15-819: Resource Aware Programming Languages

# Lecture 10-11: Precious Little Diamond

Jan Hoffmann

October 13, 2020

## 1 Introduction

An unexpected application area for programming languages is complexity theory. Can we design a programming language in which we can only write programs that run in polynomial time? Can we design a programming language in which we can implement exactly the polynomial-time functions? These questions have been positively answered by Bellantoni and Cook in 1992 [BC92]. The original motivation for the development of Bellantoni and Cook's language was to provide a syntactic characterization of polynomial time that would eventually lead to a separation of the complexity classes $\mathsf{P}$ and $\mathsf{NP}$ by proving that a specific decision function in $\mathsf{NP}$ (e.g., traveling salesman) cannot be implemented in the language. Today, there exist several syntactic characterization of $\mathsf{P}$ and there are also other interesting complexity classes (like $\mathsf{LOGSPACE}$ or $\mathsf{PSPACE}$), which have been characterized by programming languages. The research area that studies such languages is called *implicit computational complexity*. We restrict our attention to $\mathsf{FP}$ and $\mathsf{P}$ in this lecture.

The title of the lecture is inspired by the 1984 song *Precious Little Diamond* by *Fox the Fox* and the diamond type $\Diamond$ that has been introduced by Hofmann [Hof99] to develop *LFPL*, my favorite programming language for $\mathsf{P}$. A variant of this language characterizes the complexity class $\mathsf{EXP}$ [Hof02].

## 2 Complexity Classes

We say that a (mathematical) function is *polynomial time* or in the class $\mathsf{FP}$ if it can be implemented by an algorithm whose runtime is bounded by a polynomial. The class of functions $\mathsf{P}$ is the subset of $\mathsf{FP}$ in which we only consider functions that encode *decision problems*, that is, functions with a boolean result type. Similarly, the class $\mathsf{NP}$ consists of decision problems that can be implemented in non-deterministic polynomial time.

When defining complexity classes, the representation of data is of central importance. How are the inputs to, say, a Turing machine encoded? This choice impacts what mathematical functions end up in the complexity class. The convention for natural number is to use binary representation:

$$\begin{array}{lcll}
\mathsf{FP} & = & \{ f : \mathbb{N}^k \to \mathbb{N} & \mid \ f \text{ can be implemented with a polynomial time algorithm} \\
 & & & \quad \text{that uses binary representation} \} \\
\mathsf{P} & = & \{ f : \mathbb{N}^k \to \{0,1\} & \mid \ f \text{ can be implemented with a polynomial time algorithm} \\
 & & & \quad \text{that uses binary representation} \}
\end{array}$$

We follow this convention in this lecture and corresponding use binary representation for natural numbers in the programming languages for $\mathsf{FP}$ and $\mathsf{P}$. Note however that this choice is not essential and the respective characterizations would carry over to a setting in which we would use unary representation in both the definition of the complexity classes and the programming languages.

$\boxed{\Gamma \vdash e : \tau \qquad \text{“expression } e \text{ has type } \tau \text{ under context } \Gamma\text{”}}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\text{T:Var}) \qquad \frac{\Gamma, x{:}\tau' \vdash e : \tau}{\Gamma \vdash \mathrm{lam}\{\tau'\}(x.e) : \tau' \to \tau}\ (\text{T:Abs}) \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \mathrm{app}(e_1; e_2) : \tau}\ (\text{T:App})$$

$$\frac{}{\Gamma \vdash \mathrm{z} : \mathrm{nat}}\ (\text{T:Zero}) \qquad \frac{\Gamma \vdash e : \mathrm{nat}}{\Gamma \vdash \mathrm{s}(e) : \mathrm{nat}}\ (\text{T:Succ})$$

$$\frac{\Gamma \vdash e : \mathrm{nat} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathrm{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \mathrm{rec}\{e_0; x, y.e_1\}(e) : \tau}\ (\text{T:Rec})$$

**Figure 1:** Static semantics of System T.

## 3 Structural Recursion

Let us approach the problem of designing a language for FP step by step. Our first observation is that the language should be total since every function in FP is total. This means that we have to restrict recursion and cannot add recursive functions $\mathrm{fun}\{\tau,\tau'\}(f, x.e)$ like in the lecture on cost semantics.

**System T** A prominent example of a total language is Gödel's *System T* [Har12]. System T has been designed by Gödel around 1941 and presented in a talk at Yale University. However, it was published much later. The context of this development was Gödel's interest in proof theory. He had already shown that there are effective proof systems in his Completeness Theorem (1929) and that such proof system are necessarily incomplete for logics that are at least as expressive as Peano Arithmetic (PA). PA roughly corresponds to first-order theorems that can be proved by (nested) induction over natural numbers. In particular, Gödel's Incompleteness Theorem showed that it is impossible to prove the consistency of PA inside PA. System T was presented as a higher-order version of PA and Gödel's main result was that System T is expressive enough to show the consistency of PA. Here, System T is presented as a programming language for total functions.

In System T, general recursion is replaced with a recursor over natural numbers (see below). The idea is to define a function $f : \mathrm{nat} \to \tau$ by defining the base case $f(0) : \tau$ and a recursive case $f(n+1) : \mathrm{nat} \to \tau \to \tau$ that computes the result as a function of the predecessor $n$ and the recursive call $f(n)$.

The types of System T are defined as follows.

$$\begin{array}{lll} \tau & ::= & \mathrm{nat} \\ & & \tau_1 \to \tau_2 \end{array}$$

The expressions of System T of the are defined as follows.

$$\begin{array}{lllll} e & ::= & x & & x \\ & & \mathrm{lam}\{\tau\}(x.e) & & \lambda(x:\tau)e \\ & & \mathrm{app}(e_1; e_2) & & e_1(e_2) \\ & & \mathrm{z} & & \mathsf{z} \\ & & \mathrm{s}(e) & & \mathsf{s}(e) \\ & & \mathrm{rec}\{e_0; x, y.e_1\}(e) & & \mathsf{rec}\ e\ \{z \hookrightarrow e_0 \mid \mathsf{s}(x)\ \mathsf{with}\ y \hookrightarrow e_1\} \end{array}$$

The recursor $\mathrm{rec}\{e_0; x, y.e_1\}(e)$ defines a (terminating) recursion on the value $n$ of $e$. The base case ($n = 0$) is given by $e_0$ and the recursive case ($n = n' + 1$) is given by $e_1$ where the predecessor $n'$ of $n$ is bound to $x$ and the recursive result is bound to $y$.

The static semantics of System T $e$ is given by the judgment $\Gamma \vdash e : \tau$ as defined by the rules in Figure 1. Note that there are no restrictions on the result type $\tau$ in the rule T:Rec. So $\tau$ can

$$\boxed{e \Downarrow v \qquad \text{"expression } e \text{ evaluates to value } v\text{"}}$$

$$\frac{}{\mathrm{lam}\{\tau\}(x.e) \Downarrow \mathrm{lam}\{\tau\}(x.e)}\ (\text{E:Lam}) \qquad \frac{e_1 \Downarrow \mathrm{lam}\{\tau\}(x.e) \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{\mathrm{app}(e_1;e_2) \Downarrow v}\ (\text{E:App})$$

$$\frac{}{\mathrm{z} \Downarrow \mathrm{z}}\ (\text{E:Zero}) \qquad \frac{e \Downarrow v}{\mathrm{s}(e) \Downarrow \mathrm{s}(v)}\ (\text{E:Succ}) \qquad \frac{e \Downarrow \mathrm{z} \qquad e_0 \Downarrow v}{\mathrm{rec}\{e_0;x,y.e_1\}(e) \Downarrow v}\ (\text{E:Rec-Z})$$

$$\frac{e \Downarrow \mathrm{s}(v_x) \qquad \mathrm{rec}\{e_0;x,y.e_1\}(v_x) \Downarrow v_y \qquad [v_x,v_y/x,y]e_1 \Downarrow v}{\mathrm{rec}\{e_0;x,y.e_1\}(e) \Downarrow v}\ (\text{E:Rec-S})$$

**Figure 2:** Dynamic semantics of System T.

be a function type. This ability makes System T surprisingly powerful. For example, we can define Ackermann's function which is an extremely fast growing function which is hopelessly far outside of the class FP.

