Lecture Notes on Elaboration

15-814: Types and Programming Languages
Frank Pfenning

Lecture 11
Tuesday, October 5, 2021

1 Introduction

We have spent a lot of time analyzing and designing the essence of a programming language, starting from first principles. The focus has been on the *statics* (the type system), the *dynamics* (the rules for how to evaluate programs), and understanding the relationship between them in a mathematically rigorous way.

There is, of course, a lot more to a real programming language. At the "front end" there is the *concrete syntax* according to which the program text is parsed. The result of parsing is either some *abstract syntax* or an error message if the program is not well-formed according to the grammar defining its syntax. At the "back end" there are concerns about how a language might be executed efficiently, or *compiled* to machine language so it can run even faster. In this course we will say little about issues of grammar, concrete syntax, parsers or parser generators, because we want to focus on the deeper semantic issues where we have accumulated a lot of knowledge about language design.

In today's lecture we will look at *elaboration*, which is a translation mediating between specific forms of concrete syntax and internal representation in abstract syntax. Elaborating the program allows us to provide some conveniences that make it easy to write and read concise programs without giving up the sound underlying principles we have learned about in this course so far.

We will also look at how declarations and definitions that exist at the top level of the LAMBDA implementation and how they are elaborated.

2 Variadic Sums

Once we know that sums are associative and commutative with unit 0 we can introduce a more general notation that is useful for practical purposes: rather than just using labels $\mathbf{l}$ and $\mathbf{r}$ for a binary sum, we can allow a *finite set I* of *tags* or *label* (think of them as strings) and write

$$(i_1 : \tau_1) + \cdots + (i_n : \tau_n)$$

where each summand is marked with a distinct label i. We also write this in abstract syntax as

$$\sum_{i \in I} (i : \tau_i)$$

The empty type 0 arises from $I = \{\,\}$ and we might define

$$\begin{array}{lcl}
\textit{bool} & = & (\mathbf{true} : 1) + (\mathbf{false} : 1) \\
\textit{option}\ \tau & = & (\mathbf{none} : 1) + (\mathbf{some} : \tau) \\
\textit{order} & = & (\mathbf{less} : 1) + (\mathbf{equal} : 1) + (\mathbf{greater} : 1) \\
\textit{nat} & \cong & (\mathbf{zero} : 1) + (\mathbf{succ} : \textit{nat}) \\
 & = & \mu\alpha.\, (\mathbf{zero} : 1) + (\mathbf{succ} : \alpha) \\[1ex]
\textit{list}\ \tau & \cong & (\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \textit{list}\ \tau) \\
 & = & \mu\alpha.\, (\mathbf{nil} : 1) + (\mathbf{cons} : \tau \times \alpha) \\[1ex]
\textit{bin} & \cong & (\mathbf{b0} : \textit{bin}) + (\mathbf{b1} : \textit{bin}) + (\mathbf{e} : 1) \\
 & = & \mu\alpha.\, (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1)
\end{array}$$

This generalized form of sum also comes with a generalized constructor (allowing any label of a sum) and case expression (requiring a branch for each label of a sum). For example, we might have the following definitions.

$$\begin{array}{lcl}
\textit{bin} & = & \mu\alpha.\, (\mathbf{b0} : \alpha) + (\mathbf{b1} : \alpha) + (\mathbf{e} : 1) \\[1ex]
\textit{b0} & : & \textit{bin} \to \textit{bin} \\
\textit{b1} & : & \textit{bin} \to \textit{bin} \\
\textit{e} & : & \textit{bin} \\[1ex]
\textit{b0} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b0} \cdot x) \\
\textit{b1} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b1} \cdot \cdot x) \\
\textit{e} & = & \mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle) \\[1ex]
\textit{inc} & : & \textit{bin} \to \textit{bin} \\
\textit{inc} & = & \mathsf{fix}\ \textit{inc}.\, \lambda x.\, \mathsf{case}\ (\mathsf{unfold}\ x) \\
 & & \qquad (\, \mathbf{b0} \cdot y \Rightarrow \textit{b1}\ y \\
 & & \qquad \mid \mathbf{b1} \cdot y \Rightarrow \textit{b0}\ (\textit{inc}\ y) \\
 & & \qquad \mid \mathbf{e} \cdot \_ \Rightarrow \textit{b1}\ \textit{e})
\end{array}$$

We now proceed to complete the language with the more general form of sum. At first it might seem this is merely a programming convenience, even if an important one. But then we note, for example, that every value $v : (\mathbf{succ} : \mathit{nat})$ (a unary sum) also has type $v : (\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{nat})$ and realize that variadic sums allow us to exploit the inherent ambiguity in the type of expression to a greater degree than binary sums. We allow a finite set of labels I and write $\sum_{i \in I}(i : \tau_i)$. The case construct for the sums then has a branch for each $i \in I$. Our previous constructs represent a special case, with $\tau_1 + \tau_2 \triangleq \sum_{i \in \{\mathbf{l},\mathbf{r}\}}(i : \tau_i) = (\mathbf{l} : \tau_1) + (\mathbf{r} : \tau_2)$ and $0 \triangleq \sum_{i \in \emptyset}(i : \tau_i)$.

