Adjoint Functional Programming

NICHOLAS COLTHARP, ANTON LORENZEN, WESLEY NUZZO, and XIAOTIAN ZHOU

These are the lecture notes for Frank Pfenning's course at OPLSS 2024.

1 LECTURE 2: FROM PL TO LOGIC AND BACK (x2?)

We will go back and forth between logic and PL. Logic will inform our PL approach. It is important to be aware of the connection: it is inevitable post-hoc but these features may be confusing to implement without the knowledge.

The rules from last lecture, summarized:

$$\frac{}{x : A \vdash x : A}$$

$$\frac{}{\cdot \vdash () : 1} \qquad \frac{\Delta \vdash e : 1 \quad \Gamma \vdash e' : C}{\Delta, \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ () \Rightarrow\ e' : C}$$

$$\frac{\Delta \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Delta, \Gamma \vdash (e_1,\ e_2) : A \times B} \qquad \frac{\Delta \vdash e : A \times B \quad \Gamma,\ x : A,\ y : B \vdash e' : C}{\Delta, \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ (x, y) \Rightarrow\ e' : C}$$

$$\frac{\Gamma \vdash e : A_k\ (k \in L)}{\Gamma \vdash k(e) : +\{\ l : A_l\ \}_{l \in L}} \qquad \frac{\Delta \vdash e : +\{\ l : A_l\ \}_{l \in L} \quad \Gamma,\ x : A_l \vdash e'_l : C\ (\forall l \in L)}{\Delta, \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ (l(x) \Rightarrow\ e'_l)_{l \in L} : C}$$

We have to use every variable exactly once.

1.1 Reduction relation

In the lecture, we will see the intuition for the theorems, but not include proofs. You can do them yourself if you want.

Next, we will look at a reduction relation for the language. In the dynamic semantics, we want to show type soundness and also something different: we want to show that there is no garbage at the end of the evaluation.

We can set up a close correspondence between the static rules and the dynamic rules. The proof will be easier if the rules are very close. What should then be our runtime interpretation of a judgement such as:

$\Gamma \vdash e : A$

We interpret the expression e as the program getting evaluated, the type A will not be carried around and Γ will be a variable map. We write the variable map as η, and define a judgement for it as $\eta : \Gamma$ with:

$$\frac{}{(\cdot) : (\cdot)} \qquad \frac{\eta : \Gamma \quad \cdot \vdash v : A}{\eta,\ x \mapsto v : (\Gamma,\ x : A)}$$

We say that η is an *environment* and Γ is a *context*.

Under these preconditions, we want to run the program as $\eta \vdash e \hookrightarrow v$. But splitting the context for pairs will create a problem:

$$\frac{? \vdash e_1 \hookrightarrow v_1 \quad ? \vdash e_2 \hookrightarrow v_2}{\eta \vdash (e_1,\ e_2) \hookrightarrow (v_1,\ v_2)}$$

How will we split the environment and fill in the "?" in the rule? We can not split η, because then we would always have to traverse e_1 (at runtime!) to see which the variables are to figure out how to do the split.

1.1.1 The subtractive approach. However, we do not actually have to split η: under the assumption that this type checks, we can put η on both sides, since we know this will be well-formed. One way to do this: use the *subtractive* approach. After e_1 is finished, we get back an η_1 of variables that are unused. We then pass η_1 to the evaluation of e_2 and get back an empty environment. But actually, we have to do this everywhere: e_2 returns an environment η_2, which we return from the rule:

$$\frac{\eta \vdash e_1 \hookrightarrow v_1 \setminus \eta_1 \quad \eta_1 \vdash e_2 \hookrightarrow v_2 \setminus \eta_2}{\eta \vdash (e_1,\ e_2) \hookrightarrow (v_1,\ v_2) \setminus \eta_2}$$

This also corresponds to how the type checker might check that variables are only used once.

- Q: Is there are more logical way to do this? It seems like much gets hidden here?
- A: Yes, taking some shortcuts here. In the subtractive approach we would write the type rule as:

$$\frac{\Gamma \vdash e_1 : A \setminus \Delta \quad \Delta \vdash e_2 : B \setminus \Delta'}{\Gamma \vdash (e_1,\ e_2) : A \times B \setminus \Delta'}$$

Can we write the rest of the rules now? Yes, let's look at variables:

$$\frac{}{\Gamma,\ x : A \vdash x : A \setminus \Gamma}$$

1.1.2 The additive approach. However, it is much better to do things *additive*. Subtractive has an issue: It forces left-to-right evaluation, where you have to look at e_1 before you look at e_2.

In the additive approach: We have a context $\Gamma \vdash e : A \setminus \Omega$ where Ω are the variables that are *actually used*. In contrast, in the subtractive approach we return the *remainder*. The pair rule becomes:

$$\frac{\Gamma \vdash e_1 : A \setminus \Omega_1 \quad \Gamma \vdash e_2 : B \setminus \Omega_2}{\Gamma \vdash (e_1,\ e_2) : A \times B \setminus (\Omega_1,\ \Omega_2)}$$

The result $\Omega_1,\ \Omega_2$ is undefined if there is any overlap between Ω_1 and Ω_2 (eg. if they share a variable). Note that in the rule above, Γ can go into both of the preconditions. That is, because we treat Γ purely as a typing context now, while Ω returns the used variables.

