LFPL: Revisited and Mechanized

Nathaniel Glover
Carnegie Mellon University, Pittsburgh, PA, USA

Jan Hoffmann
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Hofmann (1999) introduced the functional programming language LFPL to characterize the functions computable in polynomial time using an affine type system. LFPL enables a natural programming style, including nested recursion, and has inspired the development of type systems for automatic cost analysis, linear dependent type theories, and efficient memory management in functional programming languages. Despite its prominence, there does not exist a self-contained presentation, let alone a full mechanization, of LFPL and its core metatheory.

This article presents a modern account and mechanization of LFPL and its metatheory with the goal of being self-contained and accessible while streamlining the strongest-known soundness and completeness results. The soundness proof works with the language LFPL$^+$, which extends LFPL with additional language features. The proof is novel, adapting a technique by Aehlig and Schwichtenberg (2002) to construct explicit polynomials that bound the cost of an LFPL$^+$ expression with respect to a big-step cost semantics. The completeness proof shows that LFPL programs can simulate polynomial-time Turing machines while only relying on restricted forms of linear functions and lists. It has the same structure as the original proof by Hofmann (2002) but greatly simplifies the core argument with a novel stack-like data structure that is implemented with first-class functions and lists. The mechanization includes the full soundness and completeness proofs, and serves as one of the first case studies of mechanized metatheory in the recently developed proof assistant Istari.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Complexity theory and logic; Theory of computation → Linear logic

Keywords and phrases Type Theory, Implicit Computational Complexity, Affine Logic

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.18348212` [14]

1 Introduction

The goal of implicit computational complexity (ICC) is to provide a programming language which characterizes a given complexity class. Concretely, this means that a procedure is representable in the language if and only if it inhabits the desired complexity class. ICC has been extensively studied for many complexity classes including P [17, 18, 20, 12], PSPACE [20, 12, 13], EXPTIME [19], and LOGSPACE [31, 30]. Such languages often achieve their desired complexity class via modifications of standard linear or affine type systems [9].

One prominent such programming language for the complexity classes P and FP is Martin Hofmann's Linear Function Programming Language (LFPL). It was first defined in 1999, when Hofmann [17] showed that LFPL is polynomial-time sound, that is, all functions definable in LFPL belong to FP. In 2002, Hofmann [19] proved the logical inverse of this statement, polynomial-time completeness. LFPL is notable for characterizing FP while also supporting a natural programming style and giving rise to the development of automatic amortized resource analysis [16]. Recently, there has been renewed interest in LFPL. Atkey [3] extends LFPL with dependent types and unrestricted computation at the type level, creating a language equipped with the powerful guarantees of dependent type theory while still enforcing that computation is polynomial-time at the term level. Lorenzen et al. [24] utilize the type system of LFPL in their language Koka to ensure that certain functional programs can be executed fully in-place.

Despite its prominence, we are not aware of a fully self-contained presentation of LFPL and its core metatheory. The original presentation by Hofmann [17] contains a semantic soundness proof but only a weak completeness result for linear-space polynomial-time Turing machines. A somewhat informal completeness result for the original LFPL is given as a corollary in Hofmann [19], where polynomial time is not the main focus. Aehlig and Schwichtenberg [2] provide a syntactic soundness proof for a small-step semantics and a weaker completeness proof for a strong extension of LFPL. Atkey [3] presents a similar completeness proof, as well as a soundness proof for LFPL extended with dependent type theory. Lorenzen et al. [24] present LFPL's type system without the metatheory. None of these works contains both a soundness proof and a completeness proof of the original LFPL without extensions.

In this article, we give a modern account and mechanization [14] of LFPL and its metatheory. Our primary goal is for this article to be a self-contained and accessible resource for those who are interested in learning about LFPL, with all of our results guaranteed correct by the mechanization. We have witnessed firsthand the necessity of such a resource. It was challenging to collect all this information on LFPL from across the various different papers we have mentioned, and during the process we discovered some errors in Hofmann's completeness proof [19], the only full completeness proof that we are aware of.

Our presentation of LFPL involves a denotational semantics similar to the original given by Hofmann [17] as well as a big-step operational cost semantics, which, to our knowledge, was not developed for LFPL. An advantage of both semantics – which greatly reduce the complexity of our mechanization – is that they both avoid making use of syntactic substitution, which is a notoriously tedious operation to support and reason about in proof assistants.

The mechanized completeness proof is a simplified and streamlined version of the one given by Hofmann [19]. The main novelty is the implementation of a stack-like data structure in LFPL that allows us to avoid much of the complexity of Hofmann's proof as well as sidestep the errors we discovered in that proof. To our knowledge, ours is the strongest completeness result for LFPL. Hofmann also gave a weaker completeness theorem for linear-time Turing machines only [17]. Both Aehlig and Schwichtenberg [2] and Atkey [3] provide full completeness proofs but for powerful extensions of LFPL, thereby weakening the completeness result. Like Hofmann's version [19], our completeness proof works with a minimal version of LFPL, using the fewest tools to simulate a polynomial-time Turing machine.

The mechanized soundness proof is inspired by Aehlig and Schwichtenberg [2]. We give a modern presentation of their result for a big-step operational cost semantics and an extended language LFPL^+. Using this cost semantics allows us to reason about concrete program executions and avoids the substitution-related tedium that arises when mechanizing a small-step semantics. While a big-step operational cost semantics is further away from a concrete machine model, it is well-known how to link it to small-step semantics, which have been shown to be reasonable with respect to lower-level semantic models [6, 1]. Notably, for each LFPL^+ term, the proof constructs a concrete polynomial that bounds the execution cost of the term as defined by the cost semantics. To our knowledge, this is the strongest known soundness result for LFPL; we strengthen it by extending the language with several features such as a built-in stack data structure and lazy products.

In summary, we make the following contributions: the first complete presentation of the LFPL metatheory, a novel completeness proof, a novel soundness proof, and the first complete mechanization of the LFPL metatheory. The novelties in the proofs make them accessible and amenable to mechanization. Our desire to mechanize the LFPL metatheory is partially responsible for driving us to discover these proofs. Our completeness proof resolves some errors we discovered in the only other completeness proof of equal strength to ours.

2 LFPL

In Gödel's System T, recursion is controlled in such a way as to ensure the language is still terminating. This is known as primitive or structural recursion. The only way to write variable-time programs in LFPL is using structural recursion, but that alone is not enough to achieve polynomial time. This pseudocode is an example of an exponential-time program using only structural recursion:

```
double zero = zero
double (succ n) = succ (succ (double n))

exp zero = succ zero
exp (succ n) = double (exp n)
```

The main idea of LFPL is to enforce non-size-increasing computation. In the above example, the exponential behavior relies on the fact that `double` increases the size of its input. LFPL forbids such size-increasing computations. The non-size-increasing property is enforced with an affine type system, which treats input size as a limited resource that cannot be duplicated. Formally, size is represented by a type $\Diamond$, so that an element of type $\Diamond$ corresponds to one unit of size. The natural number n requires n elements of type $\Diamond$ to construct. These resources, which we shall henceforth call diamonds, are returned when n is consumed. The program `double` is no longer possible to write; in the `succ` case, one diamond is made available to us, yet we need two diamonds due to the two calls to `succ`.

In contrast to other polynomial-time systems that might restrict iteration such that nested iterations are impossible, LFPL enables a fairly natural programming style. While many polynomial-time languages are complete with respect to polynomial-time *computations*, they are often unable to express many natural *algorithms*. Consider this pseudocode for an inefficient identity function on lists:

```
append nil l2 = l2
append (n :: l1) l2 = n :: append l1 l2

id l = append (append l nil) nil
```

In a system that disallows nested iterations, expressing a program like this can be difficult or impossible when the output from a call to the iterative procedure like `append` is fed into another such call. Meanwhile, LFPL has no such restrictions, so the above program can be directly translated into LFPL with minimal modifications. Due to this flexibility, LFPL supports many other natural algorithms such as linear-time list reversal, insertion sort, and integer division by a constant.

2.1 Properties of LFPL

The defining property of LFPL is that computation is non-size-increasing. Consider a list resulting from the evaluation of an LFPL program with some lists as inputs. The non-size-increasing theorem intuitively states that the length of this output list is no greater than

Type	A, B	$::=$	Term	M, N	$::=$
	$\Diamond$	diamond		x	variable
	$\mathtt{1}$	unit		$\langle\rangle$	unit intro
	$A + B$	sum		$i \cdot M \quad (i \in \{1, 2\})$	sum intro
	$A \otimes B$	tensor		$\mathtt{case}\ M\ \{x_1.N_1 \mid x_2.N_2\}$	sum elim
	$A \multimap B$	arrow		$\langle M_1, M_2\rangle$	tensor intro
	$\mathtt{L}(A)$	list		$\mathtt{letp}\ \langle x_1, x_2\rangle = M\ \mathtt{in}\ N$	tensor elim
				$\lambda x.M$	arrow intro
				$M\ N$	arrow elim
				$\mathtt{nil}$	list intro
				$\mathtt{cons}\ (M_d; M_h; M_t)$	list intro
				$\mathtt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\}$	list elim

Figure 1 LFPL's type and term syntax.

the sum of the lengths of the input lists. We present this in detail in Section 4.2, but for now the main takeaway is that LFPL does not support problematic size-increasing functions like `double`. As expected of the defining property of the language, the non-size-increasing theorem is a main tool in showing that LFPL is polynomial-time sound.

The most complete version of the polynomial-time soundness theorem states that any function on the natural numbers expressible in LFPL is also computable by a polynomial-time Turing machine. We only prove one step in the path from LFPL to Turing machines: In a high-level cost semantics, given an LFPL program and some inputs with size n, there exists a polynomial $P : \mathbb{N} \to \mathbb{N}$ such that evaluating the program costs at most $P(n)$. Our theorem is the only step in this path that contains reasoning specific to LFPL: Given this theorem, it is a well-known and standard procedure to justify the polynomial bound for our high-level cost semantics in terms of a low-level model such as a Turing machine [1, 6].

The polynomial-time completeness theorem states the logical inverse of soundness: Any function on the natural numbers that is computable by a polynomial-time Turing machine is expressible in LFPL. As stated, this is not quite true for LFPL; `double` is computable in linear time yet is not expressible in LFPL. We can fix this by instead proving completeness for all polynomial-time computable functions that do not increase the size of their input. Perhaps surprisingly, we see in Theorem 17 that the more general statement of completeness is only *barely* false: There is a very reasonable perspective from which LFPL is able to express all polynomial-time computations, even the size-increasing ones.

2.2 Syntax and Typing Rules

The syntax of LFPL is given in Figure 1. Other than the type $\Diamond$, and the additional argument and variable appearing in the list introduction and elimination forms respectively, the syntax looks like System T with lists. The first argument M_d in the second list introduction form $\mathtt{cons}\ (M_d; M_h; M_t)$ is intended to have type $\Diamond$; to add an element to a list, we are forced to pay a diamond. Likewise, the first variable x_d in the list elimination $\mathtt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\}$ is intended to have type $\Diamond$; when eliminating a list, we earn back the diamonds we paid to construct it, allowing us to construct new lists during recursion. In this way, we use elements of type $\Diamond$ to control the lengths of lists, ensuring that the number of diamonds available to us bounds the maximum length of a list that we can construct at any point.

$$\textsc{Ty:Var}\ \frac{}{\Gamma, x : A \vdash x : A} \qquad \textsc{Ty:UnitI}\ \frac{}{\Gamma \vdash \langle\rangle : \mathtt{1}} \qquad \textsc{Ty:ArrowI}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B} \qquad \textsc{Ty:ArrowE}\ \frac{\Gamma \vdash M : A \multimap B \qquad \Gamma' \vdash N : A}{\Gamma; \Gamma' \vdash M\ N : B}$$

$$\textsc{Ty:SumI}_i\ \frac{\Gamma \vdash M : A_i}{\Gamma \vdash i \cdot M : A_1 + A_2} \qquad \textsc{Ty:SumE}\ \frac{\Gamma \vdash M : A_1 + A_2 \qquad \Gamma', x_1 : A_1 \vdash N_1 : B \qquad \Gamma', x_2 : A_2 \vdash N_2 : B}{\Gamma; \Gamma' \vdash \mathtt{case}\ M\ \{x_1.N_1 \mid x_2.N_2\} : B}$$

$$\textsc{Ty:TensorI}\ \frac{\Gamma_1 \vdash M_1 : A_1 \qquad \Gamma_2 \vdash M_2 : A_2}{\Gamma_1; \Gamma_2 \vdash \langle M_1, M_2\rangle : A_1 \otimes A_2} \qquad \textsc{Ty:TensorE}\ \frac{\Gamma \vdash M : A_1 \otimes A_2 \qquad \Gamma', x_1 : A_1, x_2 : A_2 \vdash N : B}{\Gamma; \Gamma' \vdash \mathtt{letp}\ \langle x_1, x_2\rangle = M\ \mathtt{in}\ N : B}$$

$$\textsc{Ty:ListI}_1\ \frac{}{\Gamma \vdash \mathtt{nil} : \mathtt{L}(A)} \qquad \textsc{Ty:ListI}_2\ \frac{\Gamma_d \vdash M_d : \Diamond \qquad \Gamma_h \vdash M_h : A \qquad \Gamma_t \vdash M_t : \mathtt{L}(A)}{\Gamma_d; \Gamma_h; \Gamma_t \vdash \mathtt{cons}\,(M_d; M_h; M_t) : \mathtt{L}(A)}$$

$$\textsc{Ty:ListE}\ \frac{\Gamma \vdash M : \mathtt{L}(A) \qquad \Gamma' \vdash N_1 : B \qquad \cdot, x_d : \Diamond, x_h : A, x_t : B \vdash N_2 : B}{\Gamma; \Gamma' \vdash \mathtt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\} : B}$$

Figure 2 LFPL's affine typing rules.

The typing judgement is written $\Gamma \vdash M : A$ and read as "M has type A under context Γ". Importantly, LFPL has an *affine* type system, so contraction is not allowed and we must partition the context when we type an expression containing sequential subexpressions. The typing rules are given in Figure 2. Notice that for $\mathtt{cons}\,(M_d; M_h; M_t)$ to be typed under $\Gamma_d; \Gamma_h; \Gamma_t$, we must have $\Gamma_d \vdash M_d : \Diamond$. Likewise, notice that for $\mathtt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\}$ to be typed, we must type N_2 under a context with $x_d : \Diamond$.