We note that higher-order functions often pose challenges to resource analysis, as we will see later in the course. Our first step is to look a version of System T in which results of recursive computations are restricted to natural numbers.

The dynamic semantics of System T is given in Figure 2. We use a vanilla evaluation dynamics using the judgment $e \Downarrow v$. We do not need a cost semantics since we use an intrinsic notion of complexity given by the mathematical functions in the classes FP and FP in this lecture.

It is surprisingly difficult to show that System T is indeed a total language, that is, that every closed expression evaluates to a value. The key is to prove the stronger property of hereditary termination of an (open) expression $\Gamma \vdash e : \tau$ by induction on the type $\tau$.

**Theorem 1.** *If $e : \tau$ is a closed expression then $e \Downarrow v$ for some value $v$.*

**Expressivity of System T** It is not surprising that the proof of Theorem 1 is not straightforward if we consider the expressivity of System T.

We define a translation from natural numbers to numerals as follows.

$$\begin{array}{lcl} \overline{0} & = & \mathrm{z} \\ \overline{n+1} & = & \mathrm{s}(\overline{n}) \end{array}$$

**Definition.** *We say that a function $h : \mathbb{N}^k \to \mathbb{N}$ is definable in System T if there is an expression $e_h : \mathrm{nat} \to \cdots \to \mathrm{nat}$ such that $e_h(\overline{n_1}) \cdots (\overline{n_k}) \Downarrow \overline{h(n_1,\ldots,n_k)}$ for all $n_1,\ldots,n_k$.*

We can show that the functions definable in System T correspond exactly to the function that can be proved to be terminating in PA. Intuitively, that corresponds to functions for which we can show termination by a (nested) induction on the natural numbers.

One prominent function that is definable in System T (how?) is Ackermann's functions $A : \mathbb{N}^2 \to \mathbb{N}$ defined as

$$\begin{array}{lcl} A(0,n) & = & n+1 \\ A(m+1,0) & = & A(n,1) \\ A(m+1,n+1) & = & A(m,A(m+1,n)) \end{array}$$

Note that it follows from the computational version of Gödel's Incompleteness Theorem that for every total language, there are (total) computable functions that are not definable in the language. One such function is an interpreter for the total language. In the case of System T, this is a function that takes an encoding of a System T expression $e : \mathrm{nat}$ and returns $n$ such that $e \vdash \overline{n}$. Theorem 1 states that such an interpreter is indeed total and thus intuitively computable.

**Primitive Recursion** The next step down to our way to a language for FP is to reduce the expressivity of System T by restricting recursion to the type nat. This leads to the primitive recursive functions, which do not contain extremely fast growing functions like Ackermann's function. The *primitive recursive* functions are first-order functions $\mathbb{N}^k \to \mathbb{N}$ that are usually defined inductively on (mathematical) functions. Here, we define a higher-order version that is very similar to System T. We call this language *System P*.

The types, expressions, and dynamic semantics carry over from System T. Moreover, most rules of the static semantics are identical. The only change is that we replace the rule T:Rec with the rule P:Rec below.

$$\frac{\Gamma \vdash e : \mathrm{nat} \qquad \Gamma \vdash e_0 : \mathrm{nat} \qquad \Gamma, x : \mathrm{nat}, y : \mathrm{nat} \vdash e_1 : \mathrm{nat}}{\Gamma \vdash \mathrm{rec}\{e_0; x, y.e_1\}(e) : \mathrm{nat}} \ (\text{P:Rec})$$

The rule P:Rec requires that the result type of the recursive computation is nat instead of an arbitrary type $\tau$ as in System T.

## Expressivity of Primitive Recursion

**Definition.** *We say that a function $h : \mathbb{N}^k \to \mathbb{N}$ is primitive recursive (with unary representation) if there is an expression $e_h : \mathrm{nat} \to \cdots \to \mathrm{nat}$ such that $e(\overline{n_1}) \cdots (\overline{n_k}) \Downarrow \overline{h(n_1, \ldots, n_k)}$ for all $n_1, \ldots, n_k$.*

Primitive recursive functions cannot grow as fast as functions definable in System T.

**Theorem 2.** *Ackermann's function is not primitive recursive.*

Nevertheless, it is easy implement primitive recursive functions that are not in the class FP. Consider for example the function *exp* below.

$$\begin{array}{lcl} \texttt{double} & \equiv & \lambda(x : \mathrm{nat})\ \mathsf{rec}\ x\ \{z \hookrightarrow z \mid \mathsf{s}(\_)\ \mathsf{with}\ y \hookrightarrow s(s(y))\} \\ \texttt{exp} & \equiv & \lambda(x : \mathrm{nat})\ \mathsf{rec}\ x\ \{z \hookrightarrow s(z) \mid \mathsf{s}(\_)\ \mathsf{with}\ y \hookrightarrow \texttt{double}(y)\} \end{array}$$

Then $exp(\overline{n}) = \overline{2^n}$. Further iteration on the function *exp* leads to enormous growth. We did not define a cost semantics in this lecture but it should be intuitively clear that *(*exp) has exponential cost and thus should not be in the class FP.

**Binary Representation** In complexity theory,[1] it is standard to work with a binary representation of natural numbers. This means that we look for algorithms that are polynomial-time in the sizes $|n|$ of their inputs, where

$$|n| = \lceil \log_2(n+1) \rceil \, .$$

We extend the size operation point-wise to tuples $\vec{n}$ and define $|(n_1, \ldots, n_k)| = (|n_1|, \ldots, |n_k|)$. So the class FP contains exactly the functions $f : \mathbb{N}^k \to \mathbb{N}$ for which $f(\vec{n})$ is computable with a Turing machine whose steps are bounded by a polynomial in $|\vec{n}|$.

In the presented version of System P, we used a unary representation of natural numbers like the numerals $\overline{n}$. This is not a good fit for the binary representation: With a natural cost semantics, the cost of the function *double* is already exponential in $|n|$. So it is beneficial to use a binary representation of the natural numbers to match the definition of the class FP.[2] To this end, we change our expression language as follows.

$$\begin{array}{lcll} e & ::= & x & x \\ & & \mathrm{lam}\{\tau\}(x.e) & \lambda(x : \tau)e \\ & & \mathrm{app}(e_1; e_2) & e_1(e_2) \\ & & \mathrm{z} & \mathsf{z} \\ & & \mathrm{s}_0(e) & \mathsf{s}_0\,(e) \\ & & \mathrm{s}_1(e) & \mathsf{s}_1\,(e) \\ & & \mathrm{case}\{e_0; x.e_1; y.e_2\}(e) & \mathsf{case}\ e\ \{z \hookrightarrow e_0 \mid \mathsf{s}_0(x) \hookrightarrow e_1 \mid \mathsf{s}_1(y) \hookrightarrow e_2\} \\ & & \mathrm{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) & \mathsf{rec}\ e\ \{z \hookrightarrow e_0 \mid \mathsf{s}_0(x_1)\ \mathsf{with}\ y_1 \hookrightarrow e_1 \mid \mathsf{s}_1(x_2)\ \mathsf{with}\ y_2 \hookrightarrow e_2\} \end{array}$$

[1] The issue of representation of inputs is crucial to complexity theory and algorithm design but often not discussed in depth.

