Lecture Notes on Bidirectional Type-Checking

15-814: Types and Programming Languages
Frank Pfenning

Lecture 9
Tue Sep 23, 2025

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. This is followed by type checking, and we formulate the algorithm used by the LAMBDA implementation in the form of two judgments. This is a prototypical example of how a set of inference rules can be interpreted as an algorithm and then implemented.

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

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

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.

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

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 $0 \triangleq \sum_{i \in \emptyset}(i : \tau_i)$.

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

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[1]

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.

[1] not discussed in lecture

4 Elaborating Definitions

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

```
1  type bool = ('true : 1) + ('false : 1)
2
3  decl true : bool
4  decl false : bool
5
6  defn true = 'true ()
7  defn false = 'false ()
8
9  decl and : bool -> bool -> bool
10 defn and = \b. \c. case b of ('true u => c | 'false u => false)
11
12 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.

In these example, the expressions do not contain any types. This raises the question of how the implementation performs type checking. We see from this example that there are two distinct but likely related problems to be solved. The first (for the definitions of `true`, `false`, and `and`) is that we are given a type via a declaration and we have to check the definition against this type. For example, we can see that the types of `b` and `c` in the definition of `and` should be `bool`. The second is that the definition of `example1` does not require a type. Instead, the implementation infers that it must be `bool` from the type of `and`.

What emerges is that some information is given by the programmer, and that this information is propagated throughout the expression to verify that all expressions are well-typed. The particular algorithm used by LAMBDA is *bidirectional type-checking*, which we discuss in the next section. It represents an example of a general technique to interpret judgments as describing algorithms.

5 Bidirectional Type-Checking

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.

This does not work very well in general. Let's consider the typing rule for function application.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{tp/app}$$

Imagine we are trying to construct a typing derivation for a *given* Γ, $e_1\, e_2$, and τ_1. In order to construct such derivation we should construct derivations for both premises. But we can't ask the same question for the first or second premise since we don't know what τ_2 is!

The solution has already been suggested in the previous section: sometimes we can check an expression against a given type (as for **`decl`** and **`defn`**), and sometimes we can synthesize the type

of an expression (as for **`eval`**). This means that single typing judgment bifurcates:

Declarative	Algorithmic
e has type τ $\quad \Gamma \vdash e : \tau$	$\Delta \vdash e \Leftarrow \tau$ $\quad e$ checks against τ $\Delta \vdash e \Rightarrow \tau$ $\quad e$ synthesizes τ

Here, $\Delta ::= \Delta, x \Rightarrow \tau \mid \cdot$ expresses that each variable in the context synthesizes its type, that is, there are no unknowns.

We say a judgment is *declarative* if the rules do not give rise to an obvious interpretation as describing an algorithm. Instead, they just *define* a judgment by a set of rules. If it does, we call it *algorithmic*. An algorithmic judgment is still defined by its inference rules, and it must satisfy some additional condition so we can turn it into an implementation.

There are various forms of conditions that allow us to interpret a judgment algorithmically. The simplest one is that it is *well-moded*. We interpret the constituents of the judgments as either *inputs* (denoted by $+$) or *outputs* (denoted by $-$). For the two judgments above we might write

$$\Delta^+ \vdash e^+ \Leftarrow \tau^+ \quad \Delta, e, \text{ and } \tau \text{ given}$$
$$\Delta^+ \vdash e^+ \Rightarrow \tau^- \quad \Delta, e \text{ given, } \tau \text{ computed}$$

Next, we need to construct rules that actually satisfy the given modes. This is not always possible: we cannot assign arbitrary modes to some judgment to hope it becomes an algorithm. Luckily, in this case it does. This means that at the level of an implementation of we would expect two functions:

type $\mathsf{tp} = \ldots$ % type of types
type $\mathsf{exp} = \ldots$ % type of expressions
type $\mathsf{ctx} = \ldots$ % type of contexts

decl *check* : $\mathsf{ctx} \to \mathsf{exp} \to \mathsf{tp} \to 1 + 1$
decl *synth* : $\mathsf{ctx} \to \mathsf{exp} \to 1 + \mathsf{tp}$

The first function, *check*, would return a Boolean, say, true if term has the proposed type and false otherwise. The second function, *synth* would return a (tagged) type if it has one and a (tagged) unit otherwise.

But how do we ascertain that our judgments admit such an implementation? We have to verify that the judgment is *well-moded* in the following sense:

1. For each rule, given the inputs for its conclusion, we have enough information to know the inputs to its premises.
2. For each rule, given the inputs for its conclusion and the outputs of the premises, we have enough information to construct the output of the conclusion.

We will have occasion to generalize definition slightly in short order. The requirements for the type checking and synthesis judgment should now be clear:

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

Here, Γ (with ":") is related to Δ (with "$\Rightarrow$") in the obvious way. In lecture, we did not even formally distinguish them.

Bidirectional type-checking [Dunfield and Krishnaswami, 2022] 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 begin with the fragment relevant to function types, $\tau_1 \to \tau_2$, and later add products. Reasoning through the rules we can categorize each of the typing rules to synthesize or check. Let's start with the rule tp/lam.

$$\frac{\Delta, x \Rightarrow \tau_1 \vdash e \Leftarrow \tau_2}{\Delta \vdash \lambda x.\, e \Leftarrow \tau_1 \to \tau_2}\ \mathsf{chk/lam}$$

Here is how we reason:

1. We are given Δ, $\lambda x.\, e$ and $\tau_1 \to \tau_2$.
2. That's sufficient to know $(\Delta, x \Rightarrow \tau_1)$ and e and τ_2.
3. So we can ask if there is a derivation of the premise.
4. If so, there is one for the conclusion.
5. If not, there is none for the conclusion since there is only one rule for λ-abstraction (the rules are *syntax-directed*)

The rules for variables is straightforwardly an instance of synthesis, but the rule for application is tricky.

