Wesleyan ⧫ University

EXTRACTING COST RECURRENCES FROM SEQUENTIAL AND PARALLEL FUNCTIONAL PROGRAMS

By

Justin Raymond

Faculty Advisor: Norman Danner

A Dissertation submitted to the Faculty of Wesleyan University in partial fulfillment of the requirements for the degree of Master of Arts

Middletown, Connecticut TDS

Acknowledgements

Thank you to my adviser Norman Danner for having the patience to put up with me the past year. Without him, this thesis would not have made it past the title page. Thanks also to Jim Lipton, Dan Licata, and Danny Krizanc who, along with Norman Danner, taught me everything I know about computer science.

Abstract

Complexity analysis aims to predict the resources, most often time and space, which a program requires. We build on previous work by Danner et al. [2013] and Danner et al. [2015] which formalizes the extraction of recurrences for evaluation cost from higher order functional programs. Source language programs are translated into a complexity language. The translation of an expression is a pair of a cost, a bound on the cost of evaluating the program to a value, and a potential, the cost of future use of the value. We use the formalization to analyze the time complexity of higher order functional programs. We also demonstrate the flexibility of the method by extending it to parallel cost semantics. Costs are cost graphs, which express dependencies between subcomputations in the program. We prove by logical relations that the extracted recurrences are an upper bound on the evaluation cost of the original program. We also give examples of the analysis of higher order functional programs under the parallel evaluation semantics.

Contents

CHAPTER 1

Introduction

1. Complexity Analysis

The efficiency of programs is categorized by how the resource usage of a program increases with the input size in the limit. This is often called the asymptotic efficiency or complexity of a program. Asymptotic efficiency abstracts away the details of efficiency, allowing programs to be compared without knowledge of specific hardware architecture or the size and shape of the programs input (Cormen et al. [2001]). However, traditional complexity analysis is first-order; the asymptotic efficiency of a program is only expressed in terms of its input. Consider the following function.

```
let rec map f xs =
  match xs with
  | [] -> []
  | (x::xs') -> f x :: map f xs'
```

The function `map` applies a function to every element in a list. Traditional analysis assumes the cost of applying its first argument is constant.

Traditional complexity analysis proceeds as follows. First we write a recurrence for the cost.

$$T(n) = c + T(n-1)$$

The variable n is the length of the list and the constant `c` is the cost of applying the function f to an element of the list and then applying the cons function `::`. The result is the asymptotic efficiency of `map` is linear in the length of the list.

The are two problems with this approach. The first is that there is no formal connection between the implementation of `map` and the recurrence $T(n)$. The consequence is extraction of the correct recurrence relies on the absence of human error, which the author of this thesis can attest the difficulty of doing. A formalization of the connection between the source program and the recurrence would prevent this. The translation from the source program to the recurrence could be done using application of a series of rules and the translation could even be automated.

The second is that the analysis assumes the cost of applying the function `f` to each element in the list has a constant cost. If the cost function is has a constant cost, such as fixed width integer addition, then this first-order analysis is sufficient. The cost of mapping a constant cost function over a list will increase linearly in the size of the list. However, first-order complexity analysis will not accurately describe the cost of mapping a nontrivial function over a list. The cost of mapping a quadratic time function such as insertion sort over a list of lists depends not only on the length of the list, but also on the length of the sublists. A more accurate prediction of the cost of this function can be obtained by taking into account the cost of insertion sort.

For an example such `map`, it is simple enough to change our analysis to reflect that applying the functions `f` does not have constant cost c, but instead has cost $f_c(x)$, where x is some notion of size of the elements of the list. If the elements of the list are fixed width integers, then all x would be equivalent, and $f_c(x)$ would be constant. If the elements of the list are strings or lists than the notion of size could be their length. If we only interpret the size of lists to be their lengths, because then we have no information about the size of the elements we apply f to. So we interpret lists as a pair of their largest element and their length. The recurrence for the cost of `map` becomes $T(n, x) = (f_c(x) + c)n$. Our analysis of the cost of `map` is now parameterized by the cost applying `f` to the elements of the list. However, this does not allow us to analyze the composition of functions. For example to analyze the cost of $g \circ f$, we need to have a notion of the size of the result of f, as well as the cost. We need to have a notion of the size of the result of f in order to analyze the cost of applying g to the result of applying f to some value.

The method of extracting recurrences presented in Danner et al. [2013] and Danner et al. [2015] addresses these problems. The first problem with traditional analysis is there is no formal connection between a program and the recurrence for its cost. There is a language which progams are written in, which we will refer to as the source language, and a language for recurrences, which we will refer to as the complexity language. In traditional complexity analysis, the source language is some programming language and the complexity language is mathematics, and there is no formalization of the translation from the source language to the complexity language. There is also no proof that the mathematical recurrence for the cost of the source language program is an upper bound on the cost of executing the source language program. Danner et al. [2013] address this problem by formally defining a translation from a source language to a complexity language and proving the recurrence in the complexity language is an upper bound on the cost of executing the source language program. The complexity language is also higher order. So if the source language program is higher order, such as `map`, the recurrence in the complexity language will reflect the cost of applying the higher order function. Finally, functions in the complexity language are from potentials to complexitys, so the result of applying a function results in both a cost of the function and a potential which can be used to analyze the cost of applying another function to the result.

2. Previous Work

Danner and Royer [2007], building on the work of others, introduced the idea that the complexity of an expression consists of a cost, representing an upper bound on the time it takes to evaluate the expression, and a potential, representing the cost of future uses of the expression. The notion of a potential allows the analysis of higher-order expressions. The complexity of a higher-order function such as `map` depends on the potential of its argument function. They developed a type system for ATR, a call-by-value version of System T, that consists of a part restricting the sizes of values of expressions and a part restricting the cost of evaluating a expression. Programs written in ATR are constrained by the type system to run in less than type-2 polynomial time. Danner and Royer [2009] extended this work to express more forms of recursion, in particular those required by insertion sort and selection sort.

Danner et al. [2013] utilized the notion of thinking of the complexity of an expression as a pair of a cost and a potential to statically analyze the complexity of a higher-order functional language with structural list recursion. The expressions in the higher-order functional language with structural list recursion, referred to as the source language, are mapped to expressions in a complexity language by a translation function. The translated expression describes an upper bound on the complexity of the original programs.

Danner et al. [2015] built on this work to formalize the extraction of recurrences from a higher-order functional language with structural recursion on arbitrary inductive data types. Programs are written in a higher order functional language, referred to as the source language. The programs are translated into a complexity language, which is essentially for recurrences. The result of the translation of an expression is a pair of a cost and a potential. The cost is a bound on the steps required to evaluate the expression to a value, the potential is a size which represents cost of future use of the value. A bounding relation is used to prove the translation and denotational semantics of the complexity language give an upper bound on the operational cost of running the source program. The paper also presents a syntactic bounding theorem, where the abstraction of values to sizes done syntactically instead of semantically. Arbitrary inductive data types are handled semantically using programmer specified sizes of data types. Sizes must be programmer specified because the structure of a data type does not always determine the interpretation of the size of a data type. There also exist different reasonable interpretations of size, and some may be preferable to others depending on what is being analyzed.

3. Contribution

This thesis comes in three parts.

Chapter two contains a catalog of examples of the extraction of recurrences from functional programs using the approach given by Danner et al. [2015]. These examples illustrate how to apply the method to nontrivial examples. They also serve to demonstrate common techniques for solving the extracted recurrences. The examples include reversing a list in quadratic, reversing a list in linear time, insertion sort, parametric insertion sort, list map, and tree map.

Chapter three extends the analysis to parallel programs. We change costs from from natural numbers to the cost graphs described in Harper [2012]. A cost graph represents the dependencies between subcomputations in a program. The nodes of the graph are subcomputations of the program and an edge between two nodes indicates the result of one computation is an input to the other. The cost graph can be used to determine an optimal strategy for scheduling the computation on multiple processors. The cost graph has two properties that we are interested in, work and span. The work is the total steps required to run the program, which corresponds to the steps a single processor must execute to run the program. The span is the critical path; the longest number of steps from the start to the end of the cost graph.

Chapter four demonstrates the recurrence for the potential does not depend on the recurrence for the cost. Consequently, we can extract the recurrence for the potential and analyze it independently. This is useful because it is often easier to solve the cost and potential recurrences independently than it is to solve the initial recurrence. We are also sometimes only interested in just the potential or just the cost of a recurrence.

CHAPTER 2

Higher Order Complexity Analysis

Programs are written in the source language. Then the program is translated to a complexity language. The semantic interpretation of the complexity language program may be used to analyse the complexity of the original program.

1. Source Language

The source language is the simply typed lambda calculus with `Unit`, products, suspensions, user-defined inductive datatypes and a recursion construct. Valid signatuers, types, and constructor arguments are given in Figure 2. The types, expressions, and typing judgments of the source language are given in Figure 1. Evaluation is call-by-value and the rules for evaluation are given in Figure 3.

`unit` is a singleton type with only one inhabitant, the value $\langle\rangle$, also called unit.

Product types are a compound types consisting of an ordered pair of types. Products are introduced using $\langle e_0, e_1\rangle$ Since evaluation is call-by-value, products are strict. So both expressions of in product must be evaluated before the product may be destructured. Products are eliminated using `split`.

A suspension is an unevaluated computation. A suspension has type `susp` τ where τ is the type of the suspended computation. Suspensions are introduced using the `delay`(e) operator. Suspensions are eliminated using the `force`(e) operator, which evaluates the suspended computation.

A program using datatypes must have a top-level signature ψ consisting of datatype declarations of the form

$$\texttt{datatype}\delta = C_0^\delta \texttt{of} \phi_{C_0}[\delta] \mid \ldots \mid C_{n-1}^\delta \texttt{of} \phi_{C_{n-1}}[\delta]$$

Each datatype may only refer to datatypes declared earlier in the signature. This prevents general recursive datatypes. The argument to each constructor is given by a strictly positive functor ϕ, which is one of t, τ, $\phi_0 \times \phi_1$, and $\tau \to \phi$. The identity functor t represents recursive occurence of the datatype. The constant functor τ represents a non-recursive type. The product functor $\phi_0 \times \phi_1$ represents a pair of arguments. The constant exponential $\tau \to \phi$ represents a function type. The introduction forms for datatypes are the constructors. The elimination form for a datatype is the `rec` construct.

The `rec` construct allows for structural recursion. `rec` is given an argument to recurse on and a sequence of statements corresponding to each constructor for the datatype of the first argument. The first argument to `rec` is evaluated to a value, and then depending on the outermost constructor of the value, `rec` evaluates to the appropriate branch.

`map` is used to lift functions from $\sigma \to \tau$ to $\phi[\sigma] \to \phi[\tau]$. `map` is restricted to syntactic values and is used in the operational semantics to insert recursive calls in their places. For example, if recursing on a value that does not contain a recursive occurence of a datatype, such as a boolean or a tree leaf, then `map` does not insert a recursive call anywhere.

FIGURE 1. Source language syntax and types

Types

$$\tau ::= \texttt{unit} \mid \tau \times \tau \mid \tau \to \tau \mid \texttt{susp}\ \tau \mid \delta$$

$$\phi ::= t \mid \tau \mid \phi \times \phi \mid \tau \to \phi$$

$$\texttt{datatype}\ \delta = C_0^\delta \texttt{of} \phi_{C_0}[\delta] \mid \ldots \mid C_{n-1}^\delta \texttt{of} \phi_{C_{n-1}}[\delta]$$

Expressions

$$v ::= x \mid \langle\rangle \mid \langle v, v\rangle \mid \lambda x.e \mid \texttt{delay}(e) \mid C\ v$$

$$e ::= x \mid \langle\rangle \mid \langle e, e\rangle \mid \texttt{split}(e, x.x.e) \mid \lambda x.e \mid e\ e$$

$$\mid \texttt{delay}(e) \mid \texttt{force}(e) \mid C^\delta\ e \mid \texttt{rec}^\delta(e, \overline{C \mapsto x.e_C})$$

$$\mid \texttt{map}^\phi(x.v, v) \mid \texttt{let}(e, x.e)$$

$$n ::= 0 \mid 1 \mid n + n$$

Typing Judgments

$$\frac{}{\gamma, x : \sigma \vdash x : \sigma} \quad \frac{}{\gamma \vdash \langle\rangle : \texttt{unit}}$$

$$\frac{\gamma \vdash e_0 : \tau_0 \qquad \gamma \vdash e_1 : \tau_1}{\langle e_0, e_1\rangle : \tau_0 \times \tau_1} \quad \frac{\gamma \vdash e_0 : \tau_0 \times \tau_1 \qquad \gamma, x_0 : \tau_0, x_1 : \tau_1 \vdash e_1 : \tau}{\gamma \vdash \texttt{split}(e_0, x_0.x_1.e_1) : \tau}$$

$$\frac{\gamma, x : \sigma \vdash e : \tau}{\gamma \vdash \lambda x.e : \sigma \to \tau} \quad \frac{\gamma \vdash e_0 : \sigma \to \tau \qquad \gamma \vdash e_1 : \sigma}{\gamma \vdash e_0\ e_1 : \tau}$$

$$\frac{\gamma \vdash e : \tau}{\gamma \vdash \texttt{delay}(e) : \texttt{susp}\ \tau} \quad \frac{\gamma \vdash e : \texttt{susp}\ \tau}{\gamma \vdash \texttt{force}(e) : \tau}$$

$$\frac{\gamma \vdash e : \phi_C[\delta]}{\gamma \vdash C^\delta\ e : \delta} \quad \frac{\gamma \vdash e : \delta \qquad \forall C.\gamma, x : \phi_C[\delta \times \texttt{susp}\ \tau] \vdash e_C : \tau}{\gamma \vdash \texttt{rec}^\delta(e, \overline{C \mapsto x.e_C}) : \tau}$$

$$\frac{\gamma, x : \tau_0 \vdash v_1 : \tau_1 \qquad \gamma \vdash v_0 : \phi[\tau_0]}{\texttt{map}^\phi(x.v_1, v_0) : \phi[\tau_1]} \quad \frac{\gamma \vdash e_0 : \sigma \qquad \gamma, x : \sigma \vdash e_1 : \tau}{\texttt{let}(e_0, x.e_1) : \tau}$$

FIGURE 2. Source language valid signatures, types, and constructor arguments

Signatures: ψ $\texttt{sig}$

$$\frac{}{\langle\rangle\ \texttt{sig}} \qquad \frac{\delta \notin \forall \mathrm{C}(\psi \vdash \phi_C \texttt{ok})}{\psi,\ \texttt{datatype}\ \delta = \overline{C\ \texttt{of}\ \phi_C[\delta]}\ \texttt{sig}}$$

Types : $\psi \vdash \tau$ $\texttt{type}$

$$\frac{}{\psi \vdash \texttt{unit type}} \qquad \frac{\psi \vdash \tau_0\ \texttt{type} \qquad \psi \vdash \tau_1\ \texttt{type}}{\psi \vdash \tau_0 \times \tau_1\ \texttt{type}}$$

$$\frac{\psi \vdash \tau_0\ \texttt{type} \qquad \psi \vdash \tau_1\ \texttt{type}}{\psi \vdash \tau_0 \to \tau_1\ \texttt{type}} \qquad \frac{\psi \vdash \tau\ \texttt{type}}{\psi \vdash \texttt{susp}\ \tau\ \texttt{type}} \qquad \frac{\delta \in \psi}{\psi \vdash \delta\ \texttt{type}}$$

Constructor arguments: $\psi \vdash \phi$ $\texttt{ok}$

$$\frac{}{\psi \vdash t\ \ \texttt{ok}} \qquad \frac{\psi \vdash \tau\ \ \texttt{type}}{\psi \vdash \tau\ \ \texttt{ok}}$$

$$\frac{\psi \vdash \phi_0\ \texttt{ok} \qquad \psi \vdash \phi_1\ \texttt{ok}}{\psi \vdash \phi_0 \times \phi_1\ \texttt{ok}} \qquad \frac{\psi \vdash \tau\ \texttt{type} \qquad \psi \vdash \phi\ \texttt{ok}}{\psi \vdash \tau \to \phi\ \texttt{ok}}$$

FIGURE 3. Source language operational semantics

$$\frac{e_0 \downarrow^{n_0} v_0 \qquad e_1 \downarrow^{n_1} v_1}{\langle e_0, e_1\rangle \downarrow^{n_0+n_1} \langle v_0, v_1\rangle} \qquad \frac{e_0 \downarrow^{n_0} \langle v_0, v_1\rangle \qquad e_1[v_0/x_0, v_1/x_1] \downarrow^{n_1} v}{\texttt{split}(e_0, x_0.x_1.e_1) \downarrow^{n_0+n_1} v}$$

$$\frac{e_0 \downarrow^{n_0} \lambda x.e_0' \qquad e_1 \downarrow^{n_1} v_1 \qquad e_0'[v_1/x] \downarrow^{n} v}{e_0\ e_1 \downarrow^{1+n_0+n_1+n} v} \qquad \frac{}{\texttt{delay}(e) \downarrow^{0} \texttt{delay}(e)}$$

$$\frac{e \downarrow^{n_0} \texttt{delay}(e_0) \qquad e_0 \downarrow^{n_1} v}{\texttt{force}(e) \downarrow^{n_0+n_1} v} \qquad \frac{e \downarrow^{n} v}{Ce \downarrow^{n} Cv}$$

$$\frac{e \downarrow^{n_0} Cv_0 \qquad \texttt{map}^{\phi_C}(y.\langle y, \texttt{delay}(rec(y, \overline{C \mapsto x.e_C}))\rangle, v_0) \downarrow^{n_1} v_1 \qquad e_C[v_1/x] \downarrow^{n_2} v}{rec(e, \overline{C \mapsto x.e_C}) \downarrow^{1+n_0+n_1+n_2} v}$$

$$\frac{}{\texttt{map}^{t}(x.v, v_0) \downarrow^{0} v[v_0/x]} \qquad \frac{}{\texttt{map}^{\tau}(x.v, v_0) \downarrow^{0} v_0}$$

$$\frac{\texttt{map}^{\phi_0}(x.v, v_0) \downarrow^{n_0} v_0' \qquad \texttt{map}^{\phi_1}(x.v, v_1) \downarrow^{n_1} v_1'}{\texttt{map}^{\phi_0 \times \phi_1}(x.v, \langle v_0, v_1\rangle) \downarrow^{n_0+n_1} \langle v_0', v_1'\rangle}$$

$$\frac{}{\texttt{map}^{\tau \to \phi}(x.v, \lambda y.e) \downarrow^{0} \lambda y.\texttt{let}(e, z.\texttt{map}^{\phi}(x.v, z))} \qquad \frac{e_0 \downarrow^{n_0} v_0 \qquad e_1[v_0/x] \downarrow^{n_1} v}{\texttt{let}(e_0, x.e_1) \downarrow^{n_0+n_1} v}$$

2. Complexity Language

The types, expressions, and typing judgments of the complexity language are given in Figure 4. The complexity language is similar to the source language with a few exceptions.

Suspensions are no longer present.

Tuples are destructured using projections instead of `split`.

FIGURE 4. Complexity language types, expressions, and typing judgments

Types

$$T ::= \mathbf{C} \mid \mathtt{unit} \mid \Delta \mid T \times T \mid T \to T$$

$$\Phi ::= t \mid T \mid \Phi \times \Phi \mid T \to \Phi$$

$$\mathbf{C} ::= 0 \mid 1 \mid 2 \mid ...$$

$$\mathtt{datatype}\Delta = C_0^\Delta \mathtt{of} \Phi_{C_0}[\Delta] \mid ... \mid C_{n-1}^\Delta \mathtt{of} \Phi_{C_{n-1}}[\Delta]$$

Expressions

$$E ::= x|0|1|E+E|\langle\rangle|\langle E,E\rangle|$$
$$\pi_0 E|\pi_1 E|\lambda x.E|E\ E|C^\delta\ E|\mathrm{rec}^\Delta(E, \overline{C \mapsto x.E_C})$$

Typing Judgments

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{}{\Gamma \vdash 0 : \mathbf{C}} \qquad \frac{}{\Gamma \vdash 1 : \mathbf{C}} \qquad \frac{}{\Gamma \vdash \langle\rangle : \mathbf{unit}}$$

$$\frac{\Gamma \vdash E_0 : \mathbf{C} \qquad \Gamma \vdash E_1 : \mathbf{C}}{\Gamma \vdash E_0 + E_1 : \mathbf{C}} \qquad \frac{\Gamma \vdash E_0 : T_0 \qquad \Gamma \vdash E_1 : T_1}{\Gamma \vdash \langle E_0, E_1\rangle : T_0 \times T_1}$$

$$\frac{\Gamma \vdash E : T_0 \times T_1}{\Gamma \vdash \pi_i E : T_i} \qquad \frac{\Gamma, x : T_0 \vdash E : T_1}{\Gamma \vdash \lambda x.E : T_0 \to T_1}$$

$$\frac{\Gamma \vdash E_0 : T_0 \to T_1 \qquad \Gamma \vdash E_1 : T_0}{\Gamma \vdash E_0\ E_1 : T_1} \qquad \frac{\Gamma \vdash E : \Phi_C[\Delta]}{\Gamma \vdash C^\Delta E : \Delta}$$

$$\frac{\Gamma \vdash E : \Delta \qquad \forall C. \Gamma, x : \Phi_C[\Delta \times T] \vdash E_C : T}{\Gamma \vdash \mathrm{rec}^\Delta(E, \overline{C \mapsto x.E_C}) : T}$$

The translation from the source language to the complexity language is given in Figure 5 and Figure 6.

FIGURE 5. Translation from source language to complexity language types.

$$\begin{aligned}
\|\tau\| &= \mathbf{C} \times \langle\langle \tau \rangle\rangle \\
\langle\langle \mathtt{unit} \rangle\rangle &= \mathtt{unit} \\
\langle\langle \sigma \times \tau \rangle\rangle &= \langle\langle \sigma \rangle\rangle \times \langle\langle \tau \rangle\rangle \\
\langle\langle \sigma \to \tau \rangle\rangle &= \langle\langle \sigma \rangle\rangle \to \|\tau\| \\
\langle\langle \mathtt{susp}\ \tau \rangle\rangle &= \|\tau\| \\
\langle\langle \delta \rangle\rangle &= \delta
\end{aligned}$$

$$\begin{aligned}
\|\phi\| &= \mathbf{C} \times \langle\langle \phi \rangle\rangle \\
\langle\langle t \rangle\rangle &= t \\
\langle\langle \tau \rangle\rangle &= \langle\langle \tau \rangle\rangle \\
\langle\langle \phi_0 \times \phi_1 \rangle\rangle &= \langle\langle \phi_0 \rangle\rangle \times \langle\langle phi_1 \rangle\rangle \\
\langle\langle \tau \to \phi \rangle\rangle &= \langle\langle \phi \rangle\rangle \to \|\phi\|
\end{aligned}$$

$$\langle\langle \psi \rangle\rangle = \text{ for each } \delta \in \psi, \delta = C_0^\delta \ \mathtt{of} \ \langle\langle \phi_{C_0} \rangle\rangle[\delta], ..., C_{n-1}^\delta \ \mathtt{of} \ \langle\langle \phi_{n-1} \rangle\rangle[\delta]$$

FIGURE 6. Translation from source language to complexity language expressions.

$$
\begin{aligned}
\|x\| &= \langle 0, x\rangle \\
\|\langle\rangle\| &= \langle 0, \langle\rangle\rangle \\
\|\langle e_0, e_1\rangle\| &= \langle \|e_0\|_c + \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p\rangle\rangle \\
\|\mathtt{split}(e_0, x_0.x_1.e_1)\| &= \|e_0\|_c +_c \|e_1\|[\pi_0\|e_0\|_p/x_0, \pi_1\|e_0\|_p/x_1] \\
\|\lambda x.e\| &= \langle 0, \lambda x.\|e\|\rangle \\
\|e_0\ e_1\| &= (1 + \|e_0\|_c + \|e_1\|_c) +_c \|e_0\|_p\|e_1\|_p \\
\|\mathtt{delay}(e)\| &= \langle 0, \|e\|\rangle \\
\|\mathtt{force}(e)\| &= \|e\|_c +_c \|e\|_p \\
\|\mathrm{C}_i^\delta\ e\| &= \langle \|e\|_c, \mathrm{C}_i^\delta\|e\|_p\rangle \\
\|\mathtt{rec}^\delta(e, \overline{C \mapsto x.e_C})\| &= \|e\|_c +_c \mathtt{rec}^\delta(\|e\|_p, \overline{C \mapsto x.1 +_c \|e_C\|}) \\
\|\mathtt{map}^\phi(x.v_0, v_1)\| &= \langle 0, \mathtt{map}^{\langle\langle\phi\rangle\rangle}(x.\|v_0\|_p, \|v_1\|_p)\rangle \\
\|\mathtt{let}(e_0, x.e_1)\rangle\rangle &= \|e_0\|_c +_c \|e_1\|[\|e_0\|_p/x]
\end{aligned}
$$

CHAPTER 3

Sequential Recurrence Extraction Examples

1. Fast Reverse

Fast reverse is an implementation reverse in linear time complexity. A naive implementation of reverse appends the head of the list to recursively reversed tail of the list. Fast reverse instead uses an abstraction to delay the consing. As this is the first example, we will walk through the translation and interpretation in gory detail. In following examples we will relegate the walk-through of the translation to the appendices, where the reader can peruse them, perhaps over a glass of carbenet sauvignon, as a relaxing end to a stressful day.

