Wesleyan ♦ University

COST SEMANTICS FOR PLOTKIN'S PCF

By

Theodore S. Kim

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 **May, 2016**

Recurses! Called again.
-Anonymous

Acknowledgements

First and foremost, I wish to thank my advisor, Norman Danner, for guiding me the past four years. Beginning with Data Structures and Special Topics and all the way through summer research, he has spent hours pointing me in the right direction, clarifying confusing proofs and notations, and questioning the validity of my own proofs. I have never met a more passionate person about compute science and am thankful for his guidance and the opportunity to work with him.

I would also like to thank my reading committee, Dan Licata and Olivier Hermant, for their time in evaluating this work and lecturing some of my favorite classes at Wesleyan. Special thanks to Dan for giving me key insights into proofs regarding general recursion and generally being an awesome guy.

I sincerely thank my officemates Ben Hudson and Justin Raymond for providing hours of entertainment and a source of knowledge. I could always count on you guys to help me grind through some of the tougher proofs and keep me sane.

Lastly, I thank my parents and my siblings for everything they have done for me.

Abstract

We develop a formalism for extracting recurrences from programs written in a variant of Plotkin's PCF, a typed higher-order functional language that integrates functions and natural numbers through general recursion. The complexity of an expression is defined as a pair consisting of a cost and a potential. The former is the explicit cost of evaluating the expression to a value while the latter is the cost of future use of the value. The extraction function is a language-to-language translation that makes evaluation costs explicit by mapping source language expressions to complexity language expressions. We prove by a logical relations argument that recurrences extracted by the extraction function are upper bounds for evaluation cost. The proof makes use of bounded recursion, a key property of which is the principle of fixed point induction, which allows us to be able to reason about recursive computations.

Contents

CHAPTER 1

Introduction

A key component of understanding the efficiency of an algorithm in computer science is understanding the resources necessary to execute the algorithm, or the concept of complexity. Complexity can be thought of as a function which takes a program and an input to the program and returns the cost of evaluating the program with respect to the size of the input. The cost can be specified as the time the program takes to finish, the number of reduction steps the program takes to return a value, the amount of space needed for its computations, or other resources necessary for the program to evaluate. Programmers generally concern themselves with the running time of an algorithm; as while time itself is infinite, programmers and end-users are not.

Analyzing time complexity of an algorithm is crucial to both students studying computer science in academia as well as to programmers working in industry. Time complexity analysis can give reasonable upper bounds on the running times of algorithms and programs. This information can be useful to programmers who wish to deploy a faster program or students who wish to compare the running time of two algorithms or understand the complexity of their own programs. Automated complexity analysis would provide a powerful tool to programmers wishing to better understand the efficiency of their programs.

1. Complexity Analysis

The standard method for analyzing the complexity of a recursive functional program is to obtain a big-O bound, which drops coefficients and lower order terms, through the extraction of a recurrence that relates the function's running time and the size of the function's input, and solving that recurrence for a closed form. The goal of

```
let rec insert x xs =
match xs with
[] -> [x]
| y::ys -> if x < y then x :: y :: ys
           else y :: insert x ys
```

$$T(0) = 1$$
$$T(n+2) = 2 + T(n).$$

Figure 1: List insert and its recurrence

automated complexity analysis is to establish a formal relationship between a program and a recurrence that claims to describe its cost of evaluation. Take for example an executable program written in OCaml that adds an element to a list of sorted elements and its supposed recurrence given in Figure 1. There is no guarantee that the recurrence does in fact describe the program's evaluation cost. It requires some form of heuristics to derive the supposed recurrence from the program. We recognize that we might recursively call the insert function on the tail of the list. In the worst case, we will have to call insert as many times as the number of elements in the list, since the element x that we are trying to insert to the sorted list might be larger than all the other elements in the list. The problem becomes even more obtuse when we no longer assume that the less than comparison function takes constant time to evaluate. We might observe that the growth rate of the solution to this recurrence corresponds to the growth rate of this specific implementation, but there is no formalism to guarantee this always occurs. The goal of our automatic extraction process is to establish a formal connection between the source code and the recurrence that describes its evaluation cost.

To achieve automated complexity analysis, we define a source language which we use as a target for our cost analysis and write our program e that we wish to analyze. We assign costs to evaluation steps in the operational semantics of the source language. This allows us to specify the evaluation steps of which we want to explicitly keep track. Programmers are often interested in the number of recursive calls the

function makes or in the number of function applications. We define a complexity language that serves as a programming language for recurrences. We define an extraction function from the source language to the complexity language, where we write $\|e\|$ for the recurrence extracted from program e. This extraction function can be interpreted as a language to language translation which extracts the cost from the source language program and makes it explicit. We present a bounding theorem that guarantees the cost of evaluating the source language program is bounded by the cost returned from the translated complexity program. However, this process of automating the extract-and-solve method functional programs becomes further complicated when considering a language with higher-order functions, programmer-defined datatypes, lazy datatypes, general recursion, or any combination of these features. This work builds upon previous research that addresses the issue of analyzing complexities of programs written in a higher-order functional language with recursive datatypes to analyze complexities for programs written in a higher-order functional language extended with general recursion.

2. A Static Complexity Analysis for a Higher-order Language

A topic of significant interest, there is a wide literature on the topic of automatically constructing the resource bounds of programs from source code. One notion of recurrence for higher-order functions was developed in previous work by **?**. In order to compose recurrences the same way we compose functions in the source language, it is insufficient to use a complexity measure that only considers cost. Since an output of a function can be the input of another function, the extracted recurrence needs to provide information about the running time as well as the size of the output so that the "outer" recurrence may use the information. The two recurrences can be extracted as a single mutual recurrence, which given the size of an input, returns the pair of the running time (called the *cost*) and the size of the output (called the *potential*). While the first is the explicit cost of evaluating a program to a value, the latter can be interpreted as the cost of using that value later on. Therefore, a recurrence for a higher-order function

is a higher-order function which expresses the recurrence of the result in terms of a recurrence of the argument function.

? use this notion of recurrence for higher-order functions in a formalism which extracts costs of evaluating programs written in a higher-order functional language with recursive datatypes. The source language is akin to Gödel's System T augmented with integers, booleans, and integer lists. The integer lists datatype definition automatically provides for structural recursion through fold which returns the recursive call of a function applied to the tail of a list. The complexity language is in essence System T. The extraction function is almost the identity, but with two important changes.[1] A conditional expression is translated to the maximum of the complexity of both its branches, which ensures that the complexity of a conditional bounds the cost of any possible evaluation of that conditional. The fold constructor for recursive datatypes has a corresponding pfold constructor in the complexity language so that recursive definitions in the source language are translated to recurrences in the complexity language. The complexity constructor pfold branches on numbers that represent bounds on the size of the corresponding recursive argument in the source language. The extraction function from the source language to the complexity language is proven to be sound through the use of a logical relation, called the bounding relation, between source language and complexity expressions.

3. Contributions of this Thesis

While the formalism presented in **?** is limited to analyzing programs written with fold, or structural recursion, this thesis aims to build upon and widen the scope of the source language by introducing general recursion through the addition of the fix operator. Programs such as the Euclidean algorithm for greatest common divisor, first-order fast list reversal, merge sort for lists of natural numbers, and a program that returns the n-th Fibonacci number are all examples of programs naturally defined by

[1] In **?**, the translation is not a language-to-language translation. Instead, extraction is a denotation into a model of System T. However it is more convenient to think of the extraction as a translation into System T, especially with respect to **?**

general recursion, rather than structural recursion. Furthermore, programs that are previously defined by structural recursion can be defined in terms of general recursion, where the analysis should return similar recurrences as before.

We present a source language that is a variant of left-to-right evaluation call-by-name Plotkin's Programming Computable Functions, or Plotkin's PCF; a complexity language that is PCF extended with pairs and equational reasoning; and an extraction function from the source language to the complexity language in Chapter 2. A key difference to note in this framework compared to **?** is that the extension of the notion of size to type-level greater or equal to 1 is different. This difference has to do with the difference in evaluation strategies (call-by-value in **?**, call-by-name here), and we discuss it in more detail in Chapter 4.3. We try to consider programs that users may write in a call-by-name setting since users would not write programs in a call-by-name setting if they are significantly less efficient than if they were written in a call-by-value setting. Our framework correctly extract recurrences regardless of efficiency and can be used to prove that a program is better implemented in a call-by-value setting.

We present the proof of the bounding theorem that guarantees the recurrence returned from the extraction function correctly bounds the cost of evaluating the program in Chapter 3. In order to discuss bounding of possibly divergent expressions, we present the concept of bounded recursion. The idea is that a bounded recursive expression is logically and observationally equivalent to the recursive expression for up to a certain number of *unrollings*, after which the expression is divergent. A key property of bounded recursion is the principle of fixed point induction, which allows reasoning about a recursive computation by induction on the number of unrollings needed for the expression to reach a value. This allows us to reason about bounding of recursive expressions in the source and complexity languages in the proof of the bounding theorem for the fix operator. The proof of fixed point induction relies on compactness, which states that a fixed point expression only needs to be unrolled a finite number of times in an evaluation of a program that evaluates to a value and terminates. The key technical notion used in the proofs is a logical relation between the source and complexity languages

that modifies the bounding relation of **?** to be more like a typical logical relation. The main difference is that we reason about bounding of evaluation costs and potential at the base types and relegate to lower types at higher type.

Choosing the source language to be call-by-name rather than call-by-value allows us to reason about computation without reducing to values. In a call-by-value setting, we are forced to reason about computation by reducing to values even when we do not desire to. This creates issues with logical-relation-based reasoning about fixed point induction, which is needed to introduce general recursion to the framework. The new bounding relation also introduces issues to adding user-defined datatypes. The various issues that arise from the new bounding relation are further discussed in Chapter 4. We discuss in Chapter 5 related works in detail and future work to further extend the work presented in this thesis.

CHAPTER 2

Source Language, Complexity Language, and the Extraction Function

We define a source language in which we write our target program $e : \tau$ for our cost analysis, a corresponding complexity language in which we have complexity expressions $E : T$, and a extraction function from the source language program e to the complexity language program $\|e\| : \|\tau\|$. The extraction function returns a mutual recurrence that captures both the cost and potential components of a complexity. Since the languages are significantly different from **?**, we give full descriptions and motivations of the languages. We give several examples of programs that are naturally defined by general recursion rather than by structural recursion such as `fold` or `map`.

1. The Source Language

The source language in which we write our target program for our cost analysis is essentially a variant on call-by-name PCF, a typed functional language that integrates functions and natural numbers using general recursion, which allows for defining self-referential expressions. Compared to Gödel's System T, expressions defined in PCF are not guaranteed to terminate when evaluated; that is, the type system no longer guarantees termination. We define the expressions e and typing judgment $\gamma \vdash e : \tau$ in Figure 1. The language provides higher-order programming over natural numbers and lists of natural numbers. Natural numbers have type `nat` and are defined as either 0 or the successor of a natural number $S(n)$, where n is a natural number. Values are any of the following:

- 0 and $S(n)$ where n is a value.
- Any expressions of the form $\lambda x.e$

Types:

$$\tau ::= \mathsf{nat} \mid \tau \to \tau$$

Expressions:

$$\begin{aligned} v &::= \lambda x.e \mid 0 \mid S(v) \\ e &::= x \mid S(e) \mid \mathsf{case}\ (e,\ 0 \mapsto e \mid S(n) \mapsto e) \\ &\quad \mid \lambda x.e \mid e\ e \mid \mathsf{fix}\ x = e \end{aligned}$$

Typing: $\gamma \vdash e : \tau$

$$\frac{}{\gamma, x : \sigma \vdash x : \sigma} \qquad \frac{}{\gamma \vdash 0 : \mathsf{nat}}$$

$$\frac{\gamma \vdash e : \mathsf{nat}}{\gamma \vdash S(e) : \mathsf{nat}} \qquad \frac{\gamma \vdash e : \mathsf{nat} \qquad \gamma \vdash e_0 : \tau \qquad \gamma, n : \mathsf{nat} \vdash e_1 : \tau}{\gamma \vdash \mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) : \tau}$$

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

$$\frac{\gamma, x : \tau \vdash e : \tau}{\gamma \vdash \mathsf{fix}\ x = e : \tau}$$

Figure 1: Source language syntax and typing.

We have a case expression for casing on natural numbers.

$$\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)$$

The second match for the case expression allows for structural recursion, where n is the predecessor of $S(n)$. We give an example demonstrating an implementation of fold via fix in Section 4 for the case expression.

The key extension to previous work, the language also supports general recursion through the expression $\mathsf{fix}\ x = e$. The typing rule for fix reflects how the expression is self-referential. To prove that $\mathsf{fix}\ x = e$ has type τ, we need to assume that it does indeed have type τ by assigning the variable x standing for the self-referential expression type τ and confirm that the body e has type τ under the assumption.

The operational semantics defines a big-step call-by-name evaluation, defined in Figure 2. To evaluate an application $e_0\ e_1$, it is only necessary to evaluate the left-hand side expression e_0 down to an abstraction $\lambda x.e_0'$ before substituting e_1 for the variable x in the body e_0'. The operational semantics for $\mathsf{fix}\ x = e$ allows for self-reference

Operational semantics: $e \downarrow^m v$.

$$\frac{e \downarrow^m n}{S(e) \downarrow^m S(n)} \qquad \frac{e \downarrow^{m_0} 0 \qquad e_0 \downarrow^{m_1} v}{\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \downarrow^{m_0+m_1} v}$$

$$\frac{e \downarrow^{m_0} S(n_0) \qquad e_1[n_0/n] \downarrow^{m_1} v}{\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \downarrow^{m_0+m_1} v}$$

$$\frac{e_0 \downarrow^{m_0} \lambda x.e_0' \qquad e_0'[e_1/x] \downarrow^{m_1} v}{e_0\ e_1 \downarrow^{m_0+m_1} v} \qquad \frac{e[\mathsf{fix}\ x = e/x] \downarrow^m v}{\mathsf{fix}\ x = e \downarrow^{1+m} v}$$

Figure 2: Source language operational semantics.

by substituting the recursive expression itself for the variable x in the body e. This evaluation step is known as unwinding the recursion.

The cost semantics defines the relation $e \downarrow^m v$, which we write to mean the expression e evaluates to the value v in m steps. We define $\mathsf{cost}(e)$ as the number of steps m and $\mathsf{val}(e)$ as the value v. Our cost model charges only for the number of fix operators evaluated, or the number of times we unwind the recursion.

Substitutions are defined as usual:

DEFINITION 2.1. *We write θ for substitutions $v_1/x_1, \ldots, v_n/x_n$, and $\theta : \gamma$ to mean that Dom $\theta \subseteq$ Dom γ and $\emptyset \vdash \theta(x) : \gamma(x)$ for all $x \in$ Dom θ. We define the application of a substitution θ to an expression e as usual and denote it $e[\theta]$.*

LEMMA 2.2. *If x does not occur in θ, then $e[\theta, x/x][e_1/x] = e[\theta, e_1/x]$.*

DEFINITION 2.3 (Value Evaluation).