$$
\begin{array}{llll}
\text{Types} & \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid 1 \mid \sum_{i \in I}(i : \tau_i) \mid \mu\alpha.\,\tau \\
\text{Expressions} & e & ::= & x & \text{(variables)} \\
& & \mid & \lambda x.\, e \mid e_1\, e_2 & (\to) \\
& & \mid & \langle e_1, e_2 \rangle \mid \mathsf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') & (\times) \\
& & \mid & \langle\,\rangle \mid \mathsf{case}\ e\ (\langle\,\rangle \Rightarrow e') & (1) \\
& & \mid & i \cdot e \mid \mathsf{case}\ e\ (i \cdot x \Rightarrow e')_{i \in I} & (\textstyle\sum) \\
& & \mid & \mathsf{fold}\ e \mid \mathsf{unfold}\ e & (\mu) \\
& & \mid & f \mid \mathsf{fix}\ f.\, e & \text{(recursion)}
\end{array}
$$

For sums, we have the following generalized statics and dynamics. Key is that we have to check all branches of a case expressions, and all of them have the same type τ'.

$$\frac{(k \in I) \quad \Gamma \vdash e : \tau_k}{\Gamma \vdash k \cdot e : \sum_{i \in I}(i : \tau_i)}\ \mathsf{tp/sum}$$

$$\frac{\Gamma \vdash e : \sum_{i \in I}(i : \tau_i) \quad \Gamma, x_i : \tau_i \vdash e'_i : \tau' \quad (\text{for all } i \in I)}{\Gamma \vdash \mathsf{case}\ e\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} : \tau'}\ \mathsf{tp/cases}$$

$$\frac{e\ \mathit{value}}{i \cdot e\ \mathit{value}}\ \mathsf{val/sum} \qquad \frac{e \mapsto e'}{i \cdot e \mapsto i \cdot e'}\ \mathsf{step/inject}$$

$$\frac{e_0 \mapsto e'_0}{\mathsf{case}\ e_0\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto \mathsf{case}\ e'_0\ (i \cdot x_i \Rightarrow e'_i)_{i \in I}}\ \mathsf{step/cases}_0$$

$$\frac{k \in I \quad v\ \mathit{value}}{\mathsf{case}\ (k \cdot v)\ (i \cdot x_i \Rightarrow e'_i)_{i \in I} \mapsto [v/x_k]e'_k}\ \mathsf{step/cases/inject}$$

3 "Syntactic Sugar"

Except for functions and recursive types, the destructors are of the form $\mathsf{case}\ e\ (\ldots)$. We will now unify these constructs even more, replacing the primitive $\mathsf{unfold}\ e$ by a new one, $\mathsf{case}\ e\ (\mathsf{fold}\ x \Rightarrow e')$. We can then define *Unfold* as a function

$$\begin{array}{lcl} \mathit{Unfold} & : & (\mu\alpha.\,\tau) \to [\mu\alpha.\,\tau/\alpha]\tau \\ \mathit{Unfold} & \triangleq & \lambda x.\,\mathsf{case}\ x\ (\mathsf{fold}\ x \Rightarrow x) \end{array}$$

See Exercise 3 for more on this restructuring of the language. One issue we see is that we cannot give a uniform type to *Unfold* if considered as a function in our language. That's because we cannot express $[\mu\alpha.\,\tau/\alpha]\tau$. For this, we would need first-class *functions from types to types* which are available in a language such as F^ω but not here so far.

So instead we would like to continue to use unfold but *elaborate* it to the appropriate case expression. This is an example of so-called "*syntactic sugar*" which extends the surface syntax without impacting the underlying theory in an essential way.

Another example of syntactic sugar would be the following definitions:

$$\begin{array}{lcl} \mathsf{bool} & \triangleq & (\mathbf{true} : 1) + (\mathbf{false} : 1) \\ \mathsf{true} & \triangleq & \mathbf{true} \cdot \langle\,\rangle \\ \mathsf{false} & \triangleq & \mathbf{false} \cdot \langle\,\rangle \\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \triangleq & \mathsf{case}\ e_1\ (\mathbf{true} \cdot \_ \Rightarrow e_2 \mid \mathbf{false} \cdot \_ \Rightarrow e_3) \end{array}$$

Here, we used another common convention, name we use an underscore (_) in place of a variable name if that variable does not occur in its scope (here, this scope would be e_2 for the first underscore and e_3 for the second. Such a syntactic transformation could take place before or after type checking.