Another important subtlety is in Rule Ty:ListE, where N_2 is typed under a context containing *only* the variables x_d, x_h, and x_t. One is not allowed to use additional variables from the surrounding environment. This is because N_2 represents the body of the structural recursor and therefore, despite appearing once statically, may be run many times during the execution of a program. Consequently, even if N_2 would statically appear to use a variable once, at runtime that variable would be used many times and thus violate the main principle of affine type theory. For example, N_2 could take just one diamond from the context and use it to create a list of length 10, which would break LFPL's non-size-increasing property.

2.3 Denotational Semantics

To reason about the behavior of LFPL programs, we need to reason about evaluating functions which may take lists as input. For instance, we will see in Section 2.4 a closed LFPL term $\mathtt{reverse} : \mathtt{L}(A) \multimap \mathtt{L}(A)$, but we can never actually "run" `reverse` on a nonempty input list, since that would require a closed term of type $\Diamond$ and the whole point of LFPL's type system is to make that impossible. Thus, it is difficult to study LFPL by itself; we instead need to reason about terms in a domain where $\Diamond$ is inhabited.

We now give a simple set-theoretic denotational semantics. To each type A we assign a set $[\![A]\!]$, and to each well-typed term $\Gamma \vdash M : A$ we assign an element $[\![M]\!]\,(\mathcal{V})$ of $[\![A]\!]$, where $\mathcal{V}$ is an environment such that $\mathcal{V}(x) \in [\![A]\!]$ for every $x : A \in \Gamma$, written as $\mathcal{V} \in [\![\Gamma]\!]$. Per the above discussion, we define $[\![\Diamond]\!]$ as a nonempty set to ensure $\Diamond$ is semantically inhabited.

$$\llbracket \Diamond \rrbracket = \{\blacklozenge\} \qquad \llbracket A \otimes B \rrbracket = \llbracket A \rrbracket \times \llbracket B \rrbracket \qquad \llbracket \mathtt{L}(A) \rrbracket = \llbracket A \rrbracket^*$$
$$\llbracket \mathtt{1} \rrbracket = \{*\} \qquad \llbracket A + B \rrbracket = \llbracket A \rrbracket \sqcup \llbracket B \rrbracket \qquad \llbracket A \multimap B \rrbracket = \llbracket A \rrbracket \to \llbracket B \rrbracket$$

$$\begin{array}{rcll}
\llbracket x \rrbracket(\mathcal{V}) & = & \mathcal{V}(x) & \\
\llbracket \langle\rangle \rrbracket(\mathcal{V}) & = & * & \\
\llbracket i \cdot M \rrbracket(\mathcal{V}) & = & (i, \llbracket M \rrbracket(\mathcal{V})) & \\
\llbracket \mathtt{case}\ M\ \{x_1.N_1 \mid x_2.N_2\} \rrbracket(\mathcal{V}) & = & \llbracket N_i \rrbracket(\mathcal{V}[x_i \mapsto v]) & \text{where } \llbracket M \rrbracket(\mathcal{V}) = (i, v) \\
\llbracket \langle M_1, M_2 \rangle \rrbracket(\mathcal{V}) & = & (\llbracket M_1 \rrbracket(\mathcal{V}), \llbracket M_2 \rrbracket(\mathcal{V})) & \\
\llbracket \mathtt{letp}\ \langle x_1, x_2 \rangle = M\ \mathtt{in}\ N \rrbracket(\mathcal{V}) & = & \llbracket N \rrbracket(\mathcal{V}[x_1 \mapsto v, x_2 \mapsto u]) & \text{where } \llbracket M \rrbracket(\mathcal{V}) = (v, u) \\
\llbracket \lambda x.M \rrbracket(\mathcal{V}) & = & \lambda\, v.\ \llbracket M \rrbracket(\mathcal{V}[x \mapsto v]) & \\
\llbracket M\ N \rrbracket(\mathcal{V}) & = & \llbracket M \rrbracket(\mathcal{V})(\llbracket N \rrbracket(\mathcal{V})) & \\
\llbracket \mathtt{nil} \rrbracket(\mathcal{V}) & = & [\,] & \\
\llbracket \mathtt{cons}\,(M_0; M_1; M_2) \rrbracket(\mathcal{V}) & = & \llbracket M_1 \rrbracket(\mathcal{V}) :: \llbracket M_2 \rrbracket(\mathcal{V}) & \\
\llbracket \mathtt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\} \rrbracket(\mathcal{V}) & = & \mathrm{iter}(\llbracket M \rrbracket(\mathcal{V}), \llbracket N_1 \rrbracket(\mathcal{V}), f) & \\
 & & \text{where } f(v, w) = \llbracket N_2 \rrbracket(\cdot[x_d \mapsto \blacklozenge, x_h \mapsto v, x_t \mapsto w]) &
\end{array}$$

Figure 3 LFPL's denotational semantics.

To define the semantics of the LFPL list type, we make use of the Kleene star operator. Given a set S, the set S^* is the set of finite strings of elements of S. We write $[\,]$ for the empty string, and given $x \in S$ and $\ell \in S^*$, we write $x :: \ell$ for the element of S^* obtained by putting x before the sequence of elements in ℓ. Given $\ell \in S^*$, let $|\ell| \in \mathbb{N}$ denote the length of the string ℓ. Lastly, for every set T, we inductively define the iteration operator on S^*:

$$\mathrm{iter}([\,], b, f) = b$$
$$\mathrm{iter}(x :: \ell, b, f) = f(x, \mathrm{iter}(\ell, b, f))$$

For the semantics of the sum type, we make use of the disjoint union (or coproduct) operator $S \sqcup T := (\{1\} \times S) \cup (\{2\} \times T)$. The definitions of $\llbracket A \rrbracket$ and $\llbracket M \rrbracket(\mathcal{V})$ are given in Figure 3, inspired by Hofmann's set-theoretic interpretation [19]. Notice the argument of type $\Diamond$ to `cons` is ignored; the presence of $\Diamond$ in lists is purely a means of statically enforcing polynomial-time computation, so the dynamic semantics for `cons` need not involve $\blacklozenge$.

The notion of size and the non-size-increasing property were originally formulated for the denotational semantics [17]. The basic idea is to define a partial function $\mathsf{size}_A : \llbracket A \rrbracket \to \mathbb{N}$ by induction on A, intuitively counting the number of diamonds within the input. For example, $\mathsf{size}_\Diamond(\blacklozenge) = 1$. The reason for partiality is that $\mathsf{size}_{A \multimap B}(f)$ is defined as the maximum difference (which might not exist) between $\mathsf{size}_B(f(x))$ and $\mathsf{size}_A(x)$, where x ranges over all $x \in \llbracket A \rrbracket$ such that $\mathsf{size}_A(x)$ and $\mathsf{size}_B(f(x))$ are defined. The non-size-increasing theorem then proves simultaneously that $\mathsf{size}_A(\llbracket M \rrbracket(\mathcal{V}))$ is defined whenever the sizes of all values in $\mathcal{V}$ are defined and that the size of $\llbracket M \rrbracket(\mathcal{V})$ is bounded by the sum of the sizes of the values in $\mathcal{V}$. In this paper, we provide an equivalent definition using an operational semantics in Section 4.2. It is simpler because the definition is total, avoiding the complexities of dealing with partial functions while remaining sufficient for proving soundness.

2.4 Example Programs

To build intuition about the expressivity of LFPL, we implement some examples of functions that demonstrate how to program in the language. Firstly, given any element type A, we implement efficient list reversal. For readability, we use the concrete syntax

```
rec M | nil ⇒ N1 | cons (xd, xh, xt) ⇒ N2
```

for $\mathtt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\}$ in the examples; and a similar syntax for sum elimination.

▶ **Example 1** (Linear-Time Reverse)**.** We can implement the standard linear-time list reversal, often seen in introductory material on functional programming.

```
revAppend : L(A) ⊸ L(A) ⊸ L(A)
revAppend = lam l1 . rec l1
| nil ⇒ lam l2 . l2
| cons (d, x, r) ⇒ lam l2 . r (cons (d, x, l2))

reverse : L(A) ⊸ L(A)
reverse = lam l1 . revAppend l1 nil
```

In the abstract syntax of Figure 1, `revAppend` would be written as:

$$\lambda\ell_1.\,\texttt{lrec}\ \ell_1\,\{\lambda\ell_2.\ell_2 \mid d.x.r.\,\lambda\ell_2.\,r\ (\texttt{cons}\,(d;x;\ell_2))\}$$

This term has type $\mathtt{L}(A) \multimap \mathtt{L}(A) \multimap \mathtt{L}(A)$. Using the rules in Figure 2, we can see this via the following partial derivation. The ellipses are straightforward to fill in.

$$\dfrac{\dfrac{\cdots \qquad \cdots \qquad \dfrac{\dfrac{\dfrac{}{d:\Diamond \vdash d:\Diamond} \quad \dfrac{}{x:A \vdash x:A} \quad \dfrac{}{\ell_2:\mathtt{L}(A) \vdash \ell_2:\mathtt{L}(A)}}{\cdots \quad d:\Diamond, x:A, \ell_2:\mathtt{L}(A) \vdash \texttt{cons}\,(d;x;\ell_2):\mathtt{L}(A)}}{d:\Diamond, x:A, r:\mathtt{L}(A) \multimap \mathtt{L}(A), \ell_2:\mathtt{L}(A) \vdash r\ (\texttt{cons}\,(d;x;\ell_2)):\mathtt{L}(A)}}{d:\Diamond, x:A, r:\mathtt{L}(A) \multimap \mathtt{L}(A) \vdash \lambda\ell_2.\,r\ (\texttt{cons}\,(d;x;\ell_2)):\mathtt{L}(A) \multimap \mathtt{L}(A)}}{\ell_1:\mathtt{L}(A) \vdash \texttt{lrec}\ \ell_1\,\{\lambda\ell_2.\ell_2 \mid d.x.r.\,\lambda\ell_2.\,r\ (\texttt{cons}\,(d;x;\ell_2))\}:\mathtt{L}(A) \multimap \mathtt{L}(A)}}{\cdot \vdash \lambda\ell_1.\,\texttt{lrec}\ \ell_1\,\{\lambda\ell_2.\ell_2 \mid d.x.r.\,\lambda\ell_2.\,r\ (\texttt{cons}\,(d;x;\ell_2))\}:\mathtt{L}(A) \multimap \mathtt{L}(A) \multimap \mathtt{L}(A)}$$

From this derivation, it is straightforward to derive $\cdot \vdash \texttt{reverse} : \mathtt{L}(A) \multimap \mathtt{L}(A)$. To prove that `reverse` is correct, one can use the semantic clauses in Figure 3 to prove $[\![\texttt{reverse}]\!]\,(\cdot) : [\![A]\!]^* \to [\![A]\!]^*$ is the reversal function on $[\![A]\!]^*$.

▶ **Example 2** (List Case Analysis)**.** Importantly, both for convenience and for use in Example 3, we can inspect whether a list is empty, which is often presented as pattern matching in standard functional programming languages. For brevity in the concrete syntax, we allow for pattern matching on tuple variables at their binding sites in place of the more verbose `letp`.

```
lfold : 1 + ◊ ⊗ A ⊗ L(A) ⊸ L(A)
lfold = lam x . case x .
| inj1 _ ⇒ nil
| inj2 (d, x, xs) ⇒ cons (d, x, xs)

lunfold : L(A) ⊸ 1 + ◊ ⊗ A ⊗ L(A)
lunfold = lam x . rec x .
| nil ⇒ inj1 <>
| cons (d, x, r) ⇒ inj2 (d, x, lfold r)
```

▶ **Example 3** (List Suspension)**.** Given a list, we can temporarily suspend its values and obtain the diamonds within the list for use elsewhere. We, of course, need those diamonds to un-suspend the list.

```
susp : L(A) ⊸ ((L(1) ⊸ L(A)) ⊗ L(1))
susp = lam x . rec x .
| nil ⇒ (lam _ . nil, nil)
| cons (d, x, r) ⇒ letp (f, m) = r in
  ((lam n . case (lunfold n) .
     | inj1 _ ⇒ nil
     | inj2 (d', _, n') ⇒ cons (d', x, f n')), cons (d, <>, m))
```

This example is important for the completeness proof, so it is worth stating its semantics: If $x \in [\![A]\!]^*$, then $[\![\texttt{susp}]\!]\,(\cdot)\,(x) = (f, m)$, where $f(m) = x$. Note that m is the unique element of $[\![\texttt{1}]\!]^*$ with length $|x|$, so we recover x as long as we give f enough diamonds.

2.5 Contraction

The defining trait of an affine type system is that contraction, the use of a variable more than once, is not permitted. The importance of disallowing contraction in LFPL is clear; if we could use a variable of type $\Diamond$ more than once, then we could easily create lists of arbitrary length and break the non-size-increasing property. Despite this restriction, there are types which still enjoy contraction. We define the judgement $\Diamond$-free on types as follows:

$$\frac{}{\texttt{1}\ \Diamond\text{-free}} \qquad \frac{A\ \Diamond\text{-free} \qquad B\ \Diamond\text{-free}}{A + B\ \Diamond\text{-free}} \qquad \frac{A\ \Diamond\text{-free} \qquad B\ \Diamond\text{-free}}{A \otimes B\ \Diamond\text{-free}}$$

Going by rule induction on the judgement $A\ \Diamond$-free for a given type A, it is straightforward to define a closed term $\texttt{dup}_A : A \multimap A \otimes A$ that returns two copies of its input, effectively implementing contraction at A.

A common approach to safely reintroducing contraction into affine type systems is the exponential modality !. The syntax and typing rules for it might look something like:

$$\frac{!\Gamma \vdash M : A}{!\Gamma \vdash \texttt{wrap}(M) : !A} \qquad \frac{\Gamma \vdash M : !A}{\Gamma \vdash \texttt{unwrap}(M) : A} \qquad \frac{\Gamma, x : !A, y : !A \vdash M : B}{\Gamma, x : !A \vdash [x/y]M : B}$$

By $!\Gamma$ we mean the context obtained by replacing $x : A$ with $x : !A$ for every x in Γ. Commonly, this modality would also contain a rule for weakening, but LFPL is affine and thus already supports weakening at every type. It is tempting to add this modality to LFPL; if we say that all values of type $!A$ have size 0, then these operations are non-size-increasing. Unfortunately, that is not enough to guarantee polynomial-time soundness. Consider the following pseudocode for LFPL (extended with the !-modality, where $[\![!A]\!] = [\![A]\!]$):

```
fnExp : L(1) ⊸ !(L(1) ⊸ L(1)) ⊸ !(L(1) ⊸ L(1))
fnExp = lam l1 . lam f . rec l1
| nil ⇒ f
| cons (_, _, r) ⇒ wrap (lam x . unwrap(r) (unwrap(r) x))
```

Then, $[\![\texttt{fnExp}]\!]\,(\cdot)\,(n)(f)$ is the function $f^{2^{|n|}}$ that calls f on its input $2^{|n|}$ times. Thus, we can use `fnExp` to implement functions that are not polynomial-time computable.