- *If $v \downarrow^m v'$, then $m \leq 0$ and $v = v'$.*
- *For all v, $v \downarrow^0 v$.*

2. The Complexity Language

The complexity language is essentially a variant on PCF extended with pairs. Costs as well as the sizes of expressions of type `nat` will be represented by values of type `nat` which range over natural number constants. Numerical binary operators such as $+$, $*$ and $-$ on expressions of type `nat` are part of the complexity language as well. The complexity language's syntax and typing are given in Figure 3. The case statement for

natural numbers has a corresponding `ifz` expression in the complexity language, which is a zero test. The key extension to the complexity language is its counterpart to the source language's fix expression, which plays the same role of allowing self-reference but for complexity expressions.

The complexity language also includes the maximum operation, $E_0 \vee E_1$, which returns the "greater" complexity expression between two complexity expressions. At type `nat`, the maximum of two natural numbers returns the larger natural number. At product type, we take the two maximums of the first projections and the second projections and return the pair with the maximum of each projection. At higher type, we take the pointwise maximum of the two functions. The reason for adding this maximum operation is discussed later in the translations for the two case expressions.

Since we are no longer concerned with the cost of evaluation in the complexity language, we remove the features that were included in the source language that were used to control evaluation costs. Instead, we present the equational semantics for the complexity language in Figure 4 that plays an analogous role to an operational semantics for the complexity language. The key concept behind equational reasoning is that equals can be replaced by equals in any context. For example, the sum $2 + 3 : \texttt{nat}$ can be replaced by the natural number constant $5 : \texttt{nat}$. In a more interesting example, given two equal expressions E_0 and E_1, we can substitute them into the same expression E for the same variable x and the two resulting expressions are equivalent. Thus we can create equational reasoning chains for expressions all the way down to values. We write $E \downarrow$ to mean that for a complexity expression E of type $\texttt{nat} \times \texttt{nat}$, there exists an equational reasoning chain starting from complexity expression E to some unique complexity value. Similary, we write $E \uparrow$ to mean that complexity expression E diverges and does not equal any complexity value.

Types:

$$T ::= \texttt{nat} \mid \texttt{unit} \mid T \times T \mid T \to T$$

Expressions:

$$\begin{aligned} V ::= {} & x \mid 0 \mid 1 \mid \ldots \\ & \mid \langle\,\rangle \mid \langle V, V\rangle \mid \lambda x.E \\ E ::= {} & x \mid 0 \mid 1 \mid \ldots \\ & \mid E + E \mid E - E \mid E * E \mid \texttt{ifz } E \texttt{ then } E \texttt{ else } E \\ & \mid \lambda x.E \mid E\ E \mid \langle\,\rangle \mid \langle E, E\rangle \\ & \mid \pi_0 E \mid \pi_1 E \mid \texttt{fix } x = E \mid E \vee E \end{aligned}$$

Typing: $\Gamma \vdash E : T$

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{}{\Gamma \vdash 0 : \texttt{nat}} \qquad \frac{}{\Gamma \vdash 1 : \texttt{nat}} \qquad \cdots$$

$$\frac{\Gamma \vdash E_0 : \texttt{nat} \qquad \Gamma \vdash E_1 : \texttt{nat}}{\Gamma \vdash E_0 + E_1 : \texttt{nat}} \qquad \frac{\Gamma \vdash E_0 : \texttt{nat} \qquad \Gamma \vdash E_1 : \texttt{nat}}{\Gamma \vdash E_0 \times E_1 : \texttt{nat}}$$

$$\frac{\Gamma \vdash E_0 : \texttt{nat} \qquad \Gamma \vdash E_1 : \texttt{nat}}{\Gamma \vdash E_0 - E_1 : \texttt{nat}}$$

$$\frac{\Gamma \vdash E : \texttt{nat} \qquad \Gamma \vdash E_0 : T \qquad \Gamma \vdash E_1 : T}{\Gamma \vdash \texttt{ifz } E \texttt{ then } E_0 \texttt{ else } E_1 : T}$$

$$\frac{\Gamma \vdash E_0 : T_0 \qquad \Gamma \vdash E_1 : T_1}{\Gamma \vdash \langle E_0, E_1\rangle : T_0 \times T_1} \qquad \frac{\Gamma \vdash E : T_0 \times T_1}{\Gamma \vdash \pi_i\ E : T_i}$$

$$\frac{\Gamma, x : T_0 \vdash e : T_1}{\Gamma \vdash \lambda x.E : T_0 \to T_1} \qquad \frac{\Gamma \vdash E_0 : T_0 \to T_1 \qquad \Gamma \vdash E_1 : T_0}{\Gamma \vdash E_0\ E_1 : T_1}$$

$$\frac{\Gamma, x : T \vdash E : T}{\Gamma \vdash \texttt{fix } x = E : T} \qquad \frac{\Gamma \vdash E_0 : T \qquad \Gamma \vdash E_1 : T}{\Gamma \vdash E_0 \vee E_1 : T}$$

Figure 3: Complexity language types, expressions, and typing.

Substitutions Θ in the complexity language are defined as usual and satisfy standard composition properties:

Lemma 2.4.

- *If x does not occur in Θ, then $E[\Theta, x/x][E_1/x] = E[\Theta, E_1/x]$.*
- *If x_1, x_2 do not occur in Θ, then $E[E_1/x_1][E_2/x_2][\Theta] = E[E_1[\Theta]/x_1, E_2[\Theta]/x_2]$.*

$$\frac{}{\Gamma \vdash E =_T E}\ \text{(reflexivity)} \qquad \frac{\Gamma \vdash E_0 =_T E_1 \qquad \Gamma \vdash E_1 =_T E_2}{\Gamma \vdash E_0 =_T E_2}\ \text{(transitivity)}$$

$$\frac{}{\Gamma \vdash 0 + E =_{\mathtt{nat}} E} \qquad \frac{}{\Gamma \vdash E + 0 =_{\mathtt{nat}} E}$$

$$\frac{}{\Gamma \vdash (E_0 + E_1) + E_2 =_{\mathtt{nat}} E_0 + (E_1 + E_2)}$$

$$\frac{}{\Gamma \vdash 0 - E =_{\mathtt{nat}} 0} \qquad \frac{}{\Gamma \vdash E - 0 =_{\mathtt{nat}} E}$$

$$\frac{}{\Gamma \vdash 0 * E =_{\mathtt{nat}} 0} \qquad \frac{}{\Gamma \vdash E * 0 =_{\mathtt{nat}} 0}$$

$$\frac{}{\Gamma \vdash (E_0 * E_1) * E_2 =_{\mathtt{nat}} E_0 * (E_1 * E_2)}$$

$$\frac{}{\Gamma \vdash E_0 =_T \mathtt{ifz}\ 0\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1} \qquad \frac{\Gamma \vdash E = n + 1}{\Gamma \vdash E_1 =_T \mathtt{ifz}\ E\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1}$$

$$\frac{\Gamma \vdash E_0 = E_1 : T}{\Gamma, x : T \vdash E[E_0/x] = E[E_1/x]}$$

$$\frac{}{\Gamma \vdash E_0[V_1/x] =_T (\lambda x.E_0)V_1} \qquad \frac{}{\Gamma \vdash E_i =_T \pi_i\langle E_0, E_1\rangle}$$

$$\frac{}{\Gamma \vdash E[\mathtt{fix}\ x = E/x] =_T \mathtt{fix}\ x = E}$$

$$\frac{}{\Gamma \vdash \underline{m \vee n} =_{\mathtt{nat}} \underline{m} \vee \underline{n}} \qquad \frac{}{\Gamma \vdash \lambda x.(E_0\ x \vee E_1\ x) =_{T_0 \to T_1} E_0 \vee E_1}$$

$$\frac{}{\Gamma \vdash \langle \pi_0 E_0 \vee \pi_0 E_1, \pi_1 E_0 \vee \pi_1 E_1\rangle =_{T_0 \times T_1} E_0 \vee E_1}$$

Figure 4: The Complexity Language Equational Semantics

3. The Extraction Function

A notion of complexity that considers only the cost is insufficient for properly extracting recurrences for higher-order functions such as

$$\mathit{summation} = \lambda g.\lambda m.(\mathsf{fix}\ f = \lambda w.\mathsf{case}\ (w,\ 0 \mapsto g\ 0 \mid S(n) \mapsto \mathit{plus}\ (g\ S(n))\ (f\ n)))\ m$$

The cost of $\mathit{summation}\ g\ m$ depends on the cost of evaluating g on the natural numbers from m to 0 and also indirectly depends on the sizes of the natural numbers.

Since *summation g m* might be used as an argument to another function, for example another *summation*, we need to take into account the sizes of the elements of *summation g m*, which depends on the size of the output of g. So to analyze the complexity of *summation*, we need to give a recurrence for the cost and size of $g\ m$ in terms of the size of m, from which we produce a recurrence for the cost and size of *summation g m* in terms of the size of m. We define the size of a value as that expression's potential since the size of the value determines the cost of using that value in the future.

$\|e\|$ is the recurrence extracted from e by applying the extraction function. Since we extract recurrences for both the cost of evaluation and size of e, the type of $\|e\|$ is $\mathtt{nat} \times \_$, where the first component is the type of the cost and the second component is the type of the size. This motivates the question, what is the appropriate type for the size (i.e., potential) of e? If $e : \tau$, we write $\langle\langle \tau \rangle\rangle$ for the type of the potential of e. So the type of $\|e\|$ is $\mathsf{nat} \times \langle\langle \tau \rangle\rangle$, which we abbreviate as $\|\tau\|$.

To gain some intuition for the full definition of potential in a call-by-name setting, we consider the type-level 0 and 1 cases. At type-level 0, the potential cost of an expression is the measure of size of the expression's value, since the size of the value determines the future cost of using that expression in later computations. Now consider a type-level 1 expression e_0. We use e_0 in an application to some type-level 0 expression e_1. The cost of the application is the sum of

(1) cost of evaluating e_0 to a value $\lambda x.e_0'$,
(2) cost of evaluating $e_0'[e_1/x]$ to a value
(3) possible charge for the β-reduction.

Since (2) depends on the cost of evaluating e_1 to a value v_1 and on the size of v_1, or the potential of e_1, the complexities must capture both cost and potential. Therefore, the potential of a type-level 1 expression should be a map from type-level 0 complexities to type-level 0 complexities. And in general, $\langle\langle \sigma \to \tau \rangle\rangle = \|\sigma\| \to \|\tau\|$.

With this notion in mind, we consider the type of *summation*. Its potential should be descriptive of the cost of future uses of *summation* respective to the potentials of the arguments. The discussion above suggests for the type of *summation*, its potential

$$\langle\!\langle (\mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat}) \to (\mathsf{nat}) \rangle\!\rangle,$$

should be

$$\|\mathsf{nat} \to \mathsf{nat}\| \to \|\mathsf{nat} \to \mathsf{nat}\|$$

$$= \mathsf{nat} \times \langle\!\langle \mathsf{nat} \to \mathsf{nat} \rangle\!\rangle \to \mathsf{nat} \times \langle\!\langle \mathsf{nat} \to \mathsf{nat} \rangle\!\rangle$$

$$= \mathsf{nat} \times (\|\mathsf{nat}\| \to \|\mathsf{nat}\|) \to \mathsf{nat} \times (\|\mathsf{nat}\| \to \|\mathsf{nat}\|)$$

$$= \mathsf{nat} \times ((\mathsf{nat} \times \langle\!\langle \mathsf{nat} \rangle\!\rangle) \to (\mathsf{nat} \times \langle\!\langle \mathsf{nat} \rangle\!\rangle))$$

$$\to \mathsf{nat} \times ((\mathsf{nat} \times \langle\!\langle \mathsf{nat} \rangle\!\rangle) \to (\mathsf{nat} \times \langle\!\langle \mathsf{nat} \rangle\!\rangle))$$

The extraction function from source types and expressions to complexity types and expressions is given in Figure 5. Given E, a complexity, we write E_c and E_p for $\pi_0 E$ and $\pi_1 E$ respectively for the cost and potential. We often need to "add cost" to a complexity while extracting recurrences. To add cost to a complexity; when E_1 is a cost and E_2 is a complexity, we write $E_1 +_c E_2$ for the complexity $\langle E_1 + (E_2)_c, (E_2)_p \rangle$. We also need to "apply" a complexity to another complexity. Given two complexity expressions E and E', we write $E \cdot E'$ for the complexity $E_c +_c E_p\ E' = \langle E_c + (E_p\ E')_c, (E_p\ E')_p \rangle$. Complexity application binds tighter than complexity cost addition. The extraction function maps the source language application $e\ e'$ to the complexity language application $\|e\| \cdot \|e'\|$.

We have discussed the size of functions, but not the size of nat. In **?**, the formalism asks for programmer-defined sizes provided the sizes follow a preorder. However this causes difficulties in the presence of general recursion, thus we fix the notion of size for nat in this setting. We fix the potential of a natural number as the number itself. Thus, $\langle\!\langle \mathsf{nat} \rangle\!\rangle = \mathtt{nat}$. When discussing functions that act on natural numbers, we can interpret the use cost of the number as the number itself.

$$
\begin{aligned}
\|\tau\| &= \mathtt{nat} \times \langle\!\langle \tau \rangle\!\rangle \\
\langle\!\langle \mathsf{nat} \rangle\!\rangle &= \mathtt{nat} \\
\langle\!\langle \sigma \to \tau \rangle\!\rangle &= \|\sigma\| \to \|\tau\|
\end{aligned}
$$

$$
\begin{aligned}
\|x\| &= x \\
\|0\| &= \langle 0, 0 \rangle \\
\|S(e)\| &= \langle \|e\|_c, \|e\|_p + 1 \rangle \\
\|\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)\| &= \|e\|_c +_c \mathtt{ifz}\ \|e\|_p\ \mathtt{then}\ \|e_0\| \\
&\qquad \mathtt{else}\ \|e_0\| \vee \|e_1\|[\langle 0, \|e\|_p - 1 \rangle / n] \\
\|\lambda x.e\| &= \langle 0, \lambda x.\|e\| \rangle \\
\|e_0\ e_1\| &= \|e_0\| \cdot \|e_1\| = \|e_0\|_c +_c \|e_0\|_p \|e_1\| \\
\|\mathsf{fix}\ x = e\| &= \mathtt{fix}\ x = 1 +_c \|e\|
\end{aligned}
$$

Figure 5: Extraction function from source types and expressions to complexity types. Recall that $\|e\|_c = \pi_0\|e\|$ and $\|e\|_p = \pi_1\|e\|$.

The reason the extraction function applied to the case expression

$$\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)$$

maps e_1 to $\|e_0\| \vee \|e_1\|$ instead of simply $\|e_1\|$ is to allow encoding of booleans. We can interpret false as 0 and true as 1. In the complexity language, booleans have the same sizes and therefore are indistinguishable. Since our extraction function explicitly defines the interpretation of sizes of datatypes, it needs to reflect that the encoded booleans have the same size. The `ifz` statement in the complexity language has to take the maximum in the non-zero case since the source language expression can evaluate to 0 but its translation have a potential greater than 0. We do not need to take the maximum in the zero case since when the source language expression evaluates to 0, it's potential cannot be greater than 0. A possible workaround is to directly add booleans to the source and complexity languages and we present an example program in which we demonstrate the need for the arbitrary maximum in Chapter 4.

