CS411 Final Examination

(Open Notes)

16 May 2001 12:00-2:30

1 Subtypes

Consider extending IMPX with *subrange* types: the constructor

$$\tau \quad ::= \quad [m \,..\, n]$$

denotes the type consisting of all the integers between m and n, inclusive. Note n and m are *numbers*, not expressions – that is, they are compile time constants.

a) (8 points) Give subtyping rules that relate the subrange types to one another and to the *int* type. □

Solution a)

$$\frac{}{\vdash\ [m,\ n]\ \prec\ \text{int}}$$

$$\frac{m\ >\ n}{\vdash\ [m,\ n]\ \prec\ [m',\ n']}$$

$$\frac{(m'\ \leq\ m)\ \ \wedge\ \ (n\ \leq\ n')}{\vdash\ [m,\ n]\ \prec\ [m',\ n']}$$

Note none of these use the logical rules in any significant way – the power is all in comparing values of m and n. □

Now consider extending IMPX with restricted **var** types:

$$\tau \quad ::= \quad \textbf{readonly var}\ \tau \ \mid\ \textbf{writeonly var}\ \tau$$

with the obvious meanings that a **readonly** variable cannot be assigned to, and a **writeonly** variable cannot be read. (**writeonly var** types appear in some languages as "result" parameters).

b) (12 points) Give subtyping rules for **readonly** and **writeonly** types, and explain why they are sound. Your rules should cover all the interesting cases:

$$\textbf{readonly var } \tau \ ? \ \textbf{var } \tau$$
$$\textbf{writeonly var } \tau \ ? \ \textbf{var } \tau$$
$$\textbf{readonly var } \tau \ ? \ \textbf{readonly var } \tau'$$
$$\textbf{writeonly var } \tau \ ? \ \textbf{writeonly var } \tau'$$

where $\tau \prec \tau'$. □

Solution b)

$$\frac{}{\vdash \textbf{var } \tau \prec \textbf{xxxonly var } \tau}$$

(Fewer operations/methods implies larger in the subtype ordering).

$$\frac{\vdash \tau \prec \tau'}{\vdash \textbf{readonly var } \tau \prec \textbf{readonly var } \tau'}$$

That is, **readonly var** τ is covariant in τ.

$$\frac{\vdash \tau' \prec \tau}{\vdash \textbf{writeonly var } \tau \prec \textbf{writeonly var } \tau'}$$

That is, **writeonly var** τ is contravariant in τ. □

IMPX does not have array types, but it does have **var** and function types, and it very recently acquired subrange types. We propose to model the Java array type by **var**-returning functions; that is,

$$\textbf{array } n \textbf{ of } \tau \qquad \text{becomes} \qquad \textbf{fun}(i : [1..n]) \textbf{ var } \tau$$

(Yes, I know Java uses 0-origin arrays; don't worry about this. Java also has a runtime mechanism to query array subscript bounds; don't worry about this either.)

Subscripts will be replaced by function application, so the translation from Java expressions using arrays to IMPX expressions will look like

$$T[\![a[i]]\!] \qquad = \qquad (T[\![a]\!])(i)$$

Below we call this representation the "IMPX translation of array types."

c) (10 points) Java uses the array subtyping rule

$$\frac{\vdash\ \tau\ \prec\ \tau'}{\vdash\ \ \textbf{array}\ n\ \textbf{of}\ \tau\ \ \prec\ \ \textbf{array}\ n\ \textbf{of}\ \tau'}$$

Show by example that this rule is not sound. (In Java implementations this problem is addressed by a runtime check). Is this rule derivable in the IMPX translation? Explain your answer. □

Solution c) Suppose $\tau \prec \tau'$; and let t' be an expression that is in τ' but not in τ. Then the program

$a : \textbf{array}\ n\ \textbf{of}\ \tau\ \leftarrow\ \textbf{new}\ \ldots;$
$a' : \textbf{array}\ n\ \textbf{of}\ \tau'\ \leftarrow\ \textbf{new}\ \ldots;$
$\ldots$
$a'\ \leftarrow\ a;$ (legal by unsound subtype rule)
$a'[i]\ \leftarrow\ t';$ (legal by types of a, t')
$a[i];$ (returns a value not in τ