3 Polynomial-Time Completeness

We now use the denotational semantics to state what it means for LFPL to be complete with respect to polynomial-time computation on the natural numbers. The interpretation $[\![\texttt{L(1)}]\!] = \{*\}^*$ of unit lists can be viewed as the set of unary-encoded natural numbers. However, it is also possible to use the binary representation $[\![\texttt{L}(\texttt{1} + \texttt{1})]\!]$ for $\mathbb{N}$. Instead of choosing a particular encoding of the natural numbers, we assume the digits of a natural number are encoded by a type A such that $A\ \Diamond$-free. We require $A\ \Diamond$-free because plenty of polynomial-time computable functions (even non-size-increasing ones) duplicate the digits of their input, which is impossible in LFPL if A contains diamonds.

▶ **Theorem 4** (Size-Restricted Polynomial-Time Completeness). *Suppose $A\ \Diamond$-free. Then, for every polynomial-time computable function $f : [\![A]\!]^* \to [\![A]\!]^*$ such that $|f(x)| \leq |x|$ for all $x \in [\![A]\!]^*$, there exists a closed term $M : \texttt{L}(A) \multimap \texttt{L}(A)$ such that $[\![M]\!]\,(\cdot) = f$.*

This theorem was originally proven for $A = \mathtt{1}$ [19]. The constraint that f does not increase the length of its input is necessary because of the non-size-increasing property enforced upon LFPL functions. As we see in Section 3.4, this constraint can be removed if we do not enforce that the output type of M is an LFPL list.

Proving Theorem 4 boils down to simulating the polynomial-time Turing machine that computes the function f, say with polynomial bound $P : \mathbb{N} \to \mathbb{N}$. This immediately presents two challenges. The first challenge is to represent the tape of the machine, which for an input list of size n could contain up to $P(n) + n$ values by the time the machine halts. We only have n diamonds from the input list, yet we have to store up to $P(n) + n$ elements.

The second challenge is to simulate the Turing machine for $P(n)$ steps. At first glance, it seems that LFPL is only suited for performing kn iterations given n diamonds and some constant k. This can be achieved for $k = 1$ by iterating down a list and building it back up during the iteration process, and then it is easy to repeat this for any constant k number of times. This is useful but does not help to perform, say, n^2 iterations.

Both of these challenges are somewhat unique to LFPL's completeness proof compared to completeness proofs for other languages for P. For example, the challenge in the proof of the Cons-free system from the work of Jones [21] lies mostly in simulating the polynomially many iterations of the machine's step function but not in providing sufficient space for the tape. This is similar to the completeness proof for LFPL$^+$ and the extension of LFPL considered by Atkey [3], which provide lists without size restrictions or support for iteration.

3.1 The Bounded Stack Data Structure

We first address the challenge of simulating the tape of the Turing machine. A common approach to representing the tape of the Turing machine in a functional language is to use a data structure consisting of a head element and two stacks, each representing one side of the tape relative to the head element [28]. If we use the type A for tape symbols, then the type of our tape data structure could be $\mathtt{L}(A) \otimes A \otimes \mathtt{L}(A)$. However, in LFPL, we cannot actually use the above data structure. Given n diamonds, available in an input list of length n, it can only hold $n + 1$ values. Instead, we construct a type $S(A)$ that depends on the bounding polynomial P and supports a stack-like interface, as long as it does not have to hold more than $P(n)$ elements. Then, we use a value of type $S(A) \otimes A \otimes S(A)$ to encode the tape.

We have in fact already seen the first key insight into how to construct $S(A)$ in Example 3. It shows that we do not need any diamonds to store data; we just need them to interact with it. Using this idea, we can design our data structure so that it does not store diamonds and its push/pop functions temporarily borrow diamonds to interface with the suspended data.

The second key insight is that we do not need to un-suspend the entire data structure to interface with it. Since both push and pop work at the front of the stack, we just need to be able to un-suspend enough of our data structure to expose the front. The number of diamonds we need for these operations depends on P and n.

▶ **Definition 5** (Bounded Stack Interface). *Given $k \in \mathbb{N}$ and an element type A, a k-stack implementation is a type S together with the following closed terms:*

- $\mathtt{empty} : S$
- $\mathtt{push} : (\mathtt{L}(\mathtt{1}))^k \multimap A \otimes S \multimap (\mathtt{L}(\mathtt{1}))^k \otimes (S \otimes (A + \mathtt{1}))$
- $\mathtt{pop} : (\mathtt{L}(\mathtt{1}))^k \multimap S \multimap (\mathtt{L}(\mathtt{1}))^k \otimes (S \otimes (\mathtt{1} + A))$

By $(\mathtt{L}(\mathtt{1}))^k$ we mean the type of k-tuples $\mathtt{L}(\mathtt{1}) \otimes \cdots \otimes \mathtt{L}(\mathtt{1})$. The diamonds stored in each list are used to un-suspend parts of the data structure and returned after the respective operation. The reason for using a k-tuple of lists instead of just one big list becomes apparent

in the proof of Lemma 8, where the use of such a tuple greatly simplifies the implementation. Both `push` and `pop` also return a stack and a sum. The left branch of the sum is intended to signal failure of the operation (in which case the input stack is returned unmodified), and the right branch signals success (in which case the returned stack has been successfully modified). For `push`, which takes in an element and a stack, failure means that the input stack is full, and so the input element is returned to us. For `pop`, failure means the input stack is empty.

This interface is inspired by the array-like data structure used in Hofmann's original completeness proof [19]. The key difference is that his array data structure has read and write operations requiring an index, given as a binary LFPL number (i.e., $\mathtt{L}(\mathtt{1}+\mathtt{1})$). The extra work of dealing with this index significantly complicates both the implementation and use of the data structure. Arrays are more general, but our stack is easier to implement and sufficient for representing a Turing machine's tape, so we believe it is the better choice of data structure for the completeness theorem.

To formalize our intuition of how `push` and `pop` should behave, we use the denotational semantics. We first define the element $m_{n,k} \in [\![(\mathtt{L}(\mathtt{1}))^k]\!]$ to be the tuple $(\ell, \ldots, \ell)$, where ℓ is the unique element of $[\![\mathtt{L}(\mathtt{1})]\!]$ such that $|\ell| = n$. This way, $m_{n,k}$ contains exactly nk diamonds.

▶ **Definition 6** (Bounded Stack Correctness). *Let $k \in \mathbb{N}$. Suppose $(S, \mathtt{empty}, \mathtt{push}, \mathtt{pop})$ is a k-stack implementation with element type A. Given a function $B : \mathbb{N} \to \mathbb{N}$, we say this implementation is correct and bounded by B if the following holds for all $n \in \mathbb{N}$:*

- *There exists a relation $V_n \subseteq [\![S]\!]$ of valid stack states.*
- *There exists a function $I_n : [\![S]\!] \to [\![A]\!]^*$ that returns the list of items in the stack.*
- *Let $s = [\![\mathtt{empty}]\!](\cdot)$. Then, $V_n(s)$ holds and $I_n(s) = [\,]$.*
- *Let $f = [\![\mathtt{push}]\!](\cdot)$. Then, for all $x \in [\![A]\!]$ and $s \in [\![S]\!]$ such that $V_n(s)$ holds, we have:*
  - *If $|I_n(s)| = B(n)$ then $f(m_{n,k})(x, s) = (m_{n,k}, s, (1, x))$.*
  - *If $|I_n(s)| < B(n)$ then $f(m_{n,k})(x, s) = (m_{n,k}, s', (2, *))$, where $V_n(s')$ holds and $I_n(s') = x :: I_n(s)$.*
- *Let $f = [\![\mathtt{pop}]\!](\cdot)$. Then, for all $s \in [\![S]\!]$ such that $V_n(s)$ holds:*
  - *If $I_n(s) = [\,]$ then $f(m_{n,k})(s) = (m_{n,k}, s, (1, *))$.*
  - *If $I_n(s) = x :: \ell$ then $f(m_{n,k})(s) = (m_{n,k}, s', (2, x))$, where $V_n(s')$ holds and $I_n(s') = \ell$.*

The intended use of the k-stack data structure is to choose $n \in \mathbb{N}$ and always pass $m_{n,k}$ to the `push` and `pop` operations. Definition 6 ensures that, in this use case, the stack can hold at most $B(n)$ items. Behavior is otherwise undefined. Whenever `push` and `pop` receive $m_{n,k}$ as input, they return $m_{n,k}$ as the first element of their output, thus making it available for future stack operations.

3.2 Bounded Stack Implementations

We now provide an implementation of a bounded stack and ways to build more implementations on top of pre-existing ones. This ultimately leads to a stack bounded by our choice of a polynomial P, which is exactly what we need to represent the tape of a polynomial-time Turing machine. While most of the proofs in this section focus on the high-level intuition and avoid writing or reasoning about concrete LFPL programs, these programs and other details are formalized in the mechanization [14].

▶ **Lemma 7** (Constant Stack Construction). *For any element type A and $c \in \mathbb{N}$, there exists a correct 0-stack implementation which is bounded by $B(n) = c$.*

Proof. We take the implementation type to be $S = (1 + A)^c$. Intuitively, a left injection represents an open space in the stack, and a right injection represents a stored element. A stack $(x_1, \ldots, x_c) \in [\![S]\!]$ is considered valid when there is some index $0 \leq j \leq c$ such that $x_i = (1, *)$ for $1 \leq i \leq j$, and x_i is a right injection for $j < i \leq c$.

To find the head of the stack, find the first i such that $x_i = (2, a)$ for some $a \in [\![A]\!]$; the value a is the head element. If i is 1, then the stack is full. If no such i exists, the stack is empty. Since c is a constant, this procedure of locating the stack head does not require iteration, so no diamonds are required for `push` and `pop`. ◀

The proof of Lemma 7 already contains intuition for how to construct stacks with non-constant bounds. In the proof of Lemma 8, instead of storing data in a fixed-size tuple, we store it in a list. That list can be "suspended" as seen in Example 3, so that it does not require any diamonds to store.

▶ **Lemma 8** (Inductive Stack Construction). *Fix an element type A and some $k \in \mathbb{N}$. Suppose $(S, \texttt{empty}, \texttt{push}, \texttt{pop})$ is a k-stack implementation with element type A that is correct and bounded by $B : \mathbb{N} \to \mathbb{N}$. Then, there is a $(k+1)$-stack implementation with element type A that is correct and bounded by $B'(n) = nB(n)$.*

Proof. We choose $S' = \texttt{L(1)} \multimap \texttt{L}(S)$, the type of suspended lists with element type S (called "sub-stacks"), to be our implementation type. Let ℓ be the unique element of $[\![\texttt{L(1)}]\!]$ with $|\ell| = n$. To unsuspend a stack $s \in [\![S']\!]$ we just call $s(\ell)$. We say s is valid whenever $s(\ell) = s_1 :: \cdots :: s_n :: [\,]$, where s_i is valid for all $1 \leq i \leq n$ and there exists some index $0 \leq j \leq n$ such that s_i is empty for $1 \leq i < j$, the substack s_j is unconstrained, and s_i is full for $j < i \leq n$. The items $I'(S')$ are the appended item lists $I(s_i)$ of each sub-stack.

It remains to correctly implement the three stack operations. Let us call them $\texttt{empty}'$, $\texttt{push}'$, and $\texttt{pop}'$. The operation $\texttt{empty}'$ is the function of type $\texttt{L(1)} \multimap \texttt{L}(S)$ which always returns the empty list. When implementing $\texttt{push}'$, we are given as our source of diamonds an input of type $(\texttt{L(1)})^{k+1} = \texttt{L(1)} \otimes (\texttt{L(1)})^k$, so we can use the first element of this tuple, call it ℓ, to un-suspend the input stack $s \in [\![S']\!]$ and obtain a list of sub-stacks $s(\ell) = s_1 :: \cdots :: s_n :: [\,]$. Starting from the last element of this list, s_n, attempt pushing to it using the function `push`. If this succeeds, we are done. Otherwise, s_n must be full, so we should try s_{n-1}. We can do this all the way up to s_1; if push still fails on s_1 then the whole stack must be full. This process can be implemented using LFPL's list recursor. Afterwards, we use Example 3 to re-suspend the modified list and return the results.

The implementation of $\texttt{pop}'$ is similar, but instead we start from s_1. If popping s_1 fails, it must be empty, so we should try s_2. We go all the way down the list, and if popping s_n still fails, then the whole stack must be empty. Note this is nontrivial to implement with the list recursor, since starting from s_1 and going down to s_n is the reverse of the recursor's execution order. We solve this with a similar version of the trick used in Example 1. ◀

Now that we have established the main result of this subsection, a straightforward induction combines the above two lemmas, producing stack implementations with monomial bounds.

▶ **Lemma 9** (Monomial Stack Construction). *Given $c, k \in \mathbb{N}$, there exists a k-stack implementation with element type A that is correct and bounded by $B(n) = cn^k$.*

Since every polynomial $P : \mathbb{N} \to \mathbb{N}$ is eventually bounded by some monomial $B(n) = cn^k$, it is technically enough to stop here, which is what Hofmann's original proof does [19]. However, this introduces an annoying edge case when $n = 0$ because $P(0)$ could be nonzero

but $B(0) = 0$ for any choice of c and k. Due to the complexity of the array data structure, we suspect it is not worth the trouble to avoid this problem. Our simpler stack data structure allows us to sidestep this annoyance and combine monomially bounded stacks to get polynomially bounded stacks with a few more relatively straightforward implementations.

▶ **Lemma 10** (Weakened Stack Construction). *Fix an element type A and some $k \in \mathbb{N}$. Suppose $(S, \texttt{empty}, \texttt{push}, \texttt{pop})$ is a k-stack implementation with element type A that is correct and bounded by $B : \mathbb{N} \to \mathbb{N}$. Then, there is a $(k+1)$-stack implementation with element type A that is correct and bounded by B.*

Proof. This construction is relatively obvious; if `push` and `pop` need only kn diamonds, accepting $(k+1)n$ diamonds instead is no problem. We can reuse the implementation type S as well as the code for `empty`. For the new stack operations $\texttt{push}'$ and $\texttt{pop}'$, they simply make use of the fact that $m_{n,k+1} = (\ell, m_{n,k})$ for some $\ell \in [\![\mathtt{L}(\mathtt{1})]\!]$. They just ignore ℓ and call `push` and `pop` (respectively) with $m_{n,k}$. ◀

The following lemma, which shows how to combine two stacks into one stack bounded by the sum of their bounds, is the main tool by which we are able to construct stacks with arbitrary polynomial bounds from stacks with monomial bounds. It is straightforward to do this construction for our stack data structure, but it is not clear how to do it for Hofmann's array data structure, which only yields monomially bounded arrays [19].