4 Elaborating Definitions

In the LAMBDA implementation, the programmer can make top level declarations and definitions. For example:

```
type bool = ('true : 1) + ('false : 1)

decl true : bool
decl false : bool

defn true = 'true ()
defn false = 'false ()

decl and : bool -> bool -> bool
defn and = \b. \c. case b of ('true u => c | 'false u => false)

eval example1 = and true false
```

The tick marks preceding a identifiers mark them as tags in variadic sums. In the abstract form we have shown them in bold instead.

So far we have only formalized types and expressions and the relations between them, but not definitions of types (`type`), declarations of types for names to be defined (`decl`), and definitions of expression names (`defn`). Our goal now is to formalize these and at the same time define how they should behave. First, a program P is a sequence of definitions D. For those, we have three possibilities.

$$\begin{array}{llcl} \text{Programs} & P & ::= & \cdot \mid D \, ; P \\ \text{Definitions} & D & ::= & \mathsf{type}\ t = \tau \\ & & \mid & \mathsf{defn}\ f : \tau = e \\ & & \mid & \mathsf{eval}\ x = e \end{array}$$

Here we have used t for a defined type name, in contrast with α, β, ... which generally stand for bound type variables. The language of types then as to be generalized to include such type names. Similarly, we have used f to stand for a defined expression name. Because it may stand for an expression (rather than a value), we wrote f instead of x, since variables defined as fixed points also stand for general expressions. Even though the notation suggests that f should be a function (and this is often the case) there is no such formal requirement.

With respect to the concrete syntax, we have also combined declarations and definitions so simplify our presentation. It would be straightforward but tedious to separate them.

Elaboration results in a *signature* Σ with definitions for types, expressions, and values.

$$\text{Signature} \quad \Sigma \quad ::= \quad \cdot \mid \Sigma, t = \tau \mid \Sigma, f : \tau = e \mid \Sigma, x : \tau = v$$

We process a program from left to right, building up the signature Σ. Later declarations and definitions are processed with respect to the signature Σ accumulated up to that point. At the end of elaboration we return the final accumulated signature Σ_F. So our judgment is

$$\Sigma \vdash P \rightsquigarrow \Sigma_F$$

The first rule just says that when P is empty, we just return the accumulated signature.

$$\frac{}{\Sigma \vdash (\cdot) \rightsquigarrow \Sigma}\ \textsf{prog/empty}$$

When we see a type definition, we need to make sure the type is a valid type, the name hasn't already been used (since we disallow shadowing), and then add the definition to the accumulated signature.

$$\frac{t \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma \tau\ \mathit{type} \quad \Sigma, t = \tau \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{type}\ t = \tau\ ;\ P \rightsquigarrow \Sigma_F}\ \textsf{prog/tpdef}$$

Here we realize that we need to pass in the accumulated signature Σ into the type-checker because τ may contain type names previously defined. An often-used convention is to subscript the turnstile. We then need a further rule for checking validity of types:

$$\frac{t = \tau \in \Sigma}{\Gamma \vdash_\Sigma t\ \mathit{type}}\ \textsf{tp/name}$$

Furthermore, we would have to change all the other rules to add the subscript Σ to all turnstiles. Since Σ never changes and only used in one rule, this step is often elided.

Next, when we see a definition $\textsf{defn}\ f : \tau = e$ we need to check that f isn't used yet, τ is a valid type, and e is a valid expression of type τ.

$$\frac{f \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma \tau\ \mathit{type} \quad \cdot \vdash_\Sigma e : \tau \quad \Sigma, f : \tau = e \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{defn}\ f : \tau = e\ ;\ P \rightsquigarrow \Sigma_F}\ \textsf{prog/expdef}$$

As in the previous rule, we need to generalize the typing judgment to carry a signature (which is often omitted).

In the last kind of definition we need to evaluate an expression. Here, we have purposely omitted a type for x, anticipating that we may be able to synthesize it from the expression. In any case, e must have some type, which then becomes the type of x in the accumulated signature.

$$\frac{x \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma e : \tau \quad e \mapsto^* v \quad v\ \mathit{value} \quad \Sigma, x : \tau = v \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{eval}\ x = e\ ;\ P \rightsquigarrow \Sigma_F}\ \textsf{prog/valdef}$$

Note that if e does not type-check, or if e has no value, then this rule is not applicable and elaboration cannot produce a final signature. In the implementation, the failure to type-check produces an error, and the failure to evaluate causes the implementation to hang.

Now we can investigate when a signature itself is valid (that is, well-formed) with a judgment such as Σ *valid*. We would then want to prove that if Σ *valid* and $\Sigma \vdash P \rightsquigarrow \Sigma_F$ then Σ_F *valid*. You can pursue this idea in Exercise 4.

5 Algorithmic Interpretations of Rules

One of the standard techniques for describing *algorithms* in the theory of programming languages is to interpret inference rules themselves algorithmically. The idea is that we are given a (potentially incomplete) judgment to derive. We nondeterministically construct a derivation bottom-up, using the inference rules to reduce the goal to subgoals. If no rule is applicable we fail; if the derivation is complete we succeed.

