Lecture Notes on Indexed Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 20
Tue Nov 18, 2025

1 Introduction

This is the first of two lectures on forms of *dependent types*. The first, indexed types (also called *dependent refinement types*) maintain a full separation between expressions used to index types and expressions used for computation. They were first proposed by Zenger [1997] and then generalized by Xi and Pfenning [1999]. They arose from the goal of eliminating array bounds checks [Xi and Pfenning, 1998], but were generalized to capture other program property through indexing. A related modern rendering of the idea are *liquid types* [Rondon et al., 2008] which has instantiations for many different programming languages including Haskell [Vazou et al., 2014b,a] and Rust [Lehmann et al., 2023].

The fundamental idea is the a type is indexed by some additional information about the elements of the type. For example, we might have a type $\mathsf{list}\{n\}$ representing lists of length n. The type system should guarantee that this intrinsic property of values is preserved by computation. For example, the list append function should obey

$$\mathsf{append} : \forall n.\, \forall k.\, \mathsf{list}\{n\} \to \mathsf{list}\{k\} \to \mathsf{list}\{n+k\}$$

and the type-checker should verify this. Here, our index domain is natural numbers. Here is how we might define the type of lists of natural numbers equirecursively:

$$\begin{array}{l} \mathsf{list} : \mathbb{N} \to \mathit{type} \\ \mathsf{list}\{n\} = (\mathbf{nil} : \{n = 0\} \wedge 1) + (\mathbf{cons} : \{n > 0\} \wedge \mathsf{nat} \times \mathsf{list}\{n-1\}) \end{array}$$

Here we see the use of constraints like $n = 0$ and $n > 0$.

2 Domain Quantification

With indexed types and dependent types in general we see a tension between a form where terms from the index domain are explicit in program expressions and one where they are implicit. Generally, the explicit form is easier to define and type-check but more verbose to write. Because of the ease of definition, we start with the explicit form of quantification. This mirrors what we have done for polymorphism: the assumptions in the context are already dependent. For example, in the typing of $\Lambda\alpha.\, \lambda x.\, x : \forall\alpha.\, \alpha \to \alpha$ we arrive at a judgment $\alpha\ \mathit{type}, x : \alpha \vdash x : \alpha$. The context

α *type*, $x : \alpha$ has a dependency of the second declaration on the first one. Putting aside polymorphism for now, we might propose the following language of types. The language is somewhat open-ended in the sense that the exact language of terms and constraints and even types may differ in concrete instantiations of the general idea. Xi and Pfenning [1999] abstract over the constraint domain and set out some general properties that it should satisfy to obtain a coherent language.

$$\begin{array}{lrcl}
\text{Index Terms} & t & ::= & c \mid n \mid t_1 + t_2 \mid t_1 - t_2 \mid t_1 \times t_2 \mid \ldots \\
\text{Index Constraints} & \phi & ::= & t_1 = t_2 \mid t_1 > t_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \ldots \\
& & \mid & \forall n.\, \phi \mid \exists n.\, \phi \\
\text{Indexed Types} & \tau & ::= & a\,\{t_1, \ldots, t_n\} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \ldots \\
& & \mid & \forall n \in \mathbb{N}.\, \tau \mid \exists n \in \mathbb{N}.\, \tau \\
& & \mid & \{\phi\} \wedge \tau \mid \{\phi\} \supset \tau \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, n \in \mathbb{N} \mid \Gamma, \phi\ \mathit{true} \mid \Gamma, x : \tau
\end{array}$$

At an intuitive level, quantification seems relatively clear, but what about $\{\phi\} \wedge \tau$ and $\{\phi\} \supset \tau$? We can think of $\{\phi\} \wedge \tau$ as "*there exists a proof of ϕ and an expression of type τ*" and $\{\phi\} \supset \tau$ as "*for all proofs of ϕ we obtain an expression of type τ*. But, actually, we don't want to write out proofs, or even generate them, but instead rely on a decision procedure for the constraint domain. In that case $\{\phi\} \wedge \tau$ just means "*ϕ is true and we have an expression of type τ*" and $\{\phi\} \supset \tau$ means "*assume ϕ is true for the expression of type τ*".

We use the judgment $\hat{\Gamma} \models \phi$ for entailment in the constraint domain, where $\hat{\Gamma}$ erases any hypotheses $x : \tau$ from Γ. We also use $\hat{\Gamma} \vdash t\ \mathit{term}$ to verify well-formedness of terms in the constraint domain, including that all free variables in t are declared in $\hat{\Gamma}$.