The rule is not derivable in the translation, because the translation uses **var** τ in place of τ, and the types **var** τ and **var** τ' are unrelated. □

d) (10 points) Is the rule

$$\frac{n > m}{\vdash\ \ \textbf{array}\ n\ \textbf{of}\ \tau\ \ \prec\ \ \textbf{array}\ m\ \textbf{of}\ \tau}$$

sound? Is it derivable in the IMPX translation? Explain your answer. □

Solution d) The rule is sound and derivable in the translation as follows: From

$$n\ >\ m$$

we can deduce

$$\vdash\ [1..m] \prec [1..n]$$

whence

$$\vdash\ \textbf{fun}([1..n])\ \textbf{var}\ \tau\ \prec\ \ \textbf{fun}([1..m])\ \textbf{var}\ \tau$$

by contravariance of function types. This is the desired ordering of the array types in translation. □

e) (10 points) Consider adding Java types

$$\textbf{readonly array } n \textbf{ of } \tau \qquad \text{and} \qquad \textbf{writeonly array } n \textbf{ of } \tau$$

What are the natural IMPX translations of these types, and what are the subtyping relations among them and ordinary array types? Explain your answers. □

Solution e) The idea here is to make the array element type **readonly** or **writeonly**. Thus, the translations are

$$T[\![\textbf{readonly array } n \textbf{ of } \tau]\!] = \textbf{fun}(i : [1..n]) \textbf{ readonly var } \tau$$
$$T[\![\textbf{writeonly array } n \textbf{ of } \tau]\!] = \textbf{fun}(i : [1..n]) \textbf{ writeonly var } \tau$$

Since the **var** τ occurs in a covariant position in the function types, the translated array types vary in the same direction that **readonly** and **writeonly** variables do –

$$\textbf{array } n \textbf{ of } \tau \prec \textbf{readonly array } n \textbf{ of } \tau$$
$$\textbf{array } n \textbf{ of } \tau \prec \textbf{writeonly array } n \textbf{ of } \tau$$

and for $\tau \prec \tau'$

$$\textbf{readonly array } n \textbf{ of } \tau \prec \textbf{readonly array } n \textbf{ of } \tau'$$
$$\textbf{writeonly array } n \textbf{ of } \tau' \prec \textbf{writeonly array } n \textbf{ of } \tau$$

□

2 Compiling Recursive Functions

This problem refers to the compile function $K[\![e]\!]\pi$ described in lecture and the handout of 1 May. That function compiled a language with function values and nonrecursive **let** bindings, but no explicit recursive functions.

Consider adding support for explicitly recursive function definitions. The added syntax is

$$e \quad ::= \quad \textbf{rec } f(x : \tau)\tau' \{ e_1 \} \textbf{ in } e_2$$

This is almost equivalent to

$$\textbf{let } f \sim \textbf{lambda } x : \tau \textbf{ dot } e_1 \textbf{ in } e_2$$

except that the definition is recursive, so the function name f can appear free in the function body e_1 and will be bound to the function being defined.

a) (40 points): Show how to add support for recursive function definitions, by adding/changing cases in the definition of $K[\![e]\!]\pi$. Explain your answer.

Note the type system does not distinguish between recursive and nonrecursive functions; so for example the following program is legal:

$$\begin{array}{l}\textbf{let } f \sim \textbf{newvar fun}(\text{int})\text{int } \textbf{in} \\ \quad \textbf{let } fn \sim \textbf{lambda } x : \text{ int } \textbf{dot } 17 \textbf{ in} \\ \quad\quad \textbf{rec } fr(x : \text{ int})\text{int } \{\ldots\ fr(x-1)\ \ldots\} \textbf{ in} \\ \quad\quad\quad \ldots\ f \leftarrow fn;\ \ldots;\ f \leftarrow fr;\ \ldots;\ (f\uparrow)(11);\ \ldots\end{array}$$

Thus, a call through a function valued variable may be to a nonrecursive or a recursive function. This cannot be decided at compile time. Hence, the calling conventions for nonrecursive and recursive functions *must be compatible.* □

Solution a) The representation of a recursive function f is as always a pair

$$\langle\ ip,\ ep\ \rangle$$

where ip is the function body code, to be executed in environment ep extended by a binding for the parameter value. The only change introduced by recursion is that, in the $\langle ip,\ ep\rangle$ pair that is the value of a recursive function f, the environment ep should contain a binding for f. A little thought should convince you that this just puts a cycle in the data structure for the function value. The code of the function body is unchanged from the nonrecursive case, and so is the code for function invocation.