▶ **Lemma 11** (Additive Stack Construction). *Fix an element type A and some $k \in \mathbb{N}$. For $i \in \{1, 2\}$, suppose $(S_i, \texttt{empty}_i, \texttt{push}_i, \texttt{pop}_i)$ is a k-stack implementation with element type A that is correct and bounded by $B_i : \mathbb{N} \to \mathbb{N}$. Then, there is a k-stack implementation with element type A that is correct and bounded by $B(n) = B_1(n) + B_2(n)$.*

Proof. We take the implementation type to be $S = S_1 \otimes S_2$. A stack $(s_1, s_2) \in [\![S]\!]$ is considered valid when s_1 and s_2 are valid, and s_1 is only nonempty when s_2 is full. Intuitively, s_1 contains the first section of our stack and s_2 contains the second. The operations are very similar to that of Lemma 7 with $c = 2$. To pop the stack, we first try popping s_1. If that fails, it must have been empty, so we pop s_2. To push some $x \in [\![A]\!]$ onto this stack, we first try pushing to s_2. If that fails, it means s_2 is full, so we push to s_1. ◀

▶ **Lemma 12** (Polynomial Stack Construction). *Given a polynomial $P : \mathbb{N} \to \mathbb{N}$ with degree k, there exists a k-stack implementation with element type A that is correct and bounded by P.*

Proof. Write $P(n)$ as $\sum_{i=0}^{k} c_i n^i$. For each $0 \le i \le k$, we can use Lemma 9 to obtain an i-stack with bound $B_i(n) = c_i n^i$. Then, we repeatedly use Lemma 10 to weaken all of them into k-stacks with the same bounds. Finally, we repeatedly use Lemma 11 to obtain the desired k-stack whose bound is the sum of the bounds of the weakened monomial stacks. ◀

3.3 Polynomial Iteration

We now address the challenge of simulating the iteration of the transition function of a P-time Turing machine for $P(n)$ steps when given an input list of length n. Additionally, we need to simulate the transition function itself. Fortunately, both of these tasks require comparably simpler solutions than the issue of polynomial storage presented in the previous section. To iterate a function on itself $P(n)$ times, we use the next two lemmas, which are inspired by Proposition 4.1 in Hofmann's work [17].

▶ **Lemma 13** (Linear Iteration). *Let $f : A \otimes \mathtt{L}(\mathtt{1}) \multimap A \otimes \mathtt{L}(\mathtt{1})$ be a closed term. Then, there exists a closed term $f^\sharp : A \otimes \mathtt{L}(\mathtt{1}) \multimap A \otimes \mathtt{L}(\mathtt{1})$ such that $[\![f^\sharp]\!](\cdot)(x, n) = ([\![f]\!](\cdot))^{|n|}(x, n)$.*

Proof. Consider the following pseudocode:

```
g : L(1) ⊸ A ⊗ L(1) ⊸ A ⊗ L(1)
g = lam m . rec m
| nil ⇒ lam s . s
| cons (d, u, r) ⇒ lam (x, n) . f (r (x, cons (d, u, n)))
```

We can then define $f^\sharp = \lambda s.\, \texttt{letp}\ \langle x, n\rangle = s\ \texttt{in}\ g\ n\ \langle x, \texttt{nil}\rangle$. ◀

▶ **Lemma 14** (Polynomial Iteration). *Let $f : A \otimes \mathtt{L}(\mathtt{1}) \multimap A \otimes \mathtt{L}(\mathtt{1})$ be a closed term and $P : \mathbb{N} \to \mathbb{N}$ be a polynomial. Then, there exists a closed term $f^{\sharp P} : A \otimes \mathtt{L}(\mathtt{1}) \multimap A \otimes \mathtt{L}(\mathtt{1})$ such that $[\![f^{\sharp P}]\!](\cdot)(x, n) = ([\![f]\!](\cdot))^{P(|n|)}(x, n)$.*

Proof. For monomials, we can repeat the $\sharp$ operation from Lemma 13, e.g., $[\![(f^\sharp)^\sharp]\!](\cdot)(x, n) = ([\![f]\!](\cdot))^{|n|^2}(x, n)$. To add these monomials together to obtain any polynomial, we can just use function composition, since $g^m \circ g^k = g^{m+k}$ for any $g : X \to X$ and $m, k \in \mathbb{N}$. ◀

We now address the task of encoding the transition function of the Turing machine. Suppose we are dealing with a tape alphabet that includes one blank symbol a_0 and the rest of the symbols are drawn from the set $[\![A]\!]$, where A $\Diamond$-free. Additionally, suppose the set of internal states of the Turing machine includes one halting state q_0 and the rest of the states are drawn from the set $[\![Q]\!]$, where again Q $\Diamond$-free. Then, the transition function g of the Turing machine has the following signature:

$$g : [\![Q]\!] \times (\{a_0\} \sqcup [\![A]\!]) \to (\{q_0\} \sqcup [\![Q]\!]) \times (\{a_0\} \sqcup [\![A]\!]) \times \{\Leftarrow, \Rightarrow\}$$

If $(q', a', d) = g(q, a)$, the intent is that q is the current state of the Turing machine, a is the symbol under the tape head, q' is the new state after taking a step, a' is the symbol to overwrite a with, and d is the direction (left or right) to shift the head. So, the type of the LFPL term encoding g should be:

$$Q \otimes (\mathtt{1} + A) \multimap (\mathtt{1} + Q) \otimes (\mathtt{1} + A) \otimes (\mathtt{1} + \mathtt{1})$$

The input and output sets of g are both denotations of $\Diamond$-free types. The following two lemmas show LFPL is complete with respect to computations only involving types A such that A $\Diamond$-free, and thus that g can be implemented.

▶ **Lemma 15.** *Suppose A $\Diamond$-free. Then, for every $a \in [\![A]\!]$, there exists a closed term $M_a : A$ such that $[\![M_a]\!](\cdot) = a$.*

▶ **Lemma 16.** *Suppose A and B are types such that A $\Diamond$-free and, for every $b \in [\![B]\!]$, that there exists a closed term $M_b : B$ such that $[\![M_b]\!](\cdot) = b$. Then, for every $f : [\![A]\!] \to [\![B]\!]$, there exists a closed term $M_f : A \multimap B$ such that $[\![M_f]\!](\cdot) = f$.*

Both of these are straightforward to prove by rule induction on A $\Diamond$-free. By combining the two, we obtain a closed term M_g encoding the transition function g of the Turing machine.

3.4 Simulating the Turing Machine

It only remains to put everything together. For this subsection, assume we have a function $f : [\![A]\!]^* \to [\![A]\!]^*$ computable by a Turing machine on a tape with alphabet $\{a_0\} \sqcup [\![A]\!]$, with internal state set $\{q_0\} \sqcup [\![Q]\!]$ and transition function g. Further assume the machine always halts in $P(n)$ steps on an input of length n, where $P : \mathbb{N} \to \mathbb{N}$ is a k-degree polynomial. Lastly, assume A $\Diamond$-free and Q $\Diamond$-free. This assumption is not restrictive; Turing machines have finite alphabets and finitely many states.

Our goal is to construct a closed term $M : \mathtt{L}(A) \to \mathtt{L}(A)$ such that $[\![M]\!](\cdot) = f$. Assuming an input of $\ell \in [\![A]\!]^*$ with length $n = |\ell|$, the body of the function for M only has access to n diamonds. We are going to need a tape data structure that can hold the input list and any additional memory used by the Turing machine, so up to $n + P(n)$ values, while only requiring n diamonds.

Since the Turing machine receives its input as a single list, giving us a pool of diamonds in the form $\mathtt{L}(\mathtt{1})$, yet stack operations require diamonds in the form $(\mathtt{L}(\mathtt{1}))^{k+1}$, we need a way to losslessly convert between diamonds collected into a list of type $\mathtt{L}(\mathtt{1})$ and diamonds in a divided form $(\mathtt{L}(\mathtt{1}))^{k+1} \otimes \mathtt{L}(\mathtt{1})$. The reverse direction is straightforward; just append all the lists together into one. The forward direction amounts to implementing unary division by the constant $k + 1$ in LFPL. This is nontrivial but not too different from an implementation of $\mathtt{divmod}_k : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ in a standard functional programming language, where $\mathtt{divmod}_k(n)$ returns the quotient and remainder of n by $k + 1$.

Consider the polynomial $P'(m) = (k + 1)(m + 1) + P((k + 1)(m + 1))$, which has degree at most $\max(1, k)$, which we further bound with $k + 1$ to avoid a degree-zero edge case. Let $(m, r) = \mathtt{divmod}_k(n)$ so that $0 \leq r \leq k$ and $n = m(k + 1) + r$. Then, $(k + 1)(m + 1) = m(k + 1) + (k + 1) > m(k + 1) + r = n$, implying $P'(m) \geq P(n) + n$. Therefore, we can apply Lemma 12 to P' to obtain a stack implementation for element type $\mathtt{1} + A$, say with underlying type S. These stacks hold up to $P'(m)$ values assuming we have $m(k + 1)$ diamonds available and therefore at least $P(n)$ values. So we can model the tape with a data structure of type $S \otimes (\mathtt{1} + A) \otimes S$, as discussed earlier.

This is another place where our stack data structure simplifies the argument from Hofmann's array-based tape implementation [19]. The array-based tape requires an index representing what element of the array is currently beneath the head of the tape. This index needs additional diamonds to store, creating the additional complexity of properly sharing the total n diamonds between the index and the tape operations.

We obtain the following more general completeness theorem alluded to earlier, which drops the size restriction of Theorem 4.

▶ **Theorem 17** (Polynomial-Time Completeness). *Let A be a type such that A $\Diamond$-free and let $P : \mathbb{N} \to \mathbb{N}$ be a polynomial with degree k. For every P-time computable function $f : [\![A]\!]^* \to [\![\mathtt{1} + A]\!]^*$, there exists a correct $(k + 1)$-stack implementation with element type $\mathtt{1} + A$, say with underlying type S and item functions $I_n : [\![S]\!] \to [\![\mathtt{1} + A]\!]^*$, and a closed term $M : \mathtt{L}(A) \multimap S$ such that for every $x \in [\![A]\!]^*$:*

$$f(x) = I_{|x|}\left([\![M]\!](\cdot)(x)\right)$$

In other words, M outputs a stack with items equal to $f(x)$.

Proof. To define M, we start with a function abstraction taking in the input $x : \mathtt{L}(A)$. Using the tape data structure discussed above, and the n diamonds from x, we can write the values of x onto the tape. Then, using the tools from Section 3.3, we take an LFPL implementation of the transition function g and iterate it on the tape (as well as an LFPL encoding of the initial machine state $q_0 \in [\![Q]\!]$) for a total of $P(n)$ steps. Finally, we return the stack representing the right half (or left; this choice just depends on the particulars of how we define a Turing machine's output) of the tape. ◀

By allowing M to output a stack rather than insisting on a list, we can implement *any* polynomial-time computation, even the size-increasing ones. Since the stack operations from Section 3.1 and the polynomial iterator from Lemma 14 do not permanently consume any

diamonds, we still have n diamonds left over after the procedure in the above proof. We can just use those to remove the first n values of the stack, store them in a list, and filter out the values representing blank symbols, thereby proving Theorem 4 as a corollary of Theorem 17.

3.5 Errors in the Original Proof

We originally intended to mechanize Hofmann's completeness proof [19], but thinking carefully about some of the details, we noticed several subtle errors and wanted to avoid the difficulties that come with resolving them. Thus, the mechanization effort led us to invent a simpler stack-like data structure for representing the tape of a Turing machine.

A main difficulty in the completeness proof is to represent the tape of a polynomial-time Turing machine in LFPL, say with polynomial P and degree k. Hofmann constructs a data structure (called a "storage device") that can hold only cn^k elements for a fixed c, as opposed to $P(n)$ elements, given temporary access to n diamonds. In Hofmann's encoding of the Turing machine, given an input list of length n, the first step is to construct a storage device capable of holding $P(2km)$ elements when given access to km temporary diamonds, which Hofmann does by finding a monomial $cm^k \geq P(2km)$. Then, m is taken to be $n/(2k)$, but the largest possible value for m is $\lfloor n/(2k) \rfloor$; any larger and we could potentially have $km > n/2$. This makes the chosen storage device unsuitable for the proof since it would mandate the availability of more than our $n/2$ diamonds. See below for why we have only $n/2$ rather than n diamonds. So, due to this upper bound on m, to have $cm^k \geq P(n)$ we actually need cm^k to bound $P(2k(m+1))$, which is one small error in the proof.

More importantly, there's the issue that $m = \lfloor n/(2k) \rfloor = 0$ when $n < 2k$, and thus $cm^k = 0$ while it may be that $P(0) > 0$, so Hofmann's construction fails for input lists of length less than $2k$ (as well as any inputs of length 0). This can be fixed by hard-coding each of these finitely many inputs. Nonetheless, neither this problem nor any potential solutions are discussed in the original proof. Our proof sidesteps this issue since our stack construction works for general polynomial bounds.

The second source of errors is that storage devices have an array-like interface, where the client can access any element by providing its index as a binary natural number. In LFPL, such an index requires some diamonds to store, whereas our stack-like interface does not have indices to worry about. In Hofmann's encoding of the Turing machine, the input list (say with length n) is immediately split into two lists of length $n/2$. One half is reserved for the storage device as discussed in the previous paragraphs. The purpose of the other half is not discussed in the proof, but it is necessary to store a storage device index representing the position of the tape head. The issue is that the tape could hold up to $P(n)$ elements, but the remaining diamonds can only represent up to $2^{n/2}$ binary indices, which is not necessarily greater than $P(n)$. This could be fixed by hard-coding more inputs or storing the index as a d-ary number for sufficiently large d. None of this is discussed in Hofmann's proof, and fortunately we are once again able to sidestep all of it with our stack data structure.

4 Polynomial-Time Soundness

This section shows that LFPL programs can be evaluated in polynomial time. To strengthen the soundness result and illustrate how to add features to LFPL while maintaining polynomial-time soundness, we extend the language and refer to the result as LFPL$^+$. The features are the standard product type, which manifests as the lazy product in an affine setting, a type of unbounded stacks, which are similar to lists but do not support recursion, and the inductive type of binary trees. The syntax and typing rules of these language constructs are given in Figure 4 and Figure 5 respectively.