[2] Alternatively, we could define FP to contain the functions $f : \mathbb{N}^k \to \mathbb{N}$ so that $f(n_1, \ldots, n_k)$ is computable with an algorithm that is polynomial in $n_1, \ldots, n_k$.

$$\boxed{\Gamma \vdash e : \tau \qquad \text{“expression } e \text{ has type } \tau \text{ under context } \Gamma\text{”}}$$

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathsf{nat}}\ (\text{T:Z-B}) \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{s}_0(e) : \mathsf{nat}}\ (\text{T:SE}) \qquad \frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{s}_1(e) : \mathsf{nat}}\ (\text{T:SO})$$

$$\frac{\Gamma \vdash e : \mathsf{nat} \qquad \Gamma \vdash e_0 : \mathsf{nat} \qquad \Gamma, x_1 : \mathsf{nat} \vdash e_1 : \mathsf{nat} \qquad \Gamma, x_2 : \mathsf{nat} \vdash e_2 : \mathsf{nat}}{\Gamma \vdash \mathsf{case}\{e_0; x_1.e_1; x_2.e_2\}(e) : \mathsf{nat}}\ (\text{T:Case})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathsf{nat} \\ \Gamma \vdash e_0 : \mathsf{nat} \qquad \Gamma, x_1 : \mathsf{nat}, y_1 : \mathsf{nat} \vdash e_1 : \mathsf{nat} \qquad \Gamma, x_2 : \mathsf{nat}, y_2 : \mathsf{nat} \vdash e_2 : \mathsf{nat}\end{array}}{\Gamma \vdash \mathsf{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) : \mathsf{nat}}\ (\text{T:Rec-B})$$

**Figure 3:** Static semantics of binary numbers.

$$\boxed{e \Downarrow v \qquad \text{“expression } e \text{ evaluates to value } v\text{”}}$$

$$\frac{}{\mathsf{z} \Downarrow \mathsf{z}}\ (\text{E:Zero}) \qquad \frac{e \Downarrow v}{\mathsf{s}_1(e) \Downarrow \mathsf{s}_1(v)}\ (\text{E:SO}) \qquad \frac{e \Downarrow v}{\mathsf{s}_1(e) \Downarrow \mathsf{s}_0(v)}\ (\text{E:SE})$$

$$\frac{e \Downarrow \mathsf{z} \qquad e_0 \Downarrow v}{\mathsf{case}\{e_0; x_1.e_1; x_2.e_2\}(e) \Downarrow v}\ (\text{E:Case-Z}) \qquad \frac{e \Downarrow \mathsf{s}_1(v') \qquad [v'/x]e_1 \Downarrow v}{\mathsf{case}\{e_0; x_1.e_1; x_2.e_2\}(e) \Downarrow v}\ (\text{E:Case-O})$$

$$\frac{e \Downarrow \mathsf{s}_0(v') \qquad [v'/x]e_2 \Downarrow v}{\mathsf{case}\{e_0; x_1.e_1; x_2.e_2\}(e) \Downarrow v}\ (\text{E:Case-E}) \qquad \frac{e \Downarrow \mathsf{z} \qquad e_0 \Downarrow v}{\mathsf{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) \Downarrow v}\ (\text{E:BRec-Z})$$

$$\frac{e \Downarrow \mathsf{s}_0(v') \qquad \mathsf{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(v') \Downarrow v_r \qquad [v', v_r/x_1, y_1]e_1 \Downarrow v}{\mathsf{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) \Downarrow v}\ (\text{E:BRec-E})$$

$$\frac{e \Downarrow \mathsf{s}_1(v') \qquad \mathsf{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(v') \Downarrow v_r \qquad [v', v_r/x_2, y_2]e_2 \Downarrow v}{\mathsf{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) \Downarrow v}\ (\text{E:BRec-O})$$

**Figure 4:** Dynamic semantics of binary numbers.

So we have two “successor” constructors $\mathsf{s}_0(e)$ and $\mathsf{s}_1(e)$, one for even and one for odd numbers. Correspondingly, we have a case construct that branches based on the constructor used.

The type rules for the binary constructs are given in Figure 3. The evaluation rules of the binary constructs are given in Figure 4.

We define the binary numerals as

$$\begin{array}{lcll} \widetilde{0} & = & \mathsf{z} & \\ \widetilde{2n+1} & = & \mathsf{s}_1(\widetilde{n}) & \\ \widetilde{2n} & = & \mathsf{s}_0(\widetilde{n}) & \text{if } n > 0 \end{array}$$

The syntactic form for case analysis is redundant. We have

$$\mathsf{case}\{e_0; x_1.e_1; x_2.e_2\}(e) \equiv \mathsf{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e)$$

if $y_1 \not\in \mathrm{FV}(e_1)$ and $y_2 \not\in \mathrm{FV}(e_2)$. The reason we added is because recursion and case analysis are treated differently in the type system of System BC in the following section.

**Definition.** *We say that a function $h : \mathbb{N}^k \to \mathbb{N}$ is primitive recursive (with binary representation) if there is an expression $e_h : \mathrm{nat} \to \cdots \to \mathrm{nat}$ such that $e(\widetilde{n_1}) \cdots (\widetilde{n_k}) \Downarrow h(\widetilde{n_1, \dots, n_k})$ for all $n_1, \dots, n_k$.*

Now we can implement a *double* function runs (intuitively) in polynomial time. However, the switch to binary representations does not help with the problem of fast growing functions. It is still possible to implement functions that are not in the class FP using binary representations.

An example for exponential growth is the function *bexp* below.

$$
\begin{array}{lcl}
\mathit{conc} & : & \mathrm{nat} \to \mathrm{nat} \to \mathrm{nat} \\
\mathit{conc} & \equiv & \lambda(n : \mathrm{nat})\ \lambda(m : \mathrm{nat}) \\
& & \mathsf{rec}\ n\ \{z \hookrightarrow m \\
& & \qquad \mid \mathsf{s}_0(x_1)\ \mathsf{with}\ y_1 \hookrightarrow \mathsf{s}_0(y_1) \\
& & \qquad \mid \mathsf{s}_1(x_2)\ \mathsf{with}\ y_2 \hookrightarrow \mathsf{s}_1(y_2)\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{bexp} & : & \mathrm{nat} \to \mathrm{nat} \\
\mathit{bexp} & \equiv & \lambda(n : \mathrm{nat}) \\
& & \mathsf{rec}\ n\ \{z \hookrightarrow \mathsf{s}_1(\mathrm{z}) \\
& & \qquad \mid \mathsf{s}_0(x_1)\ \mathsf{with}\ y_1 \hookrightarrow \mathit{conc}(y_1)(y_1) \\
& & \qquad \mid \mathsf{s}_1(x_2)\ \mathsf{with}\ y_2 \hookrightarrow \mathit{conc}(y_2)(y_2)\}
\end{array}
$$

## 4 System BC

The idea of *safe recursion* [BC92] is to restrict the growths of functions by limiting the use of recursive computations. Like with the primitive recursive functions, the original formulation of the idea is an inductive definition of mathematical functions $\mathbb{N}^k \to \mathbb{N}$. Each of these functions is of the form $f(\vec{x}; \vec{y})$ where $\vec{x}$ corresponds to normal arguments that are available for iterations and $\vec{y}$ are the *safe* arguments that cannot be used in iterations. The results of "recursive calls" can only be used as safe arguments.