For this method to be effective we interpret the constituents of the judgments as either *inputs* (denoted by $+$) or *outputs* (denoted by $-$). For elaboration, for example, we write

$$\Sigma^+ \vdash P^+ \rightsquigarrow \Sigma_F^-$$

that is, we assume the signature Σ and the program P to be given, and elaboration should construct Σ_F. These modes are then reflected in the correctness theorem for elaboration, stating that if Σ is valid and elaboration succeeds, then Σ_F is valid (see Exercise 4).

When assigning modes to a judgment in the expectation of providing an algorithmic interpretation of its rules, we have to be careful. In particular, we have to verify that the judgment is *well-moded* in the following sense:

1. Given the inputs for a conclusion, we have enough information to know the inputs for the premises.

2. Given the inputs for a conclusion and the outputs of the premises, we have enough information to construct the output of the conclusion.

Let's consider some simple examples.

We declare $\Gamma^+ \vdash_{\Sigma^+} \tau^+$ *type*, that is, we simply *check* that a given type τ is well-typed in a given context Γ and signature Σ. The outcome of the attempt to construct a derivation of such a judgment is either *success* or *failure* and it acts as a decision procedure. We show an example how we check that the rules are well-moded.

$$\frac{\Gamma \vdash_\Sigma \tau_1 \; \mathit{type} \quad \Gamma \vdash_\Sigma \tau_2 \; \mathit{type}}{\Gamma \vdash_\Sigma \tau_1 \to \tau_2 \; \mathit{type}} \; \textsf{tp/arrow}$$

According to the mode, the whole conclusion is given to us, so we know Γ, Σ, and $\tau_1 \to \tau_2$. That's enough information to know Γ, Σ and τ_1 for the first premise, and Γ, Σ and τ_2 for the second premise.

If we try to give the typing judgment the mode $\Gamma^+ \vdash_{\Sigma^+} e^+ : \tau^+$ (we just check that the expression e has type τ), then the following rule is *not* well-moded:

$$\frac{\Gamma \vdash_\Sigma e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash_\Sigma e_2 : \tau_2}{\Gamma \vdash_\Sigma e_1 \, e_2 : \tau_1} \; \textsf{tp/app}$$

Since all components are inputs, we know Γ, Σ, e_1, e_2, and τ_1 from the conclusion, but we do not know τ_2 which is necessary for both premises. So our algorithm for constructing a derivation would get stuck here.

Let's analyze our rules for elaborating a signature from this perspective. Recall that we would like to have the mode $\Sigma^+ \vdash P^+ \rightsquigarrow \Sigma_F^-$.

$$\frac{}{\Sigma \vdash (\cdot) \rightsquigarrow \Sigma} \; \textsf{prog/empty}$$

Since Σ on the left is given as input, we can construct Σ on the right as output, so this rule is well-moded. On to the next rule.

$$\frac{t \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma \tau \; \mathit{type} \quad \Sigma, t = \tau \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{type} \; t = \tau \; ; P \rightsquigarrow \Sigma_F} \; \textsf{prog/tpdef}$$

From the conclusion we know Σ, t, τ, and P. That's sufficient to check $t \notin \mathrm{dom}(\Sigma)$ and $\cdot \vdash_\Sigma \tau \; \mathit{type}$ since all inputs to this judgment are known. Finally, we can recurse into the last premise because Σ, t, τ and P are known. We obtain the output Σ_F which we have to return as output in the conclusion. So this rule is also well-moded.

Next, definition of an expression name.

$$\frac{f \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma \tau \; \mathit{type} \quad \cdot \vdash_\Sigma e : \tau \quad \Sigma, f : \tau = e \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{defn} \; f : \tau = e \; ; P \rightsquigarrow \Sigma_F} \; \textsf{prog/expdef}$$

1. From the conclusion we know Σ, f, τ, e and P.

2. With that information, we can check the first premise $f \notin \mathrm{dom}(\Sigma)$

3. We can also check the second premise $\cdot \vdash_\Sigma \tau$ *type*

4. We get stuck with the third premise, $\cdot \vdash_\Sigma e : \tau$ because we have already seen by example above that $\Gamma^+ \vdash_{\Sigma^+} e^+ : \tau^+$ is *not* well-moded!

5. But assuming we could somehow get this to work, we could recurse on the fourth premise (Σ, f, τ, e, and P are all known) and we would obtain Σ_F

6. That's sufficient to construct the output Σ_F in the conclusion.

So the prog/expdef rule is *not* well-moded!

So what we would need here would be a judgment

$$\Gamma^+ \vdash_{\Sigma^+} e^+ \Leftarrow \tau^+$$

that *checks* e against a given type τ. We'll design such a judgment in the next section.