The definition of the list datatype holds no suprises.

$$\texttt{datatype list} = \texttt{Nil of unit} \mid \texttt{Cons of int} \times \texttt{list}$$

The implementation of fast reverse is not obvious. We write a function `rev` that applies an auxilary function to an empty list to produce the result. The specification of reverse is `rev` $[x_0, \ldots, x_{n-1}]$ = $[x_{n-1}, \ldots, x_0]$. The specification of the auxilary function `rec(xs,...)` is `rec(`$[x_0, \ldots, x_{n-1}]$`, ...)` $[y_0, \ldots, y_{m-1}]$ = $[x_{n-1}, \ldots, x_0, y_0, \ldots, y_{m-1}]$.

```
rev  xs = λxs.rec(xs,
                    Nil ↦ λa.a,
                    Cons↦b.split(b,x.c.split(c,xs'.r.
                                 λa.force(r)  Cons⟨x,a⟩)))  Nil
```

Notice that the implementation of `rev` would be much cleaner if we where able to pattern match on cases of the `rec`. Below is `rev` written with this syntactic sugar.

```
rev = λxs.rec(xs,  Nil ↦ λa.a,
```

```
Cons↦ ⟨y⟨ys , r⟩⟩ . λb . force ( r )  Cons⟨x , b⟩ )  Nil
```

Each recursive call creates an abstraction that applies the recursive call on the tail of the list to the list created by consing the head of the list onto the abstraction argument. The recursive calls builds nested abstractions as deep as the length of the list which is collapsed by application of the outermost abastraction to `Nil`. Below we show the evaluation of `rev` applied to a small list of just two elements.

```
rev  (Cons⟨0 ,Cons⟨1 ,  Nil⟩⟩)  →
   rec(Cons⟨0 ,Cons⟨1 , Nil⟩⟩ ,
         Nil  ↦ λa . a
         Cons↦b . split (b , x . c . split (c , xs ' . r .
                         λa . force ( r )  Cons⟨x , a⟩ ) ) )  Nil
   →* (λa0 . ( λa1 . ( λa2 . a2 )  Cons⟨1 , a1⟩ )  Cons⟨0 , a0⟩ )  Nil
   →β  (λa1 . ( λa2 . a2 )  Cons⟨1 , a1⟩ )  Cons⟨0 , Nil⟩
   →β  (λa2 . a2 )  Cons⟨1 ,Cons⟨0 , Nil⟩⟩
   →β  Cons⟨1 ,Cons⟨0 , Nil⟩⟩
```

1.1. Translation. We will walk through the translation from the source language to the complexity language.

$$\|\texttt{rev}\| = \|\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda\texttt{a.a},$$
$$\texttt{Cons} \mapsto b.\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle\texttt{x,a}\rangle)))\ \texttt{Nil}\|$$

First we apply the rule for translating an abstraction. The rule is $\|\lambda x.e\| = \langle 0, \lambda x.\|e\|\rangle$.

$$\|\texttt{rev}\| = \|\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a,$$
$$\texttt{Cons} \mapsto b.\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\texttt{Cons}\langle x, a\rangle)))\ \texttt{Nil}\|$$
$$= \langle 0, \lambda xs.\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a,$$
$$\texttt{Cons} \mapsto b.\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\texttt{Cons}\langle x, a\rangle)))\ \texttt{Nil}\|\rangle$$

The next translation is an application. The rule for translating an application is $\|e_0\ e_1\| = (1 + \|e_0\|_c + \|e_1\|_c) +_c (\|e_0\|_p\ \|e_1\|_p)$. In this case, `rec(...)` is e_0 and `Nil` is e_1. We translate `Nil` then `rec(...)` seperately. The translation of a constructor applied to an expression is a tuple of the cost of the translated expression and the corresponding complexity language constructor applied to the potential of the translated expression. Since the expression inside `Nil` is $\langle\rangle$, and $\|\langle\rangle\| = \langle 0, \langle\rangle\rangle$, we have

$$
\begin{aligned}
|\texttt{Nil}\| &= \langle\langle 0, \langle\rangle\rangle_c, \texttt{Nil}\langle 0, \langle\rangle\rangle_p\rangle \\
&= \langle 0, \texttt{Nil}\langle\rangle\rangle
\end{aligned}
$$

The rule for translating a `rec` expression is

$$\|\texttt{rec}(e, \overline{C \mapsto x.e_C})\| = \|e\|_c +_c \texttt{rec}(\|e\|_p, \overline{C \mapsto x.\|e_C\|})$$

$$
\begin{aligned}
&\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a, \\
&\qquad \texttt{Cons} \mapsto b.\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)))\| \\
&= \|xs\|_c +_c \texttt{rec}(\|xs\|_p, \texttt{Nil} \mapsto 1 +_c \|\lambda a.a\| \\
&\qquad\quad \texttt{Cons} \mapsto b.1 +_c \|\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle))\|) \\
&= \langle 0, xs\rangle_c +_c \texttt{rec}(\langle 0, xs\rangle_p, \texttt{Nil} \mapsto 1 +_c \|\lambda a.a\| \\
&\qquad\quad \texttt{Cons} \mapsto b.1 +_c \|\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle))\|)
\end{aligned}
$$

The term xs is a variable and the rule for translating variables is $\|xs\| = \langle 0, xs\rangle$.

$$
\begin{aligned}
&= \texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \|\lambda a.a\| \\
&\qquad\quad \texttt{Cons} \mapsto b.1 +_c \|\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle))\|)
\end{aligned}
$$

The translation of the `Nil` branch is simple application of the $\|\lambda x.e\| = \langle 0, \lambda x.\|e\|\rangle$ and the variable translation rule.

$$
\begin{aligned}
&1 +_c \|\lambda a.a\| \\
&= 1 +_c \langle 0, \lambda a.\|a\|\rangle \\
&= \langle 1, \lambda a.\langle 0, a\rangle\rangle
\end{aligned}
$$

The translation of the `Cons` branch is a slightly more involved. The rule for translating `split` is

$$\|\texttt{split}(e_0, x_0.x_1.e_1)\| = \|e_0\|_c +_c \|e_1\|[\pi_0\|e_0\|_p/x_0, \pi_1\|e_0\|_p/x_1]$$

After applying the rule to the `Cons` branch we get

$$\begin{aligned}
&1 +_c \|\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle))\| \\
&= 1 +_c \|b\|_c +_c \|\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\|[\pi_0\|b\|_p/x, \pi_1\|b\|_p/c]
\end{aligned}$$

Remember that b is a variable and has type $\phi_{\texttt{Cons}}[\texttt{list} \times \texttt{susp (list} \to \texttt{list)}]$. The translation of this type is $\mathbf{C} \times \langle\langle \phi_{\texttt{Cons}} \rangle\rangle[\texttt{list} \times \langle \texttt{list} \to \langle \mathbf{C} \times \texttt{list}\rangle\rangle]$. We can say that $\pi_0\|\texttt{b}\|_p$ is the head of the list `xs`, $\pi_0\pi_1\|\texttt{b}\|_p$ is the tail of the list `xs`, and $\pi_1\pi_1\|\texttt{b}\|_p$ is the result of the recursive call. The translation of b is $\langle 0, b\rangle$.

$$1 +_c \|b\|_c +_c \|\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\|[\pi_0\|b\|_p/x, \pi_1\|b\|_p/c]$$

$$= 1 +_c \|\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\|[\pi_0\|b\|_p/x, \pi_1\|b\|_p/c]$$

We apply the rule for `split` again.

$$= 1 +_c (\|c\|_c +_c \|\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle\|[\pi_0\|c\|_p/xs', \pi_1\|c\|_p/r][\pi_0\|b\|_p/x, \pi_1\|b\|_p/c]$$

c is a variable, so its translation is $\langle 0, c\rangle$.

$$= 1 +_c \|\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle\|[\pi_0\|c\|_p/xs', \pi_1\|c\|_p/r][\pi_0\|b\|_p/x, \pi_1\|b\|_p/c]$$

We apply the rule for abstraction.

$$= 1 +_c \langle 0, \lambda a.\|\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle\|[\pi_0\|c\|_p/xs', \pi_1\|c\|_p/r][\pi_0\|b\|_p/x, \pi_1\|b\|_p/c]$$

Recall $C +_c E$ is a macro for $\langle C + E_c, E_p\rangle$. We use this to eliminate the $+_c$.

We also apply the translation rule for application.

$$\begin{aligned}
= \langle 1, \lambda a.(1 &+ \|\texttt{force}(r)\|_c + \|\texttt{Cons}\langle x, a\rangle\|_c) \\
&+_c \|\texttt{force}(r)\|_p\|\texttt{Cons}\langle x, a\rangle\|_p\rangle[\pi_0\|c\|_p/xs', \pi_1\|c\|_p/r][\pi_0\|b\|_p/x, \pi_1\|b\|_p/c]
\end{aligned}$$

We will translate $\texttt{force}(r)$ and $\texttt{Cons}\langle x, a\rangle$ individually.

First we compose the two substitutions.

$$\begin{aligned}\text{let } \Theta &= [\pi_0\|c\|_p/xs', \pi_1\|c\|_p/r][\pi_0\|b\|_p/x, \pi_1\|b\|_p/c] \\ &= [\pi_0\pi_1\|b\|_p/xs', \pi_1\pi_1\|b\|_p/r, \pi_0\|b\|_p/x]\end{aligned}$$

Since b is a variable, the potential of its translation is b.

$$\Theta = [\pi_0\pi_1 b/xs', \pi_1\pi_1 b/r, \pi_0 b/x]$$

In translation of $\texttt{force}(r)$ we apply the rule $\|\texttt{force}(e)\| = \|e\|_c +_c \|e\|_p$.

$$\|\texttt{force}(r)\|\Theta = \|r\|_c\Theta +_c \|r\|_p\Theta$$

We apply the variable translation rule to r, then apply the substitution Θ.

$$\begin{aligned}&= \langle 0, r\rangle_c\Theta +_c \langle 0, r\rangle_p\Theta \\ &= r\Theta = \pi_1\pi_1 b\end{aligned}$$

Next we do the translation of $\texttt{Cons}\langle x, a\rangle$.

$$\|\texttt{Cons}\langle x, a\rangle\| = \langle \|\langle x, a\rangle\|_c, \texttt{Cons}\|\langle x, a\rangle\|_p\rangle$$

Notice the translation of $\langle x, a\rangle$ appears twice, so we will do this seperately.

$$\|\langle x, a\rangle\| = \langle \|x\|_c + \|a\|_c, \langle \|x\|_p, \|a\|_p\rangle\rangle\Theta$$

Both x and a are variables, so they have 0 cost.

$$= \langle 0, \langle x, a\rangle\rangle\Theta$$

We apply the substitution Θ.

$$\begin{aligned}&= \langle 0, \langle \pi_1 b, \pi_1\pi_1 b\rangle\rangle \\ &= \langle 0, \langle \pi_1 b, \pi_1\pi_1 b\rangle\rangle\end{aligned}$$

We complete the translation of $\texttt{Cons}\langle x, a\rangle$ using $\langle x, a\rangle$.

$$\begin{aligned}\|\texttt{Cons}\langle x, a\rangle\| &= \langle \|\langle x, a\rangle\|_c, \texttt{Cons}\|\langle x, a\rangle\|_p\rangle \\ &= \langle 0, \texttt{Cons}\langle \pi_1 b, \pi_1\pi_1 b\rangle\rangle\end{aligned}$$

We use substitute in the translations of $\texttt{force}(r)$ and $\texttt{Cons}\langle x, a\rangle$.

$\|\texttt{force}(r)\|$ has cost $(\pi_1\pi_1 b)_c$ and $\|\texttt{Cons}\langle x, a\rangle\|$ has cost 0.

$$\langle 1, \lambda a.(1 + \|\texttt{force}(r)\|_c + \|\texttt{Cons}\langle x, a\rangle\|_c) +_c \|\texttt{force}(r)\|_p \|\texttt{Cons}\langle x, a\rangle\|_p\rangle\rangle\Theta$$

$$= \langle 1, \lambda a.(1 + (\pi_1\pi_1 b)_c) +_c (\pi_1\pi_1 b)_p\ \texttt{Cons}\langle \pi_1 b, a\rangle\rangle$$

We can now complete the translation of the rec expression.

$$\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a,$$

$$\texttt{Cons} \mapsto b.\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)))\|$$

$$= \texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \|\lambda a.a\|$$

$$\texttt{Cons} \mapsto b.1 +_c \|\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle))\|)$$

$$= \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto b.\langle 1, \lambda a.(1 + (\pi_1\pi_1 b)_c) +_c (\pi_1\pi_1 b)_p\ \texttt{Cons}\langle \pi_1 b, a\rangle\rangle)$$

We substitute the translation of rec and Nil into the translation of the application.

Let $R = \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$

$$\texttt{Cons} \mapsto b.\langle 1, \lambda a.(1 + (\pi_1\pi_1 b)_c) +_c (\pi_1\pi_1 b)_p\ \texttt{Cons}\langle \pi_1 b, a\rangle\rangle)$$

$$\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a,$$

$$\texttt{Cons} \mapsto b.\texttt{split}(b, x.c.\texttt{split}(c, xs'.r.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)))\ \texttt{Nil}\|$$

Substituting R for the translation of rec and $\langle 0, \texttt{Nil}\rangle$ for the translation of Nil.

$$= (1 + R_c) +_c R_p\ \texttt{Nil}\rangle$$

Recall $C +_c E = \langle C + E_c, E_p\rangle$, so $(1 + E_c) +_c E_p = 1 +_c E$

$$= 1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto b.\langle 1, \lambda a.(1 + (\pi_1\pi_1 b)_c) +_c (\pi_1\pi_1 b)_p\ \texttt{Cons}\langle \pi_1 b, a\rangle\rangle)\ \texttt{Nil}$$

Finally, we substitute this into the translation of `rev`.

$$\begin{aligned}\|\texttt{rev}\| &= \|(\lambda\texttt{xs.rec(xs, Nil} \mapsto \lambda\texttt{a.a,}\\ &\quad \texttt{Cons} \mapsto \texttt{b.split(b,x.c.split(c,xs'.r.}\lambda\texttt{a.force(r) Cons}\langle\texttt{x,a}\rangle\texttt{)))) Nil}\|\\ &= \langle 0, \lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle\\ &\qquad \texttt{Cons} \mapsto b.\langle 1, \lambda a.(1 + (\pi_1\pi_1 b)_c) +_c (\pi_1\pi_1 b)_p\ \texttt{Cons}\langle \pi_1 b, a\rangle\rangle)\ \texttt{Nil}\rangle\end{aligned}$$

Observe that $\|\texttt{rev}\|$ admits the same syntactic sugar as `rev`. In the complexity language, instead of taking projections of b, we can use the same pattern matching syntactic sugar as in the source language.

$$\begin{aligned}\|\texttt{rev}\| = \langle 0, \lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle\\ \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle \pi_1 x, a\rangle\rangle)\ \texttt{Nil}\rangle\end{aligned}$$

1.2. Syntactic Sugar Translation. We walk through the same translation of fast reverse, but we use the syntactic sugar for matching introducted earlier. Recall the implementation of fast using syntactic sugar. The translation is almost identical to the translation of `rev` written without syntactic sugar until we translate the `Cons` branch of the `rec`.

$$\begin{aligned}\|\texttt{rev}\| = \|\lambda\texttt{xs.rec(xs, Nil} \mapsto \lambda\texttt{a.a,}\\ \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\ \texttt{Nil}\|\end{aligned}$$

First we apply the rule for translating an abstraction. The rule is $\|\lambda x.e\| = \langle 0, \lambda x.\|e\|\rangle$.

$$\begin{aligned}\|\texttt{rev}\| = \langle 0, \lambda xs.\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a,\\ \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\ \texttt{Nil}\|\rangle\end{aligned}$$

Next we apply the rule for translating an application. The rule is $\|e_0\ e_1\| = (1 + \|e_0\|_c + \|e_1\|_c) +_c (\|e_0\|_p\ \|e_1\|_p)$. In this case, `rec(...)` is e_0 and `Nil` is e_1. We translate `Nil` then rec(...) seperately. The translation of a constructor applied to an expression is a tuple of the cost of the translated expression and the corresponding complexity language constructor applied to the potential of the translated expression. Since the expression inside Nil is $\langle\rangle$, and $\|\langle\rangle\| = \langle 0, \langle\rangle\rangle$, we have

$$\begin{aligned}\|\mathtt{Nil}\| &= \langle\langle 0, \langle\rangle\rangle_c, \mathtt{Nil}\langle 0, \langle\rangle\rangle_p\rangle \\ &= \langle 0, \mathtt{Nil}\langle\rangle\rangle\end{aligned}$$

The rule for translating a rec expression is

$$\|\mathtt{rec}(e, \overline{C \mapsto x.e_C})\| = \|e\|_c +_c \mathtt{rec}(\|e\|_p, \overline{C \mapsto x.\|e_C\|})$$

$$\begin{aligned}&\|\mathtt{rec}(xs, \mathtt{Nil} \mapsto \lambda a.a, \\ &\qquad \mathtt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\lambda a.\mathtt{force}(r)\ \mathtt{Cons}\langle x, a\rangle)\| \\ &= \|xs\|_c +_c \mathtt{rec}(\|xs\|_p, \mathtt{Nil} \mapsto 1 +_c \|\lambda a.a\| \\ &\qquad \mathtt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c \|\lambda a.\mathtt{force}(r)\ \mathtt{Cons}\langle x, a\rangle\|) \\ &= \langle 0, xs\rangle_c +_c \mathtt{rec}(\langle 0, xs\rangle_p, \mathtt{Nil} \mapsto 1 +_c \|\lambda a.a\| \\ &\qquad \mathtt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c \|\lambda a.\mathtt{force}(r)\ \mathtt{Cons}\langle x, a\rangle\|)\end{aligned}$$

The term xs is a variable and the rule for translating variables is $\|xs\| = \langle 0, xs\rangle$.

$$\begin{aligned}&= \mathtt{rec}(xs, \mathtt{Nil} \mapsto 1 +_c \|\lambda a.a\| \\ &\qquad \mathtt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c \|\lambda a.\mathtt{force}(r)\ \mathtt{Cons}\langle x, a\rangle\|)\end{aligned}$$

The translation of the Nil branch is the same as before.

$$1 +_c \|\lambda a.a\| = \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

The translation of the Cons branch is much simpler without the two splits.

$$\begin{aligned}&1 +_c \|\lambda a.\mathtt{force}(r)\ \mathtt{Cons}\langle x, a\rangle\| \\ &\quad = 1 +_c \langle 0, \lambda a.\|\mathtt{force}(r)\ \mathtt{Cons}\langle x, a\rangle\|\rangle \\ &\quad = \langle 1, \lambda a.(1 + \|\mathtt{force}(r)\|_c + \|\mathtt{Cons}\langle x, a\rangle)\|_c) +_c \|\mathtt{force}(r)\|_p\ \|\mathtt{Cons}\langle x, a\rangle\|_p\rangle\end{aligned}$$

The translation of $\texttt{force}(r)$ and $\texttt{Cons}\langle x, a\rangle$ are the same as before, except we do not have a substitution to apply.

$$\|\texttt{force}(r)\| = \|r\|_c +_c \|r\|_p = \langle 0, r\rangle_c +_c \langle 0, r\rangle_p = 0 +_c r = r$$

$$\|\texttt{Cons}\langle x, a\rangle\| = \langle 0, \texttt{Cons}\langle x, a\rangle\rangle$$

So the complete translation of the $\texttt{Cons}$ branch is

$$\begin{aligned}
&1 +_c \|\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle\| \\
&\quad = 1 +_c \langle 0, \lambda a.\|\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle\|\rangle \\
&\quad = \langle 1, \lambda a.(1 + \|\texttt{force}(r)\|_c + \|\texttt{Cons}\langle x, a\rangle)\|_c) +_c \|\texttt{force}(r)\|_p\ \|\texttt{Cons}\langle x, a\rangle\|_p\rangle \\
&\quad = \langle 1, \lambda a.(1 + r_c + 0) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle \\
&\quad = \langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle
\end{aligned}$$

The complete translation of the $\texttt{rec}$ becomes

$$\begin{aligned}
&\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a, \\
&\qquad \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\| \\
&= \texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \|\lambda a.a\| \\
&\qquad \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c \|\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle\|) \\
&= \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 0, \lambda a.\langle 0, a\rangle\rangle \\
&\qquad \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle
\end{aligned}$$

We substitute the translations of `rec(..)` and `Nil` into the application.

$$\begin{aligned}
&\text{Let } R = \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle \\
&\qquad \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle
\end{aligned}$$

$$\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a,$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\ \texttt{Nil}\|$$

Substituting R for the translation of `rec` and $\langle 0, \texttt{Nil}\rangle$ for the translation of `Nil`.

$$= (1 + R_c) +_c R_p\ \texttt{Nil}\rangle$$

$$= 1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)\ \texttt{Nil}$$

And our complete translation of `rev` is

$$\|\texttt{rev}\| = \|\lambda\texttt{xs.rec(xs, Nil} \mapsto \lambda\texttt{a.a,}$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\ \texttt{Nil}\|$$

$$= \langle 0, \lambda xs.\|\texttt{rec}(xs, \texttt{Nil} \mapsto \lambda a.a,$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\lambda a.\texttt{force}(r)\ \texttt{Cons}\langle x, a\rangle)\ \texttt{Nil}\|\rangle$$

$$= \langle 0, \lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)\ \texttt{Nil}\rangle$$

This is the same as the translation of `rev` without the syntactic sugar. We will use the syntactic sugar for the rest of this thesis.