We discuss *System BC*, a higher-order variant of safe recursion that has been introduced by Hofmann [Hof97a]. The idea is to introduce a type modality $\Box\tau$ that represents the permission to perform recursive iterations. The modality is only present in arguments of function types.

$$
\begin{array}{lcl}
\tau & ::= & \mathrm{nat} \\
& & \tau_1 \to \tau_2 \\
& & \Box\tau_1 \to \tau_2
\end{array}
$$

It is important to note that $\Box\tau$ is not in general a type. The type $\Box\tau_1 \to \tau_2$ describes functions that can iterate over values that depend on the argument. The type $\tau_1 \to \tau_2$ corresponds to functions for which is argument is "safe" to be used with arbitrary values at a call site and values that depend on it cannot be used in recursive iterations.

The expressions of System BC are identical to the expressions of System P with binary numbers.

$$
\begin{array}{lcll}
e & ::= & x & x \\
& & \mathrm{lam}\{\tau\}(x.e) & \lambda(x : \tau)e \\
& & \mathrm{app}(e_1; e_2) & e_1(e_2) \\
& & \mathrm{z} & \mathsf{z} \\
& & \mathrm{s}_0(e) & \mathsf{s}_0\,(e) \\
& & \mathrm{s}_1(e) & \mathsf{s}_1\,(e) \\
& & \mathrm{case}\{e_0; x.e_1; y.e_2\}(e) & \mathsf{case}\ e\ \{z \hookrightarrow e_0 \mid \mathsf{s}_0(x) \hookrightarrow e_1 \mid \mathsf{s}_1(y) \hookrightarrow e_2\} \\
& & \mathrm{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) & \mathsf{rec}\ e\ \{z \hookrightarrow e_0 \mid \mathsf{s}_0(x_1)\ \mathsf{with}\ y_1 \hookrightarrow e_1 \mid \mathsf{s}_1(x_2)\ \mathsf{with}\ y_2 \hookrightarrow e_2\}
\end{array}
$$

The static semantics is given by the judgment

$$\Delta; \Gamma \vdash e : \tau$$

$\boxed{\Gamma;\Delta \vdash e : \tau \qquad \text{“expression } e \text{ has type } \tau \text{ under modal context } \Gamma \text{ and safe context } \Delta\text{”}}$

$$\frac{\Delta(x) = \tau}{\Delta;\Gamma \vdash x : \tau}\ (\text{BC:Var-}\Box) \qquad \frac{\Gamma(x) = \tau}{\Delta;\Gamma \vdash x : \tau}\ (\text{BC:Var}) \qquad \frac{\Delta;\Gamma, x{:}\tau' \vdash e : \tau}{\Delta;\Gamma \vdash \mathrm{lam}\{\tau'\}(x.e) : \tau' \to \tau}\ (\text{BC:Abs})$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau' \to \tau \qquad \Delta;\Gamma \vdash e_2 : \tau'}{\Delta;\Gamma \vdash \mathrm{app}(e_1; e_2) : \tau}\ (\text{BC:App}) \qquad \frac{\Delta, x{:}\tau';\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \mathrm{lam}\{\tau'\}(x.e) : \Box\tau' \to \tau}\ (\text{BC:Abs-}\Box)$$

$$\frac{\Delta;\Gamma \vdash e_1 : \Box\tau' \to \tau \qquad \Delta;\cdot \vdash e_2 : \tau'}{\Delta;\Gamma \vdash \mathrm{app}(e_1; e_2) : \tau}\ (\text{BC:App-}\Box) \qquad \frac{}{\Delta;\Gamma \vdash \mathrm{z} : \mathrm{nat}}\ (\text{BC:Zero})$$

$$\frac{\Delta;\Gamma \vdash e : \mathrm{nat}}{\Delta;\Gamma \vdash \mathrm{s}_0(e) : \mathrm{nat}}\ (\text{BC:SE}) \qquad \frac{\Delta;\Gamma \vdash e : \mathrm{nat}}{\Delta;\Gamma \vdash \mathrm{s}_1(e) : \mathrm{nat}}\ (\text{BC:SO})$$

$$\frac{\begin{array}{c}\Delta;\cdot \vdash e : \mathrm{nat} \\ \Delta;\Gamma \vdash e_0 : \mathrm{nat} \qquad \Delta, x_1 : \mathrm{nat};\Gamma, y_1 : \mathrm{nat} \vdash e_1 : \mathrm{nat} \qquad \Delta, x_2 : \mathrm{nat};\Gamma, y_2 : \mathrm{nat} \vdash e_2 : \mathrm{nat}\end{array}}{\Delta;\Gamma \vdash \mathrm{rec}\{e_0; x_1, y_1.e_1; x_2, y_2.e_2\}(e) : \mathrm{nat}}\ (\text{BC:Rec})$$

$$\frac{\Delta;\Gamma \vdash e : \mathrm{nat} \qquad \Delta;\Gamma \vdash e_0 : \mathrm{nat} \qquad \Delta;\Gamma, x_1 : \mathrm{nat} \vdash e_1 : \mathrm{nat} \qquad \Delta;\Gamma, x_2 : \mathrm{nat} \vdash e_2 : \mathrm{nat}}{\Delta;\Gamma \vdash \mathrm{case}\{e_0; x_1.e_1; x_2.e_2\}(e) : \mathrm{nat}}\ (\text{BC:Case})$$

$$\frac{\Delta;\Gamma \vdash e : \tau' \qquad \tau' <: \tau}{\Delta;\Gamma \vdash e : \tau}\ (\text{BC:Sub})$$

**Figure 5:** Static semantics of System BC.

The context $\Delta$ contains the modal variables, which can be used for recursive iteration while $\Gamma$ contains the *safe* variables that can only be used positions that are safe, that is, do not influence recursive iterations. We maintain the invariant that $\mathrm{dom}(\Gamma) \cup \mathrm{dom}(\Delta) = \emptyset$.

The type rules are defined in Figure 5. There are variable rules, one for variables in the modal context $\Delta$ and one for the safe context $\Gamma$. Similarly, there are two abstraction rules BC:Abs and BC:Abs-$\Box$. The rule BC:Abs introduces the function type $\tau' \to \tau$. Since $\tau'$ is the type of a safe argument, we add the binding $x : \tau'$ to the safe context $\Gamma$ when typing the function body $e$. The rule BC:Abs-$\Box$ introduces the function type $\Box\tau' \to \tau$. Here, $\tau'$ is the type of a modal argument and we add the binding $x : \tau'$ to the modal context $\Delta$ when typing the function body $e$.

The difference between the modal and safe function types becomes apparent in the rules BC:App and BC:App-$\Box$ for function application. In the rule BC:App, the function argument $e_2$ can depend on the variables in $\Delta$ and $\Gamma$. However, in BC:App-$\Box$, the argument $e_2$ can only depend on the variables in the modal context $\Delta$. The intuition is that we prevent iteration over values that depend on safe variables.