The type extraction function is extended pointwise to contexts, so $x : \tau \in \gamma$ iff $x : \|\tau\| \in \|\gamma\|$. In a call-by-name source language, a function application first evaluates

the left-hand side expression to an abstraction then substitutes the bound variable for the right-hand side expression. Since the expression being substituted may need to be evaluated down to a value, we need to carry over information about the cost of evaluating that expression every time it is called. Therefore, variables range over complexities as potentials of higher-order expressions take complexities as inputs and return complexities as outputs. To illustrate, $\|x\| = x$, where the x on the left is a source variable and the x on the right is a corresponding complexity variable. In reality the actual corresponding variables of the source and complexity languages may differ, but for convenience we make use of the same variable.

We note that the extraction function preserves types.

LEMMA 2.5. *If* $\gamma \vdash e : \tau$, *then* $\|\gamma\| \vdash \|e\| : \|\tau\|$.

4. Examples

We show the results of applying the extraction function on several functions of varying complexity in this section. We begin with examples of functions that could be defined by structural recursion written using general recursion to illustrate the analysis of structurally recursive functions have not changed. We then move on to examples of functions that are naturally defined by general recursion, that were previously not possible to extract recurrences from. We assume that the functions we are analyzing are being applied to values for convenience. Without this assumption, we have to carry around the costs of evaluating the arguments to values throughout the extraction function, even though they do not have a significant effect on the analysis.

4.1. Plus. We start by considering the plus function defined by

$$plus = \lambda x.\lambda y.(\mathsf{fix}\ f = \lambda w.\mathsf{case}\ (w,\ 0 \mapsto y \mid S(n) \mapsto S(f(n))))x.$$

Translating yields,

$$\begin{aligned}\|f(n)\| &= \|f\| \cdot \|n\| \\ &= f_c +_c f_p\, n \\ \|S(f(n))\| &= \langle f_c +_c (f_p\, n)_c, (f_p\, n)_p + 1\rangle\end{aligned}$$

Let $s(f,n) = \langle f_c +_c (f_p\, n)_c, (f_p\, n)_p + 1\rangle$. Then

$$\begin{aligned}&\|\lambda w.\mathsf{case}\ (w,\ 0 \mapsto y \mid S(n) \mapsto S(f(n)))\| \\ &= \langle 0, \lambda w.w_c +_c \texttt{ifz}\ w_p\ \texttt{then}\ y\ \texttt{else}\ y \vee s(f,n)[\langle 0, w_p - 1\rangle/n]\rangle \\ &= \langle 0, \lambda w.w_c +_c \texttt{ifz}\ w_p\ \texttt{then}\ y\ \texttt{else}\ y \vee s(f, \langle 0, w_p - 1\rangle)\rangle\end{aligned}$$

Let $F(y,w,f) = w_c +_c \texttt{ifz}\ w_p\ \texttt{then}\ y\ \texttt{else}\ y \vee s(f, \langle 0, w_p - 1\rangle$. Then

$$\begin{aligned}\|\mathsf{fix}\ f = \lambda w.\mathsf{case}\ (w,\ 0 \mapsto y \mid S(n) \mapsto S(f(n)))\| &= \texttt{fix}\ f = 1 +_c \langle 0, \lambda w.F(y,w,f)\rangle \\ &= \texttt{fix}\ f = \langle 1, \lambda w.F(y,w,f)\rangle\end{aligned}$$

Let $F'(y) = \texttt{fix}\ f = \langle 1, \lambda w.F(y,w,f)\rangle$. Then

$$\|(\mathsf{fix}\ f = \lambda w.\mathsf{case}\ (w,\ 0 \mapsto y \mid S(n) \mapsto S(f(n))))x\| = F'(y)_c +_c F'(y)_p\, x$$

$$\|plus\| = \langle 0, \lambda x.\langle 0, \lambda y.F'(y)_c +_c F'(y)_p\, x\rangle\rangle$$

Let $\overline{n} = \langle 0, n\rangle$ and $\overline{m} = \langle 0, m\rangle$. Then

$$\begin{aligned}\|plus\ m\ n\| &= \langle 0, \lambda x.\langle 0, \lambda y.F'(y)_c +_c F'(y)_p\, x\rangle\rangle \cdot \|m\| \cdot \|n\| \\ &= \langle 0, \lambda y.F'(y)_c +_c F'(y)_p\, \langle 0, m\rangle\rangle \cdot \langle 0, n\rangle \\ &= F'(\overline{n})_c +_c F'(\overline{n})_p\, \overline{m} \\ &= (\texttt{fix}\ f = \langle 1, \lambda w.F(\overline{n}, w, f)\rangle)_c +_c (\texttt{fix}\ f = \langle 1, \lambda w.F(\overline{n}, w, f)\rangle)_p\, \overline{m} \\ &= (\langle 1, \lambda w.F(\overline{n}, w, F'(\overline{n}))\rangle)_c +_c (\langle 1, \lambda w.F(\overline{n}, w, F'(\overline{n}))\rangle_p)\, \overline{m} \\ &= 1 +_c \lambda w.F(\overline{n}, w, F'(\overline{n}))\ \overline{m} \\ &= 1 +_c F(\overline{n}, \overline{m}, F'(\overline{n}))\end{aligned}$$

We expect

$$F(\overline{n}, \overline{m}, F'(\overline{n})) = \langle m, m+n \rangle.$$

That is, evaluating *plus* m n takes m steps and the potential of the evaluated result is the sum of m and n. We prove this claim by induction on m.

Proof.

$$\begin{aligned} F(\overline{n}, \overline{0}, F'(\overline{n})) &= \texttt{ifz } 0 \texttt{ then } \overline{n} \texttt{ else } \_ \\ &= \overline{n} \\ &= \langle 0, n \rangle \end{aligned}$$

Let $\overline{m+1} = \langle 0, m+1 \rangle$. Then

$$\begin{aligned} F(\overline{n}, \overline{m+1}, F'(\overline{n})) &= \texttt{ifz } \overline{n} \texttt{ then } \_ \texttt{ else } \overline{n} \vee s(F'(\overline{n}), \langle 0, (m+1)-1 \rangle) \\ &= \overline{n} \vee s(F'(\overline{n}), \langle 0, m \rangle) \\ &= \overline{n} \vee \langle F'(\overline{n})_c + (F'(\overline{n})_p \langle 0, m \rangle)_c, (F'(\overline{n})_p \langle 0, m \rangle)_p + 1 \rangle \\ &= \overline{n} \vee \langle 1 + F(\overline{n}, \overline{m}, F'(\overline{n}))_c, F(\overline{n}, \overline{m}, F'(\overline{n}))_p + 1 \rangle \\ \text{(by the IH)} \quad &= \overline{n} \vee \langle 1+m, m+n+1 \rangle \\ &= \langle 1+m, m+n+1 \rangle \end{aligned}$$

□

The extra cost of 1 in the extraction function of *plus* is from the evaluation of the fix expression that always occurs, even in the case $m = 0$.

4.2. N-th Fibonacci. Another example of a function naturally defined by general recursion is the N-th Fibonacci number function. We first define the predecessor function

$$pred = \lambda m.\mathsf{case}\ (m,\ 0 \mapsto 0 \mid S(n) \mapsto n)$$

We translate the *pred* function

$$\begin{aligned}\|pred\| &= \langle 0, \lambda m.m_c +_c \texttt{ifz } m_p \texttt{ then } \langle 0,0\rangle \texttt{ else } \langle 0,0\rangle \vee n[\langle 0, m_p - 1\rangle/n]\rangle \\ &= \langle 0, \lambda m.m_c +_c \texttt{ifz } m_p \texttt{ then } \langle 0,0\rangle \texttt{ else } \langle 0,0\rangle \vee \langle 0, m_p - 1\rangle\rangle \\ &= \langle 0, \lambda m.m_c +_c \texttt{ifz } m_p \texttt{ then } \langle 0,0\rangle \texttt{ else } \langle 0, m_p - 1\rangle\rangle\end{aligned}$$

The claim that

$$\|pred\ m_1\| = \langle 0, m_1 - 1\rangle$$

is easily verifiable by induction on m_1, where m_1 is a value of type nat. We now define the N-th Fibonacci function

$$\begin{aligned}fib = \lambda m.(\mathsf{fix}\ f = \lambda w.\mathsf{case}\ (w, 0 \mapsto 0 \mid S(n) \mapsto \mathsf{case}\ (n, 0 \mapsto 1 \\ \mid S(n) \mapsto plus\ (f\ pred(w))(f\ pred(pred(w))))))\ m\end{aligned}$$

The translation yields

$$\|pred(w)\| = \langle 0, w - 1\rangle$$

$$\|f\ pred(w)\| = f_c +_c f_p\ \langle 0, w - 1\rangle$$

$$\|f\ pred(pred(w))\| = f_c +_c f_p\ \langle 0, w - 2\rangle$$

$$\begin{aligned}&\|plus\ (f\ pred(w))(f\ pred(pred(w)))\| \\ &= \langle (f_c +_c f_p\ \langle 0, w-2\rangle)_c + (f_c +_c f_p\ \langle 0, w-1\rangle)_c + (f_p\ \langle 0, w-1\rangle)_p, \\ &\qquad\qquad (f_p\ \langle 0, w-1\rangle)_p + (f_p\ \langle 0, w-2\rangle)_p\rangle\end{aligned}$$

$$\begin{aligned}\text{Let } R(f,w) = \langle &(f_c +_c f_p\ \langle 0, w-2\rangle)_c + (f_c +_c f_p\ \langle 0, w-1\rangle)_c + (f_p\ \langle 0, w-1\rangle)_p, \\ &(f_p\ \langle 0, w-1\rangle)_p + (f_p\ \langle 0, w-2\rangle)_p\rangle\end{aligned}$$

$$\begin{aligned}&\|\mathsf{case}\ (n, 0 \mapsto 1\ S(n) \mapsto plus\ (f\ pred(w))(f\ pred(pred(w))))\| \\ &n_c +_c \texttt{ifz } n_p \texttt{ then } \langle 0,1\rangle \texttt{ else } \langle 0,1\rangle \vee R(f,w)\end{aligned}$$

$$\|\mathsf{case}\ (w, 0 \mapsto 0 \mid S(n) \mapsto \mathsf{case}\ (n, 0 \mapsto 1$$
$$\mid S(n) \mapsto \mathit{plus}\ (f\ \mathit{pred}(n))(f\ \mathit{pred}(\mathit{pred}(n))))))\|$$
$$= w_c +_c \mathtt{ifz}\ w_p\ \mathtt{then}\ \langle 0, 0\rangle$$
$$\mathtt{else}\ \langle 0, 0\rangle \vee \langle 0, w_p - 1\rangle_c +_c\ \mathtt{ifz}\ \langle 0, w_p - 1\rangle_p\ \mathtt{then}\ \langle 0, 1\rangle$$
$$\mathtt{else}\ \langle 0, 1\rangle \vee R(f, w)$$
$$= w_c +_c \mathtt{ifz}\ w_p\ \mathtt{then}\ \langle 0, 0\rangle\ \mathtt{else\ ifz}\ w_p - 1\ \mathtt{then}\ \langle 0, 1\rangle\ \mathtt{else}\ \langle 0, 1\rangle \vee R(f, w)$$

$$\|\mathsf{fix}\ f = \lambda w.\mathsf{case}\ (w, 0 \mapsto 0 \mid S(n) \mapsto \mathsf{case}\ (n, 0 \mapsto 1$$
$$\mid S(n) \mapsto \mathit{plus}\ (f\ \mathit{pred}(w))(f\ \mathit{pred}(\mathit{pred}(w))))))\|$$
$$= \mathtt{fix}\ f = 1 +_c \langle 0, \lambda w.w_c +_c \mathtt{ifz}\ w_p\ \mathtt{then}\ \langle 0, 0\rangle$$
$$\mathtt{else\ ifz}\ w_p - 1\ \mathtt{then}\ \langle 0, 1\rangle\ \mathtt{else}\ \langle 0, 1\rangle \vee R(f, w)\rangle$$
$$= \mathtt{fix}\ f = \langle 1, \lambda w.w_c +_c \mathtt{ifz}\ w_p\ \mathtt{then}\ \langle 0, 0\rangle$$
$$\mathtt{else\ ifz}\ w_p - 1\ \mathtt{then}\ \langle 0, 1\rangle\ \mathtt{else}\ \langle 0, 1\rangle \vee R(f, w)\rangle$$

$$\text{Let } F(f, w) = w_c +_c \mathtt{ifz}\ w_p\ \mathtt{then}\ \langle 0, 0\rangle$$
$$\mathtt{else\ ifz}\ w_p - 1\ \mathtt{then}\ \langle 0, 1\rangle\ \mathtt{else}\ \langle 0, 1\rangle \vee R(f, w)$$

$$\text{Let } F' = \mathtt{fix}\ f = \langle 1, \lambda w.F(f, w)\rangle$$
$$= \langle 1, \lambda w.F(F', w)\rangle$$

$$\|(\mathsf{fix}\ f = \lambda w.\mathsf{case}\ (w, 0 \mapsto 0 \mid S(n) \mapsto \mathsf{case}\ (n, 0 \mapsto 1$$
$$\mid S(n) \mapsto \mathit{plus}\ (f\ \mathit{pred}(w))(f\ \mathit{pred}(\mathit{pred}(w))))))\ m\|$$
$$= F'_c +_c F'_p\ m$$

$$\begin{aligned}\|fib\| &= \langle 0, \lambda m.F'_c +_c F'_p\ m\rangle \\ \|fib\ n\| &= F'_c +_c F'_p\ \langle 0, n\rangle \\ &= 1 +_c F(F', \langle 0, n\rangle)\end{aligned}$$

Since we cannot find a closed form for the potential component of the recurrence, we focus on finding a closed form for the cost component of the recurrence. To do so, we assume that the cost of *plus* now no longer depends on the potential component of the first number. We claim

$$F(F', \langle 0, n\rangle) = \langle 2^{0 \vee n-1} - 2, F(F', \langle 0, n\rangle)_p\rangle$$

We prove by strong induction on n.