1.3. Interpretation. The interpretation of `rev` is not interesting as the cost of `rev` is always null. Instead of interpreting `rev`, we will interpret `rev` applied to a list `xs`. Below is the translation of `rev xs`.

$$\|\texttt{rev xs}\| = (1 + \|\texttt{rev}\|_c + \|xs\|_c) +_c \|\texttt{rev}\|_p\ \|xs\|_p$$

The cost of $\|\texttt{rev}\|$ is 0, and we will let `xs` be a value, which has 0 cost.

$$= (1 + 0 + 0) +_c \|\texttt{rev}\|_p\ xs$$

$$= 1 +_c (\lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)\ \texttt{Nil})\ xs$$

The cost of `rev` is driven by the auxilary function `rec(...)`. The cost of `rev` will be determined by the cost of the auxilary function `rec(...)` applied to `Nil` plus some constant factor. We will interpret the auxilary function in the following denotational semantics. We intepret the size of an `list` to be the number of list constructors.

$$
\begin{aligned}
[\![\texttt{list}]\!] &= \mathbb{N}^\infty \\
D^{list} &= \{*\} + \{1\} \times \mathbb{N}^\infty \\
size_{list}(\texttt{Nil}) &= 1 \\
size_{list}(\texttt{Cons(1,n)}) &= 1 + n
\end{aligned}
$$

We define the macro $R(xs)$ as the translation of the auxilary function `rec(...)` to avoid repeated coping of the translation.

$$
\begin{aligned}
\text{Let } R(xs) = \texttt{rec}(xs, \texttt{Nil} &\mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle \\
\texttt{Cons} &\mapsto b.\langle 1, \lambda a.(1 + \pi_1\pi_1 b_c) +_c \pi_1\pi_1 b_p\ \texttt{Cons}\langle \pi_0 b, a\rangle\rangle
\end{aligned}
$$

The recurrence $g(n)$ is the interpretation of the auxilary function $R(xs)$, where n is the interpretation of xs.

$$
\begin{aligned}
&g(n) = [\![R(xs)]\!]\{xs \mapsto n\} \\
&= \bigvee_{size\ z \le [\![xs]\!]\{xs \mapsto n\}} case(z, f_C, f_N) \\
&\text{where} \\
&f_{Nil}(x) = [\![\langle 1, \lambda a.\langle 0, a\rangle\rangle]\!]\{xs \mapsto n\} \\
&\qquad = (1, \lambda\!\!\lambda a.(0, a)) \\
&f_{Cons}(b) = [\![\langle 1, \lambda a.(1 + \pi_1\pi_1 b_c) +_c \pi_1\pi_1 b_p\ \texttt{Cons}\langle \pi_0 b, a\rangle\rangle]\!] \\
&\qquad\qquad \{xs \mapsto n, b \mapsto map^{\Phi_{Cons}}(\lambda\!\!\lambda d.(d, [\![R(w)]\!]\{w \mapsto d, xs \mapsto n\}), b)\}
\end{aligned}
$$

Let us take a moment to analyze the semantic *map*. The definition mirrors the definition of the `map` macro in the complexity language. Since b is a tuple, *map* over a tuple is defined as the tuple of the *map* over the projections of the tuple.

$$map^{\Phi_{Cons}}(\lambda\!\!\lambda d.(d, [\![R(w)]\!]\{w \mapsto d\}), b)$$

$$= (map^{int}(\lambda\!\!\lambda d.(d, [\![R(w)]\!]\{w \mapsto d\}), \pi_0 b),$$

$$map^{list}(\lambda\!\!\lambda d.(d, [\![R(w)]\!]\{w \mapsto d\}), \pi_1 b))$$

The definition of *map* over *int* is $map^{int}(\lambda\!\!\lambda x.V_0, V_1) = V_1$.

$$= (\pi_0 b, map^{list}(\lambda\!\!\lambda d.(d, [\![R(w)]\!]\{w \mapsto d\}), \pi_1 b))$$

The definition of *map* over a recursive occurence of a

a datatype is $map^T(\lambda\!\!\lambda x.V_0, V_1) = V_0[V_1/x]$.

$$= (\pi_0 b, (\pi_1 b, [\![R(w)]\!]\{w \mapsto \pi_1 b\}))$$

Observe that we can substitute $g(\pi_1 b)$ for $[\![R(w)]\!]\{w \mapsto \pi_1 b\}$.

$$= (\pi_0 b, (\pi_1 b, g(\pi_1 b)))$$

Let us resume our interpretation of `rec(...)`.

$$f_{Cons}(b) = [\![\langle 1, \lambda a.(1 + \pi_1\pi_1 b_c) +_c \pi_1\pi_1 b_p\ \texttt{Cons}\langle \pi_0 b, a\rangle\rangle]\!]$$

$$\{xs \mapsto n, b \mapsto map^{\Phi_{Cons}}(\lambda\!\!\lambda d.(d, [\![R(w)]\!]\{w \mapsto d\}), b)\}$$

$$= [\![\langle 1, \lambda a.(1 + \pi_1\pi_1 b_c) +_c \pi_1\pi_1 b_p\ \texttt{Cons}\langle \pi_0 b, a\rangle\rangle]\!]$$

$$\{xs \mapsto n, b \mapsto (\pi_0 b, (\pi_1 b, g(\pi_1 b)))\}$$

$$= (1, [\![\lambda a.(1 + \pi_1\pi_1 b_c) +_c \pi_1\pi_1 b_p\ \texttt{Cons}\langle \pi_0 b, a\rangle]\!]$$

$$\{xs \mapsto n, b \mapsto (\pi_0 b, (\pi_1 b, g(\pi_1 b)))\})$$

$$= (1, \lambda\!\!\lambda a.[\![(1 + \pi_1\pi_1 b_c) +_c \pi_1\pi_1 b_p\ \texttt{Cons}\langle \pi_0 b, a\rangle]\!]$$

$$\{xs \mapsto n, b \mapsto (\pi_0 b, (\pi_1 b, g(\pi_1 b))), a \mapsto a\})$$

$$= (1, \lambda\!\!\lambda a.(1 + g_c(\pi_1 b)) +\!\!\!+_c\, g_p(\pi_1 b)\ (1 + a))$$

So the initial extracted recurrence from `rec` is

$$g(n) = \bigvee_{size\ z \leq n} case(z, f_C, f_N)$$

where

$$f_{Nil}(x) = (1, \lambda\!\!\lambda a.(0, a))$$

$$f_{Cons}(b) = (1, \lambda\!\!\lambda a.(1 + g_c(\pi_1 b)) +\!\!\!+_c\, g_p(\pi_1 b)\ (a + 1))$$

To obtain a closed form solution for the recurrence, we must eliminate the big maximum operator. To do so we break the defnition of g into two cases.

case $n = 0$**:**

For $n = 0$, $g(0) = (1, \lambda\!\!\lambda a.(0, a))$.

case $n > 0$**:**

$$\begin{aligned}
g(n+1) &= \bigvee_{size\ ys \leq n+1} case(ys, f_{Nil}, f_{Cons}) \\
&= \bigvee_{size\ ys \leq n} case(ys, f_{Nil}, f_{Cons}) \vee \bigvee_{size\ ys = n+1} case(ys, f_{Nil}, f_{Cons}) \\
&= g(n) \vee \bigvee_{size\ ys = n+1} case(ys, \lambda\!\!\lambda().(1, \lambda\!\!\lambda a.(0, a)), \\
&\qquad\qquad \lambda\!\!\lambda(1, m).(1, \lambda\!\!\lambda a.(1 + g_c(m)) +\!\!\!+_c\, g_p(m)(a + 1))) \\
&= g(n) \vee \langle 1, \lambda\!\!\lambda a.(1 + g_c(n)) +\!\!\!+_c\, g_p(n)(a + 1)))
\end{aligned}$$

In order to eliminate the remaining max operator, we want to show that g is monotonically increasing; $\forall n.g(n) \leq g(n+1)$. By definition of $\leq$, $g(n) \leq g(n+1) \Leftrightarrow g_c(n) \leq g_c(n+1) \wedge g_p(n) \leq g_p(n+1)$. First we will show lemma 3.1, which states the cost of $g(n)$ is always one.

LEMMA 3.1. $\forall n.g_c(n) = 1$.

PROOF. We prove this by induction on n.

Base case: $n = 0$**:**

By definition, $g_c(0) = (1, \lambda\!\!\lambda a.(0, a)) = 1$.

Induction step: $n > 0$**:**

By definition $g_c(n+1) = (g(n) \vee (1, \lambda\!\!\lambda a.(1 + g_c(n)) +\!\!\!+_c\, g_p(n)\ (a+1)))_c$. We distribute the projection over the max: $g_c(n+1) = g_c(n) \vee 1$. By the induction hypothesis, $g_c(n) = 1$, so $g_c(n+1) = 1$.

□

The immediate corollary of this is $g_c(n)$ is monotonically increasing.

COROLLARY 3.1.1. $\forall n. g_c(n) \leq g_c(n+1)$.

First we prove the lemma stating the potential of $g(n)\ a$ is monotonically increasing.

LEMMA 3.2. $\forall n. g_p(n)\ a \leq g_p(n)\ (a+1)$

PROOF. We prove this by induction on n.

$n = 0$**:**

$g_p(0)\ a = (\lambda\!\!\lambda a.(0, a))\ a = (0, a)$

$g_p(0)\ (a+1) = (\lambda\!\!\lambda a.(0.a))\ (a+1) = (0, a+1)$

$(0, a) \leq (0, a+1)$.

$n > 0$**:**

We assume $g_p(n)\ a \leq g_p(n)\ (a+1)$.

$$g_p(n)\ a \leq g_p(n)\ (a+1)$$

$$g_p(n)\ a \vee (1 + g_c(n)) +\!\!\!+_c\, g_p(n)\ a \leq g_p(n)\ (a+1) \vee (1 + g_c(n)) +\!\!\!+_c\, g_p(n)\ (a+1)$$

$$g_p(n+1)\ a \leq g_p(n+1)\ (a+1)$$

□

Now we show $g_p(n) \leq g_p(n+1)$.

PROOF. By reflexivity, $g_p(n) \leq g_p(n)$. By the lemma we just proved:

$$g_p(n)\ a \leq g_p(n)\ (a+1)$$

$$g_p(n)\ a \leq (1 + g_c(n)) +\!\!\!+_c\, g_p(n)\ (a+1)$$

$$\lambda\!\!\lambda a.g_p(n)\ a \leq \lambda\!\!\lambda a.(1 + g_c(n)) +\!\!\!+_c\, g_p(n)\ (a+1)$$

$\square$

So since for all n, $g_c(n) = 1$ and $g_p(n) \leq \lambda\!\!\lambda a.(1+g_c(n)) +\!\!\!+_c\, g_p(n)\ (a+1)$, we conclude

$$g(n) \leq \langle 1, \lambda\!\!\lambda a.(1 + g_c(n)) +\!\!\!+_c\, g_p(n)\ (a+1) \rangle)$$

So

$$g(n+1) = \langle 1, \lambda\!\!\lambda a.(1 + g_c(n)) +\!\!\!+_c\, g_p(n)\ (a+1) \rangle$$

To extract a recurrence from g, we apply g to the interpretation of a list a. Let $h(n, a) = g_p(n)\ a$. For $n = 0$

$$\begin{aligned} h(0, a) &= g_p(0)a \\ &= (\lambda\!\!\lambda a.(0, a))a \\ &= (0, a) \end{aligned}$$

For $n > 0$

$$\begin{aligned} h(n, a) &= g_p(n)a \\ &= (\lambda\!\!\lambda a.(1 + g_c(n-1)) +\!\!\!+_c\, g_p(n-1)\ (a+1))\ a \\ &= (1 + g_c(n-1)) +_c g_p(n-1)(a+1) \\ &= (1+1) +_c h(n-1, a+1) \\ &= (2 + h_c(n-1, a+1), h_p(n-1, a+1)) \end{aligned}$$

From this recurrence, we can extract a recurrence for the cost. For $n = 0$

$$h_c(0, a) = (0, a)_c = 0$$

For $n > 0$

$$h_c(n, a) = (2 + h_c(n-1, a+1), h_p(n-1, a+1))_c = 2 + h_c(n-1, a+1)$$

We now have a recurrence for the cost of the auxilary function `rec(xs,...)` when applied to some list:

$$h_c(n,a) = \begin{cases} 0 & n = 0 \\ 2 + h_c(n-1, a+1) & n > 0 \end{cases} \tag{1}$$

We state the solution to the recurrence h_c is $2n$.

THEOREM 3.3. $h_c(n,a) = 2n$

PROOF. We prove this by induction on n.

: case $n = 0$

$h_c(0,a) = 0 = 2 \cdot 0$

: case $n > 0$

We assume $h_c(n, a+1) = 2n$.

$$\begin{aligned} h_c(n+1, a) &= 2 + h_c(n, a+1) \\ &= 2 + 2n \\ &= 2(n+1) \end{aligned}$$

□

So we have proved the interpretation of applying the auxilary function of `rev xs` to a list is linear in the length of `xs`.

We can also extract a recurrence for the potential. For $n = 0$

$$\begin{aligned} h_p(0,a) &= h_p(0,a) \\ &= (0,a)_p \\ &= a \end{aligned}$$

For $n > 0$

$$\begin{aligned} h_p(n,a) &= (2 + h_c(n-1, a+1), h_p(n-1, a+1))_p \\ &= h_p(n-1, a+1) \end{aligned}$$

We now have a recurrence for the potential of the auxilary function in `rev xs` when applied to some list a.

$$h_p(n,a) = \begin{cases} a & n = 0 \\ h_p(n-1, a+1) & n > 0 \end{cases} \tag{2}$$

THEOREM 3.4. $h_p(n,a) = n + a$

PROOF. We prove this by induction on n.

: case $n = 0$

$$h_p(0,a) = a$$

: case $n > 0$

$$\begin{aligned} h_p(n,a) &= h_p(n-1, a+1) \\ &= n - 1 + a + 1 && \text{by the induction hypothesis} \\ &= n + a \end{aligned}$$

$\square$

Now that we have obtained a closed form solution for the recurrence describing the cost and potential of the auxilary function that drives the cost of `rev`, we can obtain the interpretations for the cost and potential of `rev xs`. Recall the translation of `rev xs`.

$$\begin{aligned} \|\texttt{rev xs}\| = 1 +_c (\lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle \\ \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)\ \texttt{Nil})\ xs \end{aligned}$$

We can obtain an interpretation of $\|\texttt{rev xs}\|$ by substituting our interpretation of the auxilary function.

Let $n = [\![\|xs\|]\!]$.

$$[\![\|\texttt{rev xs}\|]\!] = [\![1 +_c (\lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)\ \texttt{Nil})\ xs]\!]\{xs \mapsto n\}$$

$$= 1 +\!\!\!+_c [\![\lambda xs.1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)\ \texttt{Nil}]\!]\{xs \mapsto n\}\ n$$

$$= 1 +\!\!\!+_c (\lambda\!\!\lambda xs.[\![1 +_c \texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)\ \texttt{Nil}]\!]\{xs \mapsto n\})\ n$$

$$= 1 +\!\!\!+_c (\lambda\!\!\lambda xs.1 +\!\!\!+_c [\![\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \lambda a.\langle 0, a\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\langle 1, \lambda a.(1 + r_c) +_c r_p\ \texttt{Cons}\langle x, a\rangle\rangle)]\!]\{xs \mapsto n\}\ 0)\ n$$

$$= 1 +\!\!\!+_c (\lambda\!\!\lambda xs.1 +\!\!\!+_c h(xs, 0))\ n$$

$$= 1 +\!\!\!+_c (1 +\!\!\!+_c h(n, 0))$$

$$= 1 +\!\!\!+_c (1 +\!\!\!+_c (2n, n))$$

$$= (2 + 2n, n)$$

So we see that the cost of `rev xs` is linear in the length of the list, and that the potential of the result is equal to the potential of the input.

2. Reverse

Here we present the naive implementation of list reverse. The naive implementation reverses a list in quadratic time as opposed to linear time.

$$\texttt{datatype list} = \texttt{Nil of unit} \mid \texttt{Cons of int} \times \texttt{list}$$

The implementation walks down a list, appending the head of the list to the end of the result of recursively calling itself on the tail of the list. We use the syntactic sugar introduced earlier. `rev` uses the auxilary function `snoc`. `snoc` appends an item to the end of a list.

$$\begin{aligned}\texttt{snoc} = \lambda xs.\lambda x.\texttt{rec}(&xs, \texttt{Nil} \mapsto \texttt{Cons}\langle x, \texttt{Nil}\rangle,\\ &\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{Cons}\langle y, \texttt{force}(r)\rangle)\end{aligned}$$

The quadratic time implementation of reverse recurses on the list, appending the head of the list to the recursively reversed tail of the list.

$$\begin{aligned}rev = \lambda xs.\texttt{rec}(&xs, \texttt{Nil} \mapsto \texttt{Nil},\\ &\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\texttt{snoc}\ \texttt{force}(r)\ x)\end{aligned}$$

2.1. Translation.

2.1.1. *`snoc` Translation.* First we translate the function `snoc`. To do so we apply the rule for translating an abstraction two times. Recall the rule is $\|\lambda x.e\| = \langle 0, \lambda x.\|e\|\rangle$.

$$\begin{aligned}\|\texttt{snoc}\| &= \|\lambda xs.\lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Cons}\langle x, \texttt{Nil}\rangle,\\ &\qquad\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{Cons}\langle y, \texttt{force}(r)\rangle)\|\\ &= \langle 0, \lambda xs.\|\lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Cons}\langle x, \texttt{Nil}\rangle,\\ &\qquad\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{Cons}\langle y, \texttt{force}(r)\rangle)\|\rangle\\ &= \langle 0, \lambda xs.\langle 0, \lambda x.\|\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Cons}\langle x, \texttt{Nil}\rangle,\\ &\qquad\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{Cons}\langle y, \texttt{force}(r)\rangle)\|\rangle\rangle\end{aligned}$$

Next we apply the rule for translating a `rec`.

$$= \langle 0, \lambda xs.\langle 0, \lambda x.\|xs\|_c +_c \texttt{rec}(\|xs\|_p, \texttt{Nil} \mapsto 1 +_c \|\texttt{Cons}\langle x, \texttt{Nil}\rangle\|,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\|)\rangle\rangle$$

xs is a variable, so its translation is $\langle 0, xs\rangle$.

$$= \langle 0, \lambda xs.\langle 0, \lambda x.\langle 0, xs\rangle_c +_c \texttt{rec}(\langle 0, xs\rangle_p, \texttt{Nil} \mapsto 1 +_c \|\texttt{Cons}\langle x, \texttt{Nil}\rangle\|,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\|)\rangle\rangle$$

We take the cost and potential projections of the translated term.

$$= \langle 0, \lambda xs.\langle 0, \lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \|\texttt{Cons}\langle x, \texttt{Nil}\rangle\|,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\|)\rangle\rangle$$

We will translate $\texttt{Cons}\langle x, \texttt{Nil}\rangle$. In order to do so we will first translate $\langle x, \texttt{Nil}\rangle$.

$$\|\langle x, \texttt{Nil}\rangle\| = \langle \|x\|_c + \|\texttt{Nil}\|_c, \langle \|x\|_p, \|\texttt{Nil}\|_p\rangle$$

x is a variable, so its translation is $\langle 0, x\rangle$.

The translation of Nil is $\langle 0, \texttt{Nil}\rangle$.

$$= \langle \langle 0, x\rangle_c + \langle 0, \texttt{Nil}\rangle_c, \langle \langle 0, x\rangle_p, \langle 0, \texttt{Nil}\rangle_p\rangle\rangle$$

$$= \langle 0, \langle x, \texttt{Nil}\rangle\rangle$$

We use the result in translation of $\texttt{Cons}\langle x, \texttt{Nil}\rangle$.

$$\|\texttt{Cons}\langle x, \texttt{Nil}\rangle\| = \langle \|\langle x, \texttt{Nil}\rangle\|_c, \texttt{Cons}\|\langle x, \texttt{Nil}\rangle\|_p\rangle$$

$$= \langle \langle 0, \langle x, \texttt{Nil}\rangle\rangle_c, \texttt{Cons}\langle 0, \langle x, \texttt{Nil}\rangle\rangle_p\rangle$$

$$= \langle 0, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle$$

Now that we have translated $\texttt{Cons}\langle x, \texttt{Nil}\rangle$ we return can substitute it in to the translation of `snoc` to complete the translation of the `Nil` branch of the `rec`.

$$= \langle 0, \lambda xs.\langle 0, \lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \langle 0, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\|)\rangle\rangle$$

we can expand the $+_c$ macro to simplify the `Nil` branch.

$$= \langle 0, \lambda xs.\langle 0, \lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r \rangle\rangle.1 +_c \|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\|\rangle\rangle$$

To complete the translation of `snoc` we must translate $\texttt{Cons}\langle y, \texttt{force}(r)\rangle$. To do so we first translate $\langle y, \texttt{force}(r)\rangle$.

$$\|\langle y, \texttt{force}(r)\rangle\| = \langle \|y\|_c + \|\texttt{force}(r)\|_c, \langle \|y\|_p, \|\texttt{force}(r)\|_p\rangle\rangle$$

y is a variable, so

$$\|y\| = \langle 0, y\rangle$$

$$\texttt{force}(r) = \|r\|_c +_c \|r\|_p$$

r is also a variable.

$$= \langle 0, r\rangle_c +_c \langle 0, r\rangle_p$$

$$= 0 +_c r = r$$

$$= \langle \langle 0, y\rangle_c + r_c, \langle \langle 0, y\rangle_p, r_p\rangle\rangle$$

$$= \langle r_c, \langle y, r_p\rangle\rangle$$

We use this in our translation of $\texttt{Cons}\langle y, \texttt{force}(r)\rangle$.

$$\begin{aligned}
\texttt{Cons}\langle y, \texttt{force}(r)\rangle &= \langle \|\langle y, \texttt{force}(r)\rangle\|_c, \texttt{Cons}\|\langle y, \texttt{force}(r)\rangle\|_p\rangle \\
&= \langle \langle r_c, \langle y, r_p\rangle\rangle_c, \langle r_c, \langle y, r_p\rangle\rangle_p\rangle \\
&= \langle r_c, \texttt{Cons}\langle y, r_p\rangle\rangle
\end{aligned}$$

We substitute this result into our translation of `rev`

$$\begin{aligned}
\|\texttt{snoc}\| &= \langle 0, \lambda xs.\langle 0, \lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle, \\
&\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\|\rangle\rangle \\
&= \langle 0, \lambda xs.\langle 0, \lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle, \\
&\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \langle r_c, \texttt{Cons}\langle y, r_p\rangle\rangle\rangle\rangle\rangle \\
&= \langle 0, \lambda xs.\langle 0, \lambda x.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle, \\
&\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle\rangle\rangle\rangle
\end{aligned}$$

2.1.2. *rev Translation.* The translation into the complexity language follows First we apply the abstraction translation rule: $\|\lambda x.e\| = \langle 0, \lambda x.\|e\|\rangle$.

$$\|rev\| = \|\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil},$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\texttt{snoc}\ \texttt{force}(r)\ x)$$

$$= \langle 0, \lambda xs.\|\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil},$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.\texttt{snoc}\ \texttt{force}(r)\ x)\|\rangle$$

Next we apply the rec translation rule.

$$= \langle 0, \lambda xs.\|xs\|_c +_c \texttt{rec}(\|xs\|_p, \texttt{Nil} \mapsto 1 +_c \|\texttt{Nil}\|,$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c \|\texttt{snoc}\ \texttt{force}(r)\ x\|)\rangle$$

As before, the translation of the variable xs is $\langle 0, xs\rangle$,

and the translation of Nil is $\langle 0, \texttt{Nil}\rangle$.

$$= \langle 0, \lambda xs.\langle 0, xs\rangle_c +_c \texttt{rec}(\langle 0, xs\rangle_p, \texttt{Nil} \mapsto 1 +_c \langle 0, \texttt{Nil}\rangle,$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c \|\texttt{snoc}\ xs\ x\|)\rangle$$

We take the projections of the translated expressions and expand the $+_c$ macro.

$$= \langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle,$$

$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c \|\texttt{snoc}\ xs\ x\|)\rangle$$

Next we translate the application `snoc force(r) x`.

$$\|\texttt{snoc}\ \texttt{force}(r)\ x\| = (1 + \|\texttt{snoc}\ \texttt{force}(r)\|_c + \|x\|_c) +_c \|\texttt{snoc}\ \texttt{force}(r)\|_p \|x\|_p$$

$$\|\texttt{snoc}\ \texttt{force}(r)\| = (1 + \|\texttt{snoc}\|_c + \|\texttt{force}(r)\|_c) +_c \|\texttt{snoc}\|_p \|\texttt{force}(r)\|_p$$

Next we translate the `force`.

$$\|\texttt{force}(r)\| = \|r\|_c +_c \|r\|_p$$

r is also a variable, so its translation is $\langle 0, xs\rangle$. The cost of $\|\texttt{snoc}\|$ is 0.

$$= \langle 0, r\rangle_c +_c \langle 0, r\rangle_p = r$$

$$\|\texttt{snoc}\ \texttt{force}(r)\| = (1 + 0 + r_c) +_c \|\texttt{snoc}\|_p\ r_p$$

x is a variable so its translation is $\langle 0, x\rangle$.

$$\|\texttt{snoc force}(r)\ x\| = (1 + \|\texttt{snoc force}(r)\|_c + \|x\|_c) +_c \|\texttt{snoc}\ r\|_p \|x\|_p i$$

$$= (1 + 1 + r_c + (\|\texttt{snoc}\|_p\ r_p)_c) +_c (\|\texttt{snoc}\|_p\ r_p)_p\ x$$

The cost of the partially applied function is 0.

$$= (2 + r_c) +_c (\|\texttt{snoc}\|_p\ r_p)_p\ x$$

We can use this to complete the translation of the `Cons` branch.

$$= \langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle,$$
$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.1 +_c ((2 + r_c) +_c (\|\texttt{snoc}\|_p\ r_p)_p\ x)\rangle$$
$$= \langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle,$$
$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.(3 + r_c) +_c (\|\texttt{snoc}\|_p\ r_p)_p\ x)\rangle$$

It is more interesting if we consider the translation of `rev` applied to some list `xs`. The translation of this function into the complexity language proceeds as follows. First we apply the rule for translating an application.

$$\|\texttt{rev}\ xs\| = (1 + \|\texttt{rev}\|_c + \|xs\|_c) +_c \|\texttt{rev}\|_p \|xs\|_p$$
$$= (1 + \|xs\|_c) +_c \|\texttt{rev}\|_p\ \|xs\|_p$$
$$= (1 + \|xs\|_c) +_c (\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle,$$
$$\texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.(3 + r_c) +_c (\|\texttt{snoc}\|_p\ r_p)_p\ x))\ \|xs\|_p$$

2.2. Interpretation. We intepret the size of an `list` to be the number of `Cons` constructors.

$$[\![\texttt{list}]\!] = \mathbb{N}^\infty$$
$$D^{list} = \{*\} + \{1\} \times \mathbb{N}^\infty$$
$$size_{list}(\texttt{Nil}) = 0$$
$$size_{list}(\texttt{Cons(1,n)}) = 1 + n$$

2.2.1. *`snoc` Interpretation.* We interpret $\|\texttt{snoc xs x}\|$. Recall the translation.

$$\|\texttt{snoc}\ xs\ x\| = (2 + \|xs\|_c + \|x\|_c) +_c (\|\texttt{snoc}\|_p\ \|xs\|_p)_p\ \|x\|_p$$

The cost of `snoc` is driven by the recursion. We interpret the cost of the `rec` by defining a recurrence $g(n)$. We add $x \mapsto x$ to the environment, where x is the interpretation of x.

$$\begin{aligned}
g(n) &= [\![\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle, \\
&\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)]\!]\{xs \mapsto n, x \mapsto x\} \\
&= \bigvee_{size\ z \leq n} case(z, f_{Nil}, f_{Cons})
\end{aligned}$$

where

$$\begin{aligned}
f_{Nil}(*) &= [\![\langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle]\!]\{xs \mapsto n, x \mapsto x\} \\
&= (1, 1) \\
f_{Cons}(1, m) &= [\![\langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle]\!]\{xs \mapsto n, x \mapsto x, y \mapsto 1, ys \mapsto m, r \mapsto g(m)\} \\
&= (1 + g_c(m), 1 + g_p(m))
\end{aligned}$$

To eliminate the big maximum operator, we use the same technique as in fast reverse, by splitting the big maximum into two cases: $size\ z < n$ and $size\ z = n$.

case $n = 0$**:**

The only z such that $size\ z \leq 0$ is $*$. So $g(0) = f_{Nil}(0) = (1, 1)$.

case $n > 0$**:**

$$\begin{aligned}
g(n) &= \bigvee_{size z < n} case(z, f_{Nil}, f_{Cons}) \vee \bigvee_{size\ z = n} case(z, f_{Nil}, f_{Cons}) \\
&= g(n-1) \vee (1 + g_c(n-1), 1 + g_p(n-1))
\end{aligned}$$

Since $\leq$ is symmetric, $g(n-1) \leq (g_c(n-1), g_p(n-1))$, and

$(g_c(n-1), g_p(n-1)) < (1 + g_c(n-1), g_p(n-1))$.

