Miniprojects
Advanced Control

15-814: Types and Programming Languages
Frank Pfenning

Due Friday, December 5, 2025
200 points

Please pick one among the following two mini-projects and hand in your PDF writeup and code in Gradescope by the due date. We will try to be flexible, but due to the end of semester and grading deadlines, extensions will be limited.

You may consult with the course staff, and also post publicly on Piazza if your post does not give away answers. Along similar lines, feel free to discuss the problems and approaches with your classmates, but avoid giving away detailed answers that would take away from the learning experience. As with any homework, you may consult papers and LLMs, but if you do please mention in your write-up how and what you used, including any papers you consulted that steered you towards your solutions. This information is helpful for us.

Please keep in mind that answers to the problems are not unique. To the extent possible, you should stick to the zen of the course, including the notation, style of rules, properties of interests, etc.

You may also define your own project, in which case we'd like a proposal with preliminary ideas, references, etc. by **Thursday November 20** so that we can provide feedback in a timely manner. You should submit a single `mp.zip` file containing your answers as `mp.pdf` and any code. If it should pass the LAMBDA implementation, please use the `.cbv` extension.

Your PDF report should consist of the following two sections:

Executive summary. Describe the key points of what you discovered. This is also where you should cite outside sources you consulted.

Answers to the tasks. Follow all instructions in the tasks. If you get creative, also briefly mention the intuitive idea.

1 Call/cc

In this mini-project we explore the advanced control construct callcc, which is short for "*call-with-current-continuation*". With callcc we can implement various patterns of control such as nondeterministic choice, coroutines, or backtracking search at a high level of abstraction.

We introduce a new type $\neg\tau$ which is inhabited by continuations that accept values of type τ. This is related to the *proposition* $\neg A$ from classical logic (see Tasks 12 and 13). Its introduction form is what makes it odd: it has the form callcc $k.\, e$, binding the *variable* k with scope e. When executing callcc $k.\, e$ we substitute the *current continuation* for k in e and evaluate the result. The *current continuation* captures whatever remains to be done after the evaluation of callcc until the completion of the whole program. In the interpreters of Lecture 10 or Lecture 12 we represented the continuation as metalevel function. We cannot express this behavior directly in the small-step dynamics $e \mapsto e'$ since we do not have a representation of the continuation available, so we need a somewhat different formalization of the language dynamics.

In our rules, we had destructors applied to constructors, such as $(\lambda x.\, e)\, v \mapsto [v/x]e$, and we also had congruence rules such as $e_1\, e_2 \mapsto e_1'\, e_2$ if $e_1 \mapsto e_1'$. The congruence rules determine *where* a reduction can take place. An alternative formulation, sometimes called *contextual semantics*, uses an *evaluation context* K which is an expression with a single hole (written as $[\,]$). For functions only, we define

$$\text{Evaluation Context} \quad K \quad ::= \quad [\,] \mid K\, e \mid v\, K$$

where v stands for values. We write $K[e]$ for the result of filling the hole in K with expression e. The sole rule of reduction for functions is then

$$\frac{}{K[(\lambda x.\, e)\, v] \longrightarrow K[[v/x]e]}\ \textsf{step/beta}$$

All of the desirable properties for our stepping judgment should also hold with this new formulation because $e \mapsto e'$ iff $e \longrightarrow e'$.

Task 1 Show the steps in the evaluation of $(\lambda f.\, f)\, ((\lambda x.\, \lambda y.\, x)\, 3)$ for a value 3, writing out the evaluation context at each step explicitly.

Task 2 Complete the definition of evaluation contexts and the step relation for the constructs associated with types $\tau \times \sigma$, 1, $\Sigma_{\ell \in L}(\ell : \tau_\ell)$, $\&_{\ell \in L}(\ell : \tau_\ell)$, $\mu\alpha.\, \tau$, $\forall\alpha.\, \tau$ and $\exists\alpha.\, \tau$.