PROOF. CASE: $n = 0$

$$\begin{aligned}F(F', \langle 0, 0\rangle) &= \texttt{ifz}\ 0\ \texttt{then}\ \langle 0, 0\rangle\ \texttt{else}\ \_ \\ &= \langle 0, 0\rangle\end{aligned}$$

CASE: $n = 1$

$$\begin{aligned}F(F', \langle 0, 1\rangle) &= \texttt{ifz}\ 1\ \texttt{then}\ \_\ \texttt{else ifz}\ 0\ \texttt{then}\ \langle 0, 1\rangle\ \texttt{else}\ \_ \\ &= \langle 0, 1\rangle\end{aligned}$$

CASE: $n = n' + 2$

$$\begin{aligned}&F(F', \langle 0, n' + 2\rangle) \\ &= \texttt{ifz}\ n' + 2\ \texttt{then}\ \_ \\ &\quad \texttt{else ifz}\ n' + 2 - 1\ \texttt{then}\ \langle 0, 1\rangle\ \texttt{else}\ \langle 0, 1\rangle \vee R(F', \langle 0, n' + 2\rangle) \\ &= \langle 0, 1\rangle \vee R(F', \langle 0, n' + 2\rangle) \\ &= \langle 0, 1\rangle \vee \\ &\langle (F'_c +_c F'_p\ \langle 0, n' + 2 - 2\rangle)_c + (F'_c +_c F'_p\ \langle 0, n' + 2 - 1\rangle)_c, \\ &\qquad\qquad (F'_p\ \langle 0, n' + 2 - 1\rangle)_p + (F'_p\ \langle 0, n' + 2 - 2\rangle)_p\rangle\end{aligned}$$

$= \langle 0, 1\rangle \vee$

$\langle (1 +_c F(F', \langle 0, n'\rangle))_c + (1 +_c F(F', \langle 0, n' + 1\rangle))_c,$

$(F(F', \langle 0, n' + 1\rangle))_p + (F(F', \langle 0, n'\rangle))_p\rangle$

(by the IH)

$= \langle 0, 1\rangle \vee \langle (1 +_c \langle 2^{0 \vee n'} - 2, F(F', \langle 0, n'\rangle)_p\rangle)_c + (1 +_c \langle 2^{0 \vee n'+1} - 2, F(F', \langle 0, n\rangle)\rangle)_c,$

$(\langle 2^{n'+1} - 2, F(F', \langle 0, n + 1\rangle)\rangle)_p + (\langle 2^{0 \vee n'} - 2, F(F', \langle 0, n'\rangle)\rangle)_p\rangle$

$= \langle 0, 1\rangle \vee \langle (1 + 2^{n'} - 2 + 1 + 2^{n'+1} - 2, F(F', \langle 0, n + 1\rangle)_p + F(F', \langle 0, n'\rangle)_p\rangle$

$= \langle (2^{n'+2} - 2, F(F', \langle 0, n + 1\rangle)_p + F(F', \langle 0, n'\rangle)_p\rangle$

$= \langle (2^{0 \vee n'+2} - 2, F(F', \langle 0, n + 1\rangle)_p + F(F', \langle 0, n'\rangle)_p\rangle$

□

4.3. Subtraction. An example of a function for which we do not give very precise bounds on size is the subtraction function defined by

$$subtract = \lambda m_1.\lambda m_2.(\mathsf{fix}\ f = \lambda u.\lambda v.\mathsf{case}\ (v,\ 0 \mapsto u \mid S(n) \mapsto f\ (pred1\ u)\ n))\ m_1\ m_2$$

We translate the *subtract* function

$$\|pred\ u\| = \langle u_c, u_p - 1\rangle$$

$$\|f\ (pred\ u)\| = f_c +_c f_p\ \langle u_c, u_p - 1\rangle$$

$$\|f\ (pred\ u)\ n\| = (f_c +_c f_p\ \langle u_c, u_p - 1\rangle)_c +_c (f_p\ \langle u_c, u_p - 1\rangle)_p\ n$$

$\|\lambda u.\lambda v.\mathsf{case}\ (v,\ 0 \mapsto u \mid S(n) \mapsto f\ (pred\ u)\ n)\|$

$= \langle 0, \lambda u.\langle 0, \lambda v.v_c +_c \texttt{ifz}\ v_p\ \texttt{then}\ u$

$\texttt{else}\ u \vee (f_c +_c f_p\ \langle u_c, u_p - 1\rangle)_c +_c (f_p\ \langle u_c, u_p - 1\rangle)_p\ \langle 0, v_p - 1\rangle\rangle\rangle$

Let $F(f, u, v) = v_c +_c \texttt{ifz}\ v_p\ \texttt{then}\ u$

$\texttt{else}\ u \vee (f_c +_c f_p\ \langle u_c, u_p - 1\rangle)_c +_c (f_p\ \langle u_c, u_p - 1\rangle)_p\ \langle 0, v_p - 1\rangle$

$$\|\mathsf{fix}\ f = \lambda u.\lambda v.\mathsf{case}\ (v,\ 0 \mapsto u \mid S(n) \mapsto f\ (pred\ u)\ \langle 0, v_p - 1\rangle))\|$$

$$= \mathtt{fix}\ \ f = \langle 1, \lambda u.\langle 0, \lambda v.v_c +_c \mathtt{ifz}\ \ v_p\ \ \mathtt{then}\ \ u\ \ \mathtt{else}\ \ F(f, u, \langle 0, v_p - 1\rangle)\rangle\rangle$$

Let $F' = \mathtt{fix}\ \ f = \langle 1, \lambda u.\langle 0, \lambda v.F(f, u, v)\rangle\rangle$.

$$\|\mathsf{fix}\ f = \lambda u.\lambda v.\mathsf{case}\ (v,\ 0 \mapsto u \mid S(n) \mapsto f\ (pred\ u)\ n))\ m_1\ m_2\|$$

$$= (F'\ m_1)_c +_c (F'\ m_1)_p \cdot m_2$$

$$= ((F'\ m_1)_c + ((F'\ m_1)_p\ m_2)_c) +_c ((F'\ m_1)_p\ m_2)_p$$

$$\|subtract\|$$

$$= \|\lambda m_1.\lambda m_2.(\mathsf{fix}\ f = \lambda u.\lambda v.\mathsf{case}\ (v,\ 0 \mapsto u \mid S(n) \mapsto f\ (pred\ u)\ n))\ m_1\ m_2\|$$

$$= \langle 0, \lambda m_1.\langle 0, \lambda m_2.((F'\ m_1)_c + ((F'\ m_1)_p\ m_2)_c) +_c ((F'\ m_1)_p\ m_2)_p\rangle\rangle$$

$$\|subtract\ n_1\ n_2\|$$

$$= \langle 0, \lambda m_1.\langle 0, \lambda m_2.((F'\ m_1)_c + ((F'\ m_1)_p\ m_2)_c) +_c ((F'\ m_1)_p\ m_2)_p\rangle\rangle \cdot \|n_1\| \cdot \|n_2\|$$

$$= \langle 0, \lambda m_1.\langle 0, \lambda m_2.((F'\ m_1)_c + ((F'\ m_1)_p\ m_2)_c) +_c ((F'\ m_1)_p\ m_2)_p\rangle\rangle \cdot \langle 0, n_1\rangle \cdot \langle 0, n_2\rangle$$

$$= ((F'\ \langle 0, n_1\rangle)_c + ((F'\ \langle 0, n_1\rangle)_p\ \langle 0, n_2\rangle)_c) +_c ((F'\ \langle 0, n_1\rangle)_p\ \langle 0, n_2\rangle)_p$$

$$= 1 +_c F(F', \langle 0, n_1\rangle, \langle 0, n_2\rangle)$$

We claim

$$F(F', \langle 0, n_1\rangle, \langle 0, n_2\rangle) = \langle n_2, n_1\rangle$$

We prove by induction on n_2.

Proof. CASE: $n_2 = 0$.

$$F(F', \langle 0, n_1\rangle, \langle 0, 0\rangle) = \langle 0, 0\rangle_c +_c \mathtt{ifz}\ \ \langle 0, 0\rangle_p\ \ \mathtt{then}\ \ \langle 0, 1\rangle\ \ \mathtt{else}\ \ \_$$

$$= \mathtt{ifz}\ \ 0\ \ \mathtt{then}\ \ \langle 0, 1\rangle\ \ \mathtt{else}\ \ \_$$

$$= \langle 0, 1\rangle$$

CASE: $n_2 = n_2' + 1$

$$F(F', \langle 0, n_1\rangle, \langle 0, n_2' + 1\rangle) = \langle 0, n_2' + 1\rangle_c +_c \texttt{ifz } \langle 0, n_2' + 1\rangle_p \texttt{ then \_ else } \langle 0, n_1\rangle \vee$$

$$(F_c' +_c F_p' \ \langle 0, n_1\rangle_p - 1)_c +_c (F_p' \ \langle \langle 0, n_1\rangle_c, \langle 0, n_1\rangle_p - 1\rangle)_p \ \langle 0, \langle 0, n_2' + 1\rangle_p - 1\rangle$$

$$= \texttt{ifz } n_2' + 1 \texttt{ then \_ else } \langle 0, n_1\rangle \vee (F_c' +_c F_p' \ (n_1 - 1))_c +_c (F_p' \ \langle 0, n_1 - 1\rangle)_p \ \langle 0, n_2'\rangle$$

$$= \langle 0, n_1\rangle \vee (F_c' +_c F_p' \ (n_1 - 1))_c +_c (F_p' \ \langle 0, n_1 - 1\rangle)_p \ \langle 0, n_2'\rangle$$

$$= \langle 0, n_1\rangle \vee (1 +_c F(F', \langle 0, n_1 - 1\rangle, \langle 0, n_2'\rangle)$$

$$= \langle 0, n_1\rangle \vee (1 +_c \langle n_2', n_1\rangle) \text{(by the IH)}$$

$$= \langle 0 \vee n_2' + 1, n_1 \vee n_1\rangle$$

$$= \langle n_2 + 1, n_1\rangle$$

□

While we want to prove the stronger claim that

$$F(F', \langle 0, n_1\rangle, \langle 0, n_2\rangle) = \langle n_2, n_2 - n_1\rangle,$$

we cannot prove it due to the maximum in the `else` statement, which forces us to take the larger number n_1 as the potential each time in the recursive call of *subtract*. One possible way of fixing this issue is to directly implement addition, subtraction, and multiplication into the source language, where we interpret the sizes of the operators results as the corresponding numeric values.

CHAPTER 3

The Bounding Theorem

We present a logical relation between source language expressions and complexity language expressions in order to reason about cost and potential. We introduce the concept of fixed point induction in order to reason about the bounding relation for the fix operator and the extracted recurrence. We present a proof of soundness for the extraction function, which guarantees the cost of evaluating a program defined in the source language is bounded by the cost component of its extracted recurrence.

1. The Bounding Relation

While the minimum criterion for the correctness of our technique is that the evaluation cost of a closed source program is bounded by the cost component of its extracted recurrence, to show our extracted recurrences are correct, we need to consider more than closed expressions. We need to also know that the potential of a function bounds the cost of evaluating that function applied to all arguments, and so on at higher types. We present a logical relation between closed source language programs and complexities which we then generalize to source language expressions and complexity language expressions. The $\leq$ operator is the less than or equal to operation for the natural numbers. Recall that we write $E \downarrow$ to mean there exists an equational reasoning chain that equates the complexity expression $E : \mathtt{nat} \times \mathtt{nat}$ to some unique complexity value. When $e \downarrow^m v$, we write $\mathsf{cost}(e)$ to mean m and $\mathsf{val}(e)$ to mean v. Thus we write $\mathsf{cost}(e) \leq E_c$ to mean m is less than or equal to the first projection of the unique complexity value that E is equal to.

DEFINITION 3.1.

$$E \leq E' =_{df} \exists m, m' \ s.t. \ E = m \ and \ E' = m' \ and \ m \leq m'$$

DEFINITION 3.2 (Closed Bounding Relation). *For a closed source language expression e and a closed complexity language expression E, we write $e \sqsubseteq_\tau E$ to mean:*

(1) $e \sqsubseteq_{\mathsf{nat}} E$ *: if* $E \downarrow$, *then*
   (a) $\mathit{cost}(e) \leq E_c$; *and,*
   (b) $\mathit{val}(e) \leq E_p$.
(2) $e \sqsubseteq_{\sigma \to \tau} E$ *: for all* e' *and* E', *if* $e' \sqsubseteq_\sigma E'$, *then* $e\ e' \sqsubseteq_\tau E \cdot E'$.

The bounding relation states that we only consider bounding for complexity language expressions that converge. When the complexity language expression is divergent, it trivially bounds all source language expressions, including ones that converge. On the other hand, convergent complexity language expression E only bounds convergent source language expression e at base type nat, where the cost of e is less than or equal to the cost component of E and the value that e evaluates to is less than or equal to the potential component of E. Since the size of a natural number is itself, we denote it as $\mathsf{val}(e)$. At higher type, we do not explicitly consider the cost and potential of the expression and instead we require them to be bounded at the lower type when applied respectively by bounded expressions. We extend the relation to open terms by considering all closed instances.

DEFINITION 3.3 (Environments). *We define environments for both the source and complexity languages.*

(1) *For γ a source-language context, θ a map from variables to closed source language expressions, we say θ is a γ-environment to mean that $\forall x \in \mathit{Dom}\ \gamma$, $\emptyset \vdash \theta(x) : \gamma(x)$.*
(2) *We say Θ is a Γ-environment to mean that $\forall x \in \mathit{Dom}\ \Gamma$, $\emptyset \vdash \Theta(x) : \Gamma(x)$.*

DEFINITION 3.4 (Open Bounding Relation). *The relation on open terms considers all closed instances:*

(1) *For a γ-environment θ and a Γ-environment Θ, we write $\theta \sqsubseteq_\gamma \Theta$ to mean that for all $x \in \mathit{Dom}\ \gamma$, $\theta(x) \sqsubseteq_{\gamma(x)} \Theta(x)$.*

(2) *For* $\gamma \vdash e : \tau$ *and* $\Gamma \vdash E : \|\tau\|$*, we write* $\gamma \vdash e : \tau \sqsubseteq_\tau E : \|\tau\|$ *to mean that for all* $\theta : \gamma$ *and* $\Theta : \Gamma$*, if* $\theta \sqsubseteq_\gamma \Theta$*, then* $e[\theta] \sqsubseteq_\tau E[\Theta]$*. We write* $e \sqsubseteq_\tau E$ *when* γ, Γ, τ *are clear from context.*

2. The Fundamental Theorem

We now establish some key concepts and definitions before stating several lemmas necessary for the proof of the bounding theorem.

DEFINITION 3.5 (Bounded Recursion). *We first define the concept of source language bounded recursion by adding term constructors* $\mathsf{fix}_m\ x = e$ *to the source language with the following operational semantics by induction on* $m \geq 0$*:*

$$\frac{e[\mathsf{fix}_m\ x = e/x] \downarrow^{m_1} v}{\mathsf{fix}_{m+1}\ x = e \downarrow^{m_1+1} v}$$

At $m = 0$*, the expression* $\mathsf{fix}_0\ x = e$ *is divergent. We define bounded recursion in the complexity language similarly as follows:*

$$(\mathtt{fix}_0\ x = E) = (\mathtt{fix}\ x = x)$$

$$(\mathtt{fix}_{m+1}\ x = E) = E[\mathtt{fix}_m\ x = E/x]$$

We state some facts about the complexity language's equational reasoning.