This suggests the following rules for implication.

$$\frac{\Gamma, \phi\ \mathit{true} \vdash e : \tau}{\Gamma \vdash e : \{\phi\} \supset \tau}\ \supset I \qquad \frac{\Gamma \vdash e : \{\phi\} \supset \tau \quad \hat{\Gamma} \models \phi}{\Gamma \vdash e : \tau}\ \supset E$$

Conjunction is unfortunately a bit more complicated due to the nature of natural deduction. Officially, in the elimination rule, there should be the case rule we show below.

$$\frac{\hat{\Gamma} \models \phi \quad \Gamma \vdash e : \tau}{\Gamma \vdash e : \{\phi\} \wedge \tau}\ \wedge I \qquad \frac{\Gamma \vdash e : \{\phi\} \wedge \tau \quad \Gamma, \phi\ \mathit{true}, x : \tau \vdash e' : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (x \Rightarrow e') : \sigma}\ \wedge E$$

In the examples, we piggyback the conjunction elimination with general pattern matching so that the programmer rarely has to introduce an additional explicit case construct. For natural deduction, the principled solution is *tridirectional typechecking* [Dunfield and Pfenning, 2004]. In the sequent calculus, there are simpler solutions [Das and Pfenning, 2020, Somayyajula and Pfenning, 2023].

Universal quantification is again simpler than existential quantification.

$$\frac{\Gamma, n \in \mathbb{N} \vdash e : \tau}{\Gamma \vdash e : \forall n.\, \tau}\ \forall I \qquad \frac{\Gamma \vdash e : \forall n.\, \tau \quad \hat{\Gamma} \vdash t\ \mathit{term}}{\Gamma \vdash e\,\{t\} : [t/n]\tau}\ \forall E$$

For existentials, we again need a case construct.

$$\frac{\hat{\Gamma} \vdash t\ \mathit{term} \quad \Gamma \vdash e : [t/n]\tau}{\Gamma \vdash \langle \{t\}, e \rangle : \exists n.\, \tau}\ \exists I \qquad \frac{\Gamma \vdash e : \exists n.\, \tau \quad \Gamma, n : \mathbb{N}, x : \tau \vdash e' : \sigma}{\Gamma \vdash \mathsf{case}\ e\ (\langle n, x \rangle \Rightarrow e') : \sigma}\ \exists E$$

3 The Rules in Action

Let's implement append and simulate the type checking by recording context at various points in the expression. We start with the definition including (universal) index quantification.

$$\mathsf{list}\{n\} = (\mathbf{nil} : \{n = 0\} \wedge 1) + (\mathbf{cons} : \{n > 0\} \wedge \mathsf{nat} \times \mathsf{list}\{n-1\})$$

$$\begin{array}{l}
\mathsf{append} : \forall n.\, \forall k.\, \mathsf{list}\{n\} \to \mathsf{list}\{k\} \to \mathsf{list}\{n+k\} \\
\mathsf{append} = \lambda n.\, \lambda k.\, \lambda l_1.\, \lambda l_2. \\
\quad \mathsf{case}\ l_1\ \ (\ \mathbf{nil}\,\langle\,\rangle \Rightarrow l_2 \\
\qquad\quad \mid \mathbf{cons}\,\langle x, l_1'\rangle \Rightarrow \mathbf{cons}\,\langle x, \mathsf{append}\,\{n-1\}\,\{k\}\,l_1'\,l_2\rangle\ )
\end{array}$$

Next we add, as comments, additional information we get to assume or have to prove at various points during bidirectional type checking.

$$\mathsf{list}\{n\} = (\mathbf{nil} : \{n = 0\} \wedge 1) + (\mathbf{cons} : \{n > 0\} \wedge \mathsf{nat} \times \mathsf{list}\{n-1\})$$

