computation access.

As reference cells are allocated and used dynamically, tracking the exact reference cells accessed would be difficult. However, we can use a higher-level abstraction called *regions*. The idea is that the user can create memory regions and every reference cell is allocated in a specific

region. In the type and effect system, we do not track operations on individual *reference cells*, but instead, operations on *regions*.

We assume that the region names, written as $\rho_1, \rho_2, \ldots$ are already defined. To track where a reference is allocated, we modify the language as follows:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathbf{ref}_\rho\ e \\ \tau & ::= & \ldots \mid \mathsf{intref}_\rho \end{array}$$

We use ρ as a meta-variable ranging over region names. The expression that constructs a reference cell is annotated with a region that is used and this information is attached to the intref type. The effects that we want to track are now annotated with the region in which they occur, so $F \subseteq \bigcup_\rho \{A_\rho, R_\rho, W_\rho\}$.

Assuming x is a variable of type int, the following (rather useless) program allocates a reference cell initialised to the value of x in a region ρ_1 and then immediately reads the value and returns it:

$$(\lambda r \rightarrow\ !r)\ (\mathbf{ref}_{\rho_1}\ x)$$

(d) Describe what changes do we need to make to the type and effect system from (f) in order to track more detailed effect information about memory regions.

3.2 Tracking execution environments

[This is quite a long example, but gives additional insight into non-trivial uses of effect systems.] Suppose that we have a new and fancy call-by-value distributed programming language called WebLambda. A program in WebLambda can be compiled to machine code (and executed on the server-side) or to JavaScript (and executed in the web browser). However, some expressions in WebLambda cannot be compiled to JavaScript, because they are not supported in JavaScript. Similarly, some expressions cannot be compiled to native code (perhaps, because they use user-interface functions available only in JavaScript).

The WebLambda language is, of course, based on lambda calculus and has the following syntax. For now, the language does not support any communication between client-side and server-side. The only goal is to write code once and then compile it for both server-side and client-side without code duplication:

$$e \quad ::= \quad x \mid e\ e \mid \lambda x.e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$$

The following example shows two functions. A function `twice` that multiplies a number by 2 and can be compiled to both native code and JavaScript. The functions `mainClient` and `mainServer` use `twice` to run a calculation and then show the result using either `window.alert` (available in web browser) or `console.write` (only in native code):

```
let twice = fun n -> n * 2

let mainClient = fun () ->
  window.alert (twice 21)

let showServer = fun () ->
  console.write (twice 21)
```

To make programming in WebLambda safer, we want to develop *type and effect system* that will give us a type of expression together with a set of execution environments specifying where it can be executed. The syntax of types is the following:

$$\tau \quad ::= \quad \alpha \mid \tau_1 \xrightarrow{E} \tau_2$$

The type α stands for primitive types of our language (such as integers, unit and other). The set of all possible execution environments that our language supports is $\mathcal{E} = \{\mathsf{client}, \mathsf{server}\}$. In the function type $\tau_1 \xrightarrow{E} \tau_2$, the annotation E specifies a set of execution environments where the function can be executed and it is a subset of all possible environments $E \subseteq \mathcal{E}$. In the previous example, the two primitive functions have types:

$$\begin{aligned} \texttt{window.alert} &: \mathsf{int} \xrightarrow{\{\mathsf{client}\}} \mathsf{int} \\ \texttt{console.write} &: \mathsf{int} \xrightarrow{\{\mathsf{server}\}} \mathsf{int} \end{aligned}$$

As we only have integers, the functions display a number and return 0. The typing judgements have the following form:

$$\Gamma \vdash e : \tau, E$$

The judgement means that an expression e has a type τ and can be executed in execution environments E (where $E \subseteq \mathcal{E}$). The typing judgements that describe the types of variable access and lambda abstraction look as follows:

$$(\mathit{var})\ \frac{}{\Gamma, x : \tau \vdash x : \tau, \mathcal{E}} \qquad (\mathit{fun})\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2, E}{\Gamma \vdash \lambda x.e : \tau_1 \xrightarrow{E} \tau_2, E}$$

The (*var*) rule specifies that an expression which accesses a variable does not have any additional requirements on the execution environments and thus can be execute in all environments that our language supports.

The (*fun*) rule specifies that, if a body of the function can be executed in environments E, then the function is annotated with these environments (we should be only able to call it in one of the supported environments). At the same time, creating a lambda function is only allowed in environments E (as the language does not support any communication between sides, we can only create function values in environments where we can also run them).

(e) Write application and **if** rules for the type system of WebLambda. Assuming we have expression e_1 of type $\tau_1 \xrightarrow{E_1} \tau_2$ and effect E_2 and an argument e_2 of type τ_1 and effect E_2, in what execution environment can we evaluate $e_1\ e_2$? (For **if**, assume that the condition can be an integer)

(f) Recall the discussion about effect sub-typing from Lecture 13. If we have functions f_1 and f_2 of types $\tau_1 \xrightarrow{E} \tau_2$ and $\tau_1 \xrightarrow{E'} \tau_2$, the expression **if** e **then** f_1 **else** f_2 only type-checks if the effects E and E' are equal. This is limiting, so many systems provide *subtyping* that allows us to view a function with fewer effects as a function that has more effects. This is safe for effect systems, because the system *over-approximates* the actual effects that will happen at run time. Define the sub-typing rule for the WebLambda language.