The code generation function case for a recursive function block is roughly a combination of the case for a **let** block and the case for a function valued expression, modified to introduce the "loop" in the data structure described above.

$K[\![\textbf{rec } f(x : \tau)\tau' \ \{\ e_1\ \}\ \textbf{in } e_2]\!]\pi =$

```
        p ← newframe(2);
        p[VAL+IP] ← L1;  p[VAL+EP] ← ep;
        q ← newframe(2);
        PUSH(q, p[VAL+EP]);
        q[VAL+IP] ← p[VAL+IP];  q[VAL+EP] ← p[VAL+EP];
        PUSH(p, ep);
        goto L2;
    L1:
        K[[e1]](π ⊕ (f : ...) ⊕ (x : τ))
```

goto sp[LINK][VAL+IP];

L2:

$K[\![e_2]\!](\pi \oplus (f : \ldots))$

POP(ep)

As always, L1 and L2 are globally unique labels. The code is very similar to the code for a λ expression. Putting the cycle into the environment data structure is the slightly tricky part. Both p and q are equivalent $\langle ip,\ ep \rangle$ pairs, representing the value of f. The ep component of both these pairs is q itself, which is pushed onto the ep stack of p. This is the only reason q exists – it is just "part of" the function value.

Then p is pushed onto the environment as the binding for f. We jump to L2 and execute the code for the block body e_2. We then pop the binding for f (to restore the environment) and we are finished.

Note the code for the body of f, starting at L1, is compiled in a context that includes both f and the function parameter x. At run time f corresponds to the frame q that was constructed above, while x will be supplied by the caller as usual. □

3 Self Types

Consider the following program:

class C_1 {
  m_1 : **method**()C_1 { **newobj**(C_1) } }
subclass C_2 **of** C_1 {
  f : **var** int;
  m_2 : **method**(x : **Self**) int { ... } }

a) (10 points) Argue that the expression

let $c_2 \sim$ **newobj**(C_2) **in** $c_2.m_2(c_2.m1())$

cannot be considered type correct, by giving a method body for m_2 that results in a type error. □

Solution a) If **Self** is considered equivalent to C_2 inside the class definition for C_2, then in the body of m_2 we should expect to be able to reference the field f, which is an attribute of C_2 but *not* of C_1. The value returned from m_1, which is produced by invoking the constructor for C_1, has no f attribute. This is true even if m_1 is obtained by selecting from an object of class C_2, since m_1 is inherited from C_1. Thus, the expression

$$(x.f)\uparrow$$

as the body of m_2 references a nonexistent attribute. □

b) (10 points) Suppose we change the definition of C_1 to

class C_1 {
  m_1 : **method**()**Self** { **newobj**(C_1) } }

Where is the type error now? Suggest a way to fix this problem. □

Solution b) This change moves the type error to the result of m_1. Specifically, the invocation

newobj(C_1)

in m_1 produces a result of the instance type of C_1, but the result type of m_1 is specified as **Self**. This is correct only under the assumption

$$IT[\![C_1]\!] \prec \textbf{Self}$$

but any subclass will violate this assumption.

In general, the only types we can guarantee to be subtypes of **Self** are types derived somehow from **Self** or **self**. For example, there is no fundamental problem in supporting

newobj(**Self**)

which would be guaranteed always to construct an object of the **Self** type even when invoked in an inherited method. □

4 The Pair Calculus

In this question we develop a little calculus inspired by the untyped object calculus. We call it the *pair calculus*, because values are ordered pairs.