Type	A, B	$::= \cdots$	Term	M, N		$::= \cdots$
	$A \times B$	product		(M, N)		product intro
	$\mathtt{S}(A)$	stack		$M \cdot i$	$(i \in \{1,2\})$	product elim
	$\mathtt{T}(A)$	binary tree		$\mathtt{empty}$		stack intro
				$\mathtt{push}\,(M_h; M_t)$		stack intro
				$\mathtt{pop}\ M\,\{N_1 \mid x_h.x_t.N_2\}$		stack elim
				$\mathtt{leaf}$		tree intro
				$\mathtt{node}\,(M_d; M_x; M_l; M_r)$		tree intro
				$\mathtt{trec}\ M\,\{N_1 \mid x_d.x.x_l.x_r.N_2\}$		tree elim

Figure 4 LFPL$^+$'s type and term syntax.

$$\textsc{Ty:ProdI}\ \frac{\Gamma \vdash M_1 : A_1 \qquad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2} \qquad \textsc{Ty:ProdE}_i\ \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash M \cdot i : A_i} \qquad \textsc{Ty:StackI}_1\ \frac{}{\Gamma \vdash \mathtt{empty} : \mathtt{S}(A)}$$

$$\textsc{Ty:StackI}_2\ \frac{\Gamma_h \vdash M_h : A \qquad \Gamma_t \vdash M_t : \mathtt{S}(A)}{\Gamma_h; \Gamma_t \vdash \mathtt{push}\,(M_h; M_t) : \mathtt{S}(A)} \qquad \textsc{Ty:StackE}\ \frac{\Gamma \vdash M : \mathtt{S}(A) \qquad \Gamma' \vdash N_1 : B \qquad \Gamma', x_h : A, x_t : \mathtt{S}(A) \vdash N_2 : B}{\Gamma; \Gamma' \vdash \mathtt{pop}\ M\,\{N_1 \mid x_h.x_t.N_2\} : B} \qquad \textsc{Ty:TreeI}_1\ \frac{}{\Gamma \vdash \mathtt{leaf} : \mathtt{T}(A)}$$

$$\textsc{Ty:TreeI}_2\ \frac{\Gamma_d \vdash M_d : \Diamond \qquad \Gamma_x \vdash M_x : A \qquad \Gamma_l \vdash M_l : \mathtt{T}(A) \qquad \Gamma_r \vdash M_r : \mathtt{T}(A)}{\Gamma_d; \Gamma_x; \Gamma_l; \Gamma_r \vdash \mathtt{node}\,(M_d; M_x; M_l; M_r) : \mathtt{T}(A)} \qquad \textsc{Ty:TreeE}\ \frac{\Gamma \vdash M : \mathtt{T}(A) \qquad \cdot \vdash N_1 : B \qquad \cdot, x_d : \Diamond, x : A, x_l : B, x_r : B \vdash N_2 : B}{\Gamma \vdash \mathtt{trec}\ M\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} : B}$$

Figure 5 LFPL$^+$'s typing rules.

The typing rules for stacks of type $\mathtt{S}(A)$ correspond to those for lists in a standard non-affine functional programming language, where the elimination form is case analysis rather than structural recursion and the introduction form does not require diamonds. These stacks do not violate LFPL's core principle of controlling recursion by diamonds because they do not have a recursor.

The typing rules for the tree type $\mathtt{T}(A)$, an inductive type which does support recursion, are similar to those of lists $\mathtt{L}(A)$. The primary difference is that, in Rule Ty:TreeE, the base case N_1 is typed under an empty context. Like the list recursor, the tree recursor has two cases: the inductive case and base case. The difference is that for the list recursor, the base case N_1 is only run once, so it may have access to part of the context (though this is not necessary for completeness). In the tree recursor, N_1 may be run many times, as a tree can have many leaves. So, to respect linearity, it cannot use anything from the context.

The denotational semantics for these new constructs are relatively straightforward. We set $[\![A \times B]\!] = [\![A]\!] \times [\![B]\!]$ and $[\![\mathtt{S}(A)]\!] = [\![A]\!]^*$. Giving a semantics for $\mathtt{T}(A)$ and its related terms boils down to giving a mathematical account of binary trees (in the same manner we have used the Kleene star to give a simple mathematical account of lists). The semantics for terms involving these types can be found in the mechanization.

The stack type $\mathtt{S}(A)$ is a particularly powerful addition to LFPL. The vast majority of the difficulty in proving polynomial-time completeness of LFPL, and by far the most complex aspect of our entire mechanization, was the definition and implementation of the

Val	v	::=	Env	$\mathcal{V}$	::=
	$\texttt{vdiam}$	diamond value		$\cdot$	empty envmt.
	$\texttt{vnull}$	unit value		$\mathcal{V}\,[x \mapsto v]$	nonempty envmt.
	$\texttt{vlpair}\,(\mathcal{V}; (M_1, M_2))$	lazy pair closure			
	$\texttt{vinj}_i\ v$	injection			
	$\texttt{vpair}\,(v_1; v_2)$	pair			
	$\texttt{vlam}\,(\mathcal{V}; x.M)$	function closure			
	$\texttt{vempty}$	empty stack			
	$\texttt{vpush}\,(v_h; v_t)$	nonempty stack			
	$\texttt{vnil}$	empty list			
	$\texttt{vcons}\,(v_h; v_t)$	nonempty list			
	$\texttt{vleaf}$	empty tree			
	$\texttt{vnode}\,(v; v_l; v_r)$	nonempty tree			

Figure 6 LFPL$^+$'s value and environment syntax.

bounded stack data structure discussed in Section 3.1 and Section 3.2. So, the addition of unbounded, $\Diamond$-free stacks to LFPL means LFPL$^+$ admits a much simpler completeness proof, and thus it is worthwhile to show that they are sound despite their power. In essence, LFPL$^+$ internalizes the bounded stack data structure from Section 3.1 and removes the boundedness restriction. No diamonds are required to interface with the stacks of LFPL$^+$, and there is no upper bound on the number of stack elements.

4.1 Big-Step Operational Cost Semantics

As discussed in Section 2.3, any semantics for LFPL$^+$ needs to have some way to inhabit the $\Diamond$ type. To give LFPL$^+$ an operational semantics, we introduce a separate language for LFPL$^+$ values in Figure 6. Note that the definitions of value and environment are mutually recursive. This means we usually have to define functions on values via mutual recursion with a corresponding function on environments. Like in the denotational semantics, diamonds in lists and trees are not explicit. By separating values from terms, we ensure LFPL$^+$ terms remain polynomial-time sound while obtaining a means of providing concrete inputs to observe their evaluation. We also define a typing judgement $v : A$ for values, so that $\texttt{vdiam} : \Diamond$, $\texttt{vcons}\,(v_h; v_t) : \texttt{L}(A)$ whenever $v_h : A$ and $v_t : \texttt{L}(A)$, and so on.

To account for the cost of evaluation, we give a cost-annotated big-step operational semantics, represented by the judgement $\mathcal{V} \vdash M \Downarrow_c v$, read "term M evaluates under the environment $\mathcal{V}$ to a value v with cost c". A representative set of rules for this judgement is given in Figure 7, and a complete set of rules can be found in Appendix A. Note the presence of constants $C_{\text{VAR}}, C_{\text{NULL}}, \ldots, C_{\text{TREC}} \in \mathbb{N}$ (one for each syntactic construct) in the conclusion of each rule. These are left arbitrary so that our cost model is generic with respect to the particular costs of each primitive operation of the language.

▶ **Example 18.** Consider `reverse` from Example 1. Instead of proving correctness in terms of the denotational semantics, we could show that $\cdot\,[x \mapsto v_\ell] \vdash \texttt{reverse}\ v_\ell \Downarrow_{v'_\ell}$, where v'_ℓ represents the reverse of the list represented by v_ℓ.

This semantics enjoys several standard properties such as determinism and preservation of typing, which we state here and prove in the mechanization.

▶ **Theorem 19** (Determinism). *If $\mathcal{V} \vdash M \Downarrow_c v$ and $\mathcal{V} \vdash M \Downarrow_{c'} v'$ then $c = c'$ and $v = v'$.*

$$\textsc{Eval:Var}\quad \frac{}{\mathcal{V}\,[x \mapsto v] \vdash x \Downarrow_{C_{\text{VAR}}} v} \qquad \textsc{Eval:ArrowI}\quad \frac{}{\mathcal{V} \vdash \lambda x.M \Downarrow_{C_{\text{LAM}}} \texttt{vlam}\,(\mathcal{V}; x.M)}$$

$$\textsc{Eval:SumE}_i\quad \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vinj}_i\ v \qquad \mathcal{V}'\,[x \mapsto v] \vdash N_i \Downarrow_{c'} v'}{\mathcal{V}; \mathcal{V}' \vdash \texttt{case}\ M\,\{x.N_1 \mid x.N_2\} \Downarrow_{c+c'+C_{\text{CASE}}} v'}$$

$$\textsc{Eval:ArrowE}\quad \frac{\mathcal{V}_1 \vdash M \Downarrow_{c_1} \texttt{vlam}\,(\mathcal{V}'; x.M') \qquad \mathcal{V}_2 \vdash N \Downarrow_{c_2} v \qquad \mathcal{V}'\,[x \mapsto v] \vdash M' \Downarrow_{c'} v'}{\mathcal{V}_1; \mathcal{V}_2 \vdash M\ N \Downarrow_{c_1+c_2+c'+C_{\text{APP}}} v'}$$

$$\textsc{Eval:ListI}_2\quad \frac{\mathcal{V}_d \vdash M_d \Downarrow_{c_d} \texttt{vdiam} \qquad \mathcal{V}_h \vdash M_h \Downarrow_{c_h} v_h \qquad \mathcal{V}_t \vdash M_t \Downarrow_{c_t} v_t}{\mathcal{V}_d; \mathcal{V}_h; \mathcal{V}_t \vdash \texttt{cons}\,(M_d; M_h; M_t) \Downarrow_{c_d+c_h+c_t+C_{\text{CONS}}} \texttt{vcons}\,(v_h; v_t)}$$

$$\textsc{Eval:TreeE}_2\quad \frac{\begin{array}{c}\mathcal{V} \vdash M \Downarrow_c \texttt{vnode}\,(v_x; v_l; v_r) \qquad \cdot\,[y \mapsto v_l] \vdash \texttt{trec}\ y\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} \Downarrow_{c_L} v_L \\ \cdot\,[y \mapsto v_r] \vdash \texttt{trec}\ y\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} \Downarrow_{c_R} v_R \\ \cdot[x_d \mapsto \texttt{vdiam}, x \mapsto v_x, x_l \mapsto v_L, x_r \mapsto v_R] \vdash N_2 \Downarrow_{c'} v\end{array}}{\mathcal{V} \vdash \texttt{trec}\ M\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} \Downarrow_{c+c_L+c_R+c'+C_{\text{TREC}}} v}$$

Figure 7 Selected rules for LFPL$^+$'s evaluation judgement.

▶ **Theorem 20** (Preservation). *If* $\Gamma \vdash M : A$, $\mathcal{V} : \Gamma$, *and* $\mathcal{V} \vdash M \Downarrow_c v$, *then* $v : A$.

We can also give a denotational semantics for values and environments, and prove that they cohere with the denotational semantics for terms. In particular, given $v : A$ and $\mathcal{V} : \Gamma$, we define $[\![v]\!] \in [\![A]\!]$ and $[\![\mathcal{V}]\!] \in [\![\Gamma]\!]$ by mutual induction on the structures of v and $\mathcal{V}$. Most cases are straightforward; for example, $[\![\texttt{vpair}\,(v_1; v_2)]\!] = ([\![v_1]\!], [\![v_2]\!])$. For closures such as function values, we appeal to the term denotational semantics: $[\![\texttt{vlam}\,(\mathcal{V}; x.M)]\!] = \lambda v.[\![M]\!]\,([\![\mathcal{V}]\!]\,[x \mapsto v])$. We prove the following in the mechanization:

▶ **Theorem 21** (Coherence). *If* $\Gamma \vdash M : A$, $\mathcal{V} : \Gamma$, *and* $\mathcal{V} \vdash M \Downarrow_c v$, *then* $[\![v]\!] = [\![M]\!]\,([\![\mathcal{V}]\!])$.

Polynomial-time soundness is meant to be a statement about the computational complexity of functions encoded by LFPL with respect to some machine model like a Turing machine. The big-step cost semantics is somewhat removed from a Turing machine's cost model. However, it is well understood how to justify a cost semantics for a functional language like LFPL$^+$ in terms of a low-level model such as a Turing machine [6, 1]. The only non-standard features of LFPL are the affine type system and the type $\Diamond$, and both of these can be forgotten in a simple cost-preserving translation. Thus, when proving true polynomial-time soundness of LFPL, the only part which relies on LFPL-specific reasoning is the very first step of giving a polynomial bound on the cost in LFPL's big-step semantics; after that, these standard methods will carry the polynomial bound all the way down to the machine cost model. As the focus of this paper is LFPL, we only mechanize the construction and verification of the polynomial cost bound.

4.2 The Non-Size-Increasing Property

We now make precise the non-size-increasing property referred to in previous sections. Our goal is to assign to each value v a number $\mathsf{size}\,(v) \in \mathbb{N}$ that, roughly speaking, counts the number of diamond values `vdiam` contained within v. We then show that the size of the value resulting from the evaluation of an LFPL$^+$ term is bounded by the size of the values in the environment. We define size by mutual induction on values and environments in Figure 8.

$$\begin{aligned}
\mathsf{size}(\texttt{vdiam}) &= 1 & \mathsf{size}(\texttt{vlpair}(\mathcal{V}; (M_1, M_2))) &= \mathsf{size}(\mathcal{V}) \\
\mathsf{size}(\texttt{vnull}) &= 0 & \mathsf{size}(\texttt{vlam}(\mathcal{V}; x.M)) &= \mathsf{size}(\mathcal{V}) \\
\mathsf{size}(\texttt{vinj}_i\ v) &= \mathsf{size}(v) & \mathsf{size}(\texttt{vpair}(v_1; v_2)) &= \mathsf{size}(v_1) + \mathsf{size}(v_2) \\
\mathsf{size}(\texttt{vempty}) &= 0 & \mathsf{size}(\texttt{vpush}(v_h; v_t)) &= \mathsf{size}(v_h) + \mathsf{size}(v_t) \\
\mathsf{size}(\texttt{vnil}) &= 0 & \mathsf{size}(\texttt{vcons}(v_h; v_t)) &= 1 + \mathsf{size}(v_h) + \mathsf{size}(v_t) \\
\mathsf{size}(\texttt{vleaf}) &= 0 & \mathsf{size}(\texttt{vnode}(v; v_l; v_r)) &= 1 + \mathsf{size}(v) + \mathsf{size}(v_l) + \mathsf{size}(v_r)
\end{aligned}$$

$$\mathsf{size}(\cdot) = 0 \qquad \mathsf{size}(\mathcal{V}[x \mapsto v]) = \mathsf{size}(\mathcal{V}) + \mathsf{size}(v)$$

Figure 8 Size of values and environments in LFPL$^+$.