$\leq (1 + g_c(n-1), 1 + g_p(n-1))$

The solution to this recurrence is given in lemma 3.5.

LEMMA 3.5. $g(n) = (1 + n, 1 + n)$.

PROOF. We prove this by induction on n.

case $n = 0$:

$g(0) = (1, 1)$.

case $n > 0$:

$$\begin{aligned} g(n) &= (1 + g_c(n-1), 1 + g_p(n-1)) \\ &= (1 + (n, n)_c, 1 + (n, n)_p) \\ &= (1 + n, 1 + n) \end{aligned}$$

□

This is a closed form solution for the recurrence describing the complexity of the `rec` expression in the body of `snoc`. We use this to produce the equation describing the complexity of $\|$`snoc`$\|$.

$$snoc(n, x) = g(n) = (1 + n, 1 + n)$$

2.2.2. *`rev` Interpretation.* Recall the translation of `rev xs`.

$$\begin{aligned} \texttt{rev}\ xs = (1 + \|xs\|_c) +_c (\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle,& \\ \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.(3 + r_c) +_c (\|\texttt{snoc}\|_p\ r_p)_p\ x))\ \|xs\|_p& \end{aligned}$$

We will interpret the `rec` construct first.

$$\begin{aligned} g(n) &= [\![\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle, \\ &\qquad \texttt{Cons} \mapsto \langle x, \langle xs', r\rangle\rangle.(3 + r_c) +_c (\|\texttt{snoc}\|_p\ r_p)_p\ x)]\!]\{xs \mapsto n\} \\ &= \bigvee_{size\ z \leq n} case(z, f_{Nil}, f_{Cons}) \end{aligned}$$

where

$$\begin{aligned} f_{Nil}(*) &= [\![\langle 1, \texttt{Nil}\rangle]\!]\{xs \mapsto n\} \\ &= (1, 0) \\ f_{Cons}((1, m)) &= [\![(3 + r_c) +_c (\|\texttt{snoc}\|_p\ r_p\ x)]\!]\{xs \mapsto n, x \mapsto 1, r \mapsto g(m)\} \\ &= (3 + g_c(m)) +\!\!\!+_c ([\![\|\texttt{snoc}\|_p]\!]\{xs \mapsto n, x \mapsto 1, r \mapsto g(m)\}\ g_p(m)\ 1) \end{aligned}$$

$$= (3 + g_c(m)) \mathbin{+\!\!+}_c snoc(g_p(m), 1)$$

$$= (3 + g_c(m) + snoc_c(g_p(m), 1), snoc_p(g_p(m), 1))$$

To obtain a solution to this recurrence, we apply the same technique as in the interpretation of `snoc`. We break the recurrence into the case where the argument is 0 and when the argument is greater than 0. Then we eliminate the big maximum operator by breaking the maximum into cases where $size\ z < n$ and when $size\ z = n$.

case $n = 0$**:**

The only z such that $size\ z \leq 0$ is $*$.

$$g(0) = (1, 0)$$

case $n > 0$**:**

$$g(n) = \bigvee_{size\ z \leq n} case(z, f_{Nil}, f_{Cons})$$

$$= \bigvee_{size\ z < n} case(z, f_{Nil}, f_{Cons}) \vee \bigvee_{size\ z = n} case(z, f_{Nil}, f_{Cons})$$

$$= g(n-1) \vee \bigvee_{size\ z = n} case(z, f_{Nil}, f_{Cons})$$

$$= g(n-1) \vee (3 + g_c(n-1) + snoc_c(g_p(n-1), 1), snoc_p(g_p(n-1), 1))$$

We substitute the definition of $snoc(n, x)$.

$$= g(n-1) \vee (3 + g_c(n-1) + g_p(n-1) + 1, g_p(n-1) + 1)$$

$g_p(n-1)$ is nonnegative, so we can eliminate the max.

$$= (4 + g_c(n-1) + g_p(n-1), 1 + g_p(n-1))$$

We use the substitution method to solve the recurrence.

LEMMA 3.6. $g(n) = (4n^2 + 1, n)$

PROOF. We prove this by induction on n.

case $n = 0$**:**

$g(0) = (1, 0) = (4\dot{0}^2 + 1, 0)$.

case $n > 0$:

$$
\begin{aligned}
g(n) &= (4 + g_c(n-1) + g_p(n-1), 1 + g_p(n-1)) \\
&= (4 + (4(n-1)^2 + 1) + n - 1, 1 + n - 1) \\
&= (4 + 4n^2 - 8n + 4 + n, n)
\end{aligned}
$$

$\square$

3. Parametric Insertion Sort

Parametric insertion sort is a higher order algorithm which sorts a list using a comparison function which is passed to it as an argument. The running time of insertion sort is $\mathcal{O}(n^2)$. This characterization of the complexity of parametric insertion sort does not capture role of the comparison function in the running time. When sorting a list of integers, where comparison between any two integers takes constant time, this does not matter. However, when sorting a list of strings, where the complexity of comparison is order the length of the string, the length of the strings may influence the running time more than the length of the list when sorting small lists of large strings.

We use the familiar `list` datatype.

$$\texttt{data list} = \texttt{Nil of unit | Cons of int} \times \texttt{list}$$

The function `sort` relies on the function `insert`. `insert` inserts an element into a sorted list.

$$\begin{aligned}\texttt{insert} = \lambda f.\lambda x.\lambda xs.\texttt{rec}(xs, \texttt{Nil} &\mapsto \texttt{Cons}\langle x, \texttt{Nil}\rangle,\\ &\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{rec}(f\ x\ y, \texttt{True} \mapsto \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle,\\ &\qquad \texttt{False} \mapsto \texttt{Cons}\langle y, \texttt{force}(r)\rangle))\end{aligned}$$

The `sort` function recurses on the list, using the `insert` function to insert the head of the list into the recursively sorted tail of the list.

$$\texttt{sort} = \lambda f.\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil}, \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{insert}\ f\ y\ \texttt{force}(r))$$

3.1. Translation of `insert`. We walk through the translation of `insert`. We will translate from the bottom up. We translate the `True` and `False` branches of the inner `rec`, then we translate the inner `rec`, then we translate the `Nil` and `Cons` branches of the outer `rec`, and finally complete the translation of `insert`. The translation of the `True` branch of the inner `rec` is given below. The translation of a datatype is the cost of translating its argument, and complexity language constructor applied to the potential of the translated argument.

$$\|\texttt{Cons}\langle x, \texttt{Cons}\langle y, rs\rangle\rangle\| = \langle \|\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\|_c, \texttt{Cons}\|\langle x, \texttt{Cons}\langle y, ys\rangle\|_p\rangle$$

The argument to the `Cons` constructor is a tuple. The cost of the translation of a tuple is the cost of the translation of each element and the potential is the tuple of the potentials of the translations of each element.

$$\|\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\| = \langle \|x\|_c + \|\texttt{Cons}\langle y, ys\rangle\|_c, \langle \|x\|_p, \texttt{Cons}\langle y, ys\rangle\|_p\rangle\rangle$$

The first element of the tuple is a variable, but the second element is another `list`. So we translate the second element first. To do so we apply the rule for translating a datatype.

$$\|\texttt{Cons}\langle y, ys\rangle\| = \langle \|y, ys\|_c, \texttt{Cons}\|y, ys\|_p\rangle$$

The argument to the constructor is a tuple. We apply the rule for translating a tuple again. Both element of the tuple are variables, so their translated cost is 0 and their translated potential is their corresponding variable in the complexity language.

$$\begin{aligned}\|\langle y, ys\rangle\| &= \langle \|y\|_c + \|rs\|_c, \langle \|y\|_p, \|rs\|_p\rangle\rangle \\ &= \langle \langle 0, y\rangle_c + \langle 0, rs\rangle_c, \langle \langle 0, y\rangle_p, \langle 0, rs\rangle\rangle\rangle \\ &= \langle 0, \langle y, ys\rangle\rangle\end{aligned}$$

We use this to complete the translation of $\texttt{Cons}\langle y, ys\rangle$.

$$\begin{aligned}\|\texttt{Cons}\langle y, ys\rangle\| &= \langle \|y, ys\|_c, \texttt{Cons}\|y, ys\|_p\rangle \\ &= \langle 0, \texttt{Cons}\langle y, ys\rangle\rangle\end{aligned}$$

We use this result to complete the translation of $\langle x, \texttt{Cons}\langle y, ys\rangle\rangle$.

$$\begin{aligned}\|\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\| &= \langle \|x\|_c + \|\texttt{Cons}\langle y, ys\rangle\|_c, \langle \|x\|_p, \texttt{Cons}\langle y, ys\rangle\|_p\rangle\rangle \\ &= \langle 0, \langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle\end{aligned}$$

And finally we use this to complete the translation of $\texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle$.

$$\begin{aligned}\|\texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\| &= \langle \|\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\|_c, \texttt{Cons}\|\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\|_p\rangle \\ &= \langle 0, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle\end{aligned}$$

Next we will translate the `False` branch.

$$\|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\| = \langle \|\langle y, \texttt{force}(r)\rangle\|_c, \texttt{Cons}\|\langle y, \texttt{force}(r)\rangle\|_p\rangle$$

To complete this we must first translate the tuple. The two elements of the tuple are y and $\texttt{force}(r)$. The translation of the variable y is $\langle 0, y\rangle$. The translation of $\texttt{force}(r)$ is $\|r\|_c +_c \|r\|_p$. Like y, r is a variable so its translation is $\langle 0, r\rangle$. So the translation of $\texttt{force}(r)$ is $0 +_c r$ which simplifies to r.

$$\begin{aligned}\|\langle y, \texttt{force}(r)\rangle\| &= \langle \|y\|_c + \|\texttt{force}(r)\|_c, \langle \|y\|_p, \|\texttt{force}(r)\|_p\rangle\rangle \\ &= \langle 0 + r_c, \langle y, r_p\rangle\rangle \\ &= \langle r_c, \langle y, r_p\rangle\rangle\end{aligned}$$

We substitute this into the translation of $\texttt{Cons}\langle y, \texttt{force}(r)\rangle$.

$$\begin{aligned}\|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\| &= \langle \|\langle y, \texttt{force}(r)\rangle\|_c, \texttt{Cons}\|\langle y, \texttt{force}(r)\rangle\|_p\rangle \\ &= \langle r_c, \texttt{Cons}\langle y, r_p\rangle\rangle\end{aligned}$$

We put together the translations of the `True` and `False` branches to translate the inner `rec`. We will need the translation of the comparison function f applied to x and y. To translate $f\ x\ y$ we apply the function application rule twice. First we apply the rule to $(f\ x)\ y$. Then we apply the rule to $f\ x$. Then we expand the $+_c$ macro to simplify the result.

$$\|f\ x\ y\| = (1 + \|f\ x\|_c + \|y\|_c) +_c \|f\ x\|_p\|y\|_p \tag{3}$$

$$\begin{aligned}\|f\ x\| &= (1 + \|f\|_c + \|x\|_c) +_c \|f\|_p\|x\|_p \\ &= (1 + \|f\|_c + \|x\|_c + (\|f\|_p\|x\|_p)_c, (\|f\|_p\|x\|_p)_p)\end{aligned}$$

$$\begin{aligned}&= (1 + (1 + \|f\|_c + \|x\|_c +_c (\|f\|_p\|x\|_p)_c) + \|y\|_c) +_c (\|f\|_p\|x\|_p)_p\|y\|_p \\ &= (2 + \|f\|_c + \|x\|_c + \|y\|_c + (\|f\|_p\|x\|_p)_c) +_c (\|f\|_p\|x\|_p)_p\|y\|_p\end{aligned}$$

We use the translation of `f x y` and the `True` and `False` branches to construct the translation of the inner `rec` construct.

$$\|\texttt{rec}(f\ x\ y, \texttt{True} \mapsto \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle, \texttt{False} \mapsto \texttt{Cons}\langle y, \texttt{force}(r)\rangle)\|$$

$$
\begin{aligned}
&= \|f\ x\ y\|_c +_c \texttt{rec}(\|f\ x\ y\|_p, \texttt{True} \mapsto 1 +_c \|\texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\|,\\
&\qquad\qquad \texttt{False} \mapsto 1 +_c \|\texttt{Cons}\langle y, \texttt{force}(r)\rangle\|)\\
&= \|f\ x\ y\|_c +_c \texttt{rec}(\|f\ x\ y\|_p, \texttt{True} \mapsto 1 +_c \langle 0, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle,\\
&\qquad\qquad \texttt{False} \mapsto 1 +_c \langle r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)\\
&= (2 + \|f\|_c + \|x\|_c + \|y\|_c + (\|f\|_p \|x\|_p)_c)\\
&\qquad +_c \texttt{rec}((\|f\|_p\ \|x\|_p)_p \|y\|_p, \texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle,\\
&\qquad\qquad \texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)
\end{aligned}
$$

Next we translate the `Nil` and `Cons` branches of the outer `rec` of `insert`. In this branch we append the element to an empty list.

$$
\begin{aligned}
\|\texttt{Cons}\langle x, \texttt{Nil}\rangle\| &= \langle \|\langle x, \texttt{Nil}\rangle\|_c, \texttt{Cons}\|\langle x, \texttt{Nil}\rangle\|_p\rangle\\
&= \langle 0, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle
\end{aligned}
$$

Translation of the `Cons` branch of the outer `rec` in `insert`. In this branch we recurse on a nonempty list. We check if `x` is comes before the head of the list under the ordering given by `f`, in which case we are done, otherwise we recurse on the tail of the list.

$$
\begin{aligned}
&\|\texttt{rec}(f\ x\ y, \texttt{True} \mapsto \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle, \texttt{False} \mapsto \texttt{Cons}\langle y, \texttt{force}(r)\rangle)\|\\
&\quad= \|f\ x\ y\|_c +_c \texttt{rec}(f\ x\ y, \texttt{True} \mapsto 1 +_c \|\texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\|)\\
&\quad= (2 + \|f\|_c + \|x\|_c + \|y\|_c + (\|f\|_p \|x\|_p)_c)\\
&\qquad\quad +_c \texttt{rec}((\|f\|_p\ \|x\|_p)_p \|y\|_p, \texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle,\\
&\qquad\qquad \texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)
\end{aligned}
$$

We know that f, x, and y are variables, so their translations have 0 cost.

$$
\begin{aligned}
&\quad= (2 + (f\ x)_c)\\
&\qquad\quad +_c \texttt{rec}(((f\ x)_p\ y)_p, \texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle,\\
&\qquad\qquad \texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)
\end{aligned}
$$

We complete the translation of the outer `rec` using the translated `Nil` and `Cons`.

$$\begin{aligned}
&\|\mathtt{rec}(xs, \mathtt{Nil} \mapsto \mathtt{Cons}\langle x, \mathtt{Nil}\rangle,\\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\mathtt{rec}(f\ x\ y, \mathtt{True} \mapsto \mathtt{Cons}\langle x, \mathtt{Cons}\langle y, ys\rangle\rangle,\\
&\qquad\qquad \mathtt{False} \mapsto \mathtt{Cons}\langle y, \mathtt{force}(r)\rangle))\|\\
&\quad = \|xs\|_c +_c \mathtt{rec}(\|xs\|_p, \mathtt{Nil} \mapsto 1 +_c \|\mathtt{Cons}\langle x, \mathtt{Nil}\rangle\|,\\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\mathtt{rec}(f\ x\ y, \mathtt{True} \mapsto \mathtt{Cons}\langle x, \mathtt{Cons}\langle y, ys\rangle\rangle,\\
&\qquad\qquad \mathtt{False} \mapsto \mathtt{Cons}\langle y, \mathtt{force}(r)\rangle)\|)\|
\end{aligned}$$

We substitute in our translations of the branches. Also note that xs is a variable, so its translation is $\langle 0, xs\rangle$.

$$\begin{aligned}
&\quad = \mathtt{rec}(xs, \mathtt{Nil} \mapsto 1 +_c \langle 0, \mathtt{Cons}\langle x, \mathtt{Nil}\rangle\rangle,\\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c ((2 + (fx)_c)\\
&\qquad\quad +_c \mathtt{rec}(((f\ x)_p\ y)_p, \mathtt{True} \mapsto \langle 1, \mathtt{Cons}\langle x, \mathtt{Cons}\langle y, ys\rangle\rangle\rangle,\\
&\qquad\qquad \mathtt{False} \mapsto \langle 1 + r_c, \mathtt{Cons}\langle y, r_p\rangle\rangle)))\\
&\quad = \mathtt{rec}(xs, \mathtt{Nil} \mapsto \langle 1, \mathtt{Cons}\langle x, \mathtt{Nil}\rangle\rangle,\\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(3 + (fx)_c)\\
&\qquad\quad +_c \mathtt{rec}(((f\ x)_p\ y)_p, \mathtt{True} \mapsto \langle 1, \mathtt{Cons}\langle x, \mathtt{Cons}\langle y, ys\rangle\rangle\rangle,\\
&\qquad\qquad \mathtt{False} \mapsto \langle 1 + r_c, \mathtt{Cons}\langle y, r_p\rangle\rangle)))
\end{aligned}$$

The translation of `insert` is just three applications of the application rule.

$$\begin{aligned}
\mathtt{insert} = {} & \|\lambda f.\lambda x.\lambda xs.\mathtt{rec}(xs, \mathtt{Nil} \mapsto \mathtt{Cons}\langle x, \mathtt{Nil}\rangle,\\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\mathtt{rec}(f\ x\ y, \mathtt{True} \mapsto \mathtt{Cons}\langle x, \mathtt{Cons}\langle y, ys\rangle\rangle,\\
&\qquad\qquad \mathtt{False} \mapsto \mathtt{Cons}\langle y, \mathtt{force}(r)\rangle))\|\\
= {} & \langle 0, \lambda f.\langle 0, \lambda x.\langle 0, \lambda xs.\|\mathtt{rec}(xs, \mathtt{Nil} \mapsto \mathtt{Cons}\langle x, \mathtt{Nil}\rangle,\\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\mathtt{rec}(f\ x\ y, \mathtt{True} \mapsto \mathtt{Cons}\langle x, \mathtt{Cons}\langle y, ys\rangle\rangle,\\
&\qquad\qquad \mathtt{False} \mapsto \mathtt{Cons}\langle y, \mathtt{force}(r)\rangle))\|\rangle\rangle\rangle
\end{aligned}$$

$$
\begin{aligned}
&= \langle 0, \lambda f.\langle 0, \lambda x.\langle 0, \lambda xs.\mathtt{rec}(xs, \mathtt{Nil} \mapsto \langle 1, \mathtt{Cons}\langle x, \mathtt{Nil}\rangle\rangle, \\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(3 + (fx)_c) +_c \mathtt{rec}(((f\ x)_p\ y)_p, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{True} \mapsto \langle 1, \mathtt{Cons}\langle x, \mathtt{Cons}\langle y, ys\rangle\rangle\rangle, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{False} \mapsto \langle 1 + r_c, \mathtt{Cons}\langle y, r_p\rangle\rangle))))
\end{aligned}
$$

We are interested in the interpretation of applying `insert`. So we will give a translation of `insert f x xs`.

$$
\begin{aligned}
\|\mathtt{insert}\ f\ x\ xs\| &= (1 + \|\mathtt{insert}\ f\ x\|_c + \|xs\|_c) +_c \|\mathtt{insert}\ f\ x\|_p \|xs\|_p \\
&= (1 + \|\mathtt{insert}\ f\ x\|_c + \|xs\|_c) +_c \|\mathtt{insert}\ f\ x\|_p \|xs\|_p \\
&= (2 + \|\mathtt{insert}\ f\|_c + \|x\|_c + (\|\mathtt{insert}\ f\|_p \|x\|_p)_c + \|xs\|_c) \\
&\qquad\quad +_c \|\mathtt{insert}\ f\|_p \|x\|_p \|xs\|_p \\
&= (2 + (\|\mathtt{insert}\|_p\ f)_c + \|x\|_c + \|xs\|_c) +_c \|\mathtt{insert}\ f\|_p \|x\|_p \|xs\|_p \\
&= (3 + \|\mathtt{insert}\|_c + \|f\|_c + (\|\mathtt{insert}\|_p \|f\|_p \|x\|_p)_c + \|x\|_c + \|xs\|_c) \\
&\qquad\quad +_c \|\mathtt{insert}\|_p \|f\|_p \|x\|_p \|xs\|_p \\
&= (3 + \|f\|_c + \|x\|_c + \|xs\|_c) +_c \|\mathtt{insert}\|_p \|f\|_p \|x\|_p \|xs\|_p \\
&= (3 + \|f\|_c + \|x\|_c + \|xs\|_c) +_c \mathtt{rec}(\|xs\|_p, \\
&\qquad\quad \mathtt{Nil} \mapsto \langle 1, \mathtt{Cons}\langle x, \mathtt{Nil}\rangle\rangle \\
&\qquad\quad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(3 + ((\|f\|_p \|x\|_p)_p y)_c) \\
&\qquad\qquad\quad +_c \mathtt{rec}((\|f\|_p \|x\|_p)_p y)_p, \\
&\qquad\qquad\qquad \mathtt{True} \mapsto \langle 1, \mathtt{Cons}\langle \|x\|_p, \mathtt{Cons}\langle y, ys\rangle\rangle\rangle \\
&\qquad\qquad\qquad \mathtt{False} \mapsto \langle 1 + r_c, \mathtt{Cons}\langle y, r_p\rangle\rangle))
\end{aligned}
$$

3.2. Translation of `sort`. We walk through the translation of `sort`. Recall the definition of `sort`.

$$
\mathtt{sort} = \lambda f.\lambda xs.\mathtt{rec}(xs, \mathtt{Nil} \mapsto \mathtt{Nil}, \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\mathtt{insert}\ f\ y\ \mathtt{force}(r))
$$

The translation of `sort` begins with two applications of the rule for translating abstractions.

$$\|\texttt{sort}\| = \|\lambda f.\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil}, \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{insert}\ f\ y\ \texttt{force}(r))\|$$

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\|\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil},$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{insert}\ f\ y\ \texttt{force}(r))\|\rangle\rangle$$

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\|xs\|_c +_c \texttt{rec}(\|xs\|_p, \texttt{Nil} \mapsto 1 +_c \|\texttt{Nil}\|,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\texttt{insert}\ f\ y\ \texttt{force}(r)\|)\rangle\rangle$$

As xs is a variable, $\|xs\| = \langle 0, xs\rangle$.

We have seen before $\|\texttt{Nil}\| = \langle 0, \texttt{Nil}\rangle$.

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \langle 0, \texttt{Nil}\rangle,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 +_c \|\texttt{insert}\ f\ y\ \texttt{force}(r)\|)\rangle\rangle$$

We can use our translation of `insert` applied to three arguments.

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \langle 0, \texttt{Nil}\rangle,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(4 + \|f\|_c + \|y\|_c + \|force(r)\|_c)$$

$$+_c ((\|\texttt{insert}\|_p\|f\|_p)_p\|y\|_p)_p\|force(r)\|_p)\rangle\rangle$$

The variables f, y, and r translate to $\langle 0, f\rangle$, $\langle 0, y\rangle$, and $\langle 0, r\rangle$ respectively.

The expression $\texttt{force}(r)$ translates to $\|r\|_c +_c \|r\|$, which is just r.