Here is the syntax of the pair calculus:

$$
\begin{array}{rcll}
e & ::= & x & \text{(identifier reference)} \\
 & | & e.fst & \text{(select first component)} \\
 & | & e.snd & \text{(select second component)} \\
 & | & \varsigma(x)\langle e_1,\ e_2\rangle & \text{(pair formation)} \\
 & | & (e_1 \bowtie e_2) & \text{(combining)}
\end{array}
$$

The first case is the usual identifier reference case for bound variables.

The next two cases are just field selection, as for object or product types. The meaning is closer to method invocation (object types) than to field selection (product types), as components are evaluated "lazily" as described below.

The fourth case, "pair formation," is similar to an object construction expression in the object calculus. That is,

$$\varsigma(x)\langle e_1,\ e_2\ \rangle \text{ is like } [\ fst \sim \varsigma(x)e_1,\ snd \sim \varsigma(x)e_2\ ]$$

In both cases the $\varsigma(x)$ serves as a binding construct. The parameter x is bound to the containing pair for evaluation of e_1 or e_2. Like a λ abstraction, a ς abstraction is irreducible. It shields the body expressions from evaluation until an actual selection is performed, at which point the selected component expression is evaluated "lazily."

For example, if A is an irreducible expression, the term

$$(\varsigma(x)\langle\ x.snd,\ A\ \rangle).fst$$

should evaluate to A.

The final case, "combining," is intended to serve the same role as method updating in the object calculus. It is a bit difficult to describe informally, so we present a large step evaluation rule for it:

$$\frac{\begin{array}{l} e \ \rightarrow\ \varsigma(x)\langle e_1,\ e_2\rangle \\ e' \ \rightarrow\ \varsigma(x')\langle e'_1,\ e'_2\rangle \end{array}}{(e \bowtie e') \ \rightarrow\ \varsigma(x)\langle e_2,\ [x//x'](e'_1)\rangle}$$

The substitution operator $[\cdot//\cdot]e$ is assumed to do renaming if necessary to avoid capture. Thus, the $\bowtie$ operator makes a new pair from the second component of its left operand and the first component of its right operand, renaming the bound variable appropriately.

For example, if A, B, and D are irreducible terms, then

$$(\varsigma(x)\langle\ A,\ B\ \rangle)\ \bowtie\ (\varsigma(x')\langle\ x',\ D\ \rangle)$$

reduces to

$$\varsigma(x)\langle\ B,\ x\ \rangle$$

Note the renaming of x' to x in the second component.

a) (10 points) Give eager large step evaluation rules for the pair calculus just described. □

Solution a) Note that ς behaves like λ – the component expressions of a ς abstraction are not evaluated, even by eager rules, until they are selected.

As usual, we treat as values the closed irreducible terms – in this case ς-terms. The rules are

$$\frac{}{v\ \rightarrow\ v}$$

$$\frac{e\ \rightarrow\ v' \equiv \varsigma(x)\langle\ e_1,\ e_2\ \rangle \quad ([v'/x](e_1))\ \rightarrow\ v}{e.fst\ \rightarrow\ v}$$

$$\frac{e\ \rightarrow\ v' \equiv \varsigma(x)\langle\ e_1,\ e_2\ \rangle \quad ([v'/x](e_2))\ \rightarrow\ v}{e.snd\ \rightarrow\ v}$$

There is no possibility of capture in these rules, since we substitute only (closed) values. □

b) (10 points) Let A, B, C and D be irreducible terms in the pair calculus. Use your rules to reduce

$$((\varsigma(x)\langle\ A,\ x.snd\ \rangle)\bowtie(\varsigma(y)\langle\ C,\ D\ \rangle)).fst$$

to an irreducible term. Hint: the result should be C. □

c) (5 points) Let A be some term that reduces to a pair value

$$\varsigma(x)\langle\ v_1,\ v_2\ \rangle$$

Can you find a term that reduces to

$$\varsigma(x)\langle\ v_2,\ v_1\ \rangle$$

(that is, to the "reversal" of the value)? □

Solution c) It turns out this was an ill-advised question. First, the "obvious" solution

$$\varsigma(x)\langle\ A.snd,\ A.fst\ \rangle$$

is perfectly correct for *arbitrary* A (that is, even if A diverges or reduces to a value of the form

$$\varsigma(x)\langle\ e_1,\ e_2\ \rangle$$

where e_1 and e_2 are not both closed). The more "clever" answer, which is the one I expected and which most people seem to have noticed, is

$$A\ \rightarrow\ \varsigma(x)\langle\ v_1,\ v_2\ \rangle\ \ \Rightarrow\ \ (A \bowtie A)\ \rightarrow\ \varsigma(x)\langle\ v_2,\ v_1\ \rangle$$

