CS 565 December 11, 1996

Jens Palsberg

Final Exam + Qualifying Exam

Submit your solutions on separate paper (don't submit this handout). Write your name, id number, and department on the top of the first page you submit. I will stable your pages together.

This exam is intended to take you 2 hours. Budget your time wisely. There are 6 questions on the exam. Each question is worth 10 points, for a total of 60 points maximum. Write the solutions for different questions on different sheets of paper. Make clear what problem each solution is for. I have spare paper, just ask for it.

To those who are taking the qualifying exam: to pass the exam you must get an excellent score.

SUPL: A Programming Language

Several of the questions on this final exam concerns the programming language SUPL. Contrary to popular belief, SUPL is an acronym for "Small Universal Programming Language," and *not* an acronym for "Silly Useless Programming Language." SUPL is a subset of Scheme. A SUPL program is a family of functions defined in the usual Scheme way:

$$\texttt{(define } \mathit{Id} \texttt{ (lambda (} \mathit{Id} \ldots \mathit{Id} \texttt{) } e \texttt{))}$$

Here *Id* is a nonterminal which can generate any Scheme identifier, and e is a nonterminal which can generate expressions according to the following grammar:

$$\begin{array}{lll} e & ::= & \texttt{(}e\ e_1 \ldots e_n\texttt{)} \quad | \quad \mathit{Id} \\ & | & \texttt{(cons } e_1\ e_2\texttt{)} \quad | \quad \texttt{(car } e\texttt{)} \quad | \quad \texttt{(cdr } e\texttt{)} \quad | \quad \texttt{'()} \\ & | & \texttt{(set-car! } e_1\ e_2\texttt{)} \quad | \quad \texttt{(set-cdr! } e_1\ e_2\texttt{)} \\ & | & \texttt{\#t} \quad | \quad \texttt{\#f} \quad | \quad \texttt{(if } e_1\ e_2\ e_3\texttt{)} \end{array}$$

The only thing that makes SUPL different from Scheme is that if we consider the expression $\texttt{(if e}_1\texttt{ e}_2\texttt{ e}_3\texttt{)}$ and $\texttt{e}_1$ evaluates to other than `#t` or `#f`, then we get a run-time error.

1. Identify the subset of SUPL expressions which are simple (in the sense of the lecture notes). Give the answer as a BNF-grammar. Identify also the subset of SUPL expressions which are in tailform (in the sense of the lecture notes). Give the answer as a BNF-grammar. Describe in detail how to transform an arbitrary SUPL program into a Scheme program where every function is in tailform. Write a Scheme program which implements this transformation, using records to represent both the input program and the output program. (It is possible to get partial credit for the implementation part, for example, by sketching the program structure and implementing one or more of the functions.)

2. For every expression in a SUPL program, we want to know an approximation of the values to which the expression can evaluate. Moreover, we want to compute these approximations using the solver from Homework 9. Explain informally what constraints should be generated, and justify briefly that your constraints are of a form that can be handled by the solver.

3. Consider the language of types generated from the grammar:

$$t \quad ::= \texttt{Bool} \quad | \quad t\ \texttt{List} \quad | \quad (t \times \ldots \times t) \rightarrow t$$

Using this type language we can imagine assigning a type to every SUPL expression and every SUPL function. Note that $(t \times \ldots \times t)$ contains a type for every parameter of a function. Give type rules that assign such types to every SUPL expression and every SUPL function. No subtyping please! Give also type constraints for all SUPL constructs and argue informally that a program is typable with your type rules if and only if the type constraints are solvable.

4. The following λ-term encodes Ackermann's function:

$$Ackermann \;=\; \lambda m.\lambda n.((m(\lambda f.\lambda m.f((mf)(\lambda x.x))))(\lambda m.\lambda f.\lambda x.f((mf)x)))n$$

I have used the standard convention that the body of a λ-abstraction extends "as far as possible." Write down the constraint system for *Ackermann* in the sense of Lecture note 10 (and homework 10). Decide if *Ackermann* has a simple type or not by doing unification of the constraints. If *Ackermann* has a simple type, then give a type derivation.