LEMMA 3.6 (Complexity Equality Convergence). *If* $E = E' : \mathtt{nat} \times \mathtt{nat}$*, then*

$$E \downarrow \text{ iff } E' \downarrow .$$

where E *and* E' *are equal to the same value.*

LEMMA 3.7 (Complexity Convergence Cost Weakening). $\forall k$*, if* $k +_c E \downarrow$*, then* $E \downarrow$

LEMMA 3.8 (Complexity Convergence Case Weakening). *If* $\mathtt{ifz}\ E\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 \downarrow$*, then* $E \downarrow$ *and*

$$\text{if } E = 0 \text{ then } E_0 \downarrow$$

$$\text{if } E = n + 1 \text{ then } E_1 \downarrow .$$

LEMMA 3.9 (Complexity Convergence Potential Weakening). *If* $\langle E_c, E_p + 1\rangle \downarrow$, *then* $E \downarrow$.

LEMMA 3.10 (Complexity Language Equivalence). *If* $E = E'$, *then*

$$\forall e, e \sqsubseteq_\tau E \text{ iff } e \sqsubseteq_\tau E'.$$

PROOF. We prove by induction on τ.

CASE: $\tau = \mathsf{nat}$. Assume $e \sqsubseteq_{\mathsf{nat}} E$. To show $e \sqsubseteq_{\mathsf{nat}} E'$, it suffices to show that if $E' \downarrow$, then

$$\mathsf{cost}(e) \leq E'_c; \text{ and}$$

$$\mathsf{val}(e) \leq E'_p.$$

Since $E' \downarrow$ and $E = E'$, by complexity language equivalence (Lemma 3.6),

$$E \downarrow$$

where E' and E are equal to the same value. And since $e \sqsubseteq_{\mathsf{nat}} E$,

$$\mathsf{cost}(e) \leq E_c = E'_c; \text{ and}$$

$$\mathsf{val}(e) \leq E_p = E'_p.$$

The proof for the other direction is symmetric.

CASE: $\tau = \tau_0 \to \tau_1$. Assume $e \sqsubseteq_{\tau_0 \to \tau_1} E$ and $E = E' : \|\tau\|$. We need to show $e \sqsubseteq_{\tau_0 \to \tau_1} E'$. Assume $e_0 \sqsubseteq_{\tau_0} E_0$. It suffices to show

$$e\ e_0 \sqsubseteq_{\tau_1} E' \cdot E_0$$

By the induction hypothesis, it suffices to show

$$E \cdot E_0 = E' \cdot E_0$$

and

$$e\ e_0 \sqsubseteq_{\tau_1} E \cdot E_0.$$

Since $E = E'$,

$$E \cdot E_0 = E' \cdot E_0.$$

Since $e \sqsubseteq_{\tau_0 \to \tau_1} E$ and $e_0 \sqsubseteq_{\tau_0} E$,

$$e\ e_0 \sqsubseteq_{\tau_1} E \cdot E_0.$$

The result follows and the proof for the other direction is symmetric. □

DEFINITION 3.11. *We write $e \backsimeq e'$ to mean*

$$e \downarrow^m v \text{ iff } e' \downarrow^m v.$$

LEMMA 3.12 (Source Language Equivalence). *If $e \backsimeq e'$, then*

$$\forall E, e \sqsubseteq_\tau E \text{ iff } e' \sqsubseteq_\tau E.$$

PROOF. We prove by induction on τ.

CASE: $\tau = \mathsf{nat}$. Assume $e \sqsubseteq_{\mathsf{nat}} E$ and $e \backsimeq e' : \mathsf{nat}$. If $E \downarrow$, then

$$\mathsf{cost}(e) \leq E_c; \text{ and}$$

$$\mathsf{val}(e) \leq E_p.$$

But e and e' evaluate to the same v in the same number of steps, therefore

$$\mathsf{cost}(e') = \mathsf{cost}(e) \leq E_c; \text{ and}$$

$$\mathsf{val}(e') = \mathsf{val}(e) \leq E_p.$$

The proof for the other direction is symmetric.

CASE: $\tau = \tau_0 \to \tau_1$. Suppose $e \sqsubseteq_{\tau_0 \to \tau_1}$ and $e \backsimeq e' : \tau_0 \to \tau_1$. Assume $e_0 \sqsubseteq_{\tau_0} E_0$. It suffices to show

$$e'\ e_0 \sqsubseteq_{\tau_1} E \cdot E_0.$$

By the induction hypothesis, it suffices to show

$$e\ e_0 \backsimeq e'\ e_0$$

and

$$e\ e_0 \sqsubseteq_{\tau_1} E\ E_0.$$

To show the former, suppose $e\ e_0 \downarrow^{m'} v$. It must be the case that

$$\frac{e \downarrow^{m_1} \lambda x.e'' \qquad e''[e_0/x] \downarrow^{m_2} v}{e\ e_0 \downarrow^{m_1+m_2} v}$$

Since $e \backsimeq e'$, we have that

$$\frac{e' \downarrow^{m_1} \lambda x.e'' \qquad e''[e_0/x] \downarrow^{m_2} v}{e'\ e_0 \downarrow^{m_1+m_2} v}$$

The other direction is symmetric. Since $e \sqsubseteq_{\tau_0 \to \tau_1} E$ and $e_0 \sqsubseteq_{\tau_0} E_0$,

$$e\ e_0 \sqsubseteq_{\tau_1} E\ E_0.$$

The result follows and the proof for the other direction is symmetric. □

Recall that we write $E \uparrow$ to mean E is divergent; i.e., there is no equational reasoning chain that equates E to any complexity value. We write $x \notin FV(e)$ to mean x is not a free variable in the source language expression e and correspondingly $x \notin FV(E)$ to mean x is not a free variable in the complexity language expression E.

LEMMA 3.13 (Divergence). *If $E \uparrow$, then $\forall e$, $e \sqsubseteq_\tau E$.*

PROOF. We prove by induction on τ.

CASE: $\tau = \mathsf{nat}$. By definition, $e \sqsubseteq_{\mathsf{nat}} E$ vacuously.

CASE: $\tau = \tau_0 \to \tau_1$. Assume $e' \sqsubseteq_{\tau_0} E'$. We need to show

$$e\ e' \sqsubseteq_{\tau_1} E \cdot E'.$$

But since $E \uparrow$,

$$E \cdot E' \uparrow,$$

and the desired result follows from the induction hypothesis. □

LEMMA 3.14. *If $x : \tau \vdash e : \tau \sqsubseteq_\tau x : \|\tau\| \vdash E : \|\tau\|$, then*

$$\forall m \geq 0, \mathit{fix}_m\ x = e \sqsubseteq_\tau \mathit{fix}_m\ x = 1 +_c E.$$

PROOF. We prove by induction on m.

CASE: $m = 0$. The result is immediate since

$$\mathtt{fix}_0\ x = 1 +_c E \uparrow$$

and divergent complexity expressions trivially bound all source language expressions by divergence (Lemma 3.13).

CASE: $m+1$. We prove by induction on τ that $\mathsf{fix}_{m+1}\ x = e \sqsubseteq_\tau \mathtt{fix}_{m+1}\ x = E$.

SUBCASE: $\tau = \mathsf{nat}$. Assume $x : \mathsf{nat} \vdash e \sqsubseteq_{\mathsf{nat}} x : \|\mathsf{nat}\| \vdash E : \|\mathsf{nat}\|$. We need to show if $\mathtt{fix}_{m+1}\ x = 1 +_c E\downarrow$, then

$$\mathsf{cost}(\mathsf{fix}_{m+1}\ x = e) \leq (\mathtt{fix}_{m+1}\ x = 1 +_c E)_c$$

$$\mathsf{val}(\mathsf{fix}_{m+1}\ x = e) \leq (\mathtt{fix}_{m+1}\ x = 1 +_c E)_p$$

Assume $\mathtt{fix}_{m+1}\ x = 1 +_c E\downarrow$. By definition,

$$\mathtt{fix}_{m+1}\ x = 1 +_c E = 1 +_c E[\mathtt{fix}_m\ x = 1 +_c E/x].$$

Since $\mathtt{fix}_{m+1}\ x = 1 +_c E\downarrow$, by complexity equality convergence (Lemma 3.6),

$$1 +_c E[\mathtt{fix}_m\ x = 1 +_c E/x]\downarrow$$

Then by complexity convergence cost weakening (Lemma 3.7),

$$E[\mathtt{fix}_m\ x = 1 +_c E/x]\downarrow$$

By the outer induction hypothesis, the statement holds at m for any type τ.

$$\mathsf{fix}_m\ x = e \sqsubseteq_{\mathsf{nat}} \mathtt{fix}_m\ x = 1 +_c E$$

Since we assumed $x : \mathsf{nat} \vdash e \sqsubseteq_{\mathsf{nat}} x : \|\mathsf{nat}\| \vdash E : \|\mathsf{nat}\|$, by definition,

$$e[\mathsf{fix}_m\ x = e/x] \sqsubseteq_{\mathsf{nat}} E[\mathtt{fix}_m\ x = 1 +_c E/x].$$

Therefore $e[\mathsf{fix}_m\ x = e/x]$ must evaluate to a value; say $e[\mathsf{fix}_m\ x = e/x]\downarrow^{m_1} v$. Then

$$m_1 \leq E_c[\mathtt{fix}_m\ x = 1 +_c E/x]$$

$$v \leq E_p[\mathtt{fix}_m\ x = 1 +_c E/x]$$

Now consider the evaluation at $\mathsf{fix}_{m+1}\ x = e$:

$$\frac{e[\mathsf{fix}_m\ x = e/x]\downarrow^{m_1} v}{\mathsf{fix}_{m+1}\ x = e\downarrow^{m_1+1} v}$$

Therefore,

$$\begin{aligned} 1 + m_1 &\leq 1 + E_c[\mathtt{fix}_m\ x = 1 +_c E/x] \\ &= (1 +_c E[\mathtt{fix}_m\ x = 1 +_c E/x])_c \\ &= (\mathtt{fix}_{m+1}\ x = 1 +_c E)_c \\ v &\leq E_p[\mathtt{fix}_m\ x = 1 +_c E/x] = (\mathtt{fix}_{m+1}\ x = 1 +_c E)_p. \end{aligned}$$

Thus,

$$\mathsf{fix}_{m+1}\ x = e \sqsubseteq_{\mathsf{nat}} \mathtt{fix}_{m+1}\ x = 1 +_c E$$

SUBCASE: $\tau = \tau_0 \to \tau_1$. Assume $x : \tau_0 \to \tau_1 \vdash e \sqsubseteq_{\tau_0 \to \tau_1} x : \|\tau_0 \to \tau_1\| \vdash E : \|\tau_0 \to \tau_1\|$. Assume $e_0 \sqsubseteq_{\tau_0} E_0$, where x is not a free variable in neither e_0 or E_0. We need to show

$$(\mathsf{fix}_{m+1}\ x = e)\ e_0 \sqsubseteq_{\tau_1} (\mathtt{fix}_{m+1}\ x = 1 +_c E) \cdot E_0$$

By the inner induction hypothesis, the statement holds at $m + 1$ and lower type τ_1.

$$\mathsf{fix}_{m+1}\ x = (e\ e_0) \sqsubseteq_{\tau_1} \mathtt{fix}_{m+1}\ x = (1 +_c E) \cdot E_0$$

Therefore it suffices to show

$$(\mathsf{fix}_{m+1}\ x = e)\ e_0 \backsimeq \mathsf{fix}_{m+1}\ x = e\ e_0$$

and

$$(\mathtt{fix}_{m+1}\ x = 1 +_c E) \cdot E_0 = \mathtt{fix}_{m+1}\ x = ((1 +_c E) \cdot E_0).$$

The result follows from source language equivalence (Lemma 3.12) and complexity language equivalence (Lemma 3.10). To show the former, suppose $(\mathsf{fix}_{m+1}\ x = e)\ e_0 \downarrow^{m'} v$. It must be the case that $m' = 1 + m_1 + m_2$, where

$$\frac{\dfrac{e[\mathsf{fix}_m\ x = e/x] \downarrow^{m_1} \lambda y.e'}{\mathsf{fix}_{m+1}\ x = e \downarrow^{1+m_1} \lambda y.e'} \qquad e'[e_0/y] \downarrow^{m_2} v}{(\mathsf{fix}_{m+1}\ x = e)\ e_0 \downarrow^{1+m_1+m_2} v}$$

Since x is not a free variable in e_0,

$$\frac{\dfrac{e[\mathsf{fix}_m\ x = e/x] \downarrow^{m_1} \lambda y.e' \qquad e'[e_0/y] \downarrow^{m_2} v}{(e\ e_0)[\mathsf{fix}_m\ x = e/x] \downarrow^{m_1+m_2} v}}{\mathsf{fix}_{m+1}\ x = e\ e_0 \downarrow^{1+m_1+m_2} v}$$

The other direction is symmetric. We now show the latter

Let E^* represent $E[\mathsf{fix}_m\ x = 1 +_c E/x]$.

$$\begin{aligned}
(\mathsf{fix}_{m+1}\ x = 1 +_c E) \cdot E_0 &= (\mathsf{fix}_{m+1}\ x = \langle 1 + E_c, E_p\rangle) \cdot E_0) \\
&= (\langle 1 + E_c, E_p\rangle[\mathsf{fix}_m\ x = 1 +_c E/x]) \cdot E_0) \\
&= \langle 1 + E_c^*, E_p^*\rangle \cdot E_0 \\
&= \langle 1 + E_c^* + (E_p^*\ E_{0p})_c, (E_p^*\ E_{0p})_p\rangle \\
(\text{Since } x \notin FV(E_0)) &= \langle 1 + E_c^* + (E_p^*\ E_{0p}^*)_c, (E_p^*\ E_{0p}^*)_p\rangle \\
&= \langle 1 + E_c + (E_p\ E_{0p})_c, (E_p\ E_{0p})_p\rangle[\mathsf{fix}_m\ x = 1 +_c E/x] \\
&= (\langle 1 + E_c, E_p\rangle \cdot E_0)[\mathsf{fix}_m\ x = 1 +_c E/x] \\
&= ((1 +_c E) \cdot E_0)[\mathsf{fix}_m\ x = 1 +_c E/x] \\
&= \mathtt{fix}_{m+1}\ x = ((1 +_c E) \cdot E_0)
\end{aligned}$$

The proof follows. □

Now we present the statement of Compactness for both the source and complexity languages. While we state a slightly modified Compactness theorem that explicitly states the cost of evaluating the two expressions are equal and corresponding corollary for the source language, we omit the proofs and instead refer our readers to the proof presented by **?**.

THEOREM 3.15 (Source Language Compactness). *Suppose that* $y : \tau \vdash e : \mathsf{nat}$*, where* $y \notin FV(\mathsf{fix}\ x = e_x)$*. If* $e[\mathsf{fix}\ x = e_x/y] \downarrow^k n$*, then there exists* $m \geq 0$ *such that* $e[\mathsf{fix}_m\ x = e_x/y] \downarrow^k n$*.*

COROLLARY 3.16. *For* $e : \mathsf{nat}$*,*

$$e[\mathsf{fix}\ x = e_x/y] \downarrow^k n \text{ iff there exists } m \geq 0 \text{ such that } e[\mathsf{fix}_m\ x = e_x/y] \downarrow^k n,$$

THEOREM 3.17 (Complexity Language Compactness). *Suppose that* $y : T \vdash e : \mathbf{nat}$, *where* $y \notin FV(\mathbf{fix}\ x = E_x)$. *If* $E[\mathbf{fix}\ x = E_x/y] \downarrow$, *then there exists* $m \geq 0$ *such that* $E[\mathbf{fix}_m\ x = E_x/y] \downarrow$ *and* $E[\mathbf{fix}\ x = E_x/y] = E[\mathbf{fix}_m\ x = E_x/y]$.