But let's complete our task. It remains to check the rule

$$\frac{x \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma e : \tau \quad e \mapsto^* v \quad v\ \mathit{value} \quad \Sigma, x : \tau = v \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \mathsf{eval}\ x = e\ ;\ P \rightsquigarrow \Sigma_F}\ \mathsf{prog/valdef}$$

We reason systematically as before.

1. From the conclusion we know Σ, x, e, and P.

2. This let's us check the first premise, $x \notin \mathrm{dom}(\Sigma)$

3. We get stuck in the second premise, because $\Gamma^+ \vdash_{\Sigma^+} e^+ : \tau^-$ is also not a valid mode for the typing judgment. For example, $\cdot \vdash_{(\cdot)} \lambda x.\, x : ?\tau$ would have to yield infinitely many $?\tau$ because $\lambda x.\, x$ has infinitely many types. Looking at the rules, we see that tp/lam would not be well-moded.

4. Assuming we could fix this somehow, the next premise $e^+ \mapsto^* v^-$ is well-moded (see Exercise 7). This would give use v^- as an output.

5. Then, the next premise (which has mode v^+ *value*) would be well-moded since we now know v.

6. With all this information, the last premise is well-moded and gives us Σ_F as output.

7. Therefore, the output Σ_F in the conclusion is known.

So, again, this rule is *not* well moded. What we would need is a judgment $\Gamma^+ \vdash_{\Sigma^+} e^+ \Rightarrow \tau^-$ that *synthesizes* a τ, given Γ, Σ, and e.

This rule is also illustrate that we need to construct the derivation of $e^+ \mapsto^* v^-$ before testing v^+ *value*. A general convention is that we try to construct derivations for the premises from left to right.

Summary: if we had judgments $\Gamma^+ \vdash_{\Sigma^+} e^+ \Leftarrow \tau^+$ and $\Gamma^+ \vdash_{\Sigma^+} e^+ \Rightarrow \tau^-$ then elaboration using the following rules would be well-moded and could be interpreted algorithmically.

$$\frac{}{\Sigma \vdash (\cdot) \rightsquigarrow \Sigma}\ \textsf{prog/empty}$$

$$\frac{t \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma \tau\ \textit{type} \quad \Sigma, t = \tau \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{type}\ t = \tau\ ;\ P \rightsquigarrow \Sigma_F}\ \textsf{prog/tpdef}$$

$$\frac{f \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma \tau\ \textit{type} \quad \cdot \vdash_\Sigma e \Leftarrow \tau \quad \Sigma, f : \tau = e \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{defn}\ f : \tau = e\ ;\ P \rightsquigarrow \Sigma_F}\ \textsf{prog/expdef}$$

$$\frac{x \notin \mathrm{dom}(\Sigma) \quad \cdot \vdash_\Sigma e \Rightarrow \tau \quad e \mapsto^* v \quad v\ \textit{value} \quad \Sigma, x : \tau = v \vdash P \rightsquigarrow \Sigma_F}{\Sigma \vdash \textsf{eval}\ x = e\ ;\ P \rightsquigarrow \Sigma_F}\ \textsf{prog/valdef}$$

The last rule illustrates something else. Besides the modes, there are certain presuppositions we would like to be satisfied. For example, we never evaluate an expression e unless it is closed and well-typed. We only know this if we derive $\cdot \vdash_\Sigma e \Rightarrow \tau$ *before* we try to derive $e \mapsto^* v$, even though we know the necessary input (e). We have to be careful about such hidden dependencies when writing and interpreting the rules, so understanding *presuppositions* is also important in our interpretation of the judgments. In fact, presuppositions and modes go hand in hand.

6 Bidirectional Type-Checking

In the previous section we saw that for elaboration to work as intended, we need to define two new judgments, $\Gamma^+ \vdash_{\Sigma^+} e^+ \Rightarrow \tau^-$ and $\Gamma^+ \vdash_{\Sigma^+} e^+ \Leftarrow \tau^+$. We refer to them as *type synthesis* and *type checking*, respectively. The requirements here should be clear:

1. Both judgments are well-moded and therefore describe an algorithm.
2. Both judgments entail that $\Gamma \vdash_\Sigma e : \tau$. We call this the *soundness* of bidirectional type-checking.

3. We may also wish that if $\Gamma \vdash_\Sigma e : \tau$ then one of the two judgments holds. As we will see, this is not quite possible and we need to make some modifications to obtain a related property we call *completeness*.

Bidirectional type-checking [DK19] is quite robust in the sense that it applies to many languages and constructs where other approaches (such as *type inference*) no longer work.