5. Consider the language of type expressions generated by the grammar:

$$\begin{array}{llllll} t & ::= t \to t & \mid & \alpha & \mid & \mu\alpha.t \\ & \mid \quad \texttt{Int} & \mid & \texttt{Float} & \mid & [l_1 : t_1, \ldots, l_n : t_n] \end{array}$$

where α is a type variable, and $\mu\alpha.t$ is a recursive type. A type of the form $[l_1 : t_1, \ldots, l_n : t_n]$ is a record type with fields $l_1, \ldots, l_n$ and field types $t_1, \ldots, t_n$. A type expression denotes a type which is the infinite unfolding of the type expression such that all μ have disappeared. Black Holes are defined as usual (a Black Hole does not denote a type.) There is a subtype ordering $\leq$ which is a partial order on the types such that it is true that

- $\texttt{Int} \leq \texttt{Float}$.
- $s \to t \leq s' \to t'$ if and only if $s' \leq s$ and $t \leq t'$.
- $[l_1 : t_1, \ldots, l_n : t_n, \ldots, l_{n+m} : t_{n+m}] \leq [l_1 : t'_1, \ldots, l_n : t'_n]$ if and only $t_1 \leq t'_1$ and $\ldots$ and $t_n \leq t'_n$. (It is assumed that $n \geq 0$ and $m \geq 0$.)

Give a precise definition of subtyping on these types. The definition must be in the spirit of "Efficient Recursive Subtyping." Explain in detail how to decide the type ordering in $O(n^2)$ time, where n is the size of the largest of the two types being compared. After that, demonstrate how your algorithm works on this subtype problem:

$$\mu\alpha.[l : (\texttt{Int} \to \alpha), m : \texttt{Float}] \leq \mu\beta.[l : (\texttt{Float} \to [l : (\texttt{Int} \to \beta)])] \;.$$

Is the answer to the subtype problem "true" or "false"?

6. First some motivation for the problem! Suppose we have a Scheme program

```
(define exp
  (lambda (x n)
    ;; we assume n is an integer greater than or equal to 0
    ;; (exp x n)  =  x to the power n
    (if (zero? n)
        1
        (* x (exp x (- n 1))))))
```

Suppose next that we want to invoke `exp`, and that we have a value for `n` (it is the value 8), but that we do not yet have a value for `x`. Is there some way of speeding up the call to `exp` now that we know the value of one of the parameters? Can we somehow generate a specialized function which has built in that `n` is 8, and which runs faster than `(exp x 8)` ? In computability theory, this idea is known as Kleene's S-m-n theorem, and in programming language research, the idea is known as partial evaluation. It would be lame to generate the specialized function `exp-8`:

```
(define exp-8
  (lambda (y)
    (exp y 8)))

(define exp
  (lambda (x n)
    ;; we assume n is an integer greater than or equal to 0
    ;; (exp x n)  =  x to the power n
    (if (zero? n)
        1
        (* x (exp x (- n 1))))))
```

We want to take advantage of knowing that `n` is 8. It would be good if we could generate the specialized program

```
(define exp-8-smart
  (lambda (y)
    (square (square (square y)))))

(define square
  (lambda (z)
    (* z z)))
```

On the Web you can find partial evaluators for Scheme which do this automatically![1] The approach we will consider here does the partial evaluation in two steps. The first step identifies expressions like `(zero? n)` and `(- n 1)` which clearly can be evaluated without knowing a value for `x`. The second step uses the output of the first step to do the actual specialization which leads to the faster program. End of motivation.

This problem is about doing step 1 of the above mentioned two-step partial evaluation strategy. Specifically, the problem is about identifying the expressions in a SUPL program which can be evaluated when we know the value of some of the parameters of the enclosing functions. (The problem is not about doing the subsequent specialization.) Consider a SUPL program and focus on a SUPL function `f` with parameters $\mathtt{x}_1$, ..., $\mathtt{x}_n$, ..., $\mathtt{x}_{n+m}$. (It is assumed that $n \geq 0$ and $m \geq 0$.) The body of `f` may of course call other SUPL functions in the program. Suppose we intend to call `f` and that we have already computed the actual arguments for $\mathtt{x}_1$, ..., $\mathtt{x}_n$, but that we do not yet have actual arguments for $\mathtt{x}_{n+1}$, ..., $\mathtt{x}_{n+m}$. The problem is: describe a way to determine which expressions in the program we can evaluate when knowing only values for $\mathtt{x}_1$, ..., $\mathtt{x}_n$. Discuss briefly the limitations of your approach.

[1] Shameless advertising: If you think it would be cool to implement a partial evaluator for Java, let us discuss doing a CS 590 project (after the final is over :-).