▶ **Theorem 22** (The Non-Size-Increasing Property). *Suppose that $\Gamma \vdash M : A$. Then, for all environments $\mathcal{V}$, values v, and costs c such that $\mathcal{V} \vdash M \Downarrow_c v$, we have $\mathsf{size}(v) \leq \mathsf{size}(\mathcal{V})$.*

Proof. By rule induction on $\mathcal{V} \vdash M \Downarrow_c v$. All cases are provided in the mechanization. ◀

The relevance of this result to soundness might already be clear. An immediate corollary is that there are no closed terms of type $\Diamond$, because `vdiam` has size 1 but the empty environment has size 0. This is one of the main properties LFPL's type system was designed to ensure.

4.3 Polynomial Cost Bound

Our goal for the rest of this section is to prove the following:

▶ **Theorem 23** (Concrete Polynomial-Time Soundness of LFPL$^+$). *Suppose $\cdot, \ell : \mathtt{L}(A) \vdash M : B$, where A $\Diamond$-free. Then, there exists a polynomial $P_M : \mathbb{N} \to \mathbb{N}$ such that, for any value $v_\ell : \mathtt{L}(A)$ representing a list of length n, there exists a value $v : B$ and a cost $c \in \mathbb{N}$ such that $\cdot[\ell \mapsto v_\ell] \vdash M \Downarrow_c v$ and $c \leq P(n)$.*

Note that it is important that the polynomial only depends on M, i.e., it is quantified before the input v_ℓ. Taking $B = \mathtt{L}(\mathtt{1})$ and $A = \mathtt{1}$, we recover the basic notion of soundness for LFPL$^+$ functions on natural numbers. We present this more general form to motivate our transition to the even more general form we need to successfully prove it. As usual, we must reason about arbitrary terms under arbitrary contexts. It is tempting to say the following:

> *Suppose $\Gamma \vdash M : A$. There exists a polynomial $P_M : \mathbb{N} \to \mathbb{N}$ such that, if $\mathcal{V} : \Gamma$, there exists a value $v : A$ and a cost $c \in \mathbb{N}$ such that $\mathcal{V} \vdash M \Downarrow_c v$ and $c \leq P(\mathsf{size}(\mathcal{V}))$.*

However, this statement is false. The problem is that the environment may contain values which incur cost when used, which is dynamic information and P_M only has access to static information. Consider $f : \mathtt{1} \multimap \mathtt{1} \vdash f\ \langle\rangle : \mathtt{1}$. If for simplicity $C_{\text{APP}} = C_{\text{VAR}} = C_{\text{NULL}} = 0$, we cannot statically observe how this program could incur any cost. After all, the only statically observable syntactic constructs are application, variable, and null. On the other hand, dynamically, $\mathcal{V}$ could map f to a function which incurs cost.

To resolve this issue, we must consider the potential cost of running functions (or lazy pairs) stored in the environment $\mathcal{V}$ and result value v. Suppose for a moment we have defined three families of polynomials: P_M on all terms M, P_v on all values v, and $P_{\mathcal{V}}$ on all environments $\mathcal{V}$. Our new goal is to prove the following theorem.

▶ **Theorem 24** (General Soundness). *Suppose $\Gamma \vdash M : A$. Then, for any $\mathcal{V} : \Gamma$, there exists a value $v : A$ and a cost $c \in \mathbb{N}$ such that $\mathcal{V} \vdash M \Downarrow_c v$ and:*

$$c + P_v(\mathsf{size}(\mathcal{V})) \leq P_M(\mathsf{size}(\mathcal{V})) + P_{\mathcal{V}}(\mathsf{size}(\mathcal{V}))$$

This theorem is true (for some definitions of P_v, P_M, and $P_{\mathcal{V}}$), though as stated it is not strong enough to be proven by rule induction on $\Gamma \vdash M : A$. Before discussing this further, we define the polynomials used in this theorem. By $\max(P, Q) : \mathbb{N} \to \mathbb{N}$ we mean the polynomial with coefficients equal to the pairwise maximums of the coefficients of polynomials $P, Q : \mathbb{N} \to \mathbb{N}$. This way, $\max(P(n), Q(n)) \leq \max(P, Q)(n)$.

▶ **Definition 25** (Term Polynomial). *Let M be a LFPL$^+$ term. We inductively define the polynomial $P_M : \mathbb{N} \to \mathbb{N}$ as follows.*

$$\begin{aligned}
P_x(n) &= C_{\text{VAR}} \\
P_{(M_1,M_2)}(n) &= C_{\text{RECORD}} + \max(P_{M_1}, P_{M_2})(n) \\
P_{M.i}(n) &= C_{\text{PROJ}_i} + P_M(n) \\
P_{\lambda x.M}(n) &= C_{\text{LAM}} + P_M(n) \\
P_{M\ N}(n) &= C_{\text{APP}} + P_M(n) + P_N(n) \\
P_{\texttt{case}\ M\{x_1.N_1 \mid x_2.N_2\}}(n) &= C_{\text{CASE}} + P_M(n) + \max(P_{N_1}, P_{N_2})(n) \\
P_{\texttt{lrec}\ M\{N_1 \mid x_d.x_h.x_t.N_2\}}(n) &= P_M(n) + (C_{\text{LREC}} + P_{N_1}(n)) + n(C_{\text{VAR}} + C_{\text{LREC}} + P_{N_2}(n)) \\
P_{\texttt{trec}\ M\{N_1 \mid x.y.z.w.N_2\}}(n) &= P_M(n) + (n+1)(C_{\text{TREC}} + P_{N_1}(n)) + n(2C_{\text{VAR}} + C_{\text{TREC}} + P_{N_2}(n))
\end{aligned}$$

The remaining cases are similar and provided in the mechanization.

The aim of P_M is to bound the cost of evaluating M, and the future cost of using any closures it evaluates to, by $P_M(n)$ in an environment containing n diamonds. The intuition behind $P_{\texttt{lrec}\ M\{N_1 \mid x_d.x_h.x_t.N_2\}}(n)$ is that M and N_1 are evaluated once, and N_2 could be evaluated up to n times, where $n = \mathsf{size}(\mathcal{V})$ is the number of diamonds available in the environment. Similarly, for $P_{\texttt{trec}\ M\{N_1 \mid x.y.z.w.N_2\}}$, the base case N_1 could be evaluated $n+1$ times (a tree with n internal nodes has $n+1$ leaf nodes) and N_2 could be evaluated n times.

▶ **Definition 26** (Value and Environment Polynomials). *We define polynomials $P_{\mathcal{V}} : \mathbb{N} \to \mathbb{N}$ and $P_v : \mathbb{N} \to \mathbb{N}$ via mutual induction as follows:*

$$\begin{aligned}
P_{\texttt{vdiam}} = P_{\texttt{vnull}} = P_{\texttt{vempty}} = P_{\texttt{vnil}} = P_{\texttt{vleaf}} &= 0 \qquad & P_{\cdot} &= 0 \\
P_{\texttt{vlpair}(\mathcal{V};(M_1,M_2))} &= P_{\mathcal{V}} + \max(P_{M_1}, P_{M_2}) & P_{\mathcal{V}[x\mapsto v]} &= P_{\mathcal{V}} + P_v \\
P_{\texttt{vlam}(\mathcal{V};x.M)} &= P_{\mathcal{V}} + P_M \\
P_{\texttt{vpair}(v_1;v_2)} &= P_{v_1} + P_{v_2}
\end{aligned}$$

The definition of P_v for other values v is given in the mechanization.

The idea of P_v and $P_{\mathcal{V}}$ is to bound the future cost of evaluating all of the closures (functions and lazy pairs) stored inside v and $\mathcal{V}$ respectively. With these choices of polynomials, Theorem 24 implies Theorem 23, because when $v : \mathtt{L}(A)$ and A $\Diamond$-free, it is clear that $P_v = 0$ and $\mathsf{size}(v) = n$, where n is the length of the list represented by v.

▶ **Example 27.** Consider `reverse` and `revAppend` from Example 1. For simplicity, assume our cost model has $C_{\text{LREC}} = C_{\text{APP}} = 1$ and all other constants are zero. We have:

$$P_{\texttt{reverse}\ \ell}(n) = 3C_{\text{APP}} + P_{\texttt{revAppend}}(n) = 3C_{\text{APP}} + C_{\text{LREC}} + n(C_{\text{LREC}} + C_{\text{APP}}) = 2n + 4$$

Applying Theorem 23 to this result, we learn that if `reverse` is evaluated with an input list of length n (and an $\Diamond$-free element type), we will incur at most $2n + 4$ cost, thus proving `reverse` is linear-time.

4.4 Proving the Soundness Theorem

To prove Theorem 24, we take a standard logical relations approach. We define a unary logical relation $\mathcal{R}$ for terms, values, and environments with the goal that the fundamental theorem for $\mathcal{R}$ should imply Theorem 24.

▶ **Definition 28.** *We first define the shorthand* $\mathcal{R}_A(\mathcal{V}; M)$ *to mean:*

For all $n \geq \mathsf{size}(\mathcal{V})$*, there exists* $v : A$ *and a cost* $c \in \mathbb{N}$ *such that* $\mathcal{V} \vdash M \Downarrow_c v$*,* $\mathcal{R}_A(v)$ *holds, and* $c + P_v(n) \leq P_M(n) + P_{\mathcal{V}}(n)$

By working with a general $n \geq \mathsf{size}(\mathcal{V})$ rather than just $\mathsf{size}(\mathcal{V})$ as the input to the polynomials, we can deal with the case where $\mathcal{V}$ arises as part of a larger environment during the proof. This is reminiscent of *future worlds* from Kripke-style logical relations [27].

▶ **Definition 29.** *We define* $\mathcal{R}_A(v)$ *by induction on the type* A*:*

$$
\begin{array}{rll}
\mathcal{R}_1(v) & \textit{iff} & v = \mathtt{vnull} \\
\mathcal{R}_{A_1 \otimes A_2}(v) & \textit{iff} & v = \mathtt{vpair}(v_1; v_2),\ \mathcal{R}_{A_1}(v_1),\ \textit{and}\ \mathcal{R}_{A_2}(v_2) \\
\mathcal{R}_{A_1 \times A_2}(v) & \textit{iff} & v = \mathtt{vlpair}(\mathcal{V}; (M_1, M_2)),\ \mathcal{R}_{A_1}(\mathcal{V}; M_1),\ \textit{and}\ \mathcal{R}_{A_2}(\mathcal{V}; M_2) \\
\mathcal{R}_{A \multimap B}(v) & \textit{iff} & v = \mathtt{vlam}(\mathcal{V}; x.M),\ \textit{and}\ \mathcal{R}_A(v)\ \textit{implies}\ \mathcal{R}_B(\mathcal{V}[x \mapsto v]; M) \\
\mathcal{R}_{\mathtt{L}(A)}(v) & \textit{iff} & v\ \textit{is a list}\ \ell\ \textit{such that}\ \mathcal{R}_A(-)\ \textit{holds for each element of}\ \ell
\end{array}
$$

The remaining cases are provided in the mechanization.

▶ **Definition 30.** *We define* $\mathcal{R}_\Gamma(\mathcal{V})$ *by induction on* Γ*.* $\mathcal{R}_\cdot(\cdot)$ *always holds.* $\mathcal{R}_{\Gamma, x:A}(\mathcal{V}[x \mapsto v])$ *holds whenever* $\mathcal{R}_A(v)$ *and* $\mathcal{R}_\Gamma(\mathcal{V})$ *hold.*

Now, Theorem 24 is clearly implied by the fundamental theorem for our relation $\mathcal{R}$:

▶ **Theorem 31** (Fundamental Theorem)**.** *Suppose* $\Gamma \vdash M : A$ *and* $\mathcal{V} : \Gamma$*. Then, if* $\mathcal{R}_\Gamma(\mathcal{V})$ *holds, so does* $\mathcal{R}_A(\mathcal{V}; M)$*.*

Proof. We go by rule induction on $\Gamma \vdash M : A$. Theorem 22 comes up in nearly every case, since we need to satisfy the premise $n \geq \mathsf{size}(\mathcal{V})$ of $\mathcal{R}_A(\mathcal{V}; M)$ to use any inductive hypotheses. Importantly, we actually make use of this premise in the list and tree recursor cases, where we can say for example that the length of any list value v is at most $\mathsf{size}(\mathcal{V})$ and therefore at most n. Most cases are verbose but straightforward, and a couple of the less straightforward cases can be found in Appendix B. ◀

5 Mechanization

We have mechanized all results stated in Sections 2, 3, and 4 in the proof assistant Istari [8]. To our knowledge, our mechanization serves as one of two published case studies in this relatively new proof assistant [23]. The mechanization is roughly 7,000 lines of meaningful Istari code. Around 3,000 of these are definitions and LFPL code, around 3,500 are proof scripts, and around 800 are lemma and theorem statements [14].

5.1 Defining LFPL

We begin by defining the types of LFPL as an inductive datatype `tp` within Istari. LFPL does not have type variables or similar complexities. We also define the judgement A $\Diamond$-free within Istari. Next, we represent contexts Γ as an Istari list of LFPL types. We do not

need variable names associated with the types because we use De Bruijn indices in the term structure. Importantly, we define a judgement `split` $\Gamma\ \Gamma_1\ \Gamma_2$ to indicate when Γ can be bipartitioned into $\Gamma_1; \Gamma_2$, used extensively in the typing rules. Essentially, viewing the contexts as type lists as we do in our mechanization, our definition of `split` $\Gamma\ \Gamma_1\ \Gamma_2$ ensures `append`$(\Gamma_1)(\Gamma_2)$ is equivalent to some permutation of Γ.

Instead of presenting an untyped syntax with an extrinsic typing judgement, as we do in Section 2.2, our mechanization defines LFPL's term syntax and typing rules at once, so that all terms are intrinsically well-typed under a known context. These intrinsically typed terms are represented by an inductive datatype `term : ctx -> tp -> type`, where `type` refers to Istari types, so that a value of type `term G A` corresponds to a term M such that $G \vdash M : A$. For example, the rule for the term $\langle M_1, M_2 \rangle$ is:

```
| pair : forall (G GA GB : ctx) (A B : tp) .
    split G GA GB -> term GA A -> term GB B -> term G (tensor A B)
```

It says we must bipartition the context G into $G_A; G_B$, and give a well-typed term under each of those contexts to form a pair $\langle M_1, M_2 \rangle$ which has type $A \otimes B$ under context G.