The rule *BC:Rec* contains the key idea of the type system. In the premise $\Delta, x_1 : \mathrm{nat};\Gamma, y_1 : \mathrm{nat} \vdash e_1 : \mathrm{nat}$, we add the variable $x_1$ (which will be bound to the predecessor of the value of $e$) to the modal context $\Delta$ since it is still available for further iteration. However, we add the variable $y_1$ (which will be bound to the recursive result) to the safe context $\Gamma$ to prevent iteration on the recursive result. In addition, the recursive iteration is restricted to an expression $e$ that only depends on modal variables.

The rule *BC:Case* illustrates the reason that the case construct is present in the language. If we would implement the case analysis with recursion then the argument of the case analysis would have to be modal. In *BC:Case* we the allow $e$ to be safe (premise $\Delta;\Gamma \vdash e : \mathrm{nat}$) and thus case analysis on safe expressions. This is needed to capture all functions in FP.

Finally, we have a subtyping rule BC:SUB that enables use to use modal types when safe arguments are required. The subtyping relation is defined by the following rules.

$$\frac{}{\tau_1 \to \tau_2 <: \Box\tau_1 \to \tau_2}\ (\text{SUB:1}) \qquad\qquad \frac{\sigma_1 <: \tau_1 \qquad \tau_2 <: \sigma_2}{\tau_1 \to \tau_2 <: \sigma_1 \to \sigma_2}\ (\text{SUB:2})$$

**Dynamic Semanitcs** The dynamic semantics $e \Downarrow v$ is identical to the judgment we defined previously for System P with binary numbers. In particular, we do not define a cost semantics. In the case of System BC, we are not interested in the exact running time of an individual program or function. We only need to know that each function (of the right type) corresponds to a mathematical function in the complexity class P.

Theoretically, it would even be possible that an implementation in System BC does not directly reveal the polynomial-time algorithm for the implemented mathematical function. However, it seems to be straightforward to evaluate each System BC function in polynomial time.

**Examples** Let us first consider the function *conc* again.

$$\begin{array}{lcl} \mathit{conc} & : & \Box\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} \\ \mathit{conc} & \equiv & \lambda(n:\mathsf{nat})\ \lambda(m:\mathsf{nat}) \\ & & \mathsf{rec}\ n\ \{z \hookrightarrow m \\ & & \qquad \mid \mathsf{s}_0(x_1)\ \mathsf{with}\ y_1 \hookrightarrow \mathsf{s}_0(y_1) \\ & & \qquad \mid \mathsf{s}_1(x_2)\ \mathsf{with}\ y_2 \hookrightarrow \mathsf{s}_1(y_2)\} \end{array}$$

Since we iterate over the first argument $n$. In the type rule *BC:Rec*, there are no restrictions on the base case $e_0$. So we are free to use $m$. In the recursive cases, the recursive results $y_i$ are restricted to be used in safe positions. However, the successor constructors have type $\mathsf{nat} \to \mathsf{nat}$ and can thus be used with safe arguments.

Let us consider the function *bexp*, which has exponential runtime and should not type in System BC. We first introduce a helper function.

$$\begin{array}{lcl} \mathit{doub} & : & \Box\mathsf{nat} \to \mathsf{nat} \\ \mathit{doub} & \equiv & \lambda(n:\mathsf{nat})\ \mathit{conc}(n)(n) \end{array}$$

The argument of the function *doub* has to be modal since it is used as the first argument of the function *conc*, which is modal. If we would aim to implement the function *bexp* as shown below then we are not able to type the function because the we would have to derive the judgments $n:\mathsf{nat}, x:\mathsf{nat}; y:\mathsf{nat} \vdash \mathit{doub}(y):\mathsf{nat}$, which fails because *doub* does not accept a safe argument.

$$\begin{array}{lcl} \textit{failed-bexp} & : & \Box\mathsf{nat} \to \mathsf{nat} \\ \textit{failed-bexp} & \equiv & \lambda(n:\mathsf{nat}) \\ & & \mathsf{rec}\ n\ \{z \hookrightarrow \mathsf{s}_1(\mathsf{z}) \\ & & \qquad \mid \mathsf{s}_0(x)\ \mathsf{with}\ y \hookrightarrow \mathit{doub}(y) \\ & & \qquad \mid \mathsf{s}_1(x)\ \mathsf{with}\ y \hookrightarrow \mathit{doub}(y)\} \end{array}$$

On the other hand, we are able to type the function *square(n)* below that computes an integer of size $n^2$.

$$\begin{array}{lcl} \mathit{square} & : & \Box\mathsf{nat} \to \mathsf{nat} \\ \mathit{square} & \equiv & \lambda(n:\mathsf{nat}) \\ & & \mathsf{rec}\ n\ \{z \hookrightarrow \mathsf{z} \\ & & \qquad \mid \mathsf{s}_0(x)\ \mathsf{with}\ y \hookrightarrow \mathit{conc}(n)(y) \\ & & \qquad \mid \mathsf{s}_1(x)\ \mathsf{with}\ y \hookrightarrow \mathit{conc}(n)(y)\} \end{array}$$

Here, we have to derive the judgment $n:\mathsf{nat}, x:\mathsf{nat}; y:\mathsf{nat} \vdash \mathit{conc}(n)(y):\mathsf{nat}$, which is possible because $n$ is in the modal context and the second argument of *conc* is safe.

**Expressivity**

**Definition.** *We say that a function $h : \mathbb{N}^k \to \mathbb{N}$ is definable in System BC there is an expression $e_h : \Box\mathrm{nat} \to \cdots \to \Box\mathrm{nat} \to \mathrm{nat}$ such that $e(\widetilde{n_1}) \cdots (\widetilde{n_k}) \Downarrow \widetilde{h(n_1, \ldots, n_k)}$ for all $n_1, \ldots, n_k$.*

We need to have modal arguments in the type of $e_h$ since we would not allow iteration at all otherwise. System BC is designed to enforce the invariant stated in Theorem 3.

**Theorem 3.** *Let $e$ be an expression such that $\Delta; \Gamma \vdash e : \mathrm{nat}$ for $\Delta = x_1 : \mathrm{nat}, \ldots, x_k : \mathrm{nat}$ and $\Gamma = y_1 : \mathrm{nat}, \ldots, y_\ell : \mathrm{nat}$. Let $f_e : \mathbb{N}^k \times \mathbb{N}^\ell \to \mathbb{N}$ be the induced function of $e$, that is,*

$$[\widetilde{n_1}, \ldots, \widetilde{n_k}, \widetilde{m_1}, \ldots, \widetilde{m_\ell} / \vec{x}, \vec{y}]e \Downarrow \widetilde{f_e(\vec{n}, \vec{m})} \text{ for all } \vec{n}, \vec{m}\,.$$

*Then there exists a polynomial $p$ such that $|f_e(\vec{n}, \vec{m})| \leq p(|\vec{n}|) + \max(|\vec{m}|)$.*

Theorem 3 can be proved by induction on the definition of the function set. The intuition is that the size of the result of the function is polynomial in the normal parameters $\vec{x}$ but is constant in the safe parameters $\vec{y}$.

The main result of System BC is the following theorem.

**Theorem 4.** *Let $h : \mathbb{N}^k \to \mathbb{N}$ be a function. Then $h$ definable in System BC if and only if $h$ is in the class FP.*

The proof of *only if*-direction of the theorem follows from an extension of Theorem 3. The proof of the *if*-direction is technical and involves the implementation of a simulator for polynomial time Turing machines.