We use evaluation contexts K as our representation of continuations. More precisely:

$$\frac{}{K[\textsf{callcc}\, k.\, e] \longrightarrow K[[K/k]e]}\ \textsf{step/callcc}$$

If we want to escape our local control context we can *throw* a value to a continuation. The construct is throw e_1 e_2 where e_1 evaluates to a continuation K_1, e_2 evaluates to a value v_2, and we pass v_2 to K_1.

$$\frac{}{K[\textsf{throw}\ K'\ v] \longrightarrow K'[v]}\ \textsf{step/throw}$$

Notice that we abandon K and replace it by the evaluation context K' (which is a value). Continuations K cannot appear directly in the program syntax before evaluation—they only originate from the callcc construct. Also, continuations K are *values*.

$$\frac{}{K\ \mathit{value}}\ \textsf{val/cont}$$

Task 3 Complete the dynamics of callcc by giving the new evaluation contexts.

Task 4 Here are some sample expressions using callcc. Compute their value using the stepping rules for callcc. You do not need to show the computations, just the final values if they exist. We assume the usual definitions of unary natural numbers, including *zero* and *succ*.

(i) $\textsf{callcc}\ k.\, \textsf{throw}\ k\ (\mathit{succ}\ \mathit{zero})$
(ii) $\textsf{callcc}\ k.\, \mathit{succ}\ (\textsf{throw}\ k\ \mathit{zero})$
(iii) $\mathit{succ}\ (\textsf{callcc}\ k.\, \textsf{throw}\ k\ \mathit{zero})$

(iv) $\mathit{callcc}\ k.\,(\mathsf{throw}\ (\mathsf{throw}\ k\ \mathit{zero})\ (\mathsf{throw}\ k\ (\mathit{succ}\ \mathit{zero}))$

$\mathit{capture} = \lambda u.\,\mathsf{callcc}\ k.\,\mathsf{throw}\ k\ (\mathsf{fold}\ k)$

$\mathit{invoke} = \lambda m.\,\mathsf{case}\ m\ (\mathsf{fold}\ k \Rightarrow \mathsf{throw}\ k\ (\mathsf{fold}\ k))$

(v) $\mathit{invoke}\,(\mathit{capture}\ \langle\,\rangle)$

You may ignore the types for now, but they will be relevant in next task.

Task 5 Now provide the typing rules for callcc $k.\,e$ and throw $e_1\ e_2$ where continuations accepting a value of type τ have type $\neg\tau$. Since these are static typing rules, applied before expressions are evaluated, they cannot encounter a continuation K, only variables k standing for continuations.

All examples in Task 4 should be well-typed (but you don't have to show any typing derivations).

Task 6 Provide valid types for the expressions (i)–(v) in Task 4. This should include separate types for functions *capture* and *invoke*. Feel free to make type definitions if such a shorthand would be convenient. Your types do not need to be most general.

Task 7 Since expressions may contain actual continuations K at runtime, devise a new judgment for typing expressions at runtime. Let's call it the *dynamic typing judgment*.[1] You only need to show the rules for callcc, throw, and continuations K, and explain how the rules for the static typing judgment for other forms of expressions may need to be updated. For concreteness, show the modified dynamic typing rules for eager pairs.

Task 8 Extend the canonical forms theorem with a case for types $\neg\tau$. You do not need to prove it. Does it refer to the static typing judgment or the dynamic typing judgment. Why?

Task 9 State the preservation theorem for contextual dynamics and the structure of its proof (by induction or cases on which judgment or construction). Then show all cases in the proof relevant to the transitions rules for callcc and throw. If you need any lemmas you should state them carefully, but you do not need to prove them.

Task 10 State the progress theorem for the contextual dynamics and the structure of its proof (by induction or cases on which judgment or construction). You do not need to prove it or show any cases, but you should convince yourself that it is true because this is an important test for the correctness of your rules for typing and evaluation.