$$\begin{array}{l}
\mathsf{append} : \forall n.\, \forall k.\, \mathsf{list}\{n\} \to \mathsf{list}\{k\} \to \mathsf{list}\{n+k\} \\
\mathsf{append} = \lambda n.\, \lambda k.\, \lambda l_1.\, \lambda l_2. \\
\quad \mathsf{case}\ l_1\ \ (\ \mathbf{nil}\,\langle\,\rangle \Rightarrow \qquad\qquad \%\ \text{assume } n = 0 \\
\qquad\qquad\qquad\quad l_2 \qquad\qquad \%\ \text{check } n \in \mathbb{N}, n = 0, l_1 : list\{n\}, l_2 : \mathsf{list}\{k\} \vdash l_2 : \mathsf{list}\{n+k\} \\
\qquad\qquad\qquad\qquad\quad\ \ \%\ \text{prove } n \in \mathbb{N}, n = 0 \models k = n + k \\
\qquad\quad \mid \mathbf{cons}\,\langle x, l_1'\rangle \Rightarrow \qquad \%\ \text{assume } n > 0, x : \mathsf{nat}, l_1' : \mathsf{list}\{n-1\} \\
\qquad\qquad\qquad\quad \mathbf{cons}\,\langle x, \mathsf{append}\,\{n-1\}\,\{k\}\,l_1'\,l_2\rangle \\
\qquad\qquad\qquad\qquad\quad\ \ \%\ \text{check against } \mathsf{list}\ (n+k) \\
\qquad\qquad\qquad\qquad\quad\ \ \%\ \text{prove } n > 0 \models n + k > 0 \\
\qquad\qquad\qquad\qquad\quad\ \ \%\ \text{check } \mathsf{append}\,\{n-1\}\,\{k\}\,l_1'\,l_2 \text{ against } \mathsf{list}\{n+k-1\} \\
\qquad\qquad\qquad\qquad\quad\ \ \%\ \text{prove } n \in \mathbb{N}, k \in \mathbb{N}, n > 0 \models n - 1 + k = n + k - 1 \\
\qquad\quad)
\end{array}$$

As another example, let's consider taking the length of a list. The type

$$\mathsf{length} : \forall n.\, \mathsf{list}\{n\} \to \mathsf{nat}$$

loses a lot of information, since the result should be the number n not just some unknown natural number. The way we can maintain the separation between computation and the index domain is to index natural numbers as constructed from zero and successor by their value.

$$\begin{array}{l}
\mathsf{nat} : \mathbb{N} \to \textit{type} \\
\mathsf{nat}\{n\} = (\mathbf{zero} : \{n = 0\} \wedge 1) + (\mathbf{succ} : \{n > 0\} \wedge \mathsf{nat}\{n-1\})
\end{array}$$

We say that $\mathsf{nat}\{n\}$ is a *singleton type* inhabited only by the representation of the natural number n. Then we can write and check:

$$\begin{array}{l}
\mathsf{length} : \forall n.\, \mathsf{list}\{n\} \to \mathsf{nat}\{n\} \\
\mathsf{length} = \lambda n.\, \lambda l. \\
\quad \mathsf{case}\ l\ \ (\ \mathbf{nil}\,\langle\,\rangle \Rightarrow \mathbf{zero}\,\langle\,\rangle \\
\qquad\quad \mid \mathbf{cons}\,\langle x, l'\rangle \Rightarrow \mathbf{succ}\ (\mathsf{length}\,\{n-1\}\,l')\ )
\end{array}$$

The reason is similar to append and a bit simpler.

We can take this further and index binary number representations by their value.

$$
\begin{array}{l}
\mathsf{bin} : \mathbb{N} \to \mathit{type} \\
\mathsf{bin}\{n\} \;= (\mathbf{e} : \{n = 0\} \wedge 1) \\
\qquad\quad + (\mathbf{b0} : \exists k.\, \{n = 2k\} \wedge \mathsf{bin}\{k\}) \\
\qquad\quad + (\mathbf{b1} : \exists k.\, \{n = 2k+1\} \wedge \mathsf{bin}\{k\})
\end{array}
$$

As written, $\mathsf{bin}\{n\}$ is not a singleton type because of leading zeros: in the alternative for **b0**, n and k could both be 0. If we want to make the representation unique, we could require $k > 0$:

$$
\begin{array}{l}
\mathsf{bin} : \mathbb{N} \to \mathit{type} \\
\mathsf{bin}\{n\} \;= (\mathbf{e} : \{n = 0\} \wedge 1) \\
\qquad\quad + (\mathbf{b0} : \exists k.\, \{k > 0 \wedge n = 2k\} \wedge \mathsf{bin}\{k\}) \\
\qquad\quad + (\mathbf{b1} : \exists k.\, \{n = 2k+1\} \wedge \mathsf{bin}\{k\})
\end{array}
$$

With this representation it is relatively straightforward to verify, for example, the translations between unary and binary numbers, or the correctness of addition of binary numbers.