For the sake of conciseness, we restrict ourselves to the fragment relevant to function types, $\tau_1 \to \tau_2$. Reasoning through the rules (which we did in lecture) we can categorize each of the typing rules to synthesize or check. We obtain (omitting Σ for brevity)

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}\ \mathsf{syn/var} \qquad \frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.\, e \Leftarrow \tau_1 \to \tau_2}\ \mathsf{chk/lam}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_1}\ \mathsf{syn/app}$$

The last rule syn/app is the most interesting one since it requires both judgments. It is important that the first premise be derived first, because this gives us τ_2 which is a necessary input to the second premise. It also give use τ_1 which is necessary for the output in the conclusion.

These rules are almost trivially sound because we obtain the ordinary typing rules if we replace '$\Rightarrow$' and '$\Leftarrow$' with ':'. However the rules are *incomplete* in a somewhat surprising manner. For example, we will fail to prove that $\cdot \vdash (\lambda x.\, x) \Leftarrow 1 \to 1$:

$$\frac{\begin{array}{c}??\\ x : 1 \vdash x \Leftarrow 1\end{array}}{\cdot \vdash (\lambda x.\, x) \Leftarrow 1 \to 1}\ \mathsf{chk/lam}$$

We see there is no rule with a conclusion matching the premise, so the algorithmic interpretation would fail and say that the judgment does not hold.

What we need, of course, is one more rule that connects the checking and synthesis judgment. We can check an expression e against τ if e synthesized τ' and $\tau' = \tau$.

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad (\tau' = \tau)}{\Gamma \vdash e \Leftarrow \tau}\ \mathsf{chk/syn}$$

Is this well-moded? Let's reason it out

1. We know Γ, e, and τ from the conclusion (since $\Gamma^+ \vdash e^+ \Leftarrow \tau^+$)

2. That's sufficient for the premise, since we know Γ and e. This will return an output τ' (since $\Gamma^+ \vdash e^+ \Rightarrow \tau^-$).

3. Now we have both τ and τ' and we can compare them for equality.

With this additional rule we can complete the typing for the identity function.

$$\dfrac{\dfrac{\dfrac{}{x : 1 \vdash x \Rightarrow 1}\,\mathsf{syn/var} \qquad (1 = 1)}{x : 1 \vdash x \Leftarrow 1}\,\mathsf{chk/syn}}{\cdot \vdash (\lambda x.\, x) \Leftarrow 1 \to 1}\,\mathsf{chk/lam}$$

Having become suspicious by now regarding completeness, we try to type $(\lambda x.\, x)\,\langle\,\rangle$. Since it is an application, we try to synthesize a type:

$$\dfrac{\dfrac{??}{\cdot \vdash \lambda x.\, x \Rightarrow ?\tau_2 \to ?\tau} \qquad \dfrac{??}{\cdot \vdash \langle\,\rangle \Leftarrow ?\tau_2}}{\cdot \vdash (\lambda x.\, x)\,\langle\,\rangle \Rightarrow ?\tau}\,\mathsf{syn/app}$$

Here we get stuck in first premise, because there is no rule whose conclusion matches this judgment: λ-abstraction are only checked. Therefore we do not know $?\tau_2$ and cannot even start deriving the second premise. So the algorithmic interpretation of our rules would once again indicate failure.

For λ-abstractions we can adopt a somewhat ad hoc solution: we allow annotation of variables with their intended types! The we would have one additional rule

$$\dfrac{\Gamma, x : \tau \vdash e \Rightarrow \sigma}{\Gamma \vdash (\lambda x{:}\tau.\, e) \Rightarrow \tau \to \sigma}\,\mathsf{syn/lam}$$

There is a more general solution (see Exercise 9), but for the functional fragment the above is sufficient to obtain completeness. The property then says that if $\Gamma \vdash e : \tau$ then there is an annotated version e' of e such that $\Gamma \vdash e' \Rightarrow \tau$ (and therefore also $\Gamma \vdash e' \Leftarrow \tau$ by rule chk/syn).

We can ask when annotations are needed and it turns out that it is exactly the *normal forms* that require no annotations. This is explored in Exercise 8.

Here is the summary of the rules.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau}\ \textsf{syn/var} \qquad \frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.\, e \Leftarrow \tau_1 \to \tau_2}\ \textsf{chk/lam}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \tau_1}\ \textsf{syn/app} \qquad \frac{\Gamma \vdash e \Rightarrow \tau' \quad (\tau' = \tau)}{\Gamma \vdash e \Leftarrow \tau}\ \textsf{chk/syn}$$

$$\frac{\Gamma, x : \tau \vdash e \Rightarrow \sigma}{\Gamma \vdash (\lambda x{:}\tau.\, e) \Rightarrow \tau \to \sigma}\ \textsf{syn/lam}$$

Exercises

Exercise 1 It is often intuitive to define types in a mutually recursive way. As a simple example, consider how to define binary numbers in *standard form*, that is, not allowing leading zeros. We define binary numbers in standard form (*std*) mutually recursively with strictly positive binary numbers (*pos*).