Task 11 Define a function $\mathit{andalso} : \neg\mathit{bool} \to \mathit{bool} \to \mathit{bool} \to 0$ where $\mathit{andalso}\ \ K\ \ e_1\ \ e_2$ passes the result of $x \wedge y$ to K, "short-circuiting" the conjunction as with the language C's `&&` operation. In other words, e_2 should not evaluated if e_1 evaluates to false even though we are in a call-by-value language.

Then define a function $\mathit{orelse} : \neg\mathit{bool} \to \mathit{bool} \to \mathit{bool} \to 0$ in an analogous fashion, modeling C's short-circuiting disjunction `||`.

Task 12 Under the Curry-Howard isomorphism, the *proposition* $A \vee \neg A$ is interpreted as the type $\tau + (\tau \to 0)$. Show that with callcc and *without using recursion or recursive types* we can define a closed term for every instance of the axiom $A \vee \neg A$. This can be provided uniformly as a closed term of type $\forall\alpha.\,\alpha + (\alpha \to 0)$. This means that with only callcc (and excluding recursion) we can find a closed term for every proof in *classical* propositional natural deduction (which uses excluded middle).

[1] This name does not imply that you need to check types as the program executes, just that we need to type dynamic artifacts.

Task 13 Prove the other direction, namely, that with callcc (and without recursion) every closed expression of type τ corresponds to some proof in classical propositional logic. Together with the previous task, we will then have established two-thirds of Curry-Howard isomorphism for classical propositional logic, where the only missing component is a connection between proof reduction and rules for computation.
[**Hint:** It may be helpful to understand which *axioms* your typing rules for callcc and throw correspond to.]

Task 14 Extend our implementation of the interpreter from Lecture 12 in LAMBDA with callcc and throw. Also, encode your answers in Tasks 4, 11, 12, and 13 in this implementation and verify their correctness or test them on small inputs to the extent possible.

Task 15 Differentiate your static typing rules from Task 5 into bidirectional typing rules for callcc and throw. Your rules should not require any type annotations in the expressions.

In the last task, we explore the use of continuations as a kind of exception mechanism.

Task 16 Modify the interpreter you wrote so that (at the metalevel) it "raises an exception" to signal the object-level computation "went wrong" (which should be impossible when evaluating well-typed expressions). Because the LAMBDA implementation does not support continuations you won't be able to run it.

2 Quotation

In this mini-project we explore a statically typed version of quotation in a programming language. This can be used, for example, for meta-programming, explicit staging of computation, runtime code generation, or inlining to improve efficiency.

We introduce a new type $\Box\tau$ whose values are *quoted expressions* of type τ, written as $\mathsf{quote}\ e$. We consider $\mathsf{quote}\ e$ to be *observable* regardless of the type of e. For example, after evaluation of the expression $(\lambda x.\, x)\ (\mathsf{quote}\,(\lambda x.\, \lambda y.\, x))$ the user should be able to see the value $\mathsf{quote}\ (\lambda x.\, \lambda y.\, x)$, possibly with some renaming of the bound variables. On the other hand, we do not want to prevent the usual compilation of functions, and a value of type $nat \to nat$ should remain opaque.

In order to maintain this distinction we have two different kinds of variables: Our usual variables ranging over values, and a new kind ranging over observable expressions which we will call *expression variables*. We still write $x : \tau$ for a usual variable x, where x may or may not be observable, depending on the type τ. However, for an expression variable u, we write $u :: \tau$, and u is always observable independently of type τ. In order to prevent opaque, unobservable values from infecting our observable quoted expressions, our typing rule for quotation is

$$\frac{\Gamma|_{::} \vdash e : \tau}{\Gamma \vdash \mathsf{quote}\ e : \Box\tau}\ \mathsf{tp/quote}$$

Here, $\Gamma|_{::}$ is the restriction of Γ to declarations of the form $\alpha\ type$ and $u :: \tau$. We can define it with

$$\begin{array}{lcl} (\cdot)|_{::} & = & \cdot \\ (\Gamma, x : \tau)|_{::} & = & \Gamma|_{::} \\ (\Gamma, \alpha\ type)|_{::} & = & \Gamma|_{::}, \alpha\ type \\ (\Gamma, u :: \tau)|_{::} & = & \Gamma|_{::}, u :: \tau \end{array}$$

The destructor for quote is a new form of case expression, $\mathsf{case}\ e\ (\mathsf{quote}\ u \Rightarrow e')$. If we have general pattern matching, $\mathsf{quote}\ u$ is a new pattern to match against, binding u to the quoted expression.