We can relate our new judgement to the old one:

- Soundness: If $\Gamma \vdash e : A \setminus \Omega$ then $\Omega \vdash e : A$ and $\Omega \subseteq \Gamma$.
- Completeness: If $\Omega \vdash e : A$ and $\Omega \subseteq \Gamma$, then $\Gamma \vdash e : A \setminus \Omega$.

The corresponding rule in the semantics is:

$$\frac{\eta \vdash e_1 \hookrightarrow v_1 \setminus \omega_1 \quad \eta \vdash e_2 \hookrightarrow v_2 \setminus \omega_2}{\eta \vdash (e_1,\ e_2) \hookrightarrow (v_1,\ v_2) \setminus (\omega_1,\ \omega_2)}$$

If our expression type-checks then ω_1 and ω_2 will have disjoint domains. If we add non-linear variables, these can occur on both sides.

- Q: Where would our semantics get stuck if a program does not type-check?
- A: First off, not every program that does not type-check will get stuck. But if it gets stuck: this can be because the ω_1 and ω_2 might have overlapping domains, where it would get stuck.
- Q: Is Ω an over-approximation of the variables that are used?

- A: No, we want Ω to be *exactly* the variables used in e.
- Q: If we were to set out to try and prove this, would we need two different versions of the typing rules?
- A: Yes, you would show that for each derivation of one of them, you get a derivation of the other. This is *rule induction*.
- Q: Isn't there an overapproximation in the relation to the old judgement where we write $\Omega \subseteq \Gamma$?
- A: No, since our old judgement is always precise in its typing context.
- Q: Why can Ω and Γ not be the same?
- A: Induction would fail in the pair rule, since even if $\Gamma = \Omega_1, \Omega_2$, then $\Gamma \neq \Omega_1$ or $\Gamma \neq \Omega_2$.

1.1.3 Soundness of additive approach. What do we want the program to satisfy? We have to change our soundness theorem:

Theorem 1. (*Soundness (1)*)
If $\Gamma \vdash e : A \setminus \Omega$ and $\eta : \Gamma$ and $\omega : \Omega$ then $\eta \vdash e \hookrightarrow v \setminus \omega$ (and $v : A$).

- Q: Are the v and the ω existentially quantified in this statement?
- A: Yes, great question! We do not know that e evaluates to v, since that would imply termination. We want to additionally quantify over the v and ω:

Theorem 2. (*Soundness (2)*)
If $\Gamma \vdash e : A \setminus \Omega$ and $\eta : \Gamma$ and $\eta \vdash e \hookrightarrow v \setminus \omega$, then $\omega : \Omega$ (and $v : A$).
Since we defined them in the same way, we can now relate them in the same way. We can prove this theorem with this kind of dynamics.

How do we know that in the end there is no garbage? We write $\eta \vdash e \hookrightarrow v \setminus \eta$ at the toplevel so that everything in η is actually used. We can prove this for the new dynamics. This gives us both soundness and that there will be no garbage in the end.

- Q: Since we do not have recursion in this language, we can always assume that terms terminate, right?
- A: Yes, that is true, but then we can not prove that using induction since termination is a stronger property.
- Q: Don't we use the evaluation as a precondition in the second soundness theorem and thus can not catch stuckness?
- A: Yes, this is no longer type soundness. We will use a different approach next lecture.
- Q: Can you explain what $\eta : \Gamma$ means?
- A: η is a map from variables to values. If the variable has type A in Γ, then the value has type A in η.
- Q: Will you show a small-step semantics?
- A: Not for this language, but next lecture.
- Q: How does the merge of Ω_1, Ω_2 handle top-level variables?
- A: We treat them like non-linear variables.

1.1.4 Affine types. We can play a small game: how can we make this system *affine* so that variable are used at most once? What happens to the merge operator?

At the top-level we have to check for $\Gamma \vdash e : A \setminus \Omega$ that $\Gamma = \Omega$ in the linear version to ensure that everything in Γ is used exactly once. In an affine setting, we can allow $\Omega \subseteq \Gamma$.

- Q: It seems like the typing tree is equal to the evaluation tree?
- A: Yes, since we have not looked at interesting rules. Inviting comments: is the derivation tree the same as the evaluation tree? Student: For matches there is a difference, since we pick a branch of each match in the evaluation tree.

1.1.5 Further typing rules.

$$\frac{}{\cdot \vdash () \hookrightarrow () \setminus \cdot}$$

$$\frac{\eta \vdash e \hookrightarrow (v_1,\ v_2) \setminus \omega \quad \eta,\ x \mapsto v_1,\ y \mapsto v_2 \vdash e' \hookrightarrow v' \setminus (\omega',\ x \mapsto v_1,\ y \mapsto v_2)}{\eta \vdash \text{match } e \text{ with } (x,\ y) \Rightarrow e' \hookrightarrow v' \setminus (\omega,\ \omega')}$$

1.1.6 Top-level definitions. Not much is happening in the computation. A purely linear type system, does not allow you to write many interesting programs: we need top-level definitions. However, studying the whole language is very complicated, so we study the simple case here.