COROLLARY 3.18. *For* $E : \mathbf{nat}$,

$$E[\mathbf{fix}\ x_x = 1 +_c E/y] \downarrow \textit{ iff there exists } m \geq 0 \textit{ such that } E[\mathbf{fix}_m\ x_x = 1 +_c E/y] \downarrow$$
$$\textit{and } E[\mathbf{fix}\ x_x = 1 +_c E/y] = E[\mathbf{fix}_m\ x_x = 1 +_c E/y].$$

DEFINITION 3.19 (Applicative Contexts). *We define an applicative context* a *in the source language to be either a hole* $\circ$ *or an application of the form* $a\ e$, *where* $a : \sigma \hookrightarrow \tau$ *is an applicative context. We define similarly, an applicative context* A *in the complexity language to be either a hole* $\circ$ *or an applicative of the form* $A \cdot E$, *where* $A : \|\sigma\| \hookrightarrow \|\tau\|$ *is an applicative context. We define bounding of source language applicative contexts and complexity language applicative contexts as follows:*

(1) $\circ \sqsubseteq_{\sigma \hookrightarrow \sigma} \circ$
(2) *if* $a^{\sigma \hookrightarrow (\tau_0 \to \tau_1)} \sqsubseteq_{\sigma \hookrightarrow (\tau_0 \to \tau_1)} A^{\|\sigma\| \hookrightarrow \|\tau_0 \to \tau_1\|}$ *and* $e^{\tau_0} \sqsubseteq_{\tau_0} E^{\|\tau_0\|}$, *then* $(a\ e)^{\sigma \hookrightarrow \tau_1} \sqsubseteq_{\sigma \hookrightarrow \tau_1} (A \cdot E)^{\|\sigma\| \hookrightarrow \|\tau_1\|}$.

THEOREM 3.20 (Fixed Point Induction). *Suppose that* $x : \tau \vdash e : \tau$. *If*

$$x : \tau \vdash e : \tau \sqsubseteq_\tau x : \|\tau\| \vdash E : \|\tau\|$$

and

$$(\forall m \geq 0)\ \mathit{fix}_m\ x = e \sqsubseteq_\tau \mathbf{fix}_m\ x = 1 +_c E,$$

then

$$\mathit{fix}\ x = e \sqsubseteq_\tau \mathbf{fix}\ x = 1 +_c E.$$

PROOF. We prove by induction on the structure of τ, if $a \sqsubseteq_{\sigma \hookrightarrow \tau} A$ and

$$\forall m \geq 0, a\{\mathsf{fix}_m\ x = e\} \sqsubseteq_\tau A\{\mathtt{fix}_m\ x = 1 +_c E\},$$

then

$$a\{\mathsf{fix}\ x = e\} \sqsubseteq_\tau A\{\mathtt{fix}\ x = 1 +_c E\}.$$

Choosing $a = A = \circ$ so that $\sigma = \tau$ completes the proof.

CASE: $\tau = \mathsf{nat}$. Suppose $a \sqsubseteq_{\sigma \to \mathsf{nat}} A$, $e \sqsubseteq E$, and

$$\forall m, a\{\mathsf{fix}_m\ x = e\} \sqsubseteq_{\mathsf{nat}} A\{\mathtt{fix}_m\ x = 1 +_c E\}. \tag{1}$$

We need to show

$$a\{\mathsf{fix}\ x = e\} \sqsubseteq_{\mathsf{nat}} A\{\mathtt{fix}\ \ x = 1 +_c E\}.$$

By definition, it suffices to show that if $A\{\mathtt{fix}\ \ x = 1 +_c E\} \downarrow$, then

$$\mathsf{cost}(a\{\mathsf{fix}\ x = e\}) \leq (A\{\mathtt{fix}\ \ x = 1 +_c E\})_c; \text{ and}$$

$$\mathsf{val}(a\{\mathsf{fix}\ x = e\}) \leq (A\{\mathtt{fix}\ \ x = 1 +_c E\})_p$$

Suppose $A\{\mathtt{fix}\ \ x = 1 +_c E\} \downarrow$. Then by complexity language compactness (Corollary 3.18), there exists $m \geq 0$ such that,

$$A\{\mathtt{fix}_m\ x = 1 +_c E\} \downarrow \tag{2}$$

and

$$A\{\mathtt{fix}\ \ x = 1 +_c E\} = A\{\mathtt{fix}_m\ x = 1 +_c E\}$$

Then by complexity language equivalence (Lemma 3.10), it suffices to show that

$$\mathsf{cost}(a\{\mathsf{fix}\ x = e\}) \leq (A\{\mathtt{fix}_m\ x = 1 +_c E\})_c; \text{ and}$$

$$\mathsf{val}(a\{\mathsf{fix}\ x = e\}) \leq (A\{\mathtt{fix}_m\ x = 1 +_c E\})_p$$

From (1) and (2),

$$\mathsf{cost}(a\{\mathsf{fix}_m\ x = e\}) \leq (A\{\mathtt{fix}_m\ x = 1 +_c E\})_c; \text{ and}$$

$$\mathsf{val}(a\{\mathsf{fix}_m\ x = e\}) \leq (A\{\mathtt{fix}_m\ x = 1 +_c E\})_p$$

By source language compactness (Corollary 3.16) in the other (trivial) direction,

$$\mathsf{cost}(a\{\mathsf{fix}\ x = e\}) = \mathsf{cost}(a\{\mathsf{fix}_m\ x = e\}) \leq (A\{\mathtt{fix}_m\ x = 1 +_c E\})_c; \text{ and}$$

$$\mathsf{val}(a\{\mathsf{fix}\ x = e\}) = \mathsf{val}(a\{\mathsf{fix}_m\ x = e\}) \leq (A\{\mathtt{fix}_m\ x = 1 +_c E\})_p$$

and we have shown the desired result.

CASE: $\tau = \tau_0 \to \tau_1$. Suppose $a \sqsubseteq_{\sigma \hookrightarrow \tau_0 \to \tau_1} a$, $e \sqsubseteq_{\tau_0} E$, and

$$\forall m, a\{\mathsf{fix}_m\ x = e\} \sqsubseteq_{\tau_0 \to \tau_1} A\{\mathtt{fix}_m\ x = 1 +_c E\}. \tag{3}$$

We need to show

$$a\{\mathsf{fix}\ x = e\} \sqsubseteq_{\tau_0 \to \tau_1} A\{\mathtt{fix}\ \ x = 1 +_c E\}.$$

By definition, it suffices to show that for a fixed $e' \sqsubseteq_{\tau_0} E'$,

$$a\{\mathsf{fix}\ x = e\}\ e' \sqsubseteq_{\tau_1} A\{\mathtt{fix}\ \ x = 1 +_c E\} \cdot E'.$$

Note that

$$a\{\mathsf{fix}\ x = e\}\ e' = (a\ e')\{\mathsf{fix}\ x = e\}$$

$$A\{\mathtt{fix}\ \ x = 1 +_c E\} \cdot E' = (A \cdot E')\{\mathtt{fix}\ \ x = 1 +_c E\}.$$

where $a\ e' : \sigma \hookrightarrow \tau_1$ and $A \cdot E' : \|\sigma\| \hookrightarrow \|\tau_1\|$ Then by the induction hypothesis, it suffices to show that

$$\forall m, (a\ e')\{\mathsf{fix}_m\ x = e\} \sqsubseteq_{\tau_1} (A \cdot E')\{\mathtt{fix}_m\ x = 1 +_c E\}; \text{ i.e.,}$$

$$a\{\mathsf{fix}_m\ x = e\}\ e' \sqsubseteq_{\tau_1} A\{\mathtt{fix}_m\ x = 1 +_c E\} \cdot E'$$

It follows from (3) and $e' \sqsubseteq_{\tau_0} E'$ that

$$a\{\mathsf{fix}_m\ x = e\}\ e' \sqsubseteq_{\tau_1} A\{\mathtt{fix}_m\ x = 1 +_c E\} \cdot E'$$

as desired. □

Now we state some facts about $\vee$. Note that $\vee$ at type nat is the maximum operator for the natural numbers. We extend $\vee$ for pairs and functions in the traditional way as described by the equational semantics in Chapter 2.

LEMMA 3.21 (Reflexitivity). $E \vee_\tau E = E$

LEMMA 3.22 (Absorption). *For* $E_1, E_2, E_3 : \mathtt{nat}$,

$$((E_1 \vee E_2) + E_3) \vee (E_1 + E_3) = ((E_1 \vee E_2) + E_3)$$

and

$$((E_1 \vee E_2) + E_3) \vee (E_2 + E_3) = ((E_1 \vee E_2) + E_3)$$

LEMMA 3.23 (Max Weakening). *If* $e \sqsubseteq_\tau E$, *then* $\forall E', e \sqsubseteq_\tau E \vee_{\|\tau\|} E'$.

PROOF. We prove by induction on τ.

CASE: $\tau = \mathsf{nat}$. Assume $E \vee_{\mathsf{nat}\times\mathsf{nat}} E' \downarrow$. We need to show

$$\begin{aligned}
\mathsf{cost}(e) &\leq E \vee E' \\
&= \langle \pi_0 E \vee_{\mathsf{nat}} \pi_0 E', \pi_1 E \vee_{\mathsf{nat}} \pi_1 E' \rangle_c \\
&= \pi_0 E \vee_{\mathsf{nat}} \pi_0 E' \\
\mathsf{val}(e) &= \langle \pi_0 E \vee_{\mathsf{nat}} \pi_0 E', \pi_1 E \vee_{\mathsf{nat}} \pi_1 E' \rangle_p \\
&= \pi_1 E \vee_{\mathsf{nat}} \pi_1 E'.
\end{aligned}$$

Since $e \sqsubseteq_{\mathsf{nat}} E$,

$$\begin{aligned}
\mathsf{cost}(e) &\leq E_c = \pi_0 E \\
\mathsf{val}(e) &\leq E_p = \pi_1 E.
\end{aligned}$$

the result follows immediately by the definition of $\vee$ on natural numbers.

CASE: $\tau = \tau_0 \to \tau_1$. Assume $e_0 \sqsubseteq_{\tau_0} E_0$. We need to show

$$\begin{aligned}
&e\ e_0 \sqsubseteq_{\tau_1} (E \vee_{\mathsf{nat}\times(\|\tau_0\|\to\|\tau_1\|)} E') \cdot E_0 \\
&\quad = \langle \pi_0 E \vee_{\mathsf{nat}} \pi_0 E', \pi_1 E \vee_{\|\tau_0\|\to\|\tau_1\|} \pi_1 E' \rangle \cdot E_0 \\
&\quad = (E_c \vee_{\mathsf{nat}} E'_c) +_c (E_p \vee_{\|\tau_0\|\to\|\tau_1\|} E'_p) E_0 \\
&\quad = (E_c \vee_{\mathsf{nat}} E'_c) +_c \lambda x.(E_p\ x \vee_{\|\tau_0\|\to\|\tau_1\|} E'_p\ x)\ E_0 \\
&\quad = (E_c \vee_{\mathsf{nat}} E'_c) +_c (E_p\ E_0 \vee_{\|\tau_1\|} E'_p\ E_0) \\
&\quad = \langle (E_c \vee_{\mathsf{nat}} E'_c) + (E_p\ E_0 \vee_{\|\tau_1\|} E'_p\ E_0)_c, (E_p\ E_0 \vee_{\|\tau_1\|} E'_p\ E_0)_p \rangle \\
&\quad = \langle (E_c \vee_{\mathsf{nat}} E'_c) + \langle (E_p\ E_0)_c \vee_{\mathsf{nat}} (E'_p\ E_0)_c, (E_p\ E_0)_p \vee_{\mathsf{nat}} (E'_p\ E_0)_p \rangle_c, \\
&\qquad\qquad \langle (E_p\ E_0)_c \vee_{\langle\!\langle \mathsf{nat} \rangle\!\rangle} (E'_p\ E_0)_c, (E_p\ E_0)_p \vee_{\mathsf{nat}} (E'_p\ E_0)_p \rangle_p \rangle
\end{aligned}$$

$$= \langle (E_c \vee_{\mathsf{nat}} E'_c) + ((E_p\ E_0)_c \vee_{\mathsf{nat}} (E'_p\ E_0)_c), (E_p\ E_0)_p \vee_{\langle\!\langle \mathsf{nat} \rangle\!\rangle} (E'_p\ E_0)_p \rangle$$

$$= \langle (E_c \vee_{\mathsf{nat}} E'_c) + (E_p\ E_0)_c) \vee_{\mathsf{nat}} ((E_c \vee_{\mathsf{nat}} E'_c) + (E'_p\ E_0)_c),$$

$$(E_p\ E_0)_p \vee_{\langle\!\langle \tau_1 \rangle\!\rangle} (E'_p\ E_0)_p \rangle$$

Since $e \sqsubseteq_{\tau_0 \to \tau_1} E$,

$$e\ e_0 \sqsubseteq_{\tau_1} E \cdot E_0.$$

Then by the induction hypothesis,

$$e\ e_0 \sqsubseteq_{\tau_1} (E \cdot E_0) \vee_{\|\tau_1\|} ((E \vee_{\mathsf{nat} \times (\|\tau_0\| \to \|\tau_1\|)} E') \cdot E_0)$$

$$= (E_c +_c E_p\ E_0) \vee_{\mathsf{nat} \times \langle\!\langle \tau_1 \rangle\!\rangle} ((E \vee_{\mathsf{nat} \times (\|\tau_0\| \to \|\tau_1\|)} E') \cdot E_0)$$

Noting that $((E \vee_{\mathsf{nat} \times (\|\tau_0\| \to \|\tau_1\|)} E') \cdot E_0)$ is the same as above,

$$= \langle ((E_c \vee_{\mathsf{nat}} E'_c) + (E_p\ E_0)_c) \vee_{\mathsf{nat}} ((E_c \vee_{\mathsf{nat}} E'_c) + (E'_p\ E_0)_c) \vee_{\mathsf{nat}} (E_c + (E_p\ E_0)_c),$$

$$(E_p\ E_0)_p \vee_{\langle\!\langle \tau_1 \rangle\!\rangle} (E_p\ E_0)_p \vee_{\langle\!\langle \tau_1 \rangle\!\rangle} (E'_p\ E_0)_p \rangle$$

$$= \langle ((E_c \vee_{\mathsf{nat}} E'_c) + (E_p\ E_0)_c) \vee_{\mathsf{nat}} ((E_c \vee_{\mathsf{nat}} E'_c) + (E'_p\ E_0)_c),$$

(by reflexitivity and absorption) $(E_p\ E_0)_p \vee_{\langle\!\langle \tau_1 \rangle\!\rangle} (E'_p\ E_0)_p \rangle$

and we have shown the desired result. □

LEMMA 3.24 (Value Nat Bounding). *If $k \leq l$, then $k \sqsubseteq_{\mathsf{nat}} \langle 0, l \rangle$.*

LEMMA 3.25 (Nat Case). *If $e \sqsubseteq_{\mathsf{nat}} E$, $e_0 \sqsubseteq_\tau E_0$, $\forall k, e_0 \sqsubseteq_\tau E_1[\langle 0, k \rangle / n]$, and $e_1 \sqsubseteq_\tau E_1$, then*

$$\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \sqsubseteq_\tau E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1$$

PROOF. We prove by induction on τ.