Task 1 Give the remaining typing rules for the extension of our call-by-value language with quotation.

When writing types, we assume $\Box$ is a prefix operator with higher precedence that other type constructors, so that $\Box\tau \to \tau$ means $(\Box\tau) \to \tau$ and $\Box 1 \times \Box 1$ means $(\Box 1) \times (\Box 1)$.

Task 2 For each of the following closed expressions, give a type or indicate that it cannot be typed. The type does not need to be most general, and you do not need to show any typing derivations.

(i) $\mathsf{quote}\ (\lambda x.\, x)$

(ii) $\lambda x.\, \mathsf{quote}\ x$

(iii) $\mathsf{quote}\ (\Lambda\alpha.\, \lambda x.\, x)$

(iv) $\Lambda\alpha.\, \mathsf{quote}\ (\lambda x.\, x)$

(v) $\lambda x.\mathsf{case}\ x\ (\mathsf{quote}\ u \Rightarrow \langle u, \mathsf{quote}\ u\rangle)$

We extend the value judgment for closed expressions with the rule

$$\frac{}{\mathsf{quote}\ e\ \mathit{value}}\ \mathsf{val/quote}$$

Task 3 Extend the stepping judgment $e \mapsto e'$ to account for quote and case/quote.

Task 4 State the preservation theorem and the structure of its proof (by induction or cases on which judgment or construction). Then show the new cases relevant to quotation. If you need any lemmas you should state them carefully, but you do not need to prove them.

Task 5 Extend the canonical form theorem to account for quotation. You do not need to prove it.

Task 6 State the progress theorem and the structure of its proof (by induction or cases on which judgment or construction). Then show the new cases relevant to quotation. If you need any lemmas you should state them carefully, but you do not need to prove them.

Task 7 Under the Curry-Howard isomorphism, $\Box\tau$ corresponds to propositions $\Box A$ from an intuitionistic version of the modal logic S_4. This logic is characterized by the *rule of necessitation* (for a judgment $\vdash A$ with no hypotheses rather than in natural deduction)

$$\frac{\vdash A\ \mathit{true}}{\vdash \Box A\ \mathit{true}}\ \mathsf{Nec}$$

and the following axioms for $\Box$ with the corresponding type on the right.

$\vdash \Box A \supset A$	$\forall\alpha.\, \Box\alpha \to \alpha$
$\vdash \Box A \supset \Box\Box A$	$\forall\alpha.\, \Box\alpha \to \Box\Box\alpha$
$\vdash \Box(A \supset B) \supset (\Box A \supset \Box B)$	$\forall\alpha.\, \forall\beta.\, \Box(\alpha \to \beta) \to (\Box\alpha \to \Box\beta)$

Give proof term assignments for Nec and the three characteristic axioms.

Task 8 For each of the following propositions in intuitionistic S_4 either provide a proof term for the corresponding type, or indicate you believe no such proof exists. Like for the axioms, the question is if these propositions are true parametrically in A and B.

(i) $(\Box A \supset \Box B) \supset \Box(A \supset B)$

(ii) $\Box(A \wedge B) \supset (\Box A \wedge \Box B)$

(iii) $(\Box A \wedge \Box B) \supset \Box(A \wedge B)$

(iv) $\Box(A \vee B) \supset (\Box A \vee \Box B)$

(v) $(\Box A \vee \Box B) \supset \Box(A \vee B)$

(vi) $\Box\bot \supset \bot$

(vii) $\bot \supset \Box\bot$

Task 9 Quotation can be used to generate potentially more efficient code at runtime because we can specialize code to some known inputs. In this task we explore this option with a simple example.