$$\frac{x \Rightarrow \tau \in \Delta}{\Delta \vdash x \Rightarrow \tau}\ \mathsf{syn/var} \qquad \frac{\Delta \vdash e_1 \Rightarrow \tau_2 \to \tau_1 \quad \Delta \vdash e_2 \Leftarrow \tau_2}{\Delta \vdash e_1\ e_2 \Rightarrow \tau_1}\ \mathsf{syn/app}$$

The last rule syn/app is the most interesting one since it requires both judgments. We reason as follows

1. We are given Δ and $e_1\ e_2$. We want to compute τ_2 if it exists or fail if it doesn't.
2. Since we know Δ and e_1 we can apply synthesis to the first premise. If either e_1 does not synthesize a type, or if it does not have the form of a function type, we fail.
3. If it has a function type we learn τ_2 and τ_1. This means we can ask if e_2 checks against τ_2.
4. If not, we fail.
5. If yes, we succeed and return τ_1 and the type of the application.

In this example we see that we need to ask questions for the premises in the left-to-right order. We couldn't check e_2 against τ_2 because we don't know τ_2 unless we have read it off a (successful) construction of a derivation for the first premise.

So we modify our general mode-checking procedure as follows:

1. For each rule, given the inputs for its conclusion, we have enough information to know the inputs to its first premise.
2. Go through each premise left-to-right keeping track of all the known variables. For each premise in turn, all the inputs must be known by the time we reach it, and all the outputs can then be assumed to be known.

3. At the end, we need to ascertain that all the outputs of the conclusion are known.

To make this procedure rigorous, we would need a formal, unambiguous notation for inference rules. This is precisely what a *logical framework* accomplishes. The Twelf system [Pfenning and Schürmann, 1999] implements the logical framework LF [Harper et al., 1993], including a mode checker [Rohwedder and Pfenning, 1996, Anderson and Pfenning, 2004].

The bidirectional 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 $\cdot \vdash (\lambda x.\, x) \Leftarrow 1 \to 1$:

$$\dfrac{\overset{??}{x \Rightarrow 1 \vdash x \Leftarrow 1}}{\cdot \vdash (\lambda x.\, x) \Leftarrow 1 \to 1}\ \mathsf{chk/lam}$$

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 $\tau' = \tau$.

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

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 fail or 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.

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

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:

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

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

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

There is a more general solution (see Exercise 7), 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* (β-reductions do not apply) that require no annotations. This is explored in Exercise 6.

Here is the summary of the rules.

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

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

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

The general idea is that *constructors are checked* and *destructors synthesize*. Let's see how this plays out for pair. We have

$$\frac{\Delta \vdash e_1 \Leftarrow \tau_1 \quad \Delta \vdash e_2 \Leftarrow \tau_2}{\Delta \vdash \langle e_1, e_2 \rangle \Leftarrow \tau_1 \times \tau_2}\ \mathsf{chk/pair}$$

$$\frac{\Delta \vdash e \Rightarrow \tau_1 \times \tau_2 \quad \Delta, x_1 \Rightarrow \tau_1, x_2 \Rightarrow \tau_2 \vdash e' \Leftarrow \tau'}{\Delta \vdash \mathbf{case}\ e\ (\langle x_1, x_2 \rangle \Rightarrow e') \Leftarrow \tau'}\ \mathsf{syn/casep}$$

The second rule looks a bit odd. It *does* go from synthesizing $\tau_1 \times \tau_2$ in the first premise to synthesizing τ_1 and τ_2 in the second premise. This is in the situation where we are trying to check an expression against a given type τ', which is propagated in to the branch of the **case**.

It is easy to see that this is mode-correct, reading the two premises from left to right. It is a bit more difficult to see that we should be, overall, ins the checking judgment and not synthesis. This is in part due to the connection to the philosophical notion of *verification* for natural deduction, and in part because we wish the system to be uniform over all the observable types. For sums, it seems plausible that it is unlikely that all branches will *synthesize* the same type by construction, but they can all be *checked* against a uniform type. This increases the robustness of bidirectional type-checking in the presence of richer notions such as subtyping.

We also add another mode-correct version of the first rule, going against the principle of checking only constructors. This may be convenient to type-check more programs without annotations.

$$\frac{\Delta \vdash e_1 \Rightarrow \tau_1 \quad \Delta \vdash e_2 \Rightarrow \tau_2}{\Delta \vdash \langle e_1, e_2 \rangle \Rightarrow \tau_1 \times \tau_2}\ \mathsf{syn/pair}$$

6 Some Programming Examples in LAMBDA

See lec9.cbv.

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*).

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

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

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

(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*.

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

(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

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

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:

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

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

Exercise 4 (Internalizing Definitions) We can *internalize* definitions as part of the core language. Specifically, we add

$$\begin{array}{rcll} \text{Expressions} \quad e & ::= & \mathsf{exp}\; f : \tau = e\; \mathsf{in}\; e' \\ & \mid & \mathsf{val}\; x = e\; \mathsf{in}\; e' \\ & \mid & \cdots \end{array}$$

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 sequentiality should hold, but you do not need to prove them.

Exercise 5 (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

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

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

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

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

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

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 6 (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 7 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

Penny Anderson and Frank Pfenning. Verifying uniqueness in a logical framework. In K.Slind, A.Bunker, and G.Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'04)*, pages 18–33, Park City, Utah, September 2004. Springer LNCS 3223.

Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Computing Surveys*, 54(5):98:1–98:38, 2022.

Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, editor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.