CASE: $\tau = \mathsf{nat}$. If $E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1 \uparrow$ then by divergence we are done. So assume $E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1 \downarrow$. We need to show

$$\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \sqsubseteq_{\mathsf{nat}} E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1$$

By the complexity step rules, we consider two cases.

SUBCASE 1: $E_p = 0$. Since we assumed $E_c +_c \texttt{ifz}\ E_p\ \texttt{then}\ E_0\ \texttt{else}\ E_1 \downarrow$, by complexity cost weakening,

$$\texttt{ifz}\ E_p\ \texttt{then}\ E_0\ \texttt{else}\ E_1 \downarrow$$

Then by complexity convergence case weakening (Lemma 3.8) $E \downarrow$ and $E_0 \downarrow$. Since

$$E_c +_c \texttt{ifz}\ 0\ \texttt{then}\ E_0\ \texttt{else}\ E_1 = E_c +_c E_0$$

by complexity language equivalence (Lemma 3.10) it suffices to show

$$\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \sqsubseteq_{\mathsf{nat}} E_c +_c E_0.$$

It suffices to show

$$\mathsf{cost}(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)) \leq E_c +_c (E_0)_c$$

$$\mathsf{val}(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)) \leq (E_0)_p.$$

Now consider the evaluation at $\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)$:

$$\frac{e \downarrow^{m_1} 0 \qquad e_0 \downarrow^{m_2} v}{\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \downarrow^{m_1+m_2} v}$$

Since $e \sqsubseteq_{\mathsf{nat}} E$,

$$m_1 = \mathsf{cost}(e) \leq E_c$$

$$\mathsf{val}(e) \leq E_p = 0.$$

Since $e_0 \sqsubseteq_{\mathsf{nat}} E_0$,

$$\mathsf{cost}(e_0) \leq (E_0)_c$$

$$\mathsf{val}(e_0) \leq (E_0)_p.$$

and the result follows.

SUBCASE 2: $E_p > 0$. Since we assumed $E_c +_c \texttt{ifz}\ E_p\ \texttt{then}\ E_0\ \texttt{else}\ E_1 \downarrow$, by complexity cost weakening,

$$\texttt{ifz}\ E_p\ \texttt{then}\ E_0\ \texttt{else}\ E_1 \downarrow$$

Then by complexity convergence case weakening (Lemma 3.8) $E\downarrow$ and $E_1\downarrow$. Since

$$E_c +_c \texttt{ifz}\ E_p\ \texttt{then}\ E_0\ \texttt{else}\ E_1 = E_c +_c E_1[\langle 0, E_p - 1\rangle/n]$$

by complexity language equivalence it suffices to show

$$\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \sqsubseteq_{\mathsf{nat}} E_c +_c E_1[\langle 0, E_p - 1\rangle/n]$$

It suffices to show

$$\mathsf{cost}(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)) \leq E_c +_c (E_1\langle 0, E_p - 1\rangle/n])_c$$

$$\mathsf{val}(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)) \leq (E_1\langle 0, E_p - 1\rangle/n])_p.$$

Since $e \sqsubseteq_{\mathsf{nat}} E$, we know that

$$\mathsf{cost}(e) \leq E_c$$

$$\mathsf{val}(e) \leq E_p > 0.$$

Since $\mathsf{val}(e) \geq 0$, we consider two subcases:

SUBCASE i: $\mathsf{val}(e) = 0$. Consider the evaluation at $\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)$:

$$\frac{e \downarrow^{m_1} 0 \qquad e_0 \downarrow^{m_2} v}{\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \downarrow^{m_1+m_2} v}$$

Since $e_0 \sqsubseteq_{\mathsf{nat}} E_1$ and n does not occur freely in e_0,

$$\mathsf{cost}(e_0) \leq (E_1[\langle 0, E_p - 1\rangle/n])_c$$

$$\mathsf{val}(e_0) \leq (E_1[\langle 0, E_p - 1\rangle/n])_p.$$

and the result follows.

SUBCASE ii: $\mathsf{val}(e) > 0$ Consider the evaluation at $\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)$:

$$\frac{e \downarrow^{m_1} S(n_0) \qquad e_1[n_0/n] \downarrow^{m_2} v}{\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \downarrow^{m_1+m_2} v}$$

We need to show $n_0 \leq E_p - 1$, which we have since $S(n_0) \leq E_p$. Then by value nat bounding (Lemma 3.24),

$$n_0 \sqsubseteq_{\mathsf{nat}} \langle 0, E_p - 1\rangle$$

Then since $e_1 \sqsubseteq_{\mathsf{nat}} E_1$,

$$e_1[n_0/n] \sqsubseteq_{\mathsf{nat}} E_1[\langle 0, E_p - 1\rangle/n]$$

Thus,

$$\mathsf{cost}(e_1[n_0/n]) \leq (E_1\langle 0, E_p - 1\rangle/n])_c$$

$$\mathsf{val}(e_1[n_0/n]) \leq (E_1\langle 0, E_p - 1\rangle/n])_p.$$

and the result follows.

CASE: $\tau = \tau_0 \to \tau_1$. Assume $e' \sqsubseteq_{\tau_0} E'$ where $n \notin FV(e')$ and $n \notin FV(E')$. We need to show

$$(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1))\ e' \sqsubseteq_{\tau_1} (E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1) \cdot E'$$

$$= (E_c +_c \mathtt{ifz}\ E\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1)_c +_c (\mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1)_p\ E'$$

Since $e_0 \sqsubseteq_{\tau_0 \to \tau_1} E_0$, $e_0 \sqsubseteq_{\tau_0 \to \tau_1} E_1$, and $e_1 \sqsubseteq_{\tau_0 \to \tau_1} E_1$,

$$e_0\ e' \sqsubseteq_{\tau_1} E_0 \cdot E'$$

$$e_0\ e' \sqsubseteq_{\tau_1} E_1 \cdot E'$$

$$e_1\ e' \sqsubseteq_{\tau_1} E_1 \cdot E'.$$

Then by the induction hypothesis,

$$\mathsf{case}\ (e,\ 0 \mapsto e_0\ e' \mid S(n) \mapsto e_1\ e') \sqsubseteq_{\tau_1} E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0 \cdot E'\ \mathtt{else}\ E_1 \cdot E'.$$

If we show that

$$(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1))\ e' \backsimeq \mathsf{case}\ (e,\ 0 \mapsto e_0\ e' \mid S(n) \mapsto e_1\ e')$$

and

$$E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0 \cdot E'\ \mathtt{else}\ E_1 \cdot E' = (E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1) \cdot E'.$$

Then by source language equivalence and complexity language equivalence,

$$(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1))\ e' \sqsubseteq_{\tau_1} (E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1) \cdot E',$$

and we have shown the desired result. We first prove the former. Assume $(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1))\ e' \downarrow^{m'} v$. We consider two cases.

CASE 1: It could be the case that $m' = m_1 + m_2 + m_3$, where

$$\frac{\dfrac{e \downarrow^{m_1} 0 \qquad e_0 \downarrow^{m_2} \lambda x.e_0'}{\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \downarrow^{m_1+m_2} \lambda x.e_0'} \qquad e_0'[e'/x] \downarrow^{m_3} v}{(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1))\ e' \downarrow^{m_1+m_2+m_3} v}$$

then

$$\frac{e \downarrow^{m_1} 0 \qquad \dfrac{e_0 \downarrow^{m_2} \lambda x.e_0' \qquad e_0'[e'/x] \downarrow^{m_3} v}{e_0\ e' \downarrow^{m_2+m_3} v}}{(\mathsf{case}\ (e,\ 0 \mapsto e_0\ e' \mid S(n) \mapsto e_1\ e')) \downarrow^{m_1+m_2+m_3} v}$$

CASE 2: It could be the case that $m' = m_1 + m_2 + m_3$, where

$$\frac{\dfrac{e \downarrow^{m_1} S(n_0) \qquad e_1[n_0/n] \downarrow^{m_2} \lambda x.e_1'}{\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) \downarrow^{m_1+m_2} \lambda x.e_1'} \qquad e_1'[e'/x] \downarrow^{m_3} v}{(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1))\ e' \downarrow^{m_1+m_2+m_3} v}$$

then

$$\frac{e \downarrow^{m_1} S(n_0) \qquad \dfrac{e_1[n_0/n] \downarrow^{m_2} \lambda x.e_1' \qquad e_1'[e'/x] \downarrow^{m_3} v}{e_1\ e'[n_0/n] \downarrow^{m_2+m_3} v}}{(\mathsf{case}\ (e,\ 0 \mapsto e_0\ e' \mid S(n) \mapsto e_1\ e')) \downarrow^{m_1+m_2+m_3} v}$$

The two cases for the other direction is symmetric. Now we prove the latter. We consider two cases.

CASE 1: $E_p = 0$

$$\begin{aligned}
&E_c +_c \mathtt{ifz}\ E_p\ \mathtt{then}\ E_0 \cdot E'\ \mathtt{else}\ E_1 \cdot E' \\
&= E_c +_c \mathtt{ifz}\ 0\ \mathtt{then}\ E_0 \cdot E'\ \mathtt{else}\ E_1 \cdot E' \\
&= E_c +_c (E_0 \cdot E') \\
&= E_c +_c (E_{0c} +_c E_{0p}\ E') \\
&= E_c +_c \langle E_{0c} + (E_{0p}\ E')_c, (E_{0p}\ E')_p \rangle \\
&= \langle E_c + E_{0c} + (E_{0p}\ E')_c, (E_{0p}\ E')_p \rangle \\
&= (E_c +_c E_0)_c +_c (E_{0p}\ E') \\
&= (E_c +_c \mathtt{ifz}\ 0\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1)_c +_c ((\mathtt{ifz}\ 0\ \mathtt{then}\ E_0\ \mathtt{else}\ E_1)_p\ E')
\end{aligned}$$

$$= \langle E_c + (\texttt{ifz } 0 \texttt{ then } E_0 \texttt{ else } E_1)_c, (\texttt{ifz } 0 \texttt{ then } E_0 \texttt{ else } E_1)_p \rangle \cdot E'$$

$$= (E_c +_c \texttt{ifz } 0 \texttt{ then } E_0 \texttt{ else } E_1) \cdot E'$$

$$= (E_c +_c \texttt{ifz } E_p \texttt{ then } E_0 \texttt{ else } E_1) \cdot E'$$

CASE 2: $E_p \neq 0$

$$E_c +_c \texttt{ifz } E_p \texttt{ then } E_0 \cdot E' \texttt{ else } E_1 \cdot E'$$

$$= E_c +_c (E_1 \cdot E')[\langle 0, E_p - 1\rangle/n]$$

$$= E_c +_c (E_{1c} +_c E_{1p}\ E')[\langle 0, E_p - 1\rangle/n]$$

$$= E_c +_c \langle E_{1c} + (E_{1p}\ E')_c, (E_{1p}\ E')_p\rangle[\langle 0, E_p - 1\rangle/n]$$

$$= \langle E_c + E_{1c} + (E_{1p}\ E')_c, (E_{1p}\ E')_p\rangle[\langle 0, E_p - 1\rangle/n]$$

$$= [(E_c +_c E_1)_c +_c (E_{1p}\ E')][\langle 0, E_p - 1\rangle/n]$$

$$= (E_c +_c E_1[\langle 0, E_p - 1\rangle/n])_c +_c ((E_1[\langle 0, E_p - 1\rangle/n])_p\ E')$$

$$= (E_c +_c \texttt{ifz } E_p \texttt{ then } E_0 \texttt{ else } E_1)_c +_c ((\texttt{ifz } E_p \texttt{ then } E_0 \texttt{ else } E_1)_p\ E')$$

$$= \langle E_c + (\texttt{ifz } E_p \texttt{ then } E_0 \texttt{ else } E_1)_c, (\texttt{ifz } 0 \texttt{ then } E_0 \texttt{ else } E_1)_p\rangle \cdot E'$$

$$= (E_c +_c \texttt{ifz } E_p \texttt{ then } E_0 \texttt{ else } E_1) \cdot E'$$

□

THEOREM 3.26 (Bounding Theorem). *If $\gamma \vdash e : \tau$, then $e \sqsubseteq_\tau \|e\|$.*

PROOF. The proof is by induction on the derivation of $\gamma \vdash e : \tau$. In each case we state the last line of the derivation, taking as given the premises of the typing rules in Figure 1.

CASE: $\gamma, x : \tau \vdash x : \tau$. Assume $\theta \sqsubseteq_{\gamma, x:\tau} \Theta$. We need to show

$$x[\theta] \sqsubseteq_\tau \|x\|[\Theta]$$

$$= x[\Theta]$$

By definition 3.4, we have that

$$\theta(x) \sqsubseteq_\tau \Theta(x)$$

as desired.

CASE: $\gamma \vdash 0 : \mathsf{nat}$. Assume $\theta \sqsubseteq_\gamma \Theta$. We need to show

$$0[\theta] \sqsubseteq_{\mathsf{nat}} \|0\|[\Theta]$$

$$0 \sqsubseteq_{\mathsf{nat}} \langle 0, 0\rangle$$

where $\langle 0, 0\rangle$ is already a complexity language value and $n \downarrow^0 n$. Thus it suffices to show

$$0 \leq 0$$

$$0 \leq 0$$

which is immediate.

CASE: $\gamma \vdash S(e) : \mathsf{nat}$. Assume $\theta \sqsubseteq_\gamma \Theta$. We need to show

$$S(e)[\theta] \sqsubseteq_{\mathsf{nat}} \langle \|e\|_c, \|e\|_p + 1\rangle[\Theta]$$

$$S(e[\theta]) \sqsubseteq_{\mathsf{nat}} \langle (\|e\|[\Theta])_c, (\|e\|[\Theta])_p + 1\rangle.$$

Assume $\langle (\|e\|[\Theta])_c, (\|e\|[\Theta])_p + 1\rangle \downarrow$. It suffices to show

$$\mathsf{cost}(S(e[\theta])) \leq (\|e\|[\Theta])_c$$

$$\mathsf{val}(S(e[\theta])) \leq (\|e\|[\Theta])_p + 1.$$

By the induction hypothesis, $e[\theta] \sqsubseteq_{\mathsf{nat}} \|e\|[\Theta]$. And since we assumed convergence on the right side, by complexity convergence potential weakening (Lemma 3.9), $\|e\|[\Theta] \downarrow$. Consider the evaluation of $S(e[\theta])$:

$$\frac{e[\theta] \downarrow^m n}{S(e[\theta]) \downarrow^m n + 1}$$

Then

$$m = \mathsf{cost}(e[\theta]) \leq (\|e\|[\Theta])_c$$

$$n = \mathsf{val}(e[\theta]) \leq (\|e\|[\Theta])_p.$$

The result follows.