5.2 Denotational and Operational Semantics

Instead of the set-theoretic denotational semantics we give in Section 2.3, our mechanization gives a denotational semantics from intrinsically typed LFPL terms into Istari terms of the appropriate type. This is similar to the set-theoretic formulation, since all operations we use on sets (products, coproducts, etc.) are also present in the Istari type system. One benefit is that we can rely on Istari's reduction engine to simplify Istari terms resulting from the semantics of a complicated LFPL term. These denotations have the following signatures. Here, a value of type `env G` represents an environment $\mathcal{V}$ such that $\mathcal{V} : G$.

```
tp_sem : tp -> type
term_sem : forall (G : ctx) (A : tp) . term G A -> env G -> tp_sem A
```

Similarly, for the operational semantics judgement $\mathcal{V} \vdash M \Downarrow_c v$ from Section 4.1, we define an inductive datatype encoding its rules with the following type:

```
evals : forall (G : ctx) (A : tp) . term G A -> env G -> value A -> nat -> type
```

As briefly mentioned at the end of Section 4.1, we give a denotational semantics for values and prove a coherence result (Theorem 21) in our mechanization, which is a good example of what many of our Istari theorems look like.

```
value_sem : forall (A : tp) . value A -> tp_sem A
value_sem = ...
lemma "operational_equiv_denotational"
  forall (G : ctx) (A : tp) (M : term G A) (V : env G) (v : value A) (c : nat) .
  evals M V v c -> value_sem v = term_sem M (env_sem V)
```

5.3 Takeaways from Istari

When beginning this project, we were uncertain which proof assistant would be most suitable for the mechanization, so we began by developing two mechanizations in parallel, one in Agda and one in Istari. During preliminary work for the completeness theorem, we ran into two significant difficulties with Agda.

The first issue was a lack of tactics. Agda expressions are often cleaner and more readable than the proof scripts of tactic-based proof assistants like Rocq and Istari. However, when one constantly needs to reason about equality, say by substituting e' for e in a term $f(e)$ where f is a complex expression, it becomes helpful to have a substitution tactic. In Agda, we needed to manually specify f, which is infeasible when f is large.

Secondly, Agda requires that all expressions are well-typed, even in the statement of a theorem yet to be proven. This presents some difficulties. For example, consider the LFPL type $(\mathtt{L}(\mathtt{1}))^k$ used in Section 3.1. We know that $[\![(\mathtt{L}(\mathtt{1}))^k]\!] = [\![\mathtt{L}(\mathtt{1})]\!]^k$, so we implicitly switch between these forms as needed, but in the mechanization this must be explicit. Some parts of our proof are only well-typed if they are given the form on the left-hand side of the equality, and others need the right-hand form. In Agda, we had to state a theorem where some parts of the theorem needed the left form, and others needed the right, so there was no way to formulate this such that it was well-typed without using transport functions, which are known to cause other major difficulties [32]. Istari does not require the well-typedness of a theorem before it is proven. It must be given a type by the end of the proof, but this is not difficult since we have a lemma stating $[\![(\mathtt{L}(\mathtt{1}))^k]\!] = [\![\mathtt{L}(\mathtt{1})]\!]^k$. The difference is that Istari lets us show well-typedness during the proof, whereas Agda demands it before the proof.

Such issues led us to finish the completeness and soundness proofs only in Istari. Retrospectively, we can say that it was a more than adequate tool for this task. In fact, we never encountered any Istari-related roadblocks. This is not to say there weren't annoyances; as a relatively new proof assistant, Istari sometimes struggles to infer implicit arguments and automatically eliminate vacuous proof cases, requiring manual intervention in situations where Agda would succeed.

6 Related Work

The most closely related work to ours is that of Atkey [3]. He develops a powerful extension of LFPL with full dependent types that includes lists that do not enjoy iteration, much like the stack types we introduced in Section 4. This separation of (unrestricted) construction of data-structures and iteration has been previously used by Jones [21] in his "Cons-free" system for characterizing the class P. Atkey also provides a semantic polynomial-time soundness proof using the realizability method of Dal Lago and Hofmann [10]. These developments are mechanized in Agda. Our method for proving soundness differs from his in that it is syntactic rather than semantic and constructs an explicit polynomial bound on the cost of evaluation for each term. Furthermore, we devote great attention to mechanizing polynomial-time completeness, ensuring the strongest result by working with a minimal version of LFPL.

Hofmann's original work on LFPL [17, 19] is the basis of this article and has been discussed in detail in Sections 2, 3, and 4. The soundness proof is based on previous work by Aehlig and Schwichtenberg [2], which introduces the technique of constructing explicit polynomials from LFPL syntax to bound the cost of evaluation. Ours is the first work to apply their technique to a big-step operational semantics. None of these articles comes with a mechanization or full account of both soundness and completeness.

LFPL also directly inspired automatic amortized resource analysis (AARA) [16], and so our work could perhaps inspire early steps towards mechanizing the metatheory of AARA. Additionally, there is a polynomial-time completeness result for AARA similar to the one for LFPL discussed in Section 3 [29].

Beyond LFPL, there is a large body of work on implicit computational complexity (ICC) [9]. The polynomial time and space classes are a common focus [12, 13], though there has been interest in logarithmic space usage as well [31, 30]. Interestingly, Hofmann [19] noticed

that, if the structural recursor of LFPL is replaced with general recursion, the terminating fragment of the language characterizes exponential-time computation. LFPL is an influential language in ICC, helping to inspire numerous developments in the field [10, 25, 26, 4, 22, 7].

We are only aware of two other works in ICC, besides ours and Atkey's [3], that are significantly mechanized. Heraud and Nowak [15] mechanized the soundness and completeness of the language for P of Bellantoni and Cook [5] in Rocq. This language is quite different from LFPL since it does not feature a linear type discipline or higher-order functions but instead relies on the idea of *safe arguments*. Férée et al. [11] describe a library for verifying polynomial-time complexity in Rocq by using quasi-interpretations. Instead of a higher-order language with a linear type system, they consider first-order term-rewrite systems. As a result, the soundness and completeness arguments are very different from the ones for LFPL. In particular, the completeness proofs for both, the language of Bellantoni and Cook and the system of quasi-interpretations, do not require the construction of bounded stacks, which is the main difficulty in completeness proof for LFPL.

7 Conclusion

This article provides a self-contained presentation of Hofmann's LFPL. The mechanized soundness proof constructs, for each LFPL expression, an explicit polynomial that bounds the evaluation cost with respect to a big-step cost semantics. The mechanized completeness proof shows how LFPL can simulate polynomial-time Turing machines, relying only on linear lists and functions to encode the polynomially sized tape in a non-size-increasing way. Both proofs are based on existing ideas but contain significant simplifications and improvements.

Our hope is that this work contributes to making LFPL's metatheory more accessible and to inspiring LFPL-based innovations in areas like automatic resource bound analysis [16], quantitative type theory [3], and memory management in functional languages [24].

References

1 Beniamino Accattoli. A Fresh Look at the lambda-Calculus. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*, volume 131 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.1.

2 Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *ACM Trans. Comput. Logic*, 3(3):383–401, 2002. doi:10.1145/507382.507386.

3 Robert Atkey. Polynomial time and dependent types. *Proc. ACM Program. Lang.*, 8(POPL), 2024. doi:10.1145/3632918.

4 Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Inf. Comput.*, 248(C):56–81, 2016. doi:10.1016/j.ic.2015.12.008.

5 Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the polytime functions (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 283–293. Association for Computing Machinery, 1992. doi:10.1145/129712.129740.

6 Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 213–225. Association for Computing Machinery, 1996. doi:10.1145/232627.232650.

7 G. Bonfante, J. Y. Marion, and J. Y. Moyen. Quasi-interpretations a way to control resources. *Theor. Comput. Sci.*, 412(25):2776–2796, 2011. doi:10.1016/j.tcs.2011.02.007.

8 Karl Crary. The Istari proof assistant. https://istarilogic.org/, 2025.

9 Ugo Dal Lago. Implicit computation complexity in higher-order programming languages: A Survey in Memory of Martin Hofmann. *Math. Struct. Comput. Sci.*, 32(6):760–776, 2022. doi:10.1017/S0960129521000505.

10 Ugo Dal Lago and Martin Hofmann. Realizability models and implicit complexity. *Theor. Comput. Sci.*, 412(20):2029–2047, 2011. doi:10.1016/j.tcs.2010.12.025.

11 Hugo Férée, Samuel Hym, Micaela Mayero, Jean-Yves Moyen, and David Nowak. Formal proof of polynomial-time complexity with quasi-interpretations. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '18, pages 146–157. Association for Computing Machinery, 2018. doi:10.1145/3167097.

12 Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. Soft Linear Logic and Polynomial Complexity Classes. *Electronic Notes in Theoretical Computer Science*, 205:67–87, 2007. Proceedings of the Second Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2007). doi:10.1016/j.entcs.2008.03.066.

13 Marco Gaboardi, Jean-Yves Marion, and Simona Ronchi Della Rocca. An Implicit Characterization of PSPACE. *ACM Trans. Comput. Logic*, 13(2):1–36, 2012. doi:10.1145/2159531.2159540.

14 Nathaniel Glover and Jan Hoffmann. LFPL: Revisited and Mechanized - Source Code of the Mechanization, 2026. doi:10.5281/zenodo.18348213.

15 Sylvain Heraud and David Nowak. A formalization of polytime functions. In *Proceedings of the Second International Conference on Interactive Theorem Proving*, ITP '11, pages 119–134. Springer-Verlag, 2011. doi:10.1007/978-3-642-22863-6_11.

16 Jan Hoffmann and Steffen Jost. Two decades of automatic amortized resource analysis. *Math. Struct. Comput. Sci.*, 32(6):729–759, 2022. doi:10.1017/S0960129521000487.

17 Martin Hofmann. Linear types and non size-increasing polynomial time computation. In *Proceedings of the Fourteenth Annual IEEE Symposium on Logic in Computer Science, LICS 1999*, pages 464–473. IEEE Computer Society Press, 1999. doi:10.1109/LICS.1999.782641.

18 Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic J. of Computing*, 7(4):258–289, 2000. URL: https://dl.acm.org/doi/abs/10.5555/763845.763847.

19 Martin Hofmann. The strength of non-size increasing computation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 260–269. Association for Computing Machinery, 2002. doi:10.1145/503272.503297.

20 Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003. doi:10.1016/S0890-5401(03)00009-9.

21 Neil D. Jones. The expressive power of higher-order types or, life without cons. *Journal of Functional Programming*, 11(1):55–94, 2001. doi:10.1017/S0956796800003889.

22 Ugo Lago and Martin Hofmann. Bounded linear logic, revisited. In *Proceedings of the 9th International Conference on Typed Lambda Calculi and Applications*, TLCA '09, pages 80–94. Springer-Verlag, 2009. doi:10.1007/978-3-642-02273-9_8.

23 Runming Li, Yue Yao, and Robert Harper. Mechanizing synthetic tait computability in Istari. In *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP '26, pages 231–247. Association for Computing Machinery, 2026. doi:10.1145/3779031.3779085.

24 Anton Lorenzen, Daan Leijen, and Wouter Swierstra. FP^2: Fully in-Place Functional Programming. *Proc. ACM Program. Lang.*, 7(ICFP):275–304, 2023. doi:10.1145/3607840.

25 Jean-Yves Marion. Analysing the implicit complexity of programs. *Inf. Comput.*, 183(1):2–18, 2003. doi:10.1016/S0890-5401(03)00011-7.

26 Jean-Yves Marion. A type system for complexity flow analysis. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science*, LICS '11, pages 123–132. IEEE Computer Society, 2011. doi:10.1109/LICS.2011.41.

27 John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1):99–124, 1991. doi:10.1016/0168-0072(91)90067-V.

28 Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.

29 Long Pham and Jan Hoffmann. Typable Fragments of Polynomial Automatic Amortized Resource Analysis. In *29th EACSL Annual Conference on Computer Science Logic, CSL 2021*, volume 183 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.CSL.2021.34.

30 Ramyaa Ramyaa and Daniel Leivant. Ramified Corecurrence and Logspace. *Electronic Notes in Theoretical Computer Science*, 276:247–261, 2011. Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics, MFPS 27. doi:10.1016/j.entcs.2011.09.025.

31 Ulrich Schopp. Stratified Bounded Affine Logic for Logarithmic Space. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 411–420. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.45.

32 Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. Equivalences for free: univalent parametricity for effective transport. *Proc. ACM Program. Lang.*, 2(ICFP), 2018. doi:10.1145/3236787.