That is, the $\bowtie$ operator can be used to reverse a pair of values. Note this trick works *only* because v_1 and v_2 are closed. For example, consider replacing v_2 by $x.fst$. The resulting term would be

$$\begin{aligned}\varsigma(x)\langle v_1,\ x.fst\rangle\ \bowtie\ \varsigma(x)\langle v_1,\ x.fst\rangle\\ \rightarrow\ \ \varsigma(x)\langle\ x.fst,\ v_1\ \rangle\end{aligned}$$

and the first component now diverges – definitely not reversal!

That's all right, because the question was posed for A of the restricted form where v_1 and v_2 are closed values, and either of the above solutions is correct in that case. Unfortunately, I think this question may have confused a few people about the interpretation of values and lazy/eager evaluation in the pair calculus, and for that I apologize. □

d) (5 points) Give a pair calculus term whose evaluation diverges. □

Solution d)

$$(\varsigma(x)\langle\ x.snd,\ x.fst\ \rangle).fst$$

□

Now consider a version of the untyped lambda calculus

$$e \quad ::= \quad x \ \mid\ \lambda\, x\ .\ e \ \mid\ e_1(e_2)$$

that uses the eager evaluation rules:

$$\frac{}{v \ \rightarrow\ v}$$

and

$$\frac{\begin{array}{l} e_1 \ \rightarrow\ \lambda\, x\ .\ e_1' \\ e_2 \ \rightarrow\ v_2 \\ ([v_2/x](e_1')) \ \rightarrow\ v \end{array}}{e_1(e_2) \ \rightarrow\ v}$$

As usual, the values v are closed irreducible terms.

e) (20 points) Give a translation $T[\![\cdot]\!]$ taking terms in the untyped lambda calculus to terms in the pair calculus. Your translation should preserve derivations. Show this in the forward direction – that is, prove

$$e_1 \ \rightarrow_\lambda\ e_2 \quad \Rightarrow \quad T[\![e_1]\!] \ \rightarrow_p\ T[\![e_2]\!]$$

(where $\rightarrow_\lambda$ means reduction in the lambda calculus, and $\rightarrow_p$ means reduction in the pair calculus). The proof should be by induction on derivations $\rightarrow_\lambda$. □

Solution e) The examples in the previous parts of this problem were intended to suggest the solution here. Recall how we embedded the lambda calculus into the untyped object calculus. A function was represented as an object with a *val* method. To call the function, we would update the *arg* method to something that produced the desired argument value when invoked; then we would invoke the *val* method to extract the result. This worked because the caller and callee had an agreed-upon place to put the argument value – the *arg* attribute.

We can do essentially the same thing in the pair calculus. We use the fst and snd components of a pair to store the function and argument values; and we simulate updating the components using the $\bowtie$ operator. Here are the details.

$$T[\![x]\!] \;=\; x$$

$$T[\![\lambda x.e]\!] \;=\; \varsigma(x)\langle\, x.fst,\; [x.snd/x](T[\![e]\!])\,\rangle$$

$$T[\![e_1(e_2)]\!] \;=\; ((T[\![e_1]\!]) \bowtie (\varsigma(z)\langle\, T[\![e_2]\!],\; z.snd\,\rangle)).fst$$
where z is not free in e_2

We need to prove that derivations are preserved by this translation, as required in the problem statement above. We'll do this by induction on the derivation of $e_1 \;\rightarrow_\lambda\; e_2$. First, a few preliminary observations.