**Higher-Type Recursion** A direct extension to higher-type recursion (like in System T) leads again to fast growing functions. The issue is that recursively defined objects can be used multiple times in the recursor. For example, you can compose a recursively-computed function with itself in the recursive case.[3]

It is possible to elegantly extend System P with linear function spaces to allow recursion at higher-types while maintaining the characterization of FP [Hof97b]. We are not discussing the details here since LFPL, which is described later, also allows higher-type recursion.

## 5 LFPL

While System BC is powerful enough to express all *functions* in FP, it cannot implement all polynomial time *algorithms*. Safe recursion suppresses the use of recursive results in auxiliary recursive iterations. This restriction makes programming in System BC rather inconvenient. One could even argue that it is a typical pattern of a polynomial-time computation to use a single recursive call and to perform a lower-degree computation on the recursive result in each iteration. This pattern is for example present in the insertion sort algorithm that is implemented below with an iterator for lists, which is introduced later in this section.

$$\mathit{isort} \equiv \lambda(x : L(\tau))\ \mathsf{iter}\ x\ \{\mathsf{nil} \hookrightarrow \mathsf{nil} \mid \mathsf{cons}(a, \_)\ \mathsf{with}\ y \hookrightarrow \mathit{insert}(a, y)$$

Insertion sort is not expressible using safe recursion. The problem is that safe recursion uniformly treats recursive computations as size-increasing by a polynomial factor. However, this is not the case in general. The function *insert(x,$\ell$)* increases the size of the list $\ell$ only by a constant, which leads not only to a polynomial-time computation but also to the non-size increasing function *isort*.

One of the purposes of safe recursion is to prevent super-polynomial growth of (binary) numerals. Could we gain something by taking this idea to the extreme by preventing growth completely? Hofmann [Hof99, Hof02] gave a positive answer to this question by developing the concept of *non-size increasing computation* to prevent functions from computing data

[3]It is a good exercise to implement a function with exponential growth in System BC with recursion at higher types.

structures that are larger than the sum of the sizes of their arguments. To keep track of this property we use an *affine type system*. It seems to be very restrictive to only allow non-size increasing computation but it allows us to use natural recursion schemes: If we combine non-size increasing computation with structural recursion (at higher types!) then we obtain a characterization of the non-size increasing functions in the class FP. In particular, we obtain a characterization of the class P through functions with boolean result types. Moreover, combining non-size increasing computation with general recursion leads to a characterization of the class EXP, that is, the union of the classes $\mathsf{DTIME}(2^{p(n)})$ over all polynomials $p$. Cook showed that EXP is identical to the class of functions that can be computed in linear space with an unbounded stack.

In this lecture, we focus on the version with structural recursion that is call *LFPL* (linearly-typed functional programming language). We also use Booleans and lists instead of the binary numerals. Binary numerals can be implemented as lists of Booleans. We use lists mainly for presentation purposes and to be able to implement interesting examples. All the aforementioned results equally apply to the version of LFPL with binary numerals instead of lists and Booleans, where numerals are treated exactly like binary lists in the type system.

**Lists, Booleans, and diamonds** Our goal is to design a type system that prevents size-increases. An affine type system seems to be a good starting point because it eliminates size increases through multiple uses of variables. As we have seen in the insertion sort example, we however want to allow some harmless size increases like in the function *insert*. More generally, we need to be able to construct new data. However, this should only be allowed if some other data is destructed.

The types and expressions of LFPL are defined as follows.

$$\begin{array}{rcl} \tau & ::= & \mathrm{bool} \\ & & \Diamond \\ & & \tau_1 \multimap \tau_2 \\ & & \tau_1 \otimes \tau_2 \\ & & L(\tau) \end{array}$$

The type $\tau_1 \multimap \tau_2$ is the affine arrow type and a value of type $\Diamond$ can be viewed as a permission to increase the size of a data structure or, more operationally, as a memory cell that is used to store a new element of the data structure. Lists with elements of type $\tau$ have the $L(\tau)$ and $\tau_1 \otimes \tau_2$ is the type of affine pairs (also call multiplicative conjunction).

$$\begin{array}{rcll} e & ::= & x & x \\ & & \mathrm{lam}\{\tau\}(x.e) & \lambda(x:\tau)e \\ & & \mathrm{app}(e_1;e_2) & e_1(e_2) \\ & & \mathrm{tt} & \mathsf{true} \\ & & \mathrm{ff} & \mathsf{false} \\ & & \mathrm{if}(e;e_1;e_2) & \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\ & & \mathrm{pair}(e_1;e_2) & \langle e_1, e_2 \rangle \\ & & \mathrm{letp}(e_1;x_1,x_2.e_2) & \mathsf{letp}\ \langle x_1, x_2 \rangle = e_1\ \mathsf{in}\ e_2 \\ & & \mathrm{nil} & \mathsf{nil} \\ & & \mathrm{cons}(e_1;e_2;e_3) & \mathsf{cons}\,(e_1,e_2,e_3) \\ & & \mathrm{iter_L}\{e_0;x_1,x_2,y.e_1\}(e) & \mathsf{iter}\ e\ \{\mathsf{nil} \hookrightarrow e_0 \mid \mathsf{cons}\,(x_1,x_2,\_)\ \mathsf{with}\ y \hookrightarrow e_1\} \\ & & \blacklozenge & \end{array}$$

There is no introduction form for values of type $\Diamond$ at the surface syntax. We add an expression $\blacklozenge$ that formally acts at such an introduction form. However, we only need it to define the evaluation judgment. In an affine or linear setting, it is more appropriate to use the elimination form $\mathrm{letp}(e_1;x_1,x_2.e_2)$ for pairs instead of projections. The introduction forms for lists are the usual cons and nil. However, cons accepts three arguments instead of the usual two. We will discuss this further below but, in a nutshell, the third argument is a permission (of type $\Diamond$) to increase the size of the list. Lists are eliminated with the iterator $\mathrm{iter_L}\{e_0;x_1,x_2,y.e_1\}(e)$.

$\boxed{\Gamma \vdash e : \tau \qquad \text{“expression } e \text{ has type } \tau \text{ under context } \Gamma\text{”}}$

$$\frac{}{x : \tau \vdash x : \tau}\ (\text{D:}\textsc{Var}) \qquad \frac{\Gamma, x{:}\tau' \vdash e : \tau}{\Gamma \vdash \mathrm{lam}\{\tau'\}(x.e) : \tau' \multimap \tau}\ (\text{D:}\textsc{Abs})$$

$$\frac{\Gamma_1 \vdash e_1 : \tau' \multimap \tau \qquad \Gamma_2 \vdash e_2 : \tau'}{\Gamma_1, \Gamma_2 \vdash \mathrm{app}(e_1; e_2) : \tau}\ (\text{D:}\textsc{App}) \qquad \frac{c \in \{\mathrm{tt}, \mathrm{ff}\}}{\cdot \vdash c : \mathrm{bool}}\ (\text{D:}\textsc{BConst})$$

$$\frac{\Gamma_1 \vdash e : \mathrm{bool} \qquad \Gamma_2 \vdash e_1 : \tau \qquad \Gamma_2 \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathrm{if}(e; e_1; e_2) : \tau}\ (\text{D:}\textsc{Cond}) \qquad \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash \mathrm{pair}(e_1; e_2) : \tau_1 \otimes \tau_2}\ (\text{D:}\textsc{Pair})$$