$$\textsc{Eval:Var}\ \frac{}{\mathcal{V} \vdash x \Downarrow_{C_{\text{VAR}}} \mathcal{V}(x)} \qquad \textsc{Eval:UnitI}\ \frac{}{\mathcal{V} \vdash \langle\rangle \Downarrow_{C_{\text{NULL}}} \texttt{vnull}}$$

$$\textsc{Eval:SumE}_i\ \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vinj}_i\ v \qquad \mathcal{V}'\,[x \mapsto v] \vdash N_i \Downarrow_{c'} v'}{\mathcal{V}; \mathcal{V}' \vdash \texttt{case}\ M\ \{x.N_1 \mid x.N_2\} \Downarrow_{c+c'+C_{\text{CASE}}} v'}$$

$$\textsc{Eval:SumI}_i\ \frac{\mathcal{V} \vdash M \Downarrow_c v}{\mathcal{V} \vdash i \cdot M \Downarrow_{c+C_{\text{INJ}_i}} \texttt{vinj}_i\ v} \qquad \textsc{Eval:TensorI}\ \frac{\mathcal{V}_1 \vdash M_1 \Downarrow_{c_1} v_1 \qquad \mathcal{V}_2 \vdash M_2 \Downarrow_{c_2} v_2}{\mathcal{V}_1; \mathcal{V}_2 \vdash \langle M_1, M_2\rangle \Downarrow_{c_1+c_2+C_{\text{PAIR}}} \texttt{vpair}\,(v_1; v_2)}$$

$$\textsc{Eval:TensorE}\ \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vpair}\,(v_1; v_2) \qquad \mathcal{V}'[x_1 \mapsto v_1, x_2 \mapsto v_2] \vdash N \Downarrow_{c'} v}{\mathcal{V}; \mathcal{V}' \vdash \texttt{letp}\ \langle x_1, x_2\rangle = M\ \texttt{in}\ N \Downarrow_{c+c'+C_{\text{LETP}}} v}$$

$$\textsc{Eval:ArrowE}\ \frac{\mathcal{V}_1 \vdash M \Downarrow_{c_1} \texttt{vlam}\,(\mathcal{V}'; x.M') \qquad \mathcal{V}_2 \vdash N \Downarrow_{c_2} v \qquad \mathcal{V}'\,[x \mapsto v] \vdash M' \Downarrow_{c'} v'}{\mathcal{V}_1; \mathcal{V}_2 \vdash M\ N \Downarrow_{c_1+c_2+c'+C_{\text{APP}}} v'}$$

$$\textsc{Eval:ArrowI}\ \frac{}{\mathcal{V} \vdash \lambda x.M \Downarrow_{C_{\text{LAM}}} \texttt{vlam}\,(\mathcal{V}; x.M)} \qquad \textsc{Eval:ListE}_1\ \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vnil} \qquad \mathcal{V}' \vdash N_1 \Downarrow_{c'} v}{\mathcal{V}; \mathcal{V}' \vdash \texttt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\} \Downarrow_{c+c'+C_{\text{LREC}}} v}$$

$$\textsc{Eval:ListI}_2\ \frac{\mathcal{V}_d \vdash M_d \Downarrow_{c_d} \texttt{vdiam} \qquad \mathcal{V}_h \vdash M_h \Downarrow_{c_h} v_h \qquad \mathcal{V}_t \vdash M_t \Downarrow_{c_t} v_t}{\mathcal{V}_d; \mathcal{V}_h; \mathcal{V}_t \vdash \texttt{cons}\,(M_d; M_h; M_t) \Downarrow_{c_d+c_h+c_t+C_{\text{CONS}}} \texttt{vcons}\,(v_h; v_t)} \qquad \textsc{Eval:ListI}_1\ \frac{}{\mathcal{V} \vdash \texttt{nil} \Downarrow_{C_{\text{NIL}}} \texttt{vnil}}$$

$$\textsc{Eval:ListE}_2\ \frac{\begin{array}{c}\mathcal{V} \vdash M \Downarrow_c \texttt{vcons}\,(v_h; v_t) \qquad \mathcal{V}'\,[y \mapsto v_t] \vdash \texttt{lrec}\ y\,\{N_1 \mid x_d.x_h.x_t.N_2\} \Downarrow_{c_T} v_T \\ \cdot[x_d \mapsto \texttt{vdiam}, x_h \mapsto v_h, x_t \mapsto v_T] \vdash N_2 \Downarrow_{c'} v_2\end{array}}{\mathcal{V}; \mathcal{V}' \vdash \texttt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\} \Downarrow_{c+c_T+c'+C_{\text{LREC}}} v_2}$$

Figure 9 The full definition of LFPL$^+$'s evaluation judgement. (1/2)

A Evaluation Judgement Rules

In Figure 9 and Figure 10, we provide all rules for the big-step evaluation judgement, $\mathcal{V} \vdash M \Downarrow_c v$, defined in Section 4.1. The additional language features of LFPL$^+$ are included.

B Soundness Proof Cases

Here, we provide two cases of the soundness proof. The other cases are similar to these, though often more straightforward, and can all be found in the mechanization [14]. First, we go through the function application case in moderate detail to demonstrate the need for a logical relation. Second, we sketch the list recursor case, which as a driver of variable-time computation is one of the most critical points to consider for polynomial-time soundness.

▶ **Theorem 32** (Fundamental Theorem). *Suppose* $\Gamma \vdash M : A$ *and* $\mathcal{V} : \Gamma$. *Then, if* $\mathcal{R}_\Gamma(\mathcal{V})$ *holds, so does* $\mathcal{R}_A(\mathcal{V}; M)$.

Proof. We go by rule induction on $\Gamma \vdash M : A$.

$$\textsc{Eval:ProdI}\ \frac{}{\mathcal{V} \vdash (M_1, M_2) \Downarrow_{C_{\text{RECORD}}} \texttt{vlpair}\,(\mathcal{V}; (M_1, M_2))}$$

$$\textsc{Eval:ProdE}_i\ \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vlpair}\,(\mathcal{V}'; (M_1, M_2)) \quad \mathcal{V}' \vdash M_i \Downarrow_{c'} v}{\mathcal{V} \vdash M \cdot i \Downarrow_{c+c'+C_{\text{PROJ}_i}} v}$$

$$\textsc{Eval:StackI}_1\ \frac{}{\mathcal{V} \vdash \texttt{empty} \Downarrow_{C_{\text{EMPTY}}} \texttt{vempty}}$$

$$\textsc{Eval:StackI}_2\ \frac{\mathcal{V}_h \vdash M_h \Downarrow_{c_h} v_h \quad \mathcal{V}_t \vdash M_t \Downarrow_{c_t} v_t}{\mathcal{V}_h; \mathcal{V}_t \vdash \texttt{push}\,(M_h; M_t) \Downarrow_{c_h+c_t+C_{\text{PUSH}}} \texttt{vpush}\,(v_h; v_t)}$$

$$\textsc{Eval:StackE}_1\ \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vempty} \quad \mathcal{V}' \vdash N_1 \Downarrow_{c'} v}{\mathcal{V}; \mathcal{V}' \vdash \texttt{pop}\ M\,\{N_1 \mid x_h.x_t.N_2\} \Downarrow_{c+c'+C_{\text{POP}}} v}$$

$$\textsc{Eval:StackE}_2\ \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vpush}\,(v_h; v_t) \quad \mathcal{V}'[x_h \mapsto v_h, x_t \mapsto v_t] \vdash N_2 \Downarrow_{c'} v}{\mathcal{V}; \mathcal{V}' \vdash \texttt{pop}\ M\,\{N_1 \mid x_h.x_t.N_2\} \Downarrow_{c+c'+C_{\text{POP}}} v}$$

$$\textsc{Eval:TreeI}_2\ \frac{\mathcal{V}_d \vdash M_d \Downarrow_{c_d} \texttt{vdiam} \quad \mathcal{V}_x \vdash M_x \Downarrow_{c_x} v_x \quad \mathcal{V}_l \vdash M_l \Downarrow_{c_l} v_l \quad \mathcal{V}_r \vdash M_r \Downarrow_{c_r} v_r}{\mathcal{V}_d; \mathcal{V}_x; \mathcal{V}_l; \mathcal{V}_r \vdash \texttt{node}\,(M_d; M_x; M_l; M_r) \Downarrow_{c_d+c_x+c_l+c_r+C_{\text{NODE}}} \texttt{vnode}\,(v_x; v_l; v_r)}$$

$$\textsc{Eval:TreeI}_1\ \frac{}{\mathcal{V} \vdash \texttt{leaf} \Downarrow_{C_{\text{LEAF}}} \texttt{vleaf}}$$

$$\textsc{Eval:TreeE}_1\ \frac{\mathcal{V} \vdash M \Downarrow_c \texttt{vleaf} \quad \cdot \vdash N_1 \Downarrow_{c'} v}{\mathcal{V} \vdash \texttt{trec}\ M\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} \Downarrow_{c+c'+C_{\text{TREC}}} v}$$

$$\textsc{Eval:TreeE}_2\ \frac{\begin{array}{c}\mathcal{V} \vdash M \Downarrow_c \texttt{vnode}\,(v_x; v_l; v_r) \quad \cdot[y \mapsto v_l] \vdash \texttt{trec}\ y\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} \Downarrow_{c_L} v_L \\ \cdot[y \mapsto v_r] \vdash \texttt{trec}\ y\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} \Downarrow_{c_R} v_R \\ \cdot[x_d \mapsto \texttt{vdiam}, x \mapsto v_x, x_l \mapsto v_L, x_r \mapsto v_R] \vdash N_2 \Downarrow_{c'} v\end{array}}{\mathcal{V} \vdash \texttt{trec}\ M\,\{N_1 \mid x_d.x.x_l.x_r.N_2\} \Downarrow_{c+c_L+c_R+c'+C_{\text{TREC}}} v}$$

Figure 10 The full definition of LFPL$^+$'s evaluation judgement. (2/2)

B.0.0.1 Application Case

Suppose $\Gamma_1; \Gamma_2 \vdash M\ N : B$ because $\Gamma_1 \vdash M : A \multimap B$ and $\Gamma_2 \vdash N : A$. Assuming $\mathcal{R}_\Gamma(\mathcal{V})$, our goal is to show that $\mathcal{R}_B(\mathcal{V}; M\ N)$.

Let $\mathcal{V}_1; \mathcal{V}_2$ be the bipartition of $\mathcal{V}$ corresponding to $\Gamma_1; \Gamma_2$. Unpacking the definition of $\mathcal{R}_B(\mathcal{V}; M\ N)$, we need to show that, for all $n \geq \mathsf{size}(\mathcal{V}_1) + \mathsf{size}(\mathcal{V}_2)$, there is a value $v : B$ and a cost $c \in \mathbb{N}$ such that $\mathcal{R}_B(v)$ holds, $\mathcal{V} \vdash M\ N \Downarrow_c v$, and:

$$c + P_v(n) \leq C_{\text{APP}} + P_M(n) + P_N(n) + P_{\mathcal{V}_1}(n) + P_{\mathcal{V}_2}(n)$$

Since $\mathcal{R}_\Gamma(\mathcal{V})$, we have $\mathcal{R}_{\Gamma_1}(\mathcal{V}_1)$ and $\mathcal{R}_{\Gamma_2}(\mathcal{V}_2)$. Therefore, by the induction hypotheses, we obtain $\mathcal{R}_{A \multimap B}(\mathcal{V}_1; M)$ and $\mathcal{R}_A(\mathcal{V}_2; N)$. Applying each to n (which is valid since $n \geq \mathsf{size}(\mathcal{V}) \geq \mathsf{size}(\mathcal{V}_i)$ for $i \in \{1, 2\}$), we conclude that:

- There is a value v_M and a cost c_M such that $\mathcal{R}_{A \multimap B}(v)$ holds, $\mathcal{V}_1 \vdash M \Downarrow_{c_M} v_M$, and:

  $$c_M + P_{v_M}(n) \leq P_M(n) + P_{\mathcal{V}_1}(n)$$

- There is a value v_N and a cost c_N such that $\mathcal{R}_B(v_N)$ holds, $\mathcal{V}_2 \vdash N \Downarrow_{c_N} v_N$, and:

  $$c_N + P_{v_N}(n) \leq P_N(n) + P_{\mathcal{V}_2}(n)$$

By definition of $\mathcal{R}_{A \multimap B}(v)$, we know that $v_M = \texttt{vlam}(\mathcal{V}'; x.M')$ and $\mathcal{R}_B(\mathcal{V}'[x \mapsto v_N]; M')$. In particular, by definition of $P_{\texttt{vlam}(\mathcal{V}';x.M')}$, we obtain the following inequality:

$$c_M + P_{\mathcal{V}'}(n) + P_{M'}(n) \leq P_M(n) + P_{\mathcal{V}_1}(n)$$

Also, by Theorem 22, we know:

$$\begin{aligned}\mathsf{size}(\mathcal{V}'[x \mapsto v_N]) &= \mathsf{size}(\mathcal{V}') + \mathsf{size}(v_N) = \mathsf{size}(v_M) + \mathsf{size}(v_N) \\ &\leq \mathsf{size}(\mathcal{V}_1) + \mathsf{size}(\mathcal{V}_2) \leq n\end{aligned}$$

Therefore, we may apply $\mathcal{R}_B(\mathcal{V}'[x \mapsto v_N]; M')$ to n, concluding that there is a value $v : B$ and a cost $c_{M'}$ such that $\mathcal{R}_B(v)$ holds, $\mathcal{V}'[x \mapsto v_N] \vdash M' \Downarrow_c v$, and:

$$c_{M'} + P_v(n) \leq P_{M'}(n) + P_{v_N}(n) + P_{\mathcal{V}'}(n)$$

Take $c = c_M + c_N + c_{M'} + C_{\text{APP}}$. After applying Rule Eval:ArrowE to conclude the first part of our goal, it remains to prove the cost bound. Chaining all our inequalities together, we get that:

$$c_M + c_N + c_{M'} + P_v(n) \leq P_M(n) + P_N(n) + P_{\mathcal{V}}(n)$$

Adding C_{APP} to both sides yields the desired inequality.

B.0.0.2 Recursor Case

Suppose $\Gamma_1; \Gamma_2 \vdash \texttt{lrec}\ M\ \{N_1 \mid x_d.x_h.x_t.N_2\} : B$ because:

- $\Gamma_1 \vdash M : \texttt{L}(A)$
- $\Gamma_2 \vdash N_1 : B$
- $\cdot, x_d : \Diamond, x_h : A, x_t : B \vdash N_2 : B$

Assume $\mathcal{R}_{\Gamma_1}(\mathcal{V}_1)$ and $\mathcal{R}_{\Gamma_2}(\mathcal{V}_2)$, where $\mathcal{V}_1; \mathcal{V}_2$ is the bipartition corresponding to $\Gamma_1; \Gamma_2$. By the inductive hypothesis for M, we know that there is a value v_M and cost c_M such that $\mathcal{R}_{\texttt{L}(A)}(v_M)$ holds, $\mathcal{V}_1 \vdash M \Downarrow_{c_M} v_M$, and:

$$c_M + P_{v_M}(n) \leq P_M(n) + P_{\mathcal{V}_1}(n)$$

By definition of $\mathcal{R}_{\texttt{L}(A)}(v_M)$, we know that v_M is a list ℓ of values of type A for which $\mathcal{R}_A(-)$ holds. Now, recall the definition of $P_{\texttt{lrec}\ M\{N_1 \mid x_d.x_h.x_t.N_2\}}(n)$:

$$P_M(n) + (C_{\text{LREC}} + P_{N_1}(n)) + n(C_{\text{VAR}} + C_{\text{LREC}} + P_{N_2}(n))$$

Instead of directly dealing with this bound on the cost, we use a stronger one:

$$P_M(n) + (C_{\text{LREC}} + P_{N_1}(n)) + \mathsf{length}(\ell)(C_{\text{VAR}} + C_{\text{LREC}} + P_{N_2}(n))$$

This bound makes intuitive sense; the whole point of multiplying by n there was because it was suppose to bound the maximum number of times an iteration could run. Now that we are in the midst of the proof with access to dynamic information, we know we will iterate exactly $\mathsf{length}(\ell)$ times. This bound is indeed stronger; by Theorem 22, we know $\mathsf{length}(\ell) \leq \mathsf{size}(\mathcal{V}_1) \leq n$.

From here, the strategy is to proceed by induction on the structure of ℓ and prove termination alongside this tighter bound. The cases of this induction are relatively straightforward and do not require anything more complicated than what is seen in the application case above. The base case makes use of the induction hypothesis for N_1, and the inductive case makes use of the induction hypothesis for N_2. ◀