Lemma 1. Free variables are preserved by the translation:

$$FV[\![e]\!] \;=\; FV[\![T[\![e]\!]]\!]$$

for any expression e. This is easily proved by induction on the structure of E.□

Lemma 2. Values are preserved by the translation: if e is a lambda calculus value (a closed λ-term) then $T[\![e]\!]$ is a pair calculus value (a closed ς-term). This is immediate from Lemma 1.□

Lemma 3. Substitution of closed terms commutes with the translation. That is, if e_2 is closed, then

$$T[\![\; [e_2/x]e_1 \;]\!] \;=\; [T[\![e_2]\!]/x](T[\![e_1]\!])$$

This is one of those intuitively obvious facts with a straightforward but fairly lengthy proof, which I shall not present here. But be aware that we're relying on it.□

Finally, a special case of the copy rule for terminating expressions:

Lemma 4. If e_2 is closed,

$$(e_2 \ \rightarrow_p \ v_2) \ \wedge \ (([v_2/x]e_1) \ \rightarrow_p \ v_1) \\ \Rightarrow \ (([e_2/x]e_1) \ \rightarrow_p \ v_1)$$

This is proved as usual by induction on derivations.□

Now we show

$$e_1 \ \rightarrow_\lambda \ e_2 \quad \Rightarrow \quad T[\![e_1]\!] \ \rightarrow_p \ T[\![e_2]\!]$$

by induction on the derivation of $e_1 \ \rightarrow_\lambda \ e_2$. There are only two rules.

Case 1: The derivation is $v \ \rightarrow_\lambda \ v$. This implies $e_1 = e_2$, and it is a value in the lambda calculus. By Lemma 2, this implies $T[\![e_1]\!]$ is a value in the pair calculus, and the result is immediate.

Case 2: The derivation ends with a use of the application rule

$$\frac{\begin{array}{l} e_1 \ \rightarrow \ \lambda \ x \ . \ e_1' \\ e_2 \ \rightarrow \ v_2 \\ ([v_2/x](e_1')) \ \rightarrow \ v \end{array}}{e_1(e_2) \ \rightarrow \ v}$$

Inductively we know the following

$$T[\![e_1]\!] \ \rightarrow_p \ T[\![\lambda x.e_1']\!] \ = \ \varsigma(x)\langle \ldots, \ [x.snd/x]T[\![e_1']\!]\rangle$$

$$T[\![e_2]\!] \ \rightarrow_p \ T[\![v_2]\!]$$

and

$$T[\![[v_2/x](e_1')]\!] \ = \ [T[\![v_2]\!]/x]T[\![e_1']\!] \ \rightarrow_p \ T[\![v]\!]$$

where the equality in the last line follows from Lemma 3.

We need to conclude

$$T[\![e_1(e_2)]\!] \ \rightarrow_p \ T[\![v]\!]$$

(after some renaming of the conclusion to conform to the naming conventions of of the application rule).

Substituting in the appropriate case of the definition of $T[\![\cdot]\!]$, we get

$$\begin{array}{rcl} T[\![e_1(e_2)]\!] & \hookrightarrow & (T[\![e_1]\!] \bowtie \varsigma(z)\langle T[\![e_2]\!], T[\![e_2]\!]\rangle).fst \\ & = & (\varsigma(x)\langle \ldots, [x.snd/x]T[\![e_1']\!]\rangle \ \bowtie \ \varsigma(z)\langle T[\![e_2]\!], T[\![e_2]\!]\rangle).fst \\ & \hookrightarrow & \varsigma(x)\langle [s.xnd/x]T[\![e_1']\!],\ t[\![e_2]\!]\rangle.fst \end{array}$$

where the last step follows because e_2 is closed, so substitution for z is a no-op. Expansion of the selection yields

$$\hookrightarrow \quad [T[\![e_2]\!]/x]T[\![e_1']\!]$$

From this, and the observation made above that

$$T[\![e_2]\!] \ \rightarrow_p \ T[\![v_2]\!]$$

we can apply Lemma 4 to conclude

$$([T[\![e_2]\!]/x]T[\![e_1']\!]) \quad \rightarrow \quad T[\![v]\!]$$

as desired. □