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \langle 0, \texttt{Nil}\rangle,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(4 + r_c) +_c ((\|\texttt{insert}\|_p f)_p y)_p r_p)\rangle\rangle$$

We also give the translation of `sort` applied to two arguments.

$$\|\texttt{sort}\ f\ xs\| = (1 + \|\texttt{sort}\ f\|_c + \|xs\|_c) +_c (\|\texttt{sort}\ f\|_p)_p\|xs\|_p$$

$$= (1 + (1 + \|\texttt{sort}\|_c + \|f\|_c) + \|xs\|_c) +_c (\|\texttt{sort}\|_p\|f\|_p)_p\|xs\|_p$$

$$= (2 + \|f\|_c + \|xs\|_c) +_c (\|\texttt{sort}\|_p\|f\|_p)_p\|xs\|_p$$

3.3. Interpretation of `insert`. We well use an interpretation of lists as a pair of their greatest element and their length.

$$[\![list]\!] = \mathbb{Z} \times \mathbb{N}^\infty$$

$$D^{list} = \{*\} + \{\mathbb{Z}\} \times \mathbb{N}^\infty$$

$$size_{list}(*) = (-\infty, 0)$$

$$size_{list}((i,(j,n))) = (max\{i,j\}, 1+n)$$

We use the mutual ordering on pairs. That is, $(s,n) \leq (s',n')$ if $n \leq n'$ and $s < s'$ or $n < n'$ and $s \leq s'$. First we interpret the `rec`, which drives of the cost of `insert`. As in the translation, we break the interpretation up to make it more manageable. We will write $map, \lambda\!\!\lambda$ and $+\!\!\!+_c$ in the semantics, which stand for the semantic equivalents of the syntactic `map`, λ and $+_c$. The definitions of these semantic functions mirror the definitions of their syntactic equivalents.

First we interpret the inner `rec` of $\|$`insert`$\|$.

$$[\![\texttt{rec}(((f\ x)_p\ y)_p, \texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle, \texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)]\!]\xi$$

$$= \bigvee_{size(z) \leq ((f\ x)_p\ y)_p} case(z, f_{True}, f_{False})$$

where

$$\xi = \{f \mapsto f, x \mapsto x, y \mapsto j, ys \mapsto (j,m), r \mapsto r\}$$

$$\begin{aligned} f_{True}(*) &= [\![\langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle]\!]\xi \\ &= (1, [\![\texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle]\!])\xi \\ &= (1, (max(x,j), 2+m)) \\ f_{False}(*) &= [\![\langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle]\!]\xi \\ &= (1 + r_c, (max(j, \pi_0 r_p), 1 + \pi_1 r_p)) \end{aligned}$$

Since there are only two z, we can simplify the big maximum to a maximum.

$$\bigvee_{size(z) \leq ((f\ x)_p\ y)_p} case(z, f_{True}, f_{False})$$

$$= (1, (max(x,j), 2+m)) \vee (1 + r_c, (max(j, \pi_0 r_p), 1 + \pi_1 r_p))$$

Using this, we proceed to interpret the outer recurrence.

$$
\begin{aligned}
g(i,n) = \llbracket &\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle, \\
&\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(3 + (fx)_c) \\
&\quad +_c \texttt{rec}(((f\ x)_p\ y)_p, \texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle, \\
&\qquad \texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)))\rrbracket\xi
\end{aligned}
$$

where

$$
\begin{aligned}
\xi &= \{xs \mapsto (i,n), x \mapsto x\} \\
g(i,n) &= \bigvee_{size(z) \leq (i,n)} case(z, f_{Nil}, f_{Cons})
\end{aligned}
$$

where

$$
\begin{aligned}
f_{Nil}(*) &= \llbracket\langle 1, \texttt{Cons}\langle x, \texttt{Nil}\rangle\rangle\rrbracket\xi \\
&= (1, (x, 1)) \\
f_{Cons}(j,m) &= \llbracket(4 + (f\ x)_c) +_c \texttt{rec}(((f\ x)_p\ y)_p, \texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle, \\
&\qquad \texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)\rrbracket\xi\{y \mapsto j, ys \mapsto (j,m), r \mapsto g(j,m)\} \\
&= (4 + (f\ x)_c) +\!\!+_c \llbracket\texttt{rec}(((f\ x)_p\ y)_p, \texttt{True} \mapsto \langle 1, \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle\rangle, \\
&\qquad \texttt{False} \mapsto \langle 1 + r_c, \texttt{Cons}\langle y, r_p\rangle\rangle)\rrbracket\xi\{y \mapsto j, ys \mapsto (j,m), r \mapsto g(j,m)\} \\
&= (4 + (f\ x)_c) +\!\!+_c ((1, (max(x,j), 2 + m)) \\
&\qquad \vee (1 + g_c(j,m), (max(j, \pi_0 g_p(j,m)), 1 + \pi_1 g_p(j,m))))
\end{aligned}
$$

So the interpretation of $\|\texttt{insert}\|$ is

$$
\|\texttt{insert}\| = (0, \lambda\!\!\lambda f.(0, \lambda\!\!\lambda x.(0, \lambda\!\!\lambda(i,n).g(i,n))))
$$

where

$$
g(i,n) = \bigvee_{size(z) \leq (i,n)} case(z, f_{Nil}, f_{Cons})
$$

where

$$
\begin{aligned}
f_{Nil}(*) &= (1, (x, 1)) \\
f_{Cons}(j,m) &= (4 + (f\ x)_c) +\!\!+_c ((1, (max(x,j), 2 + m))
\end{aligned}
$$

$$\vee\, (1 + g_c(j, m), (max(j, \pi_0 r_p), 1 + \pi_1 g_p(j, m))))$$

To obtain a closed form solution, we will seperate the recurrence into a recurrence for the cost and a recurrence for the potential, and solve those independently. We use the substitution method to prove a closed form solution to the cost of the `rec` construct of `insert`.

LEMMA 3.7. $g_c(i, n) \leq (4 + ((f\ x)_p\ i)_c n + 1$

PROOF. We prove this by induction on n. Recall we use the mutual ordering on pairs.

case $n = 0$**:**

$g_c(i, n) = (1, (x, 1))_c = 1$

case $n > 0$**:**

$$\begin{aligned}
g_c(i, n) &= \bigvee_{size(z) \leq (i,n)} case(z, (f_{Nil}, f_{Cons})) \\
&= \bigvee_{\substack{j<i, m\leq n \\ \text{or } j\leq i, m<n}} case((j, m), (f_{Nil}, f_{Cons})) \\
&= \bigvee_{\substack{j<i, m\leq n \\ \text{or } j\leq i, m<n}} 4 + ((f\ x)_p\ j)_c + g_c(j, m-1)) \\
&\leq \bigvee_{\substack{j<i, m\leq n \\ \text{or } j\leq i, m<n}} 4 + ((f\ x)_p\ j)_c + (4 + ((f\ x)_p\ j)_c)(m-1) + 1 \\
&\leq \bigvee_{\substack{j<i, m\leq n \\ \text{or } j\leq i, m<n}} (4 + ((f\ x)_p\ j)_c)(m-1+1) + 1 \\
&\leq \bigvee_{\substack{j<i, m\leq n \\ \text{or } j\leq i, m<n}} (4 + ((f\ x)_p\ j)_c)m + 1 \\
&\leq \bigvee_{\substack{i<j, m\leq n \\ \text{or } i\leq j, m<n}} (4 + ((f\ x)_p\ i)_c)n + 1 \\
&\leq (5 + ((f\ x)_p\ i)_c)n
\end{aligned}$$

$\square$

As expected, we find the cost of insert is bounded by the length of the list and the largest element. Now we use the same method to obtain a closed form solution for the potential of the `rec` construct of `insert`. The potential of the resulting list is bounded by the list with maximum element equal to max of the inserted element an the maximum element of the original list and one more the length of the original list.

LEMMA 3.8. $g_p(i,n) \leq (max\{x,i\}, n+1)$

PROOF. We prove this by induction on n.

case $n = 0$**:**

$g_p(i,n) = (1,(x,1))_p = (x,1).$

case $n > 0$**:**

$$\begin{aligned}
g_p(i,n) &= \bigvee_{size(z)\leq(i,n)} case(z, f_{Nil}, f_{Cons}) \\
&= \bigvee_{\substack{j<i,m\leq n \\ \text{or } j\leq i,m<n}} (max\{x,j,\pi_0 g_p(j,m-1)\}, 2+(m-1)\vee 1+\pi_1 g_p(j,m-1)) \\
&\leq \bigvee_{\substack{j<i,m\leq n \\ \text{or } j\leq i,m<n}} (max\{x,j\}, 2+(m-1)) \\
&\leq \bigvee_{\substack{j<i,m\leq n \\ \text{or } j\leq i,m<n}} (max\{x,i\}, 1+n) \qquad \text{Since } m\leq n. \\
&\leq (max\{x,i\}, 1+n)
\end{aligned}$$

□

Using lemmas 3.7 and 3.8, we can express the cost and potential of `insert` in terms of its arguments.

$$insert\ f\ x\ xs \leq (5 + ((f\ x)_p\ i)_c n, (max\{x,i\}, n+1)) \tag{4}$$

3.4. Interpretation of `sort`. We use the same denotational semantics as in the intepretation of `insert`. We interpret `list` as a tuple of the largest element in the list and the number of `Cons` constructors. As in `insert`, the `rec` construct again drives the cost and potential of `sort`. We interpret the `rec` construct, manipulate the recurrence to a closed form, and use the result to interpret the cost and potential of applying `sort` to a comparison function f and a list xs.

$$
\begin{aligned}
g(i,n) &= [\![\mathtt{rec}(xs, \mathtt{Nil} \mapsto \langle 1, \mathtt{Nil}\rangle, \\
&\qquad \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(4 + r_c) +_c ((\|\mathtt{insert}\|_p f)_p y)_p r_p)]\!]\{xs \mapsto (i,n), f \mapsto f\} \\
&= \bigvee_{size(z) \le n} case(z, f_{Nil}, f_{Cons})
\end{aligned}
$$

where

$$
\begin{aligned}
f_{Nil} &= [\![\langle 1, \mathtt{Nil}\rangle]\!]\{xs \mapsto (i,n)\} \\
&= (1, (-\infty, 0)) \\
f_{Cons} &= [\![(4 + r_c) +_c ((\|\mathtt{insert}\|_p f)_p y)_p r_p)]\!]\xi \\
&\text{where} \quad \xi = \{xs \mapsto (i,n), f \mapsto f, y \mapsto j, ys \mapsto (j,m), r \mapsto g(j,m)\} \\
&= (4 + g_c(j,m)) +\!\!\!+_c (([\![\|\mathtt{insert}\|_p]\!]\xi\ f)_p j)_p g_p(j,m) \\
&= (5 + g_c(j,m)) +\!\!\!+_c (((f\ j)_p\ j)_c g_p(j,m), (max\{j,j\}, g_p(j,m)+1)) \\
&= (5 + g_c(j,m) + (f\ j)_p\ j)_c g_p(j,m), (max\{j,j\}, g_p(j,m)+1))
\end{aligned}
$$

Observe that in equation 9, the cost is depends on the potential of the recursive call. Therefore we must solve the recurrence for the potential first.

LEMMA 3.9. $g_p(n) \le (j, n)$

PROOF. We prove this by induction on n. We use equation 4 to determine the potential of the *insert* function.

case $n = 0$**:**

$$g_p(i,n) = (i, 0)$$

case $n > 0$**:**

$$g_p(i,n) = (\bigvee_{size(z) \le n} case(z, f_{Nil}, f_{Cons}))_p$$

$$= \bigvee_{\substack{j \leq i, m < n \\ \text{or } j < i, m \leq n}} (case(z, f_{Nil}, f_{Cons}))_p$$

$$= \bigvee_{\substack{j \leq i, m < n \\ \text{or } j < i, m \leq n}} (case(z, f_{Nil}, f_{Cons}))_p \vee \bigvee_{\substack{j \leq i, m < n \\ \text{or } j < i, m \leq n}} (case(z, f_{Nil}, f_{Cons}))_p$$

$$\leq (i, n)$$

□

As in the interpretation of `insert` we are left with a less than satisfactory bound on the potential of `sort`. It would grievous mistake to write a sorting function whose output was smaller than its input. Under the current interpretation of lists, this would mean either the length of the list decreased or the size of the largest element in the list decreased. Unfortunately we are stuck with an upper bound on the size of the output because the interpretation of *case* is a maximum over a set of smaller terms.

Using the solution to the recurrence for the potential, we can solve the recurrence for the cost of `sort`.

LEMMA 3.10. $\pi_0 g(n) \leq (3 + \pi_0(\pi_1(f\ x)\ i)n^2 + 5n + 1$

PROOF. We prove this by induction on n.

case $n = 0$**:** $g_c(i, n) = 1$

case $n > 0$**:**

$$g_c(i, n) = (\bigvee_{size(z) \leq (i,n)} case(z, f_{Nil}, f_{Cons}))_c$$

$$= \bigvee_{\substack{j < i, m \leq n \\ \text{or } j \leq i, m < n}} 3 + g_c(j, m-1) + (insert\ f\ j\ \pi_1 g(j, m-1))_c$$

$$\leq \bigvee_{\substack{j < i, m \leq n \\ \text{or } j \leq i, m < n}} 3 + g_c(j, m-1) + (insert\ f\ j\ (j, m-1))_c$$

$$\leq \bigvee_{\substack{j < i, m \leq n \\ \text{or } j \leq i, m < n}} 3 + g_c(j, m-1) + (3 + ((f\ j)\ j))(m-1) + 1$$

$$
\begin{aligned}
&\text{let } c_1 = (3 + \pi_0(\pi_1(f\ j)\ j)) \\
&\le \bigvee_{\substack{j<i, m\le n \\ \text{or } j\le i, m<n}} 3 + c_1(m-1)^2 + 5(m-1) + 1 + c_1(m-1) + 1 \\
&\le \bigvee_{\substack{j<i, m\le n \\ \text{or } j\le i, m<n}} 3 + c_1m^2 - 2c_1m + c_1 + 5m - 5 + 1 + c_1m - c_1 + 1 \\
&\le \bigvee_{\substack{j<i, m\le n \\ \text{or } j\le i, m<n}} c_1m^2 - c_1m + 5m + 1 \\
&\le \bigvee_{\substack{j<i, m\le n \\ \text{or } j\le i, m<n}} (3 + \pi_0(\pi_1(f\ i)\ i))n^2 + 5n + 1 \\
&\le (4 + ((f\ i)\ i)_p)_c n^2 + 5n + 1
\end{aligned}
$$

□

As expected the cost of `sort` is $\mathcal{O}(n^2)$ where n is the length of the list. It is clear from the analysis how the cost of the comparison function determines the running time of `sort`. We can see that the comparison function is called order n^2 times.

4. Insertion Sort

Insertion sort is a quadratic time sorting algorithm which sorts a list by inserting an element from an unsorted segment of a container into a sorted segment of the container. Although the asymptotic complexity of insertion sort is less than the optimal $\mathcal{O}(nlog_2n)$, insertion sort does have redeeming attributes. Insertion sort has small constant factors, making it more efficient on small datasets. Insertion sort may be done in-place (Cormen et al. [2001]). The running time of insertion sort is $\mathcal{O}(n^2)$.

$$\texttt{data list} = \texttt{Nil of unit | Cons of int} \times \texttt{list}$$

$$\begin{aligned}\texttt{insert} = \lambda x.\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Cons}\langle x, \texttt{Nil}\rangle,\\ \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{rec}(x \leq y, \texttt{True} \mapsto \texttt{Cons}\langle x, \texttt{Cons}\langle y, ys\rangle\rangle,\\ \texttt{False} \mapsto \texttt{Cons}\langle y, \texttt{force}(r)\rangle))\end{aligned}$$

4.1. Translation. The translation of the function `insert` is shown below.

$$\texttt{sort} = \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil}, \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\texttt{insert}\ y\ \texttt{force}(r))$$

The translation of the function `sort` is shown below.

$$\begin{aligned}\|\texttt{sort}\| = \langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil},\\ \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.(3 + r_c) +_c (\|\texttt{insert}\|_p\ y)_p r_p)\end{aligned}$$

4.2. Interpretation. We will interpret lists as their lengths.

$$\begin{aligned}[\![list]\!] &= \mathbb{N}^\infty\\ D^{list} &= \{*\} + \{1\} \times \mathbb{N}^\infty\\ size_{list}(*) &= 0\\ size_{list}((1, n)) &= 1 + n\end{aligned}$$

The interpretation of the `rec` that drives the cost of insert is given below.

$$g(n) = \bigvee_{size(z) \leq n} case(z, (f_{Nil}, f_{Cons})) \tag{5}$$

$$\text{where} \tag{6}$$

$$f_{Nil}() = (1,1) \tag{7}$$

$$f_{Cons}((1,m)) = (3 + (1 \leq \ 1)_c + g_c(m), (2+m) \vee (1 + g_p(m))) \tag{8}$$

We can extract the recurrence for the cost and eliminate the big maximum operator.

$$c(n) = \begin{cases} 1 & n = 0 \\ 4 + \pi_0(\pi_1(f\ 1)\ 1) + c(n-1) & n > 0 \end{cases}$$

The closed form solution to this recurrence is:

Lemma 3.11. $c(n) = (3 + (1 \leq 1)_c)n + 1$

We also extract a recurrence for the potential and eliminate the big maximum operator.

$$p(n) = \begin{cases} 1 & n = 0 \\ 1 + p(n-1) & n > 0 \end{cases}$$

The closed form solution to this recurrence is:

$$p(n) = 1 + n$$

The interpretation of `sort` is:

$$g(n) = \bigvee_{size(z) \leq n} case(z, (\lambda(\langle\rangle).(1,0), \lambda(1,m).3 + g_c(m)) +_c [\![\|\texttt{insert}\|]\!]\ 1\ g_p(m)) \tag{9}$$

The recurrence with the big maximum operator eliminated is:

$$g(n) = \begin{cases} (1,0) & n = 0 \\ (3 + g_c(n-1) + ([\![\|\texttt{insert}\|]\!]\ 1\ g_p(n-1))_c, ([\![\|\texttt{insert}\|]\!]\ 1\ g_p(n-1))) & n > 0 \end{cases} \tag{10}$$

We extract the recurrence for the potential. Let $p = \pi_1 \circ g$.

$$p(n) = \begin{cases} 0 & n = 0 \\ ([\![\|\texttt{insert}\|]\!]\ 1\ g_p(n-1))) & n > 0 \end{cases} \tag{11}$$

The closed form solution to this recurrence is given below.

Lemma 3.12. $p(n) = n$

We extract the recurrence for the cost. Let $c = \pi_0 \circ g$.

$$c(n) = \begin{cases} 1 & n = 0 \\ 3 + g_c(n-1) + ([\![\|\texttt{insert}\|]\!]\ 1\ g_p(n-1) & n > 0 \end{cases} \tag{12}$$

5. Sequential List Map

This example is provided for comparison with the parallel list map example given later in this thesis. We use the familiar `list` datatype.

$$\texttt{list} \ = \texttt{Nil of unit | Cons of int} \times \texttt{list}$$

The `map` function recurses on the list, applying its first argument to each element.

$$\texttt{map} = \lambda f.\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil}, \texttt{Cons} \mapsto \langle y\langle ys, y\rangle\rangle.\texttt{Cons}\langle f\ y, \texttt{force}(r)\rangle)$$

5.1. Translation.

We apply the rule for translating an abstraction twice.

$$\|\texttt{map}\| = \|\lambda f.\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil}, \texttt{Cons} \mapsto \langle y\langle ys, y\rangle\rangle.\texttt{Cons}\langle f\ y, \texttt{force}(r)\rangle)\|$$

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil}, \texttt{Cons} \mapsto \langle y\langle ys, y\rangle\rangle.\texttt{Cons}\langle f\ y, \texttt{force}(r)\rangle)\rangle\rangle$$

We apply the rule for translating a `rec` construct. We use $\|xs\| = \langle 0, xs\rangle$.

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto 1 +_c \|\texttt{Nil}\|,$$

$$\texttt{Cons} \mapsto \langle y\langle ys, y\rangle\rangle.1 +_c \|\texttt{Cons}\langle f\ y, \texttt{force}(r)\rangle\|)\rangle\rangle$$

The translation of `Nil` is $\langle 0, \texttt{Nil}\rangle$.

To translate the `Cons` branch we translate the subexpressions first.

$$\|\texttt{Cons}\langle f\ x, \texttt{force}(r)\rangle\| = \langle \|\langle f\ x, \texttt{force}(r)\rangle_c, \texttt{Cons}\|\langle f\ x, \texttt{force}(r)\rangle\|\rangle$$

$$\|\langle f\ x, \texttt{force}(r)\rangle\| = \langle \|f\ x\|_c + \|\texttt{force}(r)\|_c, \langle \|f\ x\|_p, \|\texttt{force}(r)\|_p\rangle\rangle$$

$$\|f\ x\| = (1 + \|f\|_c + \|x\|_c) +_c \|f\|_p\|x\|_p = \langle 1 + (f\ x)_c, (f\ x)_p\rangle$$

$$\|\texttt{force}(r)\| = \langle 0, r\rangle +_c \langle 0, r\rangle_p = r$$

$$\|\langle f\ x, \texttt{force}(r)\rangle\| = \langle 1 + (f\ x)_c + r_c, \langle (f\ x)_p, r_p\rangle\rangle$$

$$\|\texttt{Cons}\langle f\ x, \texttt{force}(r)\rangle\| = \langle 1 + (f\ x)_c + r_c, \texttt{Cons}\langle (f\ x)_p, r_p\rangle\rangle$$

$$= \langle 0, \lambda f.\langle 0, \lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle,$$

$$\texttt{Cons} \mapsto \langle y\langle ys, y\rangle\rangle.\langle 2 + (f\ x)_c + r_c, \texttt{Cons}\langle (f\ x)_p, r_p\rangle\rangle$$

We will translate `map` applied to some function `f` and a list `xs`.

$$\|\texttt{map f xs}\| = (1 + \|\texttt{map}\ f\|_c + \|xs\|_c) +_c \|\texttt{map}\ f\|_p \|xs\|_p$$

The translation of `map` partially applied to map is:

$$\|\texttt{map}\ f\| = (1 + \|\texttt{map}\|_c + \|f\|_c) +_c \|\texttt{map}\|_p \|f\|_p$$

The cost of partially applied `map` is 0.

$$= (2 + \|f\|_c + \|xs\|_c) +_c (\|\texttt{map}\|_p\ \|f\|_p)_p \|xs\|_p$$

5.2. Interpretation. We interpret lists as a pair of their largest element and length.

$$\begin{aligned}
[\![\texttt{list}]\!] &= \mathbb{Z} \times \mathbb{N} \\
D^{\texttt{list}} &= \{*\} + ([\![\mathbb{Z}]\!] \times [\![\texttt{list}]\!]) \\
size_{\texttt{list}}(*) &= (0, 0) \\
size_{\texttt{list}}((x, (m, n))) &= (max(x, m), 1 + n)
\end{aligned}$$

The recursor of `map` drives the cost, so we will interpret the recursor first.

$$\begin{aligned}
g(i, n) &= [\![\texttt{rec}(xs, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle, \\
&\qquad\qquad \texttt{Cons} \mapsto \langle 2 + (f\ x)_c + r_c, \texttt{Cons}\langle (f\ x)_p, r_p\rangle\rangle]\!]\{xs \mapsto n, f \mapsto f\} \\
&= \bigvee_{size(z) \leq n} case(z, f_{Nil}, f_{Cons}) \\
&\quad \text{where} \\
f_{Nil}(*) &= [\![\langle 1, \texttt{Nil}\rangle]\!] \\
&= (1, 0) \\
f_{Cons}(i, m) &= [\![\langle 2 + (f\ x)_c + r_c, \texttt{Cons}\langle (f\ x)_p, r_p\rangle\rangle]\!]\xi \\
&\qquad \text{where } \xi = \{xs \mapsto n, f \mapsto f, y \mapsto i, ys \mapsto m, r \mapsto g(i, m)\} \\
&= (2 + (f\ i)_c + g_c(i, m), (max((f\ i)_p, \pi_0 g_p(i, m)), 1 + \pi_1 g_p(i, m)))
\end{aligned}$$

We break the recurrence into the cases of $n = 0$ and $n > 0$

case $n = 0$**:**

$$g(i,0) = \bigvee_{size(z) \leq 0} case(z, f_{Nil}, f_{Cons}) = (1,0)$$

case $n > 0$**:**

$$\begin{aligned}
g(i,n) &= \bigvee_{size(z) \leq n} case(z, f_{Nil}, f_{Cons}) \\
&= \bigvee_{size(z) \leq n-1} case(z, f_{Nil}, f_{Cons}) \vee \bigvee_{size(z) = n} case(z, f_{Nil}, f_{Cons}) \\
&= g(i, n-1) \vee \\
&\quad (2 + (f\ i)_c + g_c(i, n-1), (max((f\ i)_p, \pi_0 g_p(i, n-1)), 1 + \pi_1 g_p(i, n-1)))
\end{aligned}$$

Since $g_c(i, n-1) \leq g_c(i, n-1)$, then $g_c(i, n-1) \leq 2 + (f\ i)_c + g_c(i, n-1)$.