$\mathit{nat} = \mu\alpha.\, (\mathbf{zero} : 1) + (\mathbf{succ} : \alpha)$

$\mathit{zero} : \mathit{nat}$
$\mathit{zero} = \mathsf{fold}\ \mathbf{zero} \cdot \langle\, \rangle$

$\mathit{succ} : \mathit{nat} \to \mathit{nat}$
$\mathit{succ} = \lambda n.\, \mathsf{fold}\ \mathbf{succ} \cdot n$

(i) Write the function $\mathit{plus} : \mathit{nat} \to \mathit{nat} \to \mathit{nat}$

(ii) Restage the function to have type $\mathit{plus}' : \mathit{nat} \to \Box(\mathit{nat} \to \mathit{nat})$. This function will take a natural number and generate a source expression for a function that takes the second argument to complete the addition.

(iii) Show the (observable!) result of $\mathit{plus}'\ \overline{2}$.

(iv) Show how you can use β-value reduction (that is, replacing $(\lambda x.\, e)\, v$ by $[v/x]e$) as an optimization to obtain a more efficient source expression for $\mathit{plus}'\ \overline{2}$.

Task 10 In general, we cannot prove $A \supset \Box A$ and, correspondingly, there should be no proof term for $\forall\alpha.\, \alpha \to \Box\alpha$. That's because when the type α is negative (that is, its values are not observable like $\mathit{nat} \to \mathit{nat}$) we cannot build a source expression from the given value.

However, it is possible for some types. Write a function $\mathit{rep} : \mathit{nat} \to \Box\mathit{nat}$ such that $\mathit{rep}\ v \mapsto^* \mathit{quote}\ v$.

Task 11 We now generalize the observation from the previous task. Consider the purely positive (and therefore observable) types τ^+, where σ is an arbitrary (not necessarily positive) type.

$$\tau^+ \quad ::= \quad \tau_1^+ \times \tau_2^+ \mid 1 \mid \tau_1^+ + \tau_2^+ \mid 0 \mid \mu\alpha^+.\, \tau^+ \mid \alpha^+ \mid \Box\sigma$$

Specify a translation from types into expressions in our call-by-value language such that $\mathit{reify}(\tau^+) = e$ with $\cdot \vdash e : \tau^+ \to \Box\tau^+$ and $e\, v \mapsto^* \mathtt{quote}\ v$ for every value $v : \tau^+$. As a function of τ^+, is reify parametric in τ^+?

Task 12 Extend the definition of logical equality with an additional clause for $v \sim v' \in [\Box\tau]$. With your definition, can we prove that $\mathsf{quote}\ ((\lambda x.\, x)\,\langle\,\rangle) \sim \mathsf{quote}\ \langle\,\rangle \in [\Box 1]$? Either way, defend your definition and argue why this relation should or should not hold.

Task 13 Under your definition of logical equality from the previous task, are the types $\Box\tau$ and $\Box\Box\tau$ isomorphic in the sense that there are functions *forth* and *back* such that the two compositions *forth*$\circ$*back* and *back*$\circ$*forth* are logically equal to the identity? If yes, prove it. If not, can you construct a counterexample for a specific τ?

Task 14 Extend our implementation of the interpreter from Lecture 12 in LAMBDA with quote and case/quote. Because of the structure of the representation (modeling binding in the object language by a function in the meta-language), you cannot always observe the structure of quoted expressions even though our design implies you should be able to.

Implement the examples from Tasks 7, 8, 9, and 10 and execute them on small inputs to examine their behavior to the extent possible.

Task 15 Assume the meta-language (that is, the language of the interpreter) also had quote and pattern matching against values $\mathsf{quote}\ e$. Show how you could modify the meta-interpreter's representation of expressions and evaluation function so that values that had type $\Box\tau$ in the object language (that is, the language being interpreted) always could be observed, or argue that it is still not possible.