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \otimes \tau_2 \qquad \Gamma_2, x_1 : \tau_1, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathrm{letp}(e_1; x_1, x_2.e_2) : \tau}\ (\text{D:}\textsc{Letp}) \qquad \frac{}{\cdot \vdash \mathrm{nil} : L(\tau)}\ (\text{D:}\textsc{Nil})$$

$$\frac{\Gamma_1 \vdash e_1 : \lozenge \qquad \Gamma_2 \vdash e_2 : \tau \qquad \Gamma_3 \vdash e_3 : L(\tau)}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathrm{cons}(e_1; e_2; e_3) : L(\tau)}\ (\text{D:}\textsc{Cons})$$

$$\frac{\Gamma_1 \vdash e : L(\tau') \qquad \Gamma_2 \vdash e_1 : \tau \qquad x_1 : \lozenge, x_2 : \tau', y : \tau \vdash e_2 : \tau}{\Gamma_1, \Gamma_2 \vdash \mathrm{iter}_{\mathrm{L}}\{e_0; x_1, x_2, y.e_1\}(e) : \tau}\ (\text{D:}\textsc{Iter}) \qquad \frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau}\ (\text{D:}\textsc{Weak})$$

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau \qquad \text{heap-free}(\tau)}{\Gamma, x : \tau \vdash [x, x / x_1, x_2]e : \tau}\ (\text{D:}\textsc{Cntr}) \qquad \frac{}{\cdot \vdash \blacklozenge : \lozenge}\ (\text{D:}\textsc{Dia})$$

**Figure 6:** Static semantics of LFPL.

Interestingly, we do not get access to the tail of the list during the iteration. This is necessary to prevent size increases and we revisit this point when we discuss the respective type rule.

The type judgment $\Gamma \vdash e : \tau$ is defined in Figure 6. The type system substructural with a weakening rule D:WEAK and is therefore affine. We also control sharing of heap-free types in the rule D:CNTR. The heap-free types do not contain diamonds and can be freely shared. The judgement heap-free($\tau$) is defined by the following rules.

$$\frac{}{\text{heap-free}(\mathrm{bool})}\ (\text{HF-B}) \qquad \frac{\text{heap-free}(\tau_1) \qquad \text{heap-free}(\tau_2)}{\text{heap-free}(\tau_1 \otimes \tau_2)}\ (\text{HF-T})$$

It is clear that list types should not be heap-free since it would violate the non-size increasing property to create, say, a pair of lists $\langle x, x \rangle$. For similar reasons, function types cannot be heap-free. Can you give an example that shows that it would be unsound to declare function types to be heap-free?

In the rule D:CONS, we need a head, a tail, and diamond. Conversely, in the rule D:ITER the diamonds in the list can be used in the iterative step, binding it to $x_2$ in the judgment $x_1 : \lozenge, x_2 : \tau', y : \tau \vdash e_2 : \tau$. The context $\Gamma_1, \Gamma_2$ can be only used in $e$ and $e_1$ but not in $e_2$ since it is executed multiple times, which would mean to use the variables multiple times.[4] Similarly only the recursive result $y$, the diamond $x_1$, and the head $x_2$ are available in $e_2$. Allowing the use of the tail of the input $e$, like in the recursor, would mean to use it multiple times, which we have to prevent.

[4]Note that Hofmann did not allow the use of variables from the context in $e_0$. However, it is sound to alleviate this restriction.

**Dynamic semantics** The dynamic semantics can be given using a standard evaluation dynamics $e \Downarrow v$. We only give a few key rules.

$$\frac{}{\text{nil} \Downarrow \text{nil}}\ (\text{E:N}\textsc{il}) \qquad \frac{e_1 \Downarrow \blacklozenge \quad e_2 \Downarrow v_2 \quad e_3 \Downarrow v_3}{\text{cons}(e_1; e_2; e_3) \Downarrow \text{cons}(\blacklozenge; v_2; v_3)}\ (\text{E:C}\textsc{ons})$$

$$\frac{e \Downarrow \text{nil} \qquad e_0 \Downarrow v}{\text{iter}_{\text{L}}\{e_0; x_1, x_2, y.e_1\}(e) \Downarrow v}\ (\text{E:I}\textsc{terL}\text{-N})$$

$$\frac{e \Downarrow \text{cons}(\blacklozenge; v_2; v_3) \qquad \text{iter}_{\text{L}}\{e_0; x_1, x_2, y.e_1\}(v_3) \Downarrow v_r \qquad [\blacklozenge, v_2, v_r / x_1, x_2, y]e_1 \Downarrow v}{\text{iter}_{\text{L}}\{e_0; x_1, x_2, y.e_1\}(e) \Downarrow v}\ (\text{E:I}\textsc{terL}\text{-C})$$

**Examples** To see how the system works, let us consider a few examples. First, let us consider list append, which can be implemented as follows. The type $\tau$ is fixed but arbitrary (this is not a polymorphic language).

$$\begin{array}{lcl} \textit{append} & : & L(\tau) \multimap L(\tau) \multimap L(\tau) \\ \textit{append} & \equiv & \lambda(xs : L(\tau))\ \lambda(ys : L(\tau)) \\ & & \quad \mathsf{iter}\ xs\ \{\mathsf{nil} \hookrightarrow ys \\ & & \qquad\quad \mid \mathsf{cons}\,(d, x, \_)\ \mathsf{with}\ y \hookrightarrow \mathsf{cons}\,(d, x, y)\} \end{array}$$

The implementation looks almost exactly like the usual implementation. The only difference is that we have to provide a diamond $d$ when constructing the list $\mathsf{cons}\,(d, x, y)$. We cannot create this diamond out of thin air but instead have to obtain it from the destruction of the head element of the list $xs$. The type derivation proves that $|\textit{append}(xs)(ys)| \le |xs| + |ys|$.

As a negative example, consider the following function *failed-append2*, which appends each element in the first argument twice.

$$\begin{array}{lcl} \textit{failed-append2} & : & L(\tau) \multimap L(\tau) \multimap L(\tau) \\ \textit{failed-append2} & \equiv & \lambda(xs : L(\tau))\ \lambda(ys : L(\tau)) \\ & & \quad \mathsf{iter}\ xs\ \{\mathsf{nil} \hookrightarrow ys \\ & & \qquad\quad \mid \mathsf{cons}\,(d, x, \_)\ \mathsf{with}\ y \hookrightarrow \mathsf{cons}\,(d, x, \mathsf{cons}\,(d, x, y))\} \end{array}$$

This function is clearly not non-size increasing since $|\textit{failed-append2}(xs)(ys)| = 2|xs| + |ys|$. So we should be able to derive a typing in LFP. In fact, there are two potential violations of linearity in the expression $\mathsf{cons}\,(d, x, \mathsf{cons}\,(d, x, y))$ as $d$ and $x$ are used twice. We would have to use the rule D:C<span style="font-variant:small-caps">ntr</span> to justify the "contraction", that is, the duplication of the variables. However, D:CNTR is only applicable to variables with *heap-free* types. Since $\Diamond$ is not heap-free we cannot apply the contraction rule to the variable $d$ and cannot derive a type. Similarly, we cannot apply contraction to the variable $x$ of type $\tau$ for an arbitrary type $\tau$.