Since $g_p(i, n-1) \leq g_p(i, n-1)$, then $\pi_0 g_p(i, n-1) \leq max((f\ i)_p, \pi_0 g_p(i, n-1))$,

and $\pi_1 g_p(i, n-1) \leq 1 + \pi_1 g_p(i, n-1)$.

$$= (2 + (f\ i)_c + g_c(i, n-1), (max((f\ i)_p, \pi_0 g_p(i, n-1)), 1 + \pi_1 g_p(i, n-1)))$$

We obtain a closed form solution for the cost by using the substitution method.

Lemma 3.13. $g_c(i, n) = (2 + (f\ i)_c)n + 1$

Proof. The proof is by induction on n.

case $n = 0$**:**

$$g_c(i, 0) = (1, 0)_c = 1$$

case $n > 0$**:**

$$\begin{aligned}
g_c(i,n) &= 2 + (f\ i)_c + g_c(i, n-1) \\
&= 2 + (f\ i)_c + (2 + (f\ i)_c)(n-1) + 1 \\
&= 2 + (f\ i)_c + (2 + (f\ i)_c)n - 2 - (f\ i)_c + 1 \\
&= (2 + (f\ i)_c)n + 1
\end{aligned}$$

□

We obtain a closed form of the solution for the potential of the recursor.

LEMMA 3.14. $g_p(i, n) = ((f\ i)_p, n)$

PROOF. The proof is by induction on n.

case $n = 0$**:**

$g_p(i, 0) = (1, 0)_p = 0$

case $n > 0$**:**

$$\begin{aligned} g_p(i, n) &= (max((f\ i)_p, \pi_0 g_p(i, n-1)), 1 + \pi_1 g_p(i, n-1)) \\ &= (max((f\ i)_p, (f\ i)_p), 1 + n - 1) \\ &= ((f\ i)_p, n) \end{aligned}$$

□

Using the results from lemmas 3.13 and 3.14, we interpret the translation of `map f xs`. Recall the translation is $(2 + \|f\|_c + \|xs\|_c) +_c (\|\mathtt{map}\|_p\ \|f\|_p)_p \|xs\|_p$. So the interpretation is $2 +\!\!+_c ([\![\|\mathtt{map}\|]\!] f)_p(i, n)$, where (i, n) is the interpretation of xs and we assume f and xs have 0 cost.

LEMMA 3.15. *$[\![\|\mathtt{map\ f\ xs}\|_c]\!] = 3 + (2 + (f\ i)_c)n$ where f is the interpretation of the translation of* `f` *and (i, n) is the interpretation of the translation of* `xs`*.*

LEMMA 3.16. *$[\![\|\mathtt{map\ f\ xs}\|_p]\!] = ((f\ i)_p, n)$ where f is the interpretation of the translation of* `f` *and (i, n) is the interpretation of the translation of* `xs`*.*

Lemma 3.15 shows that the cost of `map f xs` is linear in the size of the list but also depends on the cost of applying `f` to the elements of `xs`. Lemma 3.16 shows the cost of future uses of `map f xs` depends on the length of `xs` and the size of the result of applying `f` to the elements of `xs`.

6. Sequential Tree Map

This example is presented for comparison with the parallel tree map given in chapter 4. We use `int` labelled binary trees.

$$\texttt{datatype tree} = \texttt{E of Unit | N of int}\times\texttt{tree}\times\texttt{tree}$$

Tree `map f t` applies the function `f` to every label in the tree `t`.

$$\texttt{map} = \lambda f.\lambda t.\texttt{rec}(t, \texttt{E} \mapsto \texttt{E}, \texttt{N} \mapsto \langle x, \langle t_0, r_0\rangle, \langle t_1, r_1\rangle\rangle.\texttt{N}\langle f\ x, \texttt{force}(r_0), \texttt{force}(r_1)\rangle)$$

6.1. Translation.

We apply the abstraction rule twice.

$$\|\texttt{map}\| = \langle 0, \lambda f.\langle 0, \lambda t.\|\texttt{rec}(t, \texttt{E} \mapsto \texttt{E},$$
$$\texttt{N} \mapsto \langle x, \langle t_0, r_0\rangle, \langle t_1, r_1\rangle\rangle.\texttt{N}\langle f\ x, \texttt{force}(r_0), \texttt{force}(r_1)\rangle)\|\rangle\rangle$$

We apply the recursor translation rule.

$$= \langle 0, \lambda f.\langle 0, \lambda t.\|t\|_c +_c \texttt{rec}(\|t\|_p, \texttt{E} \mapsto 1 +_c \|\texttt{E}\|,$$
$$\texttt{N} \mapsto \langle x, \langle t_0, r_0\rangle, \langle t_1, r_1\rangle\rangle.1 +_c \|\texttt{N}\langle f\ x, \texttt{force}(r_0), \texttt{force}(r_1)\rangle\|)\rangle\rangle$$

The translation of the variable t is $\langle 0, t\rangle$.

The translation of the constructor `E` with $\langle\rangle$ as its argument is $\langle 0, \texttt{E}\rangle$.

The translation of the constructor $\texttt{N}\langle e\rangle$ is $\langle \|e\|_c, \texttt{N}\langle \|e\|_p\rangle\rangle$.

f and x are variables, so their translations are $\langle 0, f\rangle$ and $\langle 0, x\rangle$ respectively.

$$\|f\ x\| = \langle 1 + \|f\|_c + \|x\|_c +_c \|f\|_p\|x\|_p = \langle 1 + (f\ x)_c, (f\ x)_c\rangle$$

r_0 and r_1 are also variables.

$$\|\texttt{force}(r_i)\| = \|r_i\|_c +_c \|r_i\|_p = \langle 0, r_i\rangle +_c \langle 0, r_i\rangle = r_i$$

We use this result to translate the argument to the `N` constructor.

$$\|\langle f\ x, \texttt{force}(r_0), \texttt{force}(r_1)\rangle\| =$$
$$\langle \|f\ x\|_c + \|\texttt{force}(r_0)\|_c + \|\texttt{force}(r_1)\|_c,$$
$$\langle \|f\ x\|_p, \|\texttt{force}(r_0)\|_p\rangle, \|\texttt{force}(r_1)\|_p\rangle\rangle$$

$$= \langle 1 + (f\ x)_c + r_{0c} + r_{1c}, \langle (f\ x)_p, r_{0p}, r_{1p} \rangle \rangle$$

We use this to translate the `N` constructor.

$$\mathtt{N}\langle f\ x, \mathtt{force}(r_0), \mathtt{force}(r_1) \rangle = \langle 1 + (f\ x)_c + r_{0c} + r_{1c}, \mathtt{N}\langle (f\ x)_p, r_{0p}, r_{1p} \rangle \rangle$$

We use this to complete the translation of `map`.

$$= \langle 0, \lambda f.\langle 0, \lambda t.\mathtt{rec}(t, \mathtt{E} \mapsto \langle 1, \mathtt{E} \rangle,$$
$$\mathtt{N} \mapsto \langle x, \langle t_0, r_0 \rangle, \langle t_1, r_1 \rangle \rangle.\langle 2 + (f\ x)_c + r_{0c} + r_{1c}, \mathtt{N}\langle (f\ x)_p, r_{0p}, r_{1p} \rangle \rangle)\rangle \rangle$$

The translation of `map` applied to a function `f` and a tree `t` is:

$$\|\mathtt{map\ f\ t}\| = (1 + \|\mathtt{map\ f}\|_c + \|t\|_c) +_c \|\mathtt{map\ f}\|_p \|t\|_p$$
$$\|\mathtt{map\ f}\| = (1 + \|\mathtt{map}\|_c + \|f\|_c) +_c \|\mathtt{map}\|_p \|f\|_p$$

Since the cost of `map` and `map f` are 0, we can simplify this.

$$= \langle 1 + \|f\|_c, (\|\mathtt{map}\|_p \|f\|_p)_p \rangle$$
$$= (2 + \|f\|_c + \|t\|_c) +_c (\|\mathtt{map}\|_p \|f\|_p)_p \|t\|_p$$

6.2. Interpretation. We interpret trees as a three-tuple of their largest element and number of `N` constructors.

$$[\![tree]\!] = \mathbb{Z} \times \mathbb{Z}$$
$$D_{\mathtt{tree}} = \{*\} + \mathbb{Z} \times [\![\mathtt{tree}]\!] \times [\![\mathtt{tree}]\!]$$
$$size_{\mathtt{tree}}(*) = (0, 0)$$
$$size_{\mathtt{tree}}(x, (i_0, n_0), (i_1, n_1)) = (max(x, i_0, i_1), 1 + n_0 + n_1)$$

We will interpret the `rec` construct first, since it drives the cost of `map`.

$$g(i, n) = [\![\mathtt{rec}(t, \mathtt{E} \mapsto \langle 1, \mathtt{E} \rangle,$$
$$\mathtt{N} \mapsto \langle x, \langle t_0, r_0 \rangle, \langle t_1, r_1 \rangle \rangle.$$
$$\langle 2 + (f\ x)_c + r_{0c} + r_{1c}, \mathtt{N}\langle (f\ x)_p, r_{0p}, r_{1p} \rangle \rangle)\rangle \rangle]\!]\xi$$

where $\xi = \{t \mapsto (i, n), f \mapsto f\}$

$$= \bigvee_{size(z) \leq n} case(z, f_E, f_N)$$

where

$$f_E(*) = [\![\langle 1, \mathtt{E}\rangle]\!]\xi$$

$$= (1, (0, 0))$$

$$f_N(j, (j_0, n_0), (j_1, n_1)) = [\![\langle 2 + (f\ x)_c + r_{0c} + r_{1c}, \mathtt{N}\langle (f\ x)_p, r_{0p}, r_{1p}\rangle\rangle]\!]\xi'$$

where $\xi' = \xi\{x \mapsto j, r_0 \mapsto g(j_0, n_0), r_1 \mapsto g(j_1, n_1)\}$

$$= (2 + (f\ j)_c + g_c(j_0, n_0) + g_c(j_1, n_1),$$
$$(max((f\ j)_p, \pi_0 g_p(j_0, n_0), \pi_0 g_p(j_1, n_1)),$$
$$1 + \pi_1 g_p(j_0, n_0) + \pi_1 g_p(j_1, n_1)))$$

To obtain a closed form solution for this recurrence, we first eliminate the big maximum.

$$g(i, n) = \bigvee_{size(z) \leq (i,n)} case(z, f_E, f_N)$$

$$= (1, (0, 0)) \vee \bigvee_{size(j,(j_0,n_0),(j_1,n_1)) \leq (i,n)} f_N(j, (j_0, n_0), (j_1, n_1))$$

$$\leq \bigvee_{size(j,(j_0,n_0),(j_1,n_1)) \leq (i,n)} f_N(j, (j_0, n_0), (j_1, n_1))$$

$$\leq \bigvee_{\substack{1+n_0+n_1 \leq n \\ max(j,j_0,j_1) \leq s}} (2 + (f\ j)_c + g_c(j_0, n_0) + g_c(j_1, n_1),$$
$$(max((f\ j)_p, \pi_0 g_p(j_0, n_0), \pi_0 g_p(j_1, n_1)),$$
$$1 + \pi_1 g_p(j_0, n_0) + \pi_1 g_p(j_1, n_1)))$$

$$\leq \bigvee_{\substack{1+n_0+n_1 \leq n \\ max(j,j_0,j_1) \leq s}} (2 + (f\ j)_c + g_c(j_0, n_0) + g_c(j_1, n_1),$$
$$(max((f\ j)_p, \pi_0 g_p(j_0, n_0), \pi_0 g_p(j_1, n_1)),$$
$$1 + \pi_1 g_p(j_0, n_0) + \pi_1 g_p(j_1, n_1)))$$

CHAPTER 4

Parallel Functional Program Analysis

We demonstrate the flexibility of the method developed in Danner et al. [2015] by extending it to parallel cost semantics. We introduce a parallel operational cost semantics for the source language, alter the translation into the complexity language to produce a new notion of cost, and then prove the bounding theorem for the new complexity translation function. Finally we give two examples of the recurrence extraction and interpretation.

To analyze the complexity of a sequential program, we are only interested in a measure of the steps required to run the program. To analyze the complexity of a parallel program, we need a measure which takes into account the extent to which a computation may be run on multiple processors. The cost semantics we use are work and span Harper [2012]. We give a brief overview of work and span.

1. Work and span

Work and span is a method of predicting the running time of programs that may be run on arbitrary number of processors. Instead of producing an approximation of the number of steps required to execute a program, work and span instead produces a cost graph; a specification of the dependencies between subcomputations of the program. The cost graph can be compiled into two measures, work and span. The work of a program corresponds to the total number of steps required to execute the program. The span is the number of steps in the critical path. The critical path is the longest number of steps that must be executed sequentially. The length of the critical path determines the extent to which a program may be parallelized. If the span is equal to the work, than every step in the computation depends on the previous step, and the subcomputations of the program cannot be run independently, so the program cannot be parallelized. If the span is smaller than the work, then there are subcomputations which may be run independently and the program may be parallelized. The upper bound on the running time of a program is summarized by theorem 4.1.

THEOREM 4.1 (Brent's Theorem). *A program with work w and span s may be evaluated on p processors in $O(max(w/p, s))$ steps.*

A cost graph is defined as follows.

$$\mathcal{C} ::= 0 \mid 1 \mid \mathcal{C} \oplus \mathcal{C} \mid \mathcal{C} \otimes \mathcal{C}$$

The operator $\oplus$ connects to cost graphs who must be combined sequentially. The operator $\otimes$ connects cost graphs which may be combined in parallel.

The work of a cost graph is defined as

$$work(c) = \begin{cases} 0 & \text{if } c = 0 \\ 1 & \text{if } c = 1 \\ work(c_0) + work(c_1) & \text{if } c = c_0 \otimes c_1 \\ work(c_0) + work(c_1) & \text{if } c = c_0 \oplus c_1 \end{cases}$$

Since the work of a program is the total number of steps required to run the program, we add the work of subgraphs regardless whether the may be run independently or not.

The span of a cost graph is defined as

$$span(c) = \begin{cases} 0 & \text{if } c = 0 \\ 1 & \text{if } c = 1 \\ max(span(c_0), span(c_1)) & \text{if } c = c_0 \otimes c_1 \\ span(c_0) + span(c_1) & \text{if } c = c_0 \oplus c_1 \end{cases}$$

Cost graphs connected by $\oplus$ must be run sequentially, so their span is the sum of the spans of the subgraphs. Cost graphs connected by $\otimes$ may be run independently, so their span is the maximum of the spans of the subgraphs.

Figure 1. Source language operational semantics

$$\frac{e_0 \downarrow^{n_0} v_0 \qquad e_1 \downarrow^{n_1} v_1}{\langle e_0, e_1\rangle \downarrow^{n_0 \otimes n_1} \langle v_0, v_1\rangle} \qquad \frac{e_0 \downarrow^{n_0} \langle v_0, v_1\rangle \qquad e_1[v_0/x_0, v_1/x_1] \downarrow^{n_1} v}{split(e_0, x_0.x_1.e_1) \downarrow^{n_0 \oplus n_1} v}$$

$$\frac{e_0 \downarrow^{n_0} \lambda x.e_0' \qquad e_1 \downarrow^{n_1} v_1 \qquad e_0'[v_1/x] \downarrow^{n} v}{e_0\ e_1 \downarrow^{(n_0 \otimes n_1) \oplus n \oplus 1} v} \qquad \frac{}{delay(e) \downarrow^{0} delay(e)}$$

$$\frac{e \downarrow^{n_0} delay(e_0) \qquad e_0 \downarrow^{n_1} v}{force(e) \downarrow^{n_0 \oplus n_1} v} \qquad \frac{e \downarrow^{n} v}{Ce \downarrow^{n} Cv}$$

$$\frac{e \downarrow^{n_0} Cv_0 \qquad map^{\phi_C}(y.\langle y, delay(rec(y, \overline{C \mapsto x.e_C}))\rangle, v_0) \downarrow^{n_1} v_1 \qquad e_C[v_1/x] \downarrow^{n_2} v}{rec(e, \overline{C \mapsto x.e_C}) \downarrow^{1 \oplus n_0 \oplus n_1 \oplus n_2} v}$$

$$\frac{}{map^{t}(x.v, v_0) \downarrow^{0} v[v_0/x]} \qquad \frac{}{map^{\tau}(x.v, v_0) \downarrow^{0} v_0}$$

$$\frac{map^{\phi_0}(x.v, v_0) \downarrow^{n_0} v_0' \qquad map^{\phi_1}(x.v, v_1) \downarrow^{n_1} v_1'}{map^{\phi_0 \times \phi_1}(x.v, \langle v_0, v_1\rangle) \downarrow^{n_0 \otimes n_1} \langle v_0', v_1'\rangle}$$

$$\frac{}{map^{\tau \to \phi}(x.v, \lambda y.e) \downarrow^{0} \lambda y.let(e, z.map^{\phi}(x.v, z))} \qquad \frac{e_0 \downarrow^{n_0} v_0 \qquad e_1[v_0/x] \downarrow^{n_1} v}{let(e_0, x.e_1) \downarrow^{n_0 \oplus n_1} v}$$

1.1. Operational Cost Semantics. We alter the operational cost semantics of the source language to produce a cost graph instead of a natural number. Figure 1.1 shows the new operational cost semantics. For tuples, the subexpressions may be evaluated in parallel, so the cost of evaluating a tuple is the cost graphs of the subexpressions connected by $\otimes$. For `split`, the second subexpression depends on the result of the first subexpression, so the cost of evaluating the `split` is the cost graphs of the subexpression connected with $\oplus$. In every rule except for tuples and function application, we replace $+$ with $\oplus$. Because tuples and function application are the two syntactic forms which consist of multiple subexpressions whose evaluation does not depend on each other. The complexity translation is given in Figure 1.1. The operator $E_0 \oplus_c E_1$ is syntactic sugar for $\langle E_0 \oplus E_{1c}, E_{1p}\rangle$. The translation is similar to the original translation except we replace the use of $+$ and $+_c$ with $\oplus$ and $\oplus_c$. In the tuple case and function application case, the subexpressions may be computed in parallel so the cost is the costs of the subgraphs connected with $\otimes$.

FIGURE 2. Work and span translation from source language to complexity language

$$\begin{aligned}
\|x\| &= \langle 0, x\rangle \\
\|\langle\rangle\| &= \langle 0, \langle\rangle\rangle \\
\|\langle e_0, e_1\rangle\| &= \langle \|e_0\|_c \otimes \|e_1\|_c, \langle \|e_0\|_p, \|e_1\|_p\rangle\rangle \\
\|split(e_0, x_0.x_1.e_1)\| &= \|e_0\|_c \oplus_c \|e_1\|[\pi_0\|e_0\|_p/x_0, \pi_1\|e_1\|_p/x_1] \\
\|\lambda x.e\| &= \langle 0, \lambda x.\|e\|\rangle \\
\|e_0\ e_1\| &= 1 \oplus (\|e_0\|_c \otimes \|e_1\|_c) \oplus_c \|e_0\|_p\ \|e_1\|_p \\
\|delay(e)\| &= \langle 0, \|e\|\rangle \\
\|force(e)\| &= \|e\|_c \oplus_c \|e\|_p \\
\|C_i^\delta e\| &= \langle \|e\|_c, C_i^\delta \|e\|_p\rangle \\
\|rec^\delta(e, \overline{C \mapsto x.e_C})\| &= \|e\|_c \oplus_c rec^\delta(\|e\|_p, \overline{C \mapsto x.1 \oplus_c \|e_C\|}) \\
\|map^\phi(x.v_0, v_1)\| &= \langle 0, map^{\langle\langle\phi\rangle\rangle}(x.\|v_0\|_p, \|v_1\|_p)\rangle \\
\|let(e_0, x.e_1)\| &= \|e_0\|_c \oplus_c \|e_1\|[\|e_0\|_p/x]
\end{aligned}$$

2. Bounding Relation

We verify that the the translation of a well-typed source language is bounded by its translation into the complexity language. We mutually define the following bounding relations:

(1) $e \sqsubseteq_\tau E, \emptyset \vdash_\phi e : \tau$ and $\emptyset \vdash_{\|\psi\|} E : \|\tau\|$
(2) $v \sqsubseteq_\tau^{val} E$, where $\emptyset \vdash v : \tau$ and $\emptyset \vdash_{\|\psi\|} E : \langle\langle\tau\rangle\rangle$
(3) $v \sqsubseteq_{\phi,R}^{val} E$, where $\emptyset \vdash_\psi v : \phi[\gamma]$ and $\emptyset \vdash_{\|\psi\|} E : \langle\langle\phi\rangle\rangle[\delta]$
(4) $e \sqsubseteq_{\phi,R} E$, where $\emptyset \vdash_\psi e : \phi[\delta]$ and $\emptyset \vdash_{\|\psi\|} E : \|\phi\|[\delta]$

The bounding relation $e \sqsubseteq_\tau$ defines the notion of an source language expression to be bounded by a complexity language expression. We will refer to this as "bounding". The second bounding relation $e \sqsubseteq_{\tau}^{val} E$ defines a notion of a complexity language potential bounding a source language value. We will refer to this relation as "value bounding". Relations three and four are parameterized by any relation R and interpret strictly positive functors as relation transformers. The mutual definitions of the relations are given below.

Definition 4.1 (Bounding Relation).

(1) We define $e \sqsubseteq_{\tau} E$ as if $e \downarrow^{n} v$, then
  - $n \leq E_c$
  - $v \sqsubseteq_{\tau}^{val} E_p$.

(2) We define $v \sqsubseteq_{\tau}^{val} E$ as
  - $v \sqsubseteq_{unit}^{val} E$ always.
  - $\langle v_0, v_1 \rangle \sqsubseteq_{\tau_0 \times \tau_1}^{val} E$ iff $v_0 \sqsubseteq_{\tau_0}^{val} \pi_0 E$ and $v_1 \sqsubseteq_{\tau_2}^{val} \pi_1 E$.
  - $\texttt{delay}(e) \sqsubseteq_{\texttt{susp}\ \tau}^{val}$ if $e \sqsubseteq_{\tau} E$.
  - $v \sqsubseteq_{\delta}^{val} E$ is inductively defined by

$$\frac{\mathbf{C} : \phi \to \delta \in \psi \qquad v \sqsubseteq_{\phi, -\sqsubseteq_{\delta}^{val}-}^{val} E' \qquad \mathbf{C}\ E' \leq_{\delta} E}{\mathbf{C}\ v \sqsubseteq_{\delta}^{val} E}$$

  - $\lambda x.e \sqsubseteq_{\tau \to \phi, R}^{val} E$ if for all v and E_0, if $v \sqsubseteq_{\tau}^{val} E_0$, then $e[v/x] \sqsubseteq_{\phi, R} (E\ E_0)$

(3) We define $v \sqsubseteq_{\phi,R}^{val} E_p$ as
  - $v \sqsubseteq_{t,R}^{val} E$ if $R(v, E)$
  - $v \sqsubseteq_{\tau,R}^{val} E$ if $v \sqsubseteq_{\tau}^{val} E$.
  - $\langle v_0, v_1 \rangle \sqsubseteq_{\phi_0 \times \phi_1, R}^{val} E$ if $v_0 \sqsubseteq_{\phi_0, R}^{val} \pi_0 E$ and $v_1 \sqsubseteq_{\phi_0, R}^{val} \pi_1 E$

(4) We define as $e \sqsubseteq_{\phi,R} E$ as if $e \downarrow^{n} v$, then
  - $n \leq E_c$
  - $v \sqsubseteq_{\phi,R}^{val} E_p$

The definition 1 of $e \sqsubseteq_{\tau} E$ states that if a source language expression e steps to a value v with cost n, then the cost n is less than or equal to the cost of the complexity language expression E and the value v is value bounded by potential of the complexity language expression E. In the sequential cost semantics, the cost is a natural number and $\leq$ is the ordering $a \leq b$ if there exists a natural number c such that $a + c = b$. We need to impose a partial ordering on cost graphs. We use the containment ordering. A cost graph a is less than or equal to a cost graph b if a is a subset of b. The subset relation is transitive and antisymmetric, so the ordering is transitive and antisymmetric. The ordering must satisfy these two properties in order for the proof of the bounding relation to fall through.

The bounding theorem states well-typed source language expressions are bounded by their translations into the complexity language.

THEOREM 4.2 (Bounding Theorem). *If* $\gamma \vdash e : \tau$, *then* $e \sqsubseteq_\tau \|e\|$.

PROOF. The proof proceeds by induction on the typing derivation and is identical to the proof of the bounding theorem Danner et al. [2015]. We will not reproduce it here. $\square$

3. Parallel List Map

We will analyze the cost of the `map` function from section 5. We translate the source language program using the new translation function and interpret the resulting complexity language expression in a similar denotational semantics.