As a last examples that points out some of the difficulties, we consider the filter function. First, the unindexed function.

$$
\begin{array}{l}
\mathsf{filter} : (\mathsf{nat} \to \mathsf{bool}) \to \mathsf{list} \to \mathsf{list} \\
\mathsf{filter} = \lambda p.\, \lambda l.\, \mathsf{case}\ l \\
\quad (\mathbf{nil}\,\langle\,\rangle \Rightarrow \mathbf{nil}\,\langle\,\rangle \\
\quad \mid \mathbf{cons}\,\langle x, l'\rangle \Rightarrow \mathsf{if}\ p\,x\ \mathsf{then}\ \mathbf{cons}\,\langle x, \mathsf{filter}\,p\,l'\rangle \\
\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{filter}\,p\,l'\,)
\end{array}
$$

The first difficulty is that we cannot statically predict how long the output list will be. We just know it will have some length $k \leq n$, if the input list has length n. In our language:

$$
\mathsf{filter} : \forall n.\, (\mathsf{nat} \to \mathsf{bool}) \to \mathsf{list}\{n\} \to \exists k.\, \{k \leq n\} \wedge \mathsf{list}\{k\}
$$

$$
\begin{array}{l}
\mathsf{filter} = \lambda n.\, \lambda p.\, \lambda l.\, \mathsf{case}\ l \\
\quad (\mathbf{nil}\,\langle\,\rangle \Rightarrow \langle\{0\}, \mathbf{nil}\,\langle\,\rangle\rangle \\
\quad \mid \mathbf{cons}\,\langle x, l'\rangle \Rightarrow \mathsf{if}\ p\,x\ \mathsf{then}\ \mathbf{cons}\,\langle x, \mathsf{filter}\,\{n{-}1\}\,p\,l'\rangle \qquad \text{\% not type-correct} \\
\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{filter}\,\{n{-}1\}\,p\,l'\,)
\end{array}
$$

Unfortunately, with our rules, this function will not type-check. The problem is that the recursive call to filter returns something of length k', but in the "then" branch we want to return something of length $k' + 1$. But we don't have access to k'. We can rewrite the code to introduce a case expression that allows us to create a binding for k'.

$$
\mathsf{filter} : \forall n.\, (\mathsf{nat} \to \mathsf{bool}) \to \mathsf{list}\{n\} \to \exists k.\, \{k \leq n\} \wedge \mathsf{list}\{k\}
$$

$$
\begin{array}{l}
\mathsf{filter} = \lambda n.\, \lambda p.\, \lambda l.\, \mathsf{case}\ l \\
\quad (\mathbf{nil}\,\langle\,\rangle \Rightarrow \langle\{0\}, \mathbf{nil}\,\langle\,\rangle\rangle \\
\quad \mid \mathbf{cons}\,\langle x, l'\rangle \Rightarrow \mathsf{case}\ \mathsf{filter}\,\{n{-}1\}\,p\,l' \\
\qquad\qquad\qquad\quad (\langle k', r\rangle \Rightarrow \mathsf{if}\ p\,x\ \mathsf{then}\ \langle\{k'{+}1\}, \mathbf{cons}\,\langle x, r\rangle\rangle\ \mathsf{else}\ \langle\{k'\}, r\rangle)\,)
\end{array}
$$

As mentioned above, we can make witnesses for the existentials and arguments for the universals also implicit, under some additional strain on the constraint solver.

The effectiveness of type checking for indexed types depends on the right balance between brevity of the code and the difficulty of solving the constraints. Modern day SML solvers are often quite effective even in the presence quantifiers when the theory may be undecidable. One can also try quantifier elimination procedures for, say, Presburger arithmetic. Even though their theoretical complexity is quite high, they may work well enough in practice to be usable.

References

Ankush Das and Frank Pfenning. Session types with arithmetic refinements. In I. Konnov and L. Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, pages 13:1–13:18, Vienna, Austria, September 2020. LIPIcs 171.

Jana Dunfield and Frank Pfenning. Tridirectional typechecking. In X.Leroy, editor, *Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04)*, pages 281–292, Venice, Italy, January 2004. ACM Press. Extended version available as Technical Report CMU-CS-04-117, March 2004.

Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for Rust. *Proceedings of the ACM on Programming Languages*, 7 (PLDI):1533–1557, January 2023.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In R. Gupta and S. Amarasinghe, editors, *Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 159–169, Tuscon, Arizona, June 2008. ACM.

Siva Somayyajula and Frank Pfenning. Dependent type refinements for futures. In M. Kerjean and P. Levy, editors, *39th International Conference on Mathematical Foundations of Programming Semantics (MFPS 2023)*, Bloomington, Indiana, USA, June 2023. Preliminary version.

Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquid Haskell: Experience with refinements types in the real world. In *Symposium on Haskell*, pages 39–51, Gothenburg, Sweden, September 2014a. ACM.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for Haskell. In *19th International Conference on Functional Programming (ICFP 2014)*, pages 269–292, Gothenburg, Sweden, September 2014b. ACM.

Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.

Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In A. Aiken, editor, *Conference Record of the 26th Symposium on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, January 1999.

Christoph Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.