To be able to type a function like *failed-append2*, we have to restrict the element type $\tau$ to a heap-free type like bool and add additional diamonds to the input.

$$\begin{array}{lcl} \textit{append2} & : & L(\text{bool} \otimes \Diamond) \multimap L(\text{bool}) \multimap L(\text{bool}) \\ \textit{append2} & \equiv & \lambda(xs : L(\text{bool}))\ \lambda(ys : L(\text{bool})) \\ & & \quad \mathsf{iter}\ xs\ \{\mathsf{nil} \hookrightarrow ys \\ & & \qquad\quad \mid \mathsf{cons}\,(d_0, x, \_)\ \mathsf{with}\ y \hookrightarrow \\ & & \qquad\qquad \mathsf{letp}\,\langle x', d_1 \rangle = x\ \mathsf{in} \\ & & \qquad\qquad \mathsf{cons}\,(d_0, x', \mathsf{cons}\,(d_1, x', y))\} \end{array}$$

The function *append2* can be typed in LFPL. Since the variable $x'$ has type bool we can apply contraction to use it twice. Two have two diamonds available per list element of the first input list, we add an additional diamond to the input. However, it seems like the function is now size increasing. Nevertheless, we still have $|\textit{append2}(xs)(ys)| \le |xs| + |ys|$ if we use the right definition of the size function $|\cdot|$ that counts the number of diamonds in a datastructure. Then $|xs| = 2n$ where $n$ is the length of the list. The formal definition of size follows later.

**Expressivity of LFPL** We can define binary numerals using lists of Booleans as follows.

$$\begin{array}{lcl} \widehat{0} & = & \text{nil} \\ \widehat{2n+1} & = & \text{cons}(\blacklozenge;\text{ff};\widehat{n}) \\ \widehat{2(n+1)} & = & \text{cons}(\blacklozenge;\text{tt};\widehat{n}) \end{array}$$

**Definition.** *We say that a function $h : \mathbb{N}^k \to \mathbb{N}$ is definable in LFPL there is an expression $e_h$ : $L(\text{bool}) \multimap \cdots \multimap L(\text{bool}) \multimap L(\text{bool})$ such that $e(\widehat{n_1}) \cdots (\widehat{n_k}) \Downarrow h(\widehat{n_1,\ldots,n_k})$ for all $\vec{n}$.*

The main theorem is that exactly the non-size increasing functions in the class FP are definable in LFPL.

**Theorem 5.** *A function $h : \mathbb{N}^k \to \mathbb{N}$ is definable in LFPL if and only if $h$ is in FP and $|h(\vec{n})| \leq \sum_{1\leq i\leq k} |n_i|$.*

The proof of the "only if" direction is not too difficult. If $h$ is definable then we can show $|h(\vec{n})| \leq \sum_{1\leq i\leq k} |n_i|$ using a denotational model based on length spaces that as outlined below. From this model, we also derive that the size of each intermediate data-structure that appears in the computation is bounded $K = \sum_{1\leq i\leq k} |n_i|$. The polynomial time bound then follows because each program can only nest a constant number of iterations.

The "if" direction is more involved. Hofmann [Hof02] showed how to simulate an arbitrary Turing machine whose time complexity is bounded by a polynomial.

**Non-size increasing functions** Here, we are sketching denotational semantics for LFPL. This is not required to understand the language or the soundness result. However, it is instructive to develop an intuition.

We can give a set-theoretic interpretation to types as follows.

$$\begin{array}{lcl} [\![\text{bool}]\!] & = & \{\text{tt},\text{ff}\} \\ [\![\lozenge]\!] & = & \{\blacklozenge\} \\ [\![\tau_1 \multimap \tau_2]\!] & = & [\![\tau_1]\!] \to [\![\tau_2]\!] \\ [\![\tau_1 \otimes \tau_2]\!] & = & [\![\tau_1]\!] \times [\![\tau_2]\!] \\ [\![L(\tau)]\!] & = & \{[v_1,\ldots,v_n] \mid n \in \mathbb{N}, v_i \in [\![\tau]\!]\} \end{array}$$

The size $s_v(\tau)$ of a value $v : \tau$ is defined as follows.

$$\begin{array}{lcl} s_{\text{bool}}(v) & = & 0 \\ s_{\lozenge}(\blacklozenge) & = & 1 \\ s_{\tau_1 \multimap \tau_2}(f) & = & \min\{c \mid \forall v \in [\![\tau_1]\!].s_{\tau_2}(f(v)) \leq c + s_{\tau_1}(v)\} \\ s_{\tau_1 \otimes \tau_2}(\langle v_1, v_2\rangle) & = & s_{\tau_1}(v_1) + s_{\tau_2}(v_2) \\ s_{L(\tau)}([v_1,\ldots,v_n]) & = & n + \sum_{1\leq i\leq n} s_\tau(v_i) \end{array}$$

Note that $s_{\tau_1 \multimap \tau_2}(f)$ can actually be undefined if the set is empty and no minimum exists. As a result, sizes of other types can be undefined as well. Notice that a function $f \in [\![\tau_1 \multimap \tau_2]\!]$ is non-size increasing if and only if $s_{\tau_1 \multimap \tau_2}(()f) = 0$.

Denotations of terms are non-size increasing in the following sense.

**Theorem 6.** *Let $\Gamma \vdash e : \tau$ and let $V$ be an environment for $\Gamma$ such that $s_{\Gamma(x)}(V(x))$ is defined for each $x \in dom(\Gamma)$. Then*

$$s_\tau([\![e]\!](V)) \leq \sum_{x \in dom(\Gamma)} s_{\Gamma(x)}(V(x))$$

**General recursion** If we replace structural recursion (the list iterator) with general recursion (fixed points) in LFPL then the definable functions correspond to the class EXP [Hof02], that is, the union of the classes $\mathsf{DTIME}(2^{p(n)})$ over all polynomials $p$.

It might be surprising at first that the possibility of non-termination does not make the language to powerful. However, Cook showed that EXP is identical to the class of functions that can be computed in linear space with an unbounded stack, which provides good intuition for the result.

**Beyond complexity classes** Programming in LFPL is very natural and it is interesting to study it as programming language beyond it our immediate goal of representing complexity classes. For example, we can compile the first-order fragment for LFPL to C programs without *malloc* (i.e., memory allocation).[5]

The biggest limitation of LFPL is that function are non-size increasing. However, if we look beyond the representation of functions $h : \mathbb{N}^k \to \mathbb{N}$ then the limitation can be lifted by adding function arguments of type $\Diamond$. An example, is the function *double*: $L(\Diamond \otimes \text{bool}) \multimap L(\text{bool})$ below that appends a list to itself.

$$double \equiv \lambda(x : L(\Diamond \otimes \text{bool}))\, append(x)(x)$$

The idea of adding additional diamonds to the input is the basis of automatic amortized analysis, which we will discuss in the next lecture.

## References

[BC92] Stephen Bellantoni and Stephen A. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions. *Computational Complexity*, 2:97–110, 1992.

[Har12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[Hof97a] Martin Hofmann. An application of category-theoretic semantics to the characterisation of complexity classes using higher-order function algebras. *Bull. Symbolic Logic*, 3(4):469–486, 12 1997.

[Hof97b] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Computer Science Logic, 11th International Workshop, CSL '97, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers*, pages 275–294, 1997.

[Hof99] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 464–473, 1999.

[Hof02] Martin Hofmann. The Strength of Non-Size Increasing Computation. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 260–269, 2002.

[5] Such a compiler would make for a great final project.