We use the same data type `list` and `map` definition.

$$\texttt{datatype list} = \texttt{Nil of Unit | Cons of int} \times \texttt{list}$$

$$\texttt{map} = \lambda f.\lambda xs.\texttt{rec}(xs, \texttt{Nil} \mapsto \texttt{Nil}, \texttt{Cons} \mapsto \langle y \langle ys, y \rangle \rangle.\texttt{Cons}\langle f\ y, \texttt{force}(r) \rangle)$$

The derivation of the complexity expression is given in Figure 3. The complexity language translation is

$$\|\texttt{map f xs}\| = (2 \oplus f_c \otimes xs_c) \oplus_c$$

$$\texttt{rec}(xs_p, \texttt{Nil} \mapsto \langle 1, \texttt{Nil} \rangle,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r \rangle \rangle.\langle 1 \oplus ((1 \oplus (f_p\ y)_c) \otimes r_c), \texttt{Cons}\langle (f_p\ y)_p, r_p \rangle \rangle)$$

We interpret lists as a pair of their largest element and length.

$$[\![\texttt{list}]\!] = \mathbb{Z} \times \mathbb{N}$$

FIGURE 3. Work and span complexity translation of `map f xs`.

$$\begin{aligned}
&\|\texttt{map f xs}\| = \\
&(2 \oplus f_c \otimes xs_c) \oplus_c \texttt{rec}(xs_p, \texttt{Nil} \mapsto 1 \oplus_c \|\texttt{Nil}\|, \\
&\qquad\qquad\qquad\qquad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.1 \oplus_c \|\texttt{Cons}\langle f\ y, \texttt{force}(r)\rangle\|) \\
&\quad 1 \oplus_c \|\texttt{Nil}\| = 1 \oplus_c \langle 0, Nil\rangle = \langle 1, Nil\rangle \\
&\quad \|\texttt{Cons}\langle f\ y, \texttt{force}(r)\rangle\| = \langle \|\langle f\ y, \texttt{force}(r)\rangle\|_c, \texttt{Cons}\|\langle f\ y, \texttt{force}(r)\rangle\|_p\rangle \\
&\qquad \|\langle f\ y, \texttt{force}(r)\rangle\| = \langle \|f\ y\|_c \otimes \|\texttt{force}(r)\|_c, \langle \|f\ y\|_p, \|\texttt{force}(r)\|_p\rangle\rangle \\
&\qquad\quad \|f\ y\| = (1 \oplus \|f\|_c \otimes \|y\|_c) \oplus_c \|f\|_p \|y\|_p \\
&\qquad\qquad = (1 \oplus \langle 0, f_p\rangle_c \otimes \langle 0, y\rangle_c) \oplus_c \langle 0, f_p\rangle_p \langle 0, y\rangle_p \\
&\qquad\qquad = 1 \oplus_c f_p\ y \\
&\qquad\qquad = \langle 1 \oplus (f_p y)_c, (f_p\ y)_p\rangle \\
&\qquad\quad \|\texttt{force}(r)\| = \|r\|_c \oplus_c \|r\|_p \\
&\qquad\qquad = \langle 0, r\rangle_c \oplus_c \langle 0, r\rangle_p \\
&\qquad\qquad = r \\
&\qquad \|\langle f\ y, \texttt{force}(r)\rangle\| = \langle (1 \oplus (f_p\ y)_c) \otimes r_c, \langle (f_p\ y)_p, r_p\rangle\rangle \\
&\quad \|\texttt{Cons}\langle f\ y, \texttt{force}(r)\rangle\| = \langle (1 \oplus (f_p\ y)_c) \otimes r_c, \texttt{Cons}\langle (f_p\ y)_p, r_p\rangle\rangle \\
&\|\texttt{map f xs}\| = (2 \oplus f_c \otimes xs_c) \oplus_c \\
&\quad \texttt{rec}(xs_p, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle, \\
&\qquad\quad \texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\langle 1 \oplus ((1 \oplus (f_p\ y)_c) \otimes r_c), \texttt{Cons}\langle (f_p\ y)_p, r_p\rangle\rangle)
\end{aligned}$$

$$\begin{aligned}
D^{\texttt{list}} &= \{*\} + ([\![\mathbb{Z}]\!] \times [\![\texttt{list}]\!]) \\
size_{\texttt{list}}(*) &= (0, 0) \\
size_{\texttt{list}}((x, (m, n))) &= (max(x, m), 1 + n)
\end{aligned}$$

The interpretation of the recursor is given in Figure 3. The result is

Figure 4. Interpretation of recursor in `map`

Let $\eta = \{xs \mapsto (0,(m,n)), f \mapsto (0,f)\}$

$$g(f,(m,n)) = [\![\texttt{rec}(xs_p, \texttt{Nil} \mapsto \langle 1, \texttt{Nil}\rangle,$$

$$\texttt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\langle 1 \oplus (1 \oplus (f_p\ y)_c \otimes r_c), \texttt{Cons}\langle (f_p\ y)_p, r_p\rangle\rangle)]\!]\eta$$

$$= \bigvee_{(m_1,n_1)\leq(m,n)} case((m_1,n_1), \texttt{Nil} \mapsto [\![\langle 1, \texttt{Nil}\rangle]\!]\eta,$$

$$\texttt{N} \mapsto \langle y, \langle ys, r\rangle\rangle.[\![\langle 1 \oplus ((1 \oplus (f_p\ y)_c) \otimes r_c), \texttt{Cons}\langle (f_p\ y)_p, r_p\rangle\rangle]\!]\eta_c)$$

where $\eta_c = \{xs \mapsto (0,(m,n)), f \mapsto (0,f), y \mapsto m, ys \mapsto (0,(m,n)),$

$r \mapsto g(f,(m,n-1)))\}$

`Nil` branch

$$[\![\langle 1, \texttt{Nil}\rangle]\!]\eta = ([\![1]\!]\eta, [\![\texttt{Nil}]\!]\eta) = (1,(0,0))$$

`Cons` branch

$$[\![\langle 1 \oplus ((1 \oplus (f_p\ m)_c) \otimes r_c), \texttt{Cons}\langle (f_p\ m)_p, r_p\rangle\rangle]\!]\eta_c$$

$$= (1 \oplus ((1 \oplus (f\ m)_c) \otimes g_c(f,(m,n-1))), ((f\ m)_p, \pi_1 g_p(f,(m,n-1))))$$

$$g(f,(m,n)) =$$

$$\bigvee_{(m_1,n_1)\leq(m,n)} case((m_1,n_1), \texttt{Nil} \mapsto (1,(0,0)),$$

$$\texttt{Cons} \mapsto (1 \oplus ((1 \oplus (f\ m)_c) \otimes g_c(f,(m,n-1))), ((f\ m)_p, \pi_1 g_p(f,(m,n-1)))))$$

$$g(f,(m,n)) =$$

$$\bigvee_{(m_1,n_1)\leq(m,n)} case((m_1,n_1), \texttt{Nil} \mapsto (1,(0,0)),$$

$$\texttt{Cons} \mapsto (1 \oplus ((1 \oplus (f\ y)_c) \otimes g_c(f,(m,n-1))), ((f\ y)_p, \pi_1 g_p(f,(m,n-1)))))$$

We compile the recurrence down to the work and the span to make it easier to manipulate.

$$
\begin{aligned}
&g(f,(m,n)) = \\
&\quad \bigvee_{(m_1,n_1)\leq(m,n)} case((m_1,n_1), \\
&\qquad\qquad \texttt{Nil} \mapsto ((1,1),(0,0)), \\
&\qquad\qquad \texttt{Cons} \mapsto ((2+\pi_0(f\ y)_c + \pi_0 g_c(f,(m,n-1)), \\
&\qquad\qquad\qquad 1+max(1+\pi_1(f\ y)_c, \pi_1 g_c(f,(m,n-1)))), \\
&\qquad\qquad\quad ((f\ y)_p, \pi_1 g_p(f,(m,n-1)))))
\end{aligned}
$$

We prove by induction bounds on the work and span of the cost of g.

Theorem 4.3. $\pi_0 g_c(f,(m,n)) \leq 1 + (2+\pi_0(f\ m)_c)n$

Proof. The proof is by induction on n.

case $n = 0$**:**

$$\pi_0 g_c(f,(m,0)) = \pi_0((1,1),(0,0))_c = \pi_0(1,1) = 1$$

case $n > 0$**:**

$$
\begin{aligned}
&\pi_0(g_c(f,(m,n))) = \\
&\quad \pi_0(\bigvee_{(m_1,n_1)\le(m,n)} case((m_1,n_1), \\
&\qquad\qquad \texttt{Nil} \mapsto ((1,1),(0,0)), \\
&\qquad\qquad \texttt{Cons} \mapsto ((2+\pi_0(f\ m_1)_c + \pi_0 g_c(f,(m_1,n_1-1)), \\
&\qquad\qquad\qquad 1+max(1+\pi_1(f\ m_1)_c, \pi_1 g_c(f,(m_1,n_1-1)))), \\
&\qquad\qquad\qquad ((f\ m_1)_p, \pi_1 g_p(f,(m_1,n_1-1))))))_c \\
&\quad = 1 \vee \pi_0(\bigvee_{(m_1,n_1)\le(m,n)} ((2+\pi_0(f\ m_1)_c + \pi_0 g_c(f,(m_1,n_1-1)), \\
&\qquad\qquad\qquad 1+max(1+\pi_1(f\ m_1)_c, \pi_1 g_c(f,(m_1,n_1-1)))), \\
&\qquad\qquad\qquad ((f\ m_1)_p, \pi_1 g_p(f,(m_1,n_1-1))))))_c \\
&\quad = 1 \vee \bigvee_{(m_1,n_1)\le(m,n)} 2+\pi_0(f\ m_1)_c + \pi_0 g_c(f,(m_1,n_1-1)) \\
&\quad \le \bigvee_{(m_1,n_1)\le(m,n)} 2+\pi_0(f\ m_1)_c + (1+(2+\pi_0(f\ m_1)_c)(n_1-1)) \\
&\quad \le 2+\pi_0(f\ m)_c + 1 + (2+\pi_0(f\ m)_c)(n-1) \\
&\quad \le 1+(2+\pi_0(f\ m)_c)n
\end{aligned}
$$

□

THEOREM 4.4. $\pi_1 g_c(f,(m,n)) \le 1 + \pi_1(f\ m)_c + n$

PROOF. **case** $n = 0$**:**

$$\pi_1 g_c(f,(m,0)) = \pi_1((1,1),(0,0))_c = \pi_1(1,1) = 1$$

case $n > 0$**:**

$$
\begin{aligned}
&\pi_1(g_c(f,(m,n))) = \\
&\quad \pi_1(\bigvee_{(m_1,n_1)\leq(m,n)} case((m_1,n_1), \\
&\qquad\qquad \texttt{Nil} \mapsto ((1,1),(0,0)), \\
&\qquad\qquad \texttt{Cons} \mapsto ((2+\pi_0(f\ m_1)_c + \pi_0 g_c(f,(m_1,n_1-1)), \\
&\qquad\qquad\qquad 1+max(1+\pi_1(f\ m_1)_c, \pi_1 g_c(f,(m_1,n_1-1)))), \\
&\qquad\qquad\qquad ((f\ m_1)_p, \pi_1 g_p(f,(m_1,n_1-1))))))_c \\
&\quad = 1 \vee \pi_1(\bigvee_{(m_1,n_1)\leq(m,n)} ((2+\pi_0(f\ m_1)_c + \pi_0 g_c(f,(m_1,n_1-1)), \\
&\qquad\qquad\qquad 1+max(1+\pi_1(f\ m_1)_c, \pi_1 g_c(f,(m_1,n_1-1)))), \\
&\qquad\qquad\qquad ((f\ m_1)_p, \pi_1 g_p(f,(m_1,n_1-1))))))_c \\
&\quad = 1 \vee \bigvee_{(m_1,n_1)\leq(m,n)} 1+max(1+\pi_1(f\ m_1)_c, \pi_1 g_c(f,(m_1,n_1-1))) \\
&\quad \leq \bigvee_{(m_1,n_1)\leq(m,n)} 1+max(1+\pi_1(f\ m_1)_c, 1+\pi_1(f\ m_1)_c+n_1-1) \\
&\quad \leq 1+max(1+\pi_1(f\ m)_c, 1+\pi_1(f\ m)_c+n-1) \\
&\quad \leq 1+1+\pi_1(f\ m)_c+n-1 \\
&\quad \leq 1+\pi_1(f\ m)_c+n
\end{aligned}
$$

□

Compare these results with sequential map.

4. Parallel Tree Map

A program which is embarrasingly parallel is tree map. When a function f is mapped over a tree t, each application of f to the label at each node can be done independently. Furthermore, the tree data structure itself is dividable by construction. Dividing the work requires only destruction of the node constructor to yield the left and right subtrees.

We will use `int` labelled binary trees.

$$\texttt{datatype tree} = \texttt{E of Unit | N of int}\times\texttt{tree}\times\texttt{tree}$$

`map` simply deconstructs each node, applies the function to the label, recuses on the children, and reconstructs a node using the results.

$$\texttt{map} = \lambda f.\lambda t.\texttt{rec}(t, \texttt{E} \mapsto \texttt{E}, \texttt{N} \mapsto \langle x, \langle t_0, r_0\rangle, \langle t_1, r_1\rangle\rangle.\texttt{N}\langle f\ x, \texttt{force}(r_0), \texttt{force}(r_1)\rangle)$$

The translation of `map f t` is given in figure 4.

(13)

$$\|\texttt{map f t}\| = 2 \oplus (f_c \otimes t_c) \oplus 1 \oplus_c$$
$$\texttt{rec}(t_p, \texttt{E} \mapsto \langle 1, \texttt{E}\rangle,$$
$$\texttt{N} \mapsto \langle y, \langle t_0, r_0\rangle\langle t_1, r_1\rangle\rangle.\langle 2 \oplus (f_p\ x)_c \otimes r_{0c} \otimes r_{1c}, \texttt{N}\langle (f_p\ x)_p, r_{0p}, r_{1p}\rangle\rangle$$

The result is shown in equation 13.

We interpret trees as the number of `N` constructors and the maximum label.

$$[\![tree]\!] = \mathbb{Z} \times \mathbb{Z}$$
$$D_{\texttt{tree}} = \{*\} + \mathbb{Z} \times [\![\texttt{tree}]\!] \times [\![\texttt{tree}]\!]$$
$$size_{\texttt{tree}}(*) = (0, 0)$$
$$size_{\texttt{tree}}(x, (m_0, n_0), (m_1, n_1)) = (max(x, m_0, m_1), 1 + n_0 + n_1)$$

FIGURE 5. Complexity translation of `map f t`

$$2 \oplus (f_c \otimes t_c) \oplus_c \texttt{rec}(t_p, \texttt{E} \mapsto 1 \oplus_c \|\texttt{E}\|,$$

$$\texttt{N} \mapsto \langle y, \langle t_0, r_0\rangle\langle t_1, r_1\rangle\rangle 1 \oplus_c \|\texttt{N}\langle f\ x, \texttt{force}(r_0)\texttt{force}(r_1)\rangle\|)$$

$$1 \oplus_c \|E\| = 1 \oplus \langle 0, E\rangle = \langle 1, E\rangle$$

$$\|\texttt{N}\langle f\ x, \texttt{force}(r_0)\texttt{force}(r_1)\rangle\| =$$

$$\langle \|\langle f\ x, \texttt{force}(r_0)\texttt{force}(r_1)\rangle\|_c, \texttt{N}\|\langle f\ x, \texttt{force}(r_0)\texttt{force}(r_1)\rangle\|_p\rangle$$

$$\|\langle f\ x, \texttt{force}(r_0)\texttt{force}(r_1)\rangle\| =$$

$$\langle \|f\ x\|_c \otimes \|\texttt{force}(r_0)\|_c \otimes \|\texttt{force}(r_1)\|_c, \langle \|f\ x\|_p, \|\texttt{force}(r_0)\|_p, \|\texttt{force}(r_1)\|_p\rangle\rangle$$

$$\|f\ x\| = 1 \oplus \|f\|_c \otimes \|x\|_c \oplus_c \|f\|_p \|x\|_p$$

$$= 1 \oplus \langle 0, f\rangle_c \otimes \langle 0, x\rangle_c \oplus_c \langle 0, f\rangle_p \langle 0, x\rangle_p$$

$$= 1 \oplus_c (f_p\ x)$$

$$= \langle 1 \oplus (f_p\ x)_c, (f_p\ x)_p\rangle$$

$$\|\texttt{force}(r_0)\| = \|r_0\|_c \oplus_c \|r_0\|_p$$

$$= \langle 0, r_0\rangle \oplus_c \langle 0, r_0\rangle_p$$

$$= \langle 0 + r_{0c}, r_{0p}\rangle$$

$$= r_0$$

$$\|\texttt{force}(r_1)\| = \|r_1\|_c \oplus_c \|r_1\|_p$$

$$= \langle 0, r_1\rangle \oplus_c \langle 0, r_1\rangle_p$$

$$= \langle 0 + r_{1c}, r_{1p}\rangle$$

$$= r_1$$

$$\|\langle f\ x, \texttt{force}(r_0)\texttt{force}(r_1)\rangle\| =$$

$$= \langle 1 \oplus (f_p\ x)_c \otimes r_{0c} \otimes r_{1c}, \langle (f_p\ x)_p, r_{0p}, r_{1p}\rangle\rangle$$

$$\|\texttt{N}\langle f\ x, \texttt{force}(r_0)\texttt{force}(r_1)\rangle\| =$$

$$= \langle 1 \oplus (f_p\ x)_c \otimes r_{0c} \otimes r_{1c}, \texttt{N}\langle (f_p\ x)_p, r_{0p}, r_{1p}\rangle\rangle$$

$$\|\texttt{map f t}\| =$$

$$2 \oplus (f_c \otimes t_c) \oplus 1 \oplus_c \texttt{rec}(t_p, \texttt{E} \mapsto \langle 1, \texttt{E}\rangle,$$

$$\texttt{N} \mapsto \langle y, \langle t_0, r_0\rangle\langle t_1, r_1\rangle\rangle.\langle 2 \oplus (f_p\ x)_c \otimes r_{0c} \otimes r_{1c}, \texttt{N}\langle (f_p\ x)_p, r_{0p}, r_{1p}\rangle\rangle$$

FIGURE 6. Deriviation of interpretation of `map f t`

CHAPTER 5

Mutual Recurrence

1. Motivation

The interpretation of a recursive function can be seperated into a recurrence for the cost and a recurrence for the potential. The recurrence for the cost depends on the recurrence for the potential. However, the recurrence for the potential does not depend on the cost. We prove this by designing a pure potential translation. The pure potential translation is identical to the complexity translation except that it does not keep track of the cost.

We then show by logical relations that the potential of the complexity translation is related to the pure potential relation.

2. Pure Potential Translation

Our pure potential translation is defined below. The translation of an expression is essientially the expression itself, without suspensions.

$$|x| = x$$

$$|\langle\rangle| = \langle\rangle$$

$$|\langle e_0, e_1\rangle| = \langle |e_0|, |e_1|\rangle$$

$$|\mathtt{split}(e_0, x_0.x_1.e_1)| = |e_1|[\pi_0|e_0|/x_0, \pi_0|e_0|/x_1]$$

$$|\lambda x.e| = \lambda x.|e|$$

$$|e_0\ e_1| = |e_0|\ |e_1|$$

$$|delay(e)| = |e|$$

$$|force(e)| = |e|$$

$$|C_i^\delta e| = C_i^\delta |e|$$

$$|rec^{\delta}(e, \overline{C \mapsto x.e_C})| = rec^{\delta}(|e|, \overline{C \mapsto x.|e_C|})$$

$$|map^{\phi}(x.v_0, v_1)| = map^{|\phi|}(x.|v_0|, |v_1|)$$

$$|let(e_0, x.e_1)| = |e_1|[|e_0|/x]$$

3. Logical Relation

We define our logical relation below.

$$E \sim_{\texttt{Unit}} E' \text{always}$$

$$E \sim_{\tau_0 \times \tau_1} E' \Leftrightarrow \forall k.\langle k, \pi_0 E_p\rangle \sim_{\tau_0} \pi_0 E', \forall k.\langle k, \pi_1 E_p\rangle \sim_{\tau_1} \pi_1 E'$$

$$E \sim_{\texttt{susp}\ \tau} E' \Leftrightarrow E_p \sim_{\tau} E'$$

$$E \sim_{\sigma \to \tau} E' \Leftrightarrow \forall E_0 \sim_{\sigma} E_0'.E_p E_{0p} \sim_{\tau} E' E_0'$$

$$E \sim_{\delta} E' \Leftrightarrow \exists k, k', C, V, V'.V \sim_{\phi[\delta]} V', E \downarrow \langle k, CV_p\rangle, E' \downarrow CV'$$

The relation is defined on closed terms, but we extend it to open terms. Let Θ and Θ' be any substitutions such that $\forall x : \|\tau\|, \forall k, \langle k, \Theta(x)\rangle \sim_{\tau} \Theta'(x)$. If $E\ \Theta \sim_{\tau} E'\ \Theta'$, then $E \sim_{\tau} E'$.

4. Proof

We require some lemmas.

The first states we can always ignore the cost of related terms.

Lemma 5.1 (Ignore Cost).

$$E \sim_{\tau} E' \Leftrightarrow \forall k, \langle k, E_p\rangle \sim_{\tau} E'$$

Proof. We proceed by induction on type.

Case $E \sim_{\texttt{Unit}} E'$. Then $\forall k, \langle k, E_p\rangle \sim_{\texttt{Unit}} E'$ by definition.

Case $E \sim_{\tau_0 \times \tau_1} E'$. By definition for $i \in 0, 1, \forall k_i, \langle k_i, \pi_i E_p\rangle \sim_{\tau_i} \pi_i E'$. Let k be some cost. Then $\langle k, E_p\rangle \sim_{\tau_0 \times \tau_1} E'$ by definition.

Case $E \sim_{\texttt{susp}\ \tau} E'$. By definition $E_p \sim_{\tau} E'$. Let k be some cost. Then $\langle k, E_p\rangle \sim_{\texttt{susp}\ \tau} E'$.

Case $E \sim_{\sigma\to\tau} E'$. Let E_0, E_0' by some complexity language terms such that $E_0 \sim_\sigma E_0'$. Let k be some cost. Then, $E_p\ E_0 \sim_\tau E'\ E_0'$. So $\langle k, E_p\rangle \sim_{\sigma\to\tau} E'$.

Case $E \sim_\delta E'$. Then by definition there exists costs k and k', a constructor C, and complexity language values V and V' such that $V \sim_{\Phi[\delta]} V', E \downarrow \langle k, CV_p\rangle$, and $E' \downarrow CV'$. Since $E \downarrow \langle k, CV_p\rangle$, we know $\forall k_0, \exists k_0'. \langle k_0, E_p\rangle \downarrow \langle k_0', CV_p\rangle$. So by definition we have $\forall k_0, \langle k_0, E_p\rangle \sim_\Phi E'$. □

The next lemma states that if two terms step to related terms, then those terms are related.

Lemma 5.2 (Related Step Back).

$$E \to F, E' \to F', F \sim_\sigma F' \implies E \sim_\sigma E'$$

Proof. The proof proceeds by induction on type.

Case $\texttt{Unit}$. Trivial since $E \sim_{\texttt{Unit}} E'$ always.

Case δ. By definition $\exists C, U, U', k, k'$ such that $F \downarrow \langle k, CU_p\rangle, F' \downarrow CU', U \sim_{\phi[\delta]} U'$. Since $E \to F$ and $E' \to F'$, $E \downarrow \langle k, CU_p\rangle$ and $E' \downarrow CU'$. Therefore since $U \sim_{\phi[\delta]} U'$, we have $E \sim_\delta E'$.

Case $\sigma \to \tau$. Let $E_0 \sim_\sigma E_0'$. By definition, $F\ E_0 \sim_\tau F'\ E_0'$. Since $E \to F$ and $E' \to F'$, $E\ E_0 \to F\ E_0$ and $E'\ E_0' \to F'\ E_0'$. So by the induction hypothesis, $E\ E_0 \sim_\tau E'\ E_0'$. So by definition, $E \sim_{\sigma\to\tau} E'$.

Case $\tau_0 \times \tau_1$. Since $F \sim_{\tau_0\times\tau_1} F'$, for $i \in \{0,1\}$, $\forall k_i, \langle k_i, \pi_i F_p\rangle \sim_{\tau_i} \pi_i F'$, by definition. From $E \to F$, we get $\langle k_i, \pi_i E_p\rangle \to \langle k_i', \pi_i F_p\rangle$. From $E' \to F'$, we get $\pi_i E' \to \pi_i F'$. We can apply our induction hypothesis to get $\langle k_i, \pi_i E_p\rangle \sim_{\tau_i} \pi_i E'$. By 5.1, $\forall k_i, \langle k_i, \pi_i E_p\rangle \sim_{\tau_i} \pi_i E$. So by definition $E \sim_{\tau_0\times\tau_1} E'$.