- Q: What is a top-level definition?
- A: For example:

```
plus (x : nat) (y : nat) : nat
plus x y = ...
```

These top-level definitions also have something important to tell us logically. I will tell you at the end of the lecture.

1.2 Back to Logic

We write a natural deduction system, where $\Gamma \vdash A$ says that the assumptions in Γ can prove A. $\Delta,\ \Gamma\ ::= \cdot \mid \Gamma,\ A$. However, each assumption needs to be used exactly once, giving us linear logic.

$$\frac{\Delta \vdash A \quad \Gamma \vdash B}{\Delta, \Gamma \vdash A \otimes B} \qquad \frac{}{A \vdash A}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B} \qquad \frac{\Delta \vdash A \otimes B \quad \Gamma,\ A,\ B \vdash C}{\Delta, \Gamma \vdash C}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B} \qquad \frac{\Delta \vdash A \oplus B \quad \Gamma,\ A \vdash C \quad \Gamma,\ B \vdash C}{\Delta, \Gamma \vdash C}$$

Historically, this was the first formulation of linear logic. From it, people later developed linear type systems. Our operators so far are:

$A,\ B\ ::= 1 \mid A \otimes B \mid A \oplus B$

1.2.1 The operators of linear logic. In linear logic, there is one more operator: "of course A", written $!A$. This allows us to reuse assumptions. We can model an ordinary function $A \Rightarrow B$, as $!A \multimap B$ (we will introduce this formally later). Then we have judgements of the form $\Sigma;\ \Gamma \vdash e : A$, where Σ has reused hypothesis and Γ is linear. The full syntax is:

$A,\ B ::= 1 \mid A \otimes B \mid A \oplus B$
$\quad \mid A \multimap B \mid A \,\&\, B$
$\quad \mid\ !A$

The first row is "positive" and the second row is "negative". The first row can be duplicated by copying, eg. $1 \oplus 1 \vdash (1 \oplus 1) \otimes (1 \oplus 1)$, where we duplicate the boolean $1 \oplus 1$.

- Q: How do you prove this using the logic?
- A: Use the sum-elimination rule, where $1 \oplus 1 \vdash 1 \oplus 1$. Then we have to show that $1 \vdash (1 \oplus) \otimes (1 \oplus 1)$. We use unit elimination so that we have to show $\cdot \vdash (1 \oplus 1) \otimes (1 \oplus 1)$. Then we use the introduction rules to obtain the term.

1.2.2 Linear Functions. Lastly, we will show the rules for $A \multimap B$ and $A \,\&\, B$.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B} \qquad \frac{\Delta \vdash A \multimap B \quad \Gamma \vdash A}{\Delta, \Gamma \vdash B}$$

We will not be able to prove $A \multimap (B \multimap A)$ in general, since B is not used in the result. This is different from the rules we have seen so far, since we just plug the terms together without modifying the contexts.

How do we model this in our linear programming language? We use just one arrow $\to$ and we distinguish regular from linear functions using the arguments.

$$\frac{\Gamma,\ x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B} \qquad \frac{\Delta \vdash e_1 : A \multimap B \quad \Gamma \vdash e_2 : A}{\Delta, \Gamma \vdash e_1\ e_2 : B}$$

You can not pattern-match against a lambda expression. This is a fundamental distinction between the positive and the negative types.

1.2.3 Lazy pairs. You can not match on take a function and match on it. Instead you can only apply it and see what happens. Based on this, what do you think the elements of the $A \,\&\, B$ type should be? Let's look at the logical rule:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}$$

Oh, we duplicate Γ on both sides! How can this be sound?

$$\frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}$$

We can only extract one of them! This is similar to the match-construct for sums. In the programming language:

$$\frac{\Gamma \vdash e_1 : A \quad \Gamma \vdash e_2 : B}{\Gamma \vdash (e_1,\ e_2) : A \,\&\, B} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e.\pi_1 : A} \qquad \frac{\Gamma \vdash e : A \,\&\, B}{\Gamma \vdash e.\pi_2 : B}$$

This is a *lazy pair*: we do not evaluate the components when constructing the pair. We can only evaluate one of them when we deconstruct the pair. We can generalize this to $\&\{\ l : A_l\ \}_{l \in L}$.

$$\frac{\Gamma \vdash e_l : A_l\ (\forall l \in L)}{\Gamma \vdash (l = e_l) : \&\{\ l : A_l\ \}_{l \in L}} \qquad \frac{\Gamma \vdash e : \&\{\ l : A_l\ \}_{l \in L}}{\Gamma \vdash e.k : A_k}$$

We will use the lazy records for object oriented programming.

The main takeaway: In linear logic we have positive and negative types. We can deconstruct the positive types. We can not actually deconstruct the negative types.