$$\begin{array}{rcl} \mathit{std} & \cong & (\mathbf{e} : 1) + (\mathbf{b0} : \mathit{pos}) + (\mathbf{b1} : \mathit{std}) \\ \mathit{pos} & \cong & (\mathbf{b0} : \mathit{pos}) + (\mathbf{b1} : \mathit{std}) \end{array}$$

(i) Using only *std*, *pos*, and function types formed from them, give all types of *e*, *b0*, and *b1* defined as follows:

$$\begin{array}{rcl} \mathit{b0} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b0} \cdot x) \\ \mathit{b1} & = & \lambda x.\, \mathsf{fold}\ (\mathbf{b1} \cdot x) \\ \mathit{e} & = & \mathsf{fold}\ (\mathbf{e} \cdot \langle\,\rangle) \end{array}$$

(ii) Define the types *std* and *pos* explicitly in our language using the μ type former so that the isomorphisms stated above hold.

(iii) Does the function *inc* from Section 2 have type $\mathit{std} \to \mathit{pos}$? You may use all the types for *b0*, *b1* and *e* you derived in part (i). Then either explain where the typing fails or indicate that it has that type. You do not need to write out a typing derivation.

(iv) Write a function $\mathit{pred} : \mathit{pos} \to \mathit{std}$ that returns the predecessor of a strictly positive binary number. You must make sure your function is correctly typed, where again you may use all the types from part (i).

Exercise 2 It is often convenient to define functions by mutual recursion. As a simple example, consider the following two functions on bit strings determining if it has *even or odd parity*.

$$\begin{array}{lcl}
\mathit{bin} & \cong & (\mathbf{e} : 1) + (\mathbf{b0} : \mathit{bin}) + (\mathbf{b1} : \mathit{bin}) \\[1ex]
\mathit{even} & : & \mathit{bin} \to \mathit{bool} \\
\mathit{odd} & : & \mathit{bin} \to \mathit{bool} \\[1ex]
\mathit{even}\ \mathit{e} & = & \mathit{true} \\
\mathit{even}\ (\mathit{b0}\ x) & = & \mathit{even}\ x \\
\mathit{even}\ (\mathit{b1}\ x) & = & \mathit{odd}\ x \\[1ex]
\mathit{odd}\ \mathit{e} & = & \mathit{false} \\
\mathit{odd}\ (\mathit{b0}\ x) & = & \mathit{odd}\ x \\
\mathit{odd}\ (\mathit{b1}\ x) & = & \mathit{even}\ x
\end{array}$$

(i) Write a function *parity* with a single fixed point constructor and use it to define *even* and *odd*. Also, state the type of your *parity* function explicitly.

(ii) More generally, our simple recipe for implementing a recursively specified function using the fixed point constructor in our call-by-value language goes from the specification

$$\begin{array}{lcl}
f & : & \tau_1 \to \tau_2 \\
f\,x & = & h\,f\,x
\end{array}$$

to the implementation

$$f \;=\; \mathsf{fix}\, g.\, \lambda x.\, h\,g\,x$$

It is easy to misread these, so remember that by our syntactic convention, $h\,f\,x$ stands for $(h\,f)\,x$ and similarly for $h\,g\,x$. Give the type of h and show by calculation that f satisfies the given specification by considering $f\,v$ for an arbitrary value v of type τ_1.

(iii) A more general, *mutually recursive* specification would be

$$\begin{array}{lcl}
f & : & \tau_1 \to \tau_2 \\
g & : & \sigma_1 \to \sigma_2 \\
f\,x & = & h_1\,f\,g\,x \\
g\,y & = & h_2\,f\,g\,y
\end{array}$$

Give the types of h_1 and h_2.

(iv) Show how to explicitly define f and g in our language from h_1 and h_2 using the fixed point constructor and verify its correctness by calculation as in part (ii). You may use any other types in the language introduced so far (pairs, unit, sums, polymorphic, and recursive types).

Exercise 3 In the language where the primitive unfold has been replaced by pattern matching, we can define the following two functions:

$$\begin{array}{lcl} \mathit{Unfold} & : & \mu\alpha.\,\tau \to [\mu\alpha.\,\tau/\alpha]\tau \\ \mathit{Unfold} & = & \lambda x.\,\mathsf{case}\ x\ (\mathsf{fold}\ x \Rightarrow x) \\[1ex] \mathit{Fold} & : & [\mu\alpha.\,\tau/\alpha]\tau \to \mu\alpha.\,\tau \\ \mathit{Fold} & = & \lambda x.\,\mathsf{fold}\ x \end{array}$$

Prove that *Fold* and *Unfold* are witnessing a type isomorphism.