Case $\texttt{susp}\ \tau$. Since $F \sim_{\texttt{susp}\ \tau} F'$, by definition $F_p \sim_\tau F'$. Since $E \to F$, $E_p \to F_p$. So by the induction hypothesis, since $E_p \to F_p, E' \to F', F_p \sim_\tau F'$, $E_p \sim_\tau E'$. So by definition $E \sim_{\texttt{susp}\ \tau} E'$.

□

The next lemma states that related terms step to related terms

LEMMA 5.3. *[Related Step]*

$$E \to F, E' \to F', E \sim_\sigma E' \implies F \sim_\sigma F'$$

PROOF. The proof is by induction on type.

Case `Unit`. $F \sim_{\texttt{Unit}} F'$ always.

Case δ. By definition, $E \sim_\delta E'$ implies $\exists C, V, V', k$ such that $E \downarrow \langle k, CV_p\rangle, E' \downarrow CV'$, $V \sim_{\phi[\delta]} V'$. Since $E \to F$, $F \downarrow \langle k, CV_p\rangle$; and since $E \to F'$, $F' \downarrow CV'$. By 5.1, $\langle k, V_p\rangle \sim_{\phi[\delta]} V'$. So because $F \downarrow \langle k, CV_p\rangle, F' \downarrow CV', \langle k, V_p\rangle \sim_{\phi[\delta]} V'$, we can apply our induction hypothesis to get $F \sim_\delta F'$.

Case $\tau_0 \times \tau_1$. By definition $E \sim_{\tau_0\times\tau_1} \implies \forall i \in \{0,1\}, \forall k, \langle k_i, \pi_i E_p\rangle \sim_{\tau_i} \pi_i E'$. Fix some k_i. Since $E \to F$, $\langle k_i, \pi_i E_p\rangle \to \langle k_i, \pi_i F_p\rangle$. Since $E' \to F'$, $\pi_i E' \to \pi_i F'$. From $\langle k_i, \pi_i E_p\rangle \to \langle k_i, \pi_i F_p\rangle, \langle k_i, \pi_i E_p\rangle \sim_{\tau_i} \pi_i E'$, the induction hypothesis tells us $\langle k_i, \pi_i F_p\rangle \sim_{\tau_i} \pi_i F'$. So by definition $F \sim_{\tau_0\times\tau_1} F'$.

Case `susp` τ. By definition $E \sim_{\texttt{susp}\ \tau} E' \implies E_p \sim_\tau E'$. Since $E \to F$, $E_p \to F_p$. From $E_p \to F_p, E' \to F', E_p \sim_\tau E'$, the induction hypothesis gives us $F_p \sim_\tau F'$. So by definition $F \sim_{\texttt{susp}\ \tau} F'$.

Case $\sigma \to \tau$. Let $E_0 \sim_\sigma E_0'$. By definition, $E\ E_0 \sim_\tau E'\ E_0'$. Since $E \to F$, $E\ E_0 \to F\ E_0$. Since $E' \to F'$, $E'\ E_0' \to F'\ E_0'$. From $E\ E_0 \to F\ E_0, E'\ E_0' \to F'\ E_0'$, $E\ E_0 \sim_\tau E'\ E_0'$, the induction hypothesis tells us $F\ E_0 \sim_\tau F'\ E_0'$. So by definition $F \sim_{\sigma\to\tau} F'$. □

The next lemma states that if the arguments to *map* are related, then *map* preserves the relatedness.

LEMMA 5.4. *[Related Map]*

$$E \sim_{\tau_1} E', E_0 \sim_{\tau_0} E_0' \implies \forall k. \langle k, map^\Phi(x, E_p, E_{0p})\rangle \sim_{\Phi[\tau_1]} map^\Phi(x, E', E_0')$$

PROOF. The proof proceeds by induction on type.

Recall the definition of the *map* macro.

$$map^t(x.E, E_0) = E[E_0/x]$$

$$map^{T}(x.E, E_0) = E_0$$

$$map^{\Phi_0\times\Phi_1}(x.E, E_0) = \langle map^{\Phi_0}(x.E, \pi_0 E_0), map^{\Phi_1}(x.E, \pi_1 E_0\rangle$$

$$map^{T\to\Phi}(x.E, E_0) = \lambda y.map^{\Phi}(x.E, E_0\ y)$$

Case $\Phi = t$. Then $map^{t}(x.E_p, E_{0p}) = E_p[E_{0p}/x]$ and $map^{t}(x.E', E_0') = E'[E_0'/x]$. Let k be some cost. By 5.1, $E \sim_{\tau_1} E'$ implies $\langle k, E_p\rangle \sim_{\tau_1} E'$. Since $\langle k, E_p\rangle \sim_{\tau_1} E'$ and $E_0 \sim_{\tau_0} E_0'$, $\langle k, E_p\rangle[E_{0p}/x] \sim_{\phi[\tau_0]} E'[E_0'/x]$. So $\forall k, \langle k, map^{t}(x.E_p, E_{0p})\rangle \sim_{\Phi[\tau_1]} map^{t}(x.E', E_0')$.

Case $\Phi = T$. Then $map^{T}(x.E_p, E_{0p}) = E_{0p}$ and $map^{T}(x.E', E_0') = E_0'$. By 5.1 $\forall k, \langle k, E_{0p}\rangle \sim_{\tau_0} E_0'$. So $\forall k, \langle k, map^{T}(x.E_p, E_{0p})\rangle \sim_{\Phi[\tau_1]} map^{T}(x.E', E_0')$.

Case $\Phi = \Phi_0 \times \Phi_1$. Then
$map^{\Phi_0\times\Phi_1}(x.E_p, E_{0p}) = \langle map^{\Phi_0}(x.E_p, \pi_0 E_{0p}), map^{\Phi_1}(x.E_p, \pi_1 E_{0p})\rangle$.
Similarly $map^{\Phi_0\times\Phi_1}(x.E', E_0') = \langle map^{\Phi_0}(x.E', \pi_0 E_0'), map^{\Phi_1}(x.E', \pi_1 E_0')\rangle$.
By definition, $\forall k, \langle k, \pi_0 E_{0p}\rangle \sim_{\Phi_0[\tau_0]} \pi_0 E_0'$.
By the induction hypothesis, $\forall k, \langle k, map^{\Phi_0}(x.E_p, \pi_0 E_{0p}) \sim_{\Phi_0[\tau_1]} map^{\Phi_0[\tau_1]}(x.E', E_0')$.
By definition, $\forall k, \langle k, \pi_1 E_{0p}\rangle \sim_{\Phi_1[\tau_0]} \pi_1 E_0'$.
By the induction hypothesis, $\forall k, \langle k, map^{\Phi_1}(x.E_p, \pi_1 E_{0p}) \sim_{\Phi_1[\tau_1]} map^{\Phi_1[\tau_1]}(x.E', E_0')$.
So by definition,

$$\forall k, \langle k, \langle map^{\Phi_0}(x.E_p, \pi_0 E_{0p}), map^{\Phi_1}(x.E_p, \pi_1 E_{0p})\rangle\rangle \sim_{\Phi[\tau_1]}$$
$$\langle\langle map^{\Phi_0[\tau_1]}(x.E', E_0'), map^{\Phi_1[\tau_1]}(x.E', E_0')\rangle\rangle$$

Case $T \to \Phi$. Then $map^{T\to\Phi}(x.E_p, E_{0p}) = \lambda y.map^{\Phi}(x.E_p, E_{0p}\ y)$
and $map^{T\to\Phi}(x.E', E_0') = \lambda y.map^{\Phi}(x.E', E_0'\ y)$. Let $E_1 : T$. Then
$\lambda y.map^{\Phi}(x.E_p, E_{0p}\ y)\ E_1 \to map^{\Phi}(x.E_p, E_{0p}\ E_1)$. Similarly, $\lambda y.map^{\Phi}(x.E', E_0'\ y)\ E_1' \to map^{\Phi}(x.E', E_0'\ E_1')$. Since $E_0 \sim E_0'$ and $E_1 \sim E_1'$, we have $E_{0p}\ E_1 \sim E_0'\ E_1'$. So by our induction hypothesis, $map^{\Phi}(x.E_p, E_{0p}\ E_1) \sim map^{\Phi}(x.E', E_0'\ E_1')$. So by 5.2, $\lambda y.map^{\Phi}(x.E_p, E_{0p}\ y)\ E_1 \sim \lambda y.map^{\Phi}(x.E', E_0'\ y)\ E_1'$. So by definition,
$\lambda y.map^{\Phi}(x.E_p, E_{0p}\ y) \sim \lambda y.map^{\Phi}(x.E', E_0'\ y)$.
So $map^{T\to\Phi}(x.E_p, E_{0p}) \sim map^{T\to\Phi}(x.E', E_0')$. $\square$

Our last lemma is about the relatedness of *rec* terms.

LEMMA 5.5 (Related Rec).

$$E \sim_\delta E', \forall C, E_C \sim_\tau E'_C \implies rec(E_p, \overline{C \mapsto x.E_c}) \sim_\tau rec(E', \overline{C \mapsto x.E'_c})$$

PROOF. Recall the rule for evaluating *rec* in the complexity language:

$$\frac{E \downarrow CV_0 \qquad map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E_C})\rangle, V_0) \downarrow V_1 \qquad E_C[V_1/x] \downarrow V}{rec(E, \overline{C \mapsto x.E_C}) \downarrow V}$$

By definition of $\sim_\delta$, $\exists k, C, V_0, V'_0$ such that $E \downarrow \langle k, CV_{0p}\rangle, E' \downarrow CV'_0$, and $V_0 \sim_\delta V'_0$. Our proof proceeds by induction on the number of constructors in CV_{0p}. If $\Phi = T$, then $map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E_C})\rangle, V_{0p}) = \langle y, rec(y, \overline{C \mapsto x.E_C})\rangle[V_{0p}/y] = \langle V_{0p}, rec(V_{0p}, \overline{C \mapsto x.E_C})\rangle$. Similarly for the pure potential, $map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E'_C})\rangle, V'_{0p}) = \langle y, rec(y, \overline{C \mapsto x.E'_C})\rangle[V'_0/y] = \langle V'_0, rec(V'_0, \overline{C \mapsto x.E'_C})\rangle$. By the induction hypothesis, $rec(V_{0p}, \overline{C \mapsto x.E_C}) \sim_\tau rec(V'_0, \overline{C \mapsto x.E'_C})$. By definition of $\sim_{\mathtt{susp}\ \tau}$, for any k, $\langle k, rec(V_{0p}, \overline{C \mapsto x.E_C})\rangle \sim_{\mathtt{susp}\ \tau} rec(V'_0, \overline{C \mapsto x.E'_C})$. So by definition of $\sim_{\tau_0 \times \tau_1}$, $\langle 0, \langle V_{0p}, rec(V_{0p}, C \mapsto x.E_C)\rangle\rangle \sim_{\phi[\delta \times \mathtt{susp}\ \tau]} \langle V'_0, rec(V'_0, \overline{C \mapsto x.E'_C})\rangle$. So by 5.4, $\forall k. \langle k, map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E_C})\rangle, V_{0p}) \sim_{\phi[\delta \times \mathtt{susp}\ \tau]} map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E'_C})\rangle, V'_0)$. Let $\langle 0, map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E_C})\rangle, V_{0p}) \downarrow V_1$. Let $map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E'_C})\rangle, V'_0) \downarrow V'_1$. By 5.3, $V_1 \sim_{\phi[\delta \times \mathtt{susp}\ \tau]} V'_1$.

If $\Phi = t$, then $map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E_C})\rangle, V_{0p}) = V_{0p}$. Similarly, $map^\Phi(y, \langle y, rec(y, \overline{C \mapsto x.E'_C})\rangle V'_0) = V'_0$. So in this case $V_0 = V_1$ and $V'_0 = V'_1$. We have already established $V_0 \sim_\tau V'_0$.

So in both cases $V_1 \sim_{\phi[\delta \times \mathtt{susp}\ \tau]} V'_1$.

By definition of the relation $E_C[V_{1p}/x] \sim_\tau E'_C[V'_1/x]$. Let $E_C[V_{1p}/x] \downarrow V_2$ and $E'_C[V'_1/x] \downarrow V'_2$. By 5.3, $V_2 \sim_\tau V'_2$. So by 5.2, $rec(E_p, \overline{C \mapsto x.E_C}) \sim_\tau rec(E', \overline{C \mapsto x.E'_C})$.

□

Our theorem is that for all well-typed terms in the source language, the complexity translation of the term is related to the pure potential translation of that term.

THEOREM 5.6 (Distinct Recurrence).

$$\gamma \vdash e : \tau \implies \|e\| \sim_\tau |e|$$

PROOF. Our proof is by induction on the typing derivation $\gamma \vdash e : \tau$.

Case $\dfrac{}{\gamma, x : \sigma \vdash x : \sigma}$. Then by definition of the logical relation, $\forall k, \langle k, \Theta(x)\rangle \sim_\sigma \Theta'(x)$. Since $\|x\| = \langle 0, x\rangle$ and $|x| = x$, we have $\langle 0, x\rangle \sim_\sigma x$.

Case $\dfrac{}{\gamma \vdash e : Unit}$. By definition, $\|e\| \sim_{\texttt{Unit}} |e|$ always.

Case $\dfrac{\gamma \vdash e_0 : \tau_0 \qquad \gamma \vdash e_1 : \tau_1}{\gamma \vdash \langle e_0, e_1\rangle : \tau_0 \times \tau_1}$ By the induction hypothesis, $\|e_0\| \sim_{\tau_0} |e_0|$ and $\|e_1\| \sim_{\tau_1} |e_1|$. By 5.1, $\forall k, \langle k, \|e_0\|_p\rangle \sim_{\tau_0} |e_0|$ and $\forall k, \langle k, \|e_1\|_p\rangle \sim_{\tau_1} |e_1|$. So by definition, $\|\langle e_0, e_1\rangle\| \sim_{\tau_0\times\tau_1} |\langle e_0, e_1\rangle|$.

Case $\dfrac{\gamma \vdash e_0 : \tau_0 \times \tau_1 \qquad \gamma, x_0 : \tau_0, x_1 : \tau_1 \vdash e_1 : \tau}{\gamma \vdash split(e_0, x_0.x_1.e_1) : \tau}$ By the induction hypothesis, $\|e_0\| \sim_{\tau_0\times\tau_1} |e_0|$ and $\|e_1\| \sim_\tau |e_1|$. From $\|e_0\| \sim_{\tau_0\times\tau_1} |e_0|$ it follows by definition that $\forall k, \langle k, \pi_0\|e_0\|_p\rangle \sim_{\tau_0} \pi_0|e_0|$ and $\forall k, \langle k, \pi_1\|e_1\|_p\rangle \sim_{\tau_1} \pi_1|e_1|$. The complexity translation is $\|split(e_0, x_0.x_1.e_1)\| = \|e_0\|_c +_c \|e_1\|[\pi_0\|e_0\|_p/x_0, \pi_1\|e_1\|_p/x_1]$. The pure potential translation is $|split(e_0, x_0.x_1.e_1)| = |e_1|[\pi_0|e_0|/x_0, \pi_1|e_0|/x_1]$. By 5.1, it suffices to show $\|e_1\|[\pi_0\|e_0\|_p/x_0, \pi_1\|e_1\|_p/x_1] \sim_\tau |e_1|[\pi_0|e_0|/x_0, \pi_1|e_0|/x_1]$ By definition of the relation, it suffices to show $\|e_1\| \sim_\tau |e_1|$, $\forall k, \langle k, \pi_0\|e_0\|_p\rangle \sim_{\tau_0} \pi_0|e_0|$, and $\forall k$, $\langle k, \pi_1\|e_0\|_p\rangle \sim_{\tau_1} \pi_1|e_0|$. Since we have already establed all three conditions, we have $\|split(e_0, x_0.x_1.e_1)\| \sim_\tau |split(e_0, x_0.x_1.e_1)|$.

Case $\dfrac{\gamma, x : \sigma \vdash e : \tau}{\gamma \vdash \lambda x.e : \sigma \to \tau}$ By the induction hypothesis $\|e\| \sim_\tau |e|$. The complexity translation is $\|\lambda x.e\| = \langle 0, \lambda x.\|e\|\rangle$. The pure potential translation is $|\lambda x.e| = \lambda x.|e|$. Let $E_0 : \|\sigma\|$ and $E_0' : |\sigma|$ be complexity language terms such that $E_0 \sim_\sigma E_0'$. Then $\langle 0, \lambda x.\|e\|\rangle\ E_0 \to \langle 0 + E_{0c}, \|e\|[x \mapsto E_0]\rangle$ and $\lambda x.|e|\ E_0' \to |e|[x \mapsto E_0']$. Since $\|e\| \sim_\tau |e|$ and $E_0 \sim_\sigma E_0'$, $\|e\|[x \mapsto E_0] \sim_\tau |e|[x \mapsto E_0']$. By 5.2, $\langle 0, \lambda x.\|e\|\rangle\ E_0 \sim_\tau (\lambda x.|e|)\ E_0'$. So by definition $\langle 0, \lambda x.\|e\|\rangle \sim_{\sigma\to\tau} \lambda x.|e|$. So $\|\lambda x.e\| \sim_{\sigma\to\tau} |\lambda x.e|$.

Case $\dfrac{\gamma \vdash e_0 : \sigma \to \tau \qquad \gamma \vdash e_1 : \sigma}{\gamma \vdash e_0\ e_1 : \tau}$ The complexity translation is $\|e_0\ e_1\| = (1 + \|e_0\|_c + \|e_1\|_c) +_c \|e_0\|_p\|e_1\|_p$. The pure potential translation is $|e_0\ e_1| = |e_0||e_1|$. By 5.1, it suffices to show $\|e_0\|_p\|e_1\|_p \sim_\tau |e_0||e_1|$. By the induction hypothesis, $\|e_0\| \sim_{\sigma\to\tau} |e_0|$ and $\|e_1\| \sim_\sigma |e_1|$. By definition, $\|e_0\|_p\|e_1\|_p \sim_\tau |e_0||e_1|$.

Case $\dfrac{\gamma \vdash e : \tau}{\gamma \vdash delay(e) : susp\ \tau}$ By the induction hypothesis $\|e\| \sim_\tau |e|$. So $\langle 0, \|e\|\rangle \sim_{\mathtt{susp}\ \tau}$ $|e|$. The complexity translation is $\|delay(e)\| = \langle 0, \|e\|\rangle$. The pure potential translatio is $|delay(e)| = |e|$. So $\|delay(e)\| \sim_{\mathtt{susp}\ \tau} |delay(e)|$.

Case $\dfrac{\gamma \vdash e : susp\ \tau}{\gamma \vdash force(e) : \tau}$ By the induction hypothesis $\|e\| \sim_{\mathtt{susp}\ \tau} |e|$. So by definition of the relation at $\mathtt{susp}$ type, $\|e\|_p \sim_\tau |e|$. By 5.1, $\forall k, \langle k, \|e\|_{pp}\rangle \sim_\tau |e|$. The complexity translation is $\|force(e)\| = \|e\|_c +_c \|e\|_p$. The pure potential translation is $|force(e)| = |e|$. So $\|e\|_c +_c \|e\|_p \sim_\tau |e|$. So $\|force(e)\| \sim_\tau |force(e)|$.

Case $\dfrac{\gamma \vdash e_0 : \sigma \qquad \gamma, x : \sigma \vdash e_1 : \tau}{\gamma \vdash let(e_0, x.e_1) : \tau}$ By the induction hypothesis $\|e_0\| \sim_\sigma |e_0|$ and $\|e_1\| \sim_\tau |e_1|$. So $\|e_1\|[\|e_0\|_p/x] \sim_\tau |e_1|[|e_0|/x]$. By 5.1, $\forall k, \langle k, \|e_1\|_p[\|e_0\|_p/x]\rangle \sim_\tau |e_1|[|e_0|/x]$. The complexity translation is $\|let(e_0, x.e_1)\| = \|e_0\|_c +_c \|e_1\|[\|e_0\|_p/x]$. The pure potential translation is $|let(e_0, x.e_1)| = |e_1|[|e_0|/x]$. So $\|let(e_0, x.e_1)\| \sim_\tau |let(e_0, x.e_1)|$.

Case $\dfrac{\gamma, x : \tau_0 \vdash v_1 : \tau_1 \qquad \gamma \vdash v_0 : \phi[\tau_0]}{\gamma \vdash map^\phi(x.v_1, v_0) : \phi[\tau_1]}$ By the induction hypothesis $\|v_1\| \sim_{\tau_1} |v_1|$ and $\|v_0\| \sim_{\phi[\tau_0]} |v_0|$. By 5.4, $\forall k, \langle k, map^\Phi(x.\|v_1\|_p, \|v_0\|_p)\rangle \sim_{\phi[\tau_1]} map^\Phi(x.|v_1|, |v_0|)$. The complexity translation is $\|map^\phi(x.v_1, v_0)\| = \langle 0, map^\Phi(x.\|v_0\|_p, \|v_1\|_p)\rangle$. The pure potential translation is $|map^\phi(x.v_1, v_0| = map^\Phi(x, |v_0|, |v_1|)$. So we have $\|map^\phi(x.v_1, v_0)\| \sim_{\phi[\tau_1]} |map^\phi(x.v_1, v_0|$.

Case $\dfrac{\gamma \vdash e_0 : \delta \qquad \forall C(\gamma, x : \phi_C[\delta \times susp\ \tau] \vdash e_c : \tau}{\gamma \vdash rec^\delta(e_0, \overline{C \mapsto x.e_C}) : \tau}$ By the induction hypothesis $\|e_0\| \sim_\delta |e_0|$ and $\forall C, \|e_c\| \sim_\tau |e_c|$. By 5.1, $\forall k, \langle k \|e_C\| \sim_\tau |e_c$, so $1 +_c \|e_C\| \sim_\tau |e_c|$. So by 5.5, $rec(\|e_0\|_p, \overline{C \mapsto x.1 +_c \|e_C\|}) \sim_\tau rec(|e_0|, \overline{C \mapsto x.1 +_c |e_C|})$. So by 5.1, $\|e_0\|_c +_c rec(\|e_0\|_p, \overline{C \mapsto x.1 +_c \|e_C\|}) \sim_\tau rec(|e_0|, \overline{C \mapsto x.1 +_c |e_C|})$

Case $\dfrac{\gamma \vdash e : \phi[\delta]}{\gamma \vdash Ce : \delta}$ By the induction hypothesis, $\|e\| \sim_{\phi[\delta]} |e|$. There exists V, V' such that $\|e\| \downarrow V$ and $|e| \downarrow V'$. By 5.3 $V \sim_{\phi[\delta]} V'$. Since $\|e\| \downarrow V$, $\langle k, C\ \|e\|\rangle \downarrow \langle k, C\ V_p$. Similarly, since $|e| \downarrow V'$, $C|e| \downarrow CV'$. So by definition we have $\langle k, C\ \|e\|\rangle \sim_\delta C|e|$. The complexity translation is $\|Ce\| = \langle \|e\|, C\|e\|_p\rangle$. The pure potential translation is $|Ce| = C|e|$. Therefore by 5.1, $\|Ce\| \sim_\delta |Ce|$. □

CHAPTER 6

Conclusions and Future Work

This is the conclusion.

1. Future Work

One of the criticisms leveed against traditional complexity analysis is there is no formal connection between the source language program and the recurrence for its cost. Although the method from Danner et al. [2015] does provide the formal connection, the translation from source language to complexity language and subsequent interpretation is tedious and prone to errors. Automation of the source to complexity translation and interpretation could easily be automated. Both transformations are simply forms of cross compilation between different languages. This leaves only the process of obtaining closed form solutions of recurrences initially obtained from the interpretation. Solving the recurrences requires a significant amount of cleverness. For simpler recurrences such as `list map`, the recurrences are straightforward to solve. However for more complex functions, the recurrences may be very difficult to solve. Consider a function that calculates the nth term of the Fibonacci sequence. Under an interpretation of natural numbers as the number of successor constructors, the potential of this function is the nth Fibonacci number. The closed form solution to the potential recurrence is

$$\frac{(1+\sqrt{5})^n + (1-\sqrt{5})^n}{2^n\sqrt{5}}$$

Obtaining this solution is not at all obvious.

Bibliography

Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

Norman Danner and James S. Royer. Adventures in time and space. *Logical Methods in Computer Science*, 3(1), 2007. doi: 10.2168/LMCS-3(1:9)2007.

Norman Danner and James S. Royer. Two algorithms in search of a type-system. *Theory of Computing Systems*, 45(4):787–821, 2009. doi: 10.1007/s00224-009-9181-y. URL `http://dx.doi.org/10.1007/s00224-009-9181-y`.

Norman Danner, Jennifer Paykin, and James S. Royer. A static cost analysis for a higher-order language. In *Proceedings of the 7th workshop on Programming languages meets program verification*, pages 25–34. ACM Press, 2013. doi: 10.1145/2428116.2428123. URL `http://arxiv.org/abs/1206.3523`.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *In Proceedings of the International Conference on Functional Programming*, volume abs/1506.01949, 2015. URL `http://arxiv.org/abs/1506.01949`.

Professor Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576.