(g) In the previous example, it would be useful for WebLambda to have a $\oplus$ construct which could be used to combine functions `window.alert` and `console.write` to get a single function that can be used on both client-side and server-side. When executed,

the combined function will use one of the two provided functions, depending on the current environment.

(h) Extend the language with an operation $\oplus$ that allows us to write `window.alert` $\oplus$ `console.write` and give a typing rule for this operation.

(i) Give types and effects for the following programs, discussing the interesting aspects of the inference (assume b is a global variable of type int).

(*1*) $\lambda x.$ `console.write` x

(*2*) (**if** b **then** `console.write` **else** `window.alert`) b

(*3*) (**if** b **then** `console.write` **else** `window.alert`) $\oplus$ $(\lambda x.x)$

4 Points-to and alias analysis

Consider the program:

```
int c,d,e,*p,*q;
p = &c;
*p = 3;
p = &d;
q = &e;
```

Determine both informally, and using Andersen's method, 'the' points-to relation. Which variables are in its domain and range? Does it help to have a separate points-to relation for each program point? If so, can you give dataflow equations (as earlier in the course) relating points-to at one program point with that of its neighbours? Is it forward or backwards?

Now consider the (untyped) program, where `(*)` represents a boolean expression not containing `p`, `q` or `r`, `s` whose value cannot be determined by the program analyser:

```
q = &r;
while (*)
{   p = q;
    if (*) { p = *p; print p; }
    else   { s = q; }
    r = &s;
}
print 42;
```

Calculate the overall points-to relation by Andersen's method, and also determine (e.g. by your dataflow method above) the points-to relation at each program point.

Finally, it turns out that calculating points-to sets for each program point can produce very large amounts of data (that's the reason for Andersen's method being popular). However, not all the points-to pairs are useful. Can live variable analysis be used to remove some of the points-to pairs after calculating them.

Much harder: is there a phase-order problem between liveness and points-to analyses for this sort of example?

Can one use points-to information to determine whether a location is aliased (has two or most pointers to it)? If so, does your alias information tell you there *must* be an alias or there *may* be an alias?

5 Decompilation

In this question we are going to consider a simple imperative language called FARBOO with the following syntax:

$$
\begin{aligned}
\mathit{statements} &::= \mathit{statement}\ \mathit{statements} \mid \mathit{statement} \\
\mathit{statement} &::= \texttt{for}\ \mathit{var}\ \texttt{=}\ \mathcal{N}\ \texttt{to}\ \mathcal{N}\ \texttt{do}\ \mathit{statements}\ \texttt{end} \\
&\quad \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{statements}\ \texttt{end} \\
&\quad \mid \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{statements}\ \texttt{else}\ \mathit{statements}\ \texttt{end} \\
&\quad \mid \mathit{declaration} \\
\mathit{declaration} &::= \mathit{var}\ \texttt{=}\ \mathit{expr} \\
\mathit{expr} &::= \mathit{val}\ \mathit{operator}\ \mathit{expr} \mid \mathit{val} \\
\mathit{operator} &::= + \mid - \mid * \mid \backslash \\
\mathit{val} &::= \mathcal{N} \mid \mathit{var}
\end{aligned}
$$

where $\mathcal{N}$ ranges over integer constants and *var* over variables. The `if` statements test whether the supplied expression is non-zero as in C.

1. Consider the following 3-address code generated by compiling three programs in FARBOO. Decompile the assembly code back into FARBOO giving reasons for any decisions you make.
   Hint: Try decomposing the code into basic blocks first.

(a)
```
  MOV t01, x
  MOV t02, y
  ADD t03, t01, t02
  MOV t04, #0
  CMPEQ t04,t03, label1
  B label2
label1: MOV t01, 0
label2: MOV res1, t01
```

(b)
```
  MOV t01, #1
  MOV t02, #1
  MOV t03, #10
label1: CMPEQ t02, t03, label2
  MUL t01, t01, #2
  ADD t02, t02, #1
  B label1
label2: MOV res1, t01
```

(c)
```
  MOV t01, x
  MOV t02, y
  MUL t03, t01, t01
  MUL t04, t02, t02
  ADD t05, t03, t04
  SUB t05, t05, #1
  MOV t06, #0
  CMPNE t05, t06, label1
  MOV t07, #1
  B label2
label1:
  ADD t05, t05, #1
  MOV t07, t05
  B label2
label2: MOV res1, t07
```

3. Now that you have seen some compiled code for FARBOO, define a decompilation function for FARBOO that takes 3-address code and returns FARBOO code. You may assume that the patterns of 3-address code shown above are typical for compiled FARBOO code.

   Hints:

   - What are the general 3-address code `if` and `for` structures?
   - See Lecture 16 for decompilation.
   - A simple way to describe the decompilation function may be using *pattern matches* on sections/structures of 3-address code with variables that are "filled in" by the pattern match. E.g. $[|\texttt{MOV}x, y|] = \texttt{x = y}$ or $[|\texttt{MOV}\ a, x;\ \texttt{MOV}\ b, y;\ \texttt{ADD}\ c, a, b|] = \texttt{c = x + y}$.
   - You may want pattern matches to be able to group pieces of code into a block assigned to a variable and then to call your decompile function on this block.

6 Instruction Scheduling

Please complete question 2002 Paper 9 Question 7.

Past exam questions can be found at:
`http://www.cl.cam.ac.uk/teaching/exams/pastpapers/t-OptimisingCompilers.html`