Exercise 4 (Validity of Elaboration) Based on the rules for elaboration in Section 4, define a judgment Σ *valid*. Then prove the following theorem:

Theorem 1 *If Σ valid and $\Sigma \vdash P \rightsquigarrow \Sigma_F$ then Σ_F valid.*

Exercise 5 (Internalizing Definitions) We can *internalize* definitions as part of the core language. Specifically, we add

$$\begin{array}{lccl} \text{Expressions} & e & ::= & \mathsf{exp}\ f : \tau = e\ \mathsf{in}\ e' \\ & & \mid & \mathsf{val}\ x = e\ \mathsf{in}\ e' \\ & & \mid & \ldots \end{array}$$

where $\mathsf{exp}\ f : \tau = e\ \mathsf{in}\ e'$ internalizes the definition of an expression variable f with scope e', and $\mathsf{val}\ x = e\ \mathsf{in}\ e'$ internalizes evaluation of e and binding x to the resulting value with scope e'.

Extend the rules for typing and evaluation of expressions to account for the two new constructs. Our key language properties, namely preservation, progress, finality of values, and determinacy should hold, but you do not need to prove them.

Exercise 6 (Mutually Dependent Definitions) In this exercise we explore mutually dependent definitions and the *phase distinction*. This avoids some of the code duplication and complexities from Exercise 2.

We would like all definitions (types, expressions, and values) in a signature to be able to refer to each other without requiring any particular ordering. We could then write, for example

type $\mathit{nat} = (\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{nat})$
type $\mathit{even} = (\mathbf{zero} : 1) + (\mathbf{succ} : \mathit{odd})$
type $\mathit{odd} = () + (\mathbf{succ} : \mathit{even})$

defn $\mathit{halfe} = \lambda n.\,$case $n\ \ (\,\mathbf{zero} \cdot \_ \Rightarrow \mathbf{zero} \cdot \langle\,\rangle$
$\qquad\qquad\qquad\qquad\quad \mid \mathbf{succ} \cdot n' \Rightarrow \mathbf{succ} \cdot (\mathit{halfo}\ n')\,)$

defn $\mathit{halfo} = \lambda n.\,$case $n\ \ (\,\mathbf{succ} \cdot n' \Rightarrow \mathit{halfe}\ n'\,)$

eval $\mathit{one}\ \ = \mathit{halfo}\ (\mathbf{succ} \cdot \mathbf{succ} \cdot \mathbf{succ} \cdot \mathbf{zero} \cdot \langle\,\rangle)$

decl $\mathit{halfe} : \mathit{even} \to \mathit{nat}$
decl $\mathit{halfo} : \mathit{odd} \to \mathit{nat}$

This would require either the elaboration to insert explicit fixed point expressions and types, or the expressions and types can reference each other directly. For this exercise we assume the latter.

(i) Write a three-phase elaboration algorithm that proceeds as follows:

   (1) In the first phase, we check all type definitions and type declarations for validity.

   (2) In the second phase, we check all the expressions to be well-typed.

   (3) In the third phase we evaluate expressions as part of the eval definitions.

(ii) Define the validity condition for the signature that is the ultimate outcome of the third phase (assuming elaboration succeeds). This should come in the form of inference rules that are as simple as possible.

(iii) State the theorems that characterize the assumptions and outcome of each pass so that the next pass can proceed safely. If needed, define auxiliary judgments. The ultimate outcome should be a signature as before, except that the signature entries now can mutually refer to each other.

Exercise 7 (Modes for the Stepping Judgment) Verify the following mode assignments. You only need to show the cases for function $\tau_1 \to \tau_2$ and pairs $\tau_1 \times \tau_2$.

(i) $e^+ \mapsto e'^-$.

(ii) $e^+ \mapsto^* e'^-$.

(iii) e^+ *value*

Exercise 8 (Annotation-Free Expressions) Prove the following theorems for annotation-free expressions e.

(i) If $\Gamma \vdash e : \tau$ and e *normal*, then $\Gamma \vdash e \Leftarrow \tau$

(ii) If $\Gamma \vdash e : \tau$ and e *neutral*, then $\Gamma \vdash e \Rightarrow \tau$

Furthermore, if e is annotation-free then

(iii) If $\Gamma \vdash e \Leftarrow \tau$ then e *normal* (and $\Gamma \vdash e : \tau$)

(iv) If $\Gamma \vdash e \Rightarrow \tau$ then e *neutral* (and $\Gamma \vdash e : \tau$)

Exercise 9 A general solution to the fact that only normal forms can be typed without annotations is to introduce a new general piece of syntax, $(e : \tau)$ instead of one tailored to functions only. Nevertheless, you may restrict yourself to constructs related to function types in the answers below.

(i) Add zero, one, or more rules for checking and synthesis for the new form of expression $(e : \tau)$

(ii) Prove that your new version is *sound* with respect to typing.

(iii) Prove that if $\Gamma \vdash e : \tau$ then there is an annotated version e' of e such that $\Gamma \vdash e' \Leftarrow \tau$. This is a form of the *completeness* for bidirectional typing.

References

[DK19] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *CoRR*, abs/1908.05839, 2019.