CASE: $\gamma \vdash \mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1) : \tau$. Assume $\theta \sqsubseteq_\gamma \Theta$. We need to show

$$(\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1))[\theta] \sqsubseteq_\tau (\|\mathsf{case}\ (e,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)\|)[\Theta]$$

$$\begin{aligned}\mathsf{case}\ (e[\theta],\ 0 \mapsto e_0[\theta] \mid S(n) \mapsto e_1[n/n, \theta]) &\sqsubseteq_\tau (\|e\|_c +_c \mathtt{ifz}\ \|e_0\|_p\ \mathtt{then}\ \|e_0\| \\ &\qquad \mathtt{else}\ \|e_0\| \vee \|e_1\|[\langle 0, \|e\|_p - 1\rangle/n])[\Theta] \\ &= (\|e\|[\Theta])_c +_c \mathtt{ifz}\ (\|e_0\|[\Theta])_p\ \mathtt{then}\ \|e_0\|[\Theta] \\ &\qquad \mathtt{else}\ \|e_0\|[\Theta] \vee \|e_1\|[\langle 0, \|e\|_p - 1\rangle/n, \Theta]\end{aligned}$$

By the induction hypotheses,

$$e[\theta] \sqsubseteq_{\mathsf{nat}} \|e\|[\Theta]$$

$$e_0[\theta] \sqsubseteq_\tau \|e_0\|[\Theta]$$

$$\text{if } e' \sqsubseteq_{\mathsf{nat}} E', \text{ then } e_1[\theta, e'/n] \sqsubseteq_\tau \|e_1\|[\Theta, E'/n]$$

Then by the second induction hypothesis and max weakening (Lemma 3.23),

$$e_0[\theta] \sqsubseteq_\tau \|e_0\|[\Theta] \vee \|e_1\|[\langle 0, \|e\|_p - 1\rangle/n, \Theta]$$

Since n does not occur freely in either e_0 or $\|e_0\|$, we know that

$$\forall k, e_0[\theta] \sqsubseteq_\tau (\|e_0\|[\Theta] \vee \|e_1\|[\langle 0, \|e\|_p - 1\rangle/n])[\langle 0, k\rangle/n, \Theta]$$

The result follows from nat case (Lemma 3.25).

CASE: $\gamma \vdash \lambda x.e : \tau_0 \to \tau_1$. Assume $\theta \sqsubseteq_\gamma \Theta$. We need to show

$$(\lambda x.e)[\theta] \sqsubseteq_{\tau_0 \to \tau_1} \|\lambda x.e\|[\Theta]$$

$$\begin{aligned}\lambda x.(e[\theta]) &\sqsubseteq_{\tau_0 \to \tau_1} \langle 0, \lambda x.\|e\|\rangle[\Theta] \\ &= \langle 0, \lambda x.\|e\|[\Theta]\rangle\end{aligned}$$

Assume $e' \sqsubseteq_{\tau_0} E'$. It suffices to show

$$\lambda x.(e[\theta])\ e' \sqsubseteq_{\tau_1} \langle 0, \lambda x.\|e\|[\Theta]\rangle \cdot E'$$

By the induction hypothesis, $e[\theta] \sqsubseteq_{\tau_1} \|e\|[\Theta]$ and since $e' \sqsubseteq_{\tau_0} E'$,

$$e[\theta, e'/x] \sqsubseteq_{\tau_1} \|e\|[\Theta, E'/x].$$

So it suffices to show that

$$\lambda x.(e[\theta])\ e' \backsimeq e[\theta, e'/x]$$

and

$$\langle 0, \lambda x.\|e\|[\Theta]\rangle \cdot E' = \|e\|[\Theta, E'/x].$$

We note that the former relies on 0 cost for applications to hold, while the latter is just an unwinding of definitions, and the result follows.

CASE: $\gamma \vdash e_0\ e_1 : \tau$. Assume $\theta \sqsubseteq_\gamma \Theta$. We need to show

$$(e_0\ e_1)[\theta] \sqsubseteq_\tau (\|e_0\| \cdot \|e_1\|)[\Theta]$$

$$(e_0[\theta])\ (e_1[\theta]) \sqsubseteq_\tau \|e_0\|[\Theta] \cdot \|e_1\|[\Theta]$$

$$= \|e_0\|_c[\Theta] +_c \|e_0\|_p[\Theta]\ \|e_1\|[\Theta].$$

By the induction hypotheses,

$$e_0[\theta] \sqsubseteq_{\tau_0 \to \tau} \|e_0\|[\Theta]$$

$$e_1[\theta] \sqsubseteq_{\tau_0} \|e_1\|[\Theta].$$

Then by definition,

$$(e_0[\theta])\ (e_1[\theta]) \sqsubseteq_\tau \|e_0\|[\Theta] \cdot \|e_1\|[\Theta]$$

$$= \|e_0\|_c[\Theta] +_c (\|e_0\|_p[\Theta])\ (\|e_1\|[\Theta]),$$

and the result follows.

CASE: $\gamma \vdash \mathsf{fix}\ x = e : \tau$. Assume $\theta \sqsubseteq_\gamma \Theta$. We need to show

$$(\mathsf{fix}\ x = e)[\theta] \sqsubseteq_\tau (\mathtt{fix}\ \ x = 1 +_c \|e\|)[\Theta]$$

$$\mathsf{fix}\ x = (e[\theta]) \sqsubseteq_\tau \mathtt{fix}\ \ x = (1 +_c \|e\|[\Theta])$$

By the induction hypothesis, $e[\theta] \sqsubseteq_\tau \|e\|[\Theta]$. Then by Lemma 3.14,

$$\forall m \geq 0, \mathsf{fix}_m\ x = (e[\theta]) \sqsubseteq_\tau \mathtt{fix}_m\ x = (1 +_c \|e\|[\Theta])$$

and by fixed point induction (Theorem 3.20),

$$\mathsf{fix}\ x = (e[\theta]) \sqsubseteq_\tau \texttt{fix}\ \ x = (1 +_c \|e\|[\Theta])$$

□

CHAPTER 4

Obstructions

We now discuss the various obstructions encountered while introducing general recursion to the framework. We discuss extending the framework with booleans and the need for the maximum. We also give reasons for not explicitly considering the cost at higher type, choosing a call-by-name source language rather than a call-by-value one, and lack of arbitrary inductive data types. We believe all of these are nontrivial restrictions to the framework presented and require discussion.

1. Booleans

Suppose we extend the source language with booleans: $\mathsf{true}, \mathsf{false}, \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$. We extend the complexity language in a similar fashion: $\mathtt{true}, \mathtt{false}, \mathtt{if}\ \ e\ \mathsf{then}\ e\ \mathsf{else}\ e$. The extraction function would map true and false to the pair of the corresponding complexity boolean value and the size 1, since we cannot distinguish sizes of booleans. Now we need the arbitrary maximum in order to handle "bad" programs. Take for example the following expression e:

$$\mathsf{case}\ (\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 0\ \mathsf{else}\ S(0),\ 0 \mapsto S(S(0)) \mid S(n) \mapsto 0)$$

Although the inner case argument evaluates to 0, its potential evaluates to 1. The outer case statement evaluates to $S(0)$. If we did not take the maximum in the non-zero case, we would incorrectly conclude that the potential of the outer case is 0 and have incorrect bounding.

Taking the maximum in the zero case will be fatal for most reasonable programs. If we unconditionally directly translate $\mathsf{case}\ (es,\ 0 \mapsto e_0 \mid S(n) \mapsto e_1)$ to $\|es\|_c +_c \|e_0\| \vee \|e_1\|$, because one of the arguments to the maximum will be a recursive call, the extraction is almost guaranteed to yield a non-terminating recurrence. So in order to

handle bad programs, we must take the maximum in the non-zero size case; but in order to handle good programs, we must not take the maximum in the zero size case.

2. The Bounding Relation Definition

While extending the bounding relation presented in **?** to handle divergent expressions is somewhat trivial, proving the principle of fixed point induction without modifying the bounding relation to be more like a typical logical relation is difficult. We only consider evaluation of the expression at base type since the property of compactness in PCF does not hold at higher type. The proof of fixed point induction, which makes use of compactness at base type, relies on the induction hypothesis to drive the proof down to a lower type at higher type. Having this extra information about evaluation complicates the standard proof. To illustrate, we let the bounding relation $\sqsubseteq^*$ consider evaluation on the complexity expression at higher type, forcing evaluation on the source expression if it is convergent. That is,

- $e \sqsubseteq^*_{\tau_0 \to \tau_1} E$: If $E \downarrow$, then $e \downarrow$, where
  (1) $\mathsf{cost}(e) \leq E_c$
  (2) if for all e' and E', if $e' \sqsubseteq_{\tau_0} E'$, then $e\ e' \sqsubseteq^*_{\tau_1} E \cdot E'$.

Consider the proof of fixed point induction. The proof for base type remains the same. To show bounding at a higher type, we now have to satisfy the condition that convergence of the complexity expressions forces convergence of the source expression. However, we cannot discuss evaluation without a form of compactness, which no longer holds at higher type, as it did at base type. Thus the bounding relation for higher types cannot contain information about the cost of evaluating it. Instead, we delay counting the cost of evaluating a source language expression until all the way down at base type.

We demonstrate the difference between the "incorrect" bounding relation, $\sqsubseteq^*$, and given bounding relation, $\sqsubseteq$. Take for example this call-by-name source language expression where the operational semantics of the source language charge for every function application:

$$(\lambda x.x)(\lambda y.y) : \mathsf{nat} \to \mathsf{nat}.$$

By our definition of bounding, to show that

$$(\lambda x.x)(\lambda y.y) \sqsubseteq_{\mathsf{nat}\to\mathsf{nat}} E$$

for some E, it is sufficient to assume some e' and E' such that $e' \sqsubseteq_{\mathsf{nat}} E'$ and show that

$$\|(\lambda x.x)(\lambda y.y)\|e' \sqsubseteq_{\mathsf{nat}} E \cdot E'$$

Assume $E = \langle 0, \lambda y.\langle 1, y\rangle\rangle$, $e' = 0$, $E' = \langle 0, 0\rangle$. Then,

$$\begin{aligned}(\lambda x.x)(\lambda y.y)\ 0 &\sqsubseteq_{\mathsf{nat}} \langle 0, \lambda y.\langle 1, y\rangle\rangle \cdot \langle 0, 0\rangle \\ &= (1 + 0 + 0) +_c (\lambda y.\langle 1, y\rangle)\ 0 \\ &= \langle 2, 0\rangle\end{aligned}$$

and

$$(\lambda x.x)(\lambda y.y)\ 0 \to_1 (\lambda y.y)\ 0 \to_1 0.$$

Since $\langle 2, 0\rangle$ is a value and $(\lambda x.x)(\lambda y.y)\ 0 \downarrow^2 0$, we check that the cost component does indeed bound the number of evaluation steps and that the potential component bounds the size of the value; which both do. Now consider the other bounding relation. Instead of applying the source language expression to another expression, we instead evaluate the expression down to a value first, ensuring the cost component correctly bounds the cost of evaluating the expression.

$$(\lambda x.x)(\lambda y.y) \sqsubseteq^*_{\mathsf{nat}\to\mathsf{nat}} \langle 0, \lambda y.\langle 1, y\rangle\rangle$$

and

$$(\lambda x.x)(\lambda y.y) \to_1 \lambda y.y.$$

We can clearly see that the cost component does not actually bound the cost of evaluating the source language expression. The cost is hidden within the inner abstraction and is never considered. Instead of choosing an arbitrary E, we take the actual translation

of the source language expression. The translation gives us

$$\begin{aligned}\|(\lambda x.x)(\lambda y.y)\| &= (1 + \|\lambda x.x\|_c + \|\lambda y.y\|_c) +_c \|\lambda x.x\|_p \ \|\lambda y.y\|_p \\ &= 1 +_c \langle 0, \lambda y.\langle 0, y\rangle\rangle \\ &= \langle 1, \lambda y.\langle 0, y\rangle\rangle\end{aligned}$$

which has the correct components to be bound by both $\sqsubseteq$ and $\sqsubseteq^*$. However we do not have the means necessary to directly discuss bounding on the translation of the source language expressions.

3. Call-By-Name vs. Call-By-Value Source Language

The reason for choosing a call-by-name source language rather than a call-by-value source language as in previous works stems from the previously mentioned constraint that we are no longer explicitly concerning ourselves with the evaluation cost for expressions at higher type. While we no longer have to show convergence for our bounding relation at higher type, we also do not have any additional information about the cost of evaluating the higher type expression. Having a call-by-value source language forces us to reason about the cost at higher types when we wish not to. Again to illustrate, keeping our current definition for the bounding relation, let us assume a call-by-value source language. Take for example the proof of the bounding theorem for the abstraction case, $\gamma \vdash \lambda x.e : \tau_0 \to \tau_1$. The definition for the bounding relation requires us to assume some source language expression e' and complexity language expression E' such that $e' \sqsubseteq_{\tau_0} E'$. Then it is sufficient for us to show that $(\lambda x.e)\ e' \sqsubseteq_{\tau_1} E \cdot E'$. With a call-by-value source language, we first have to evaluate e' to some value v' in k steps. We wish to say that the cost component of E' should bound k, but since our bounding relation no longer guarantees information about cost at higher type, we cannot with certainty. Having a call-by-value source language would also take us to a setting in which potentials of higher type expressions take a potential and return a complexity. Then the information about the cost of evaluating e' to a value, while not explicitly available, is lost being passed on as a potential.

While our bounding relation delays counting the cost of evaluating the complexity language expression until base type, having a call-by-value operational semantics raises conflict. Thus intuitively, not evaluating the expression down to a value before applying the substitution is one method of delaying considering information about cost. Therefore we adopt a call-by-name operational semantics for the source language, directly making the substitution in an application without evaluating the right-hand side expression down to a value first. This choice is not trivial, as it does affect the cost analysis of the programs that we consider. Some functions have different costs depending on whether the operational semantics is call-by-name or call-by-value.

4. Arbitrary Inductive Datatypes

Another formalism presented in **?** based on **?** has already been successful in extending the framework to allow arbitrary inductive datatypes. The formalism presented in this thesis however only allows for natural numbers, where we explicitly give the interpretation for the size of `nat` in the translation. The other two frameworks allow for user-defined datatypes and different interpretations for these datatypes given a few restrictions about the ordering. The reason for restricting the scope of the source and complexity languages is the difficulty of extending the bounding relation that is more like a typical logical relation with arbitrary inductive datatypes.

Another issue that needs to be discussed in order to add arbitrary data types is the discussion of arbitrary maximums. Take for example the case expression in the source language. We have already presented a discussion of the need for taking the maximum of both branches for the second branch in the `ifz` expression in the complexity language. This issue becomes more prominent when we consider arbitrary data types. There is no obvious intuition as to how to handle the different case constructs without losing significant information by taking arbitrary maximums across all the different cases.

CHAPTER 5

Conclusions

We have described a static complexity analysis for a higher-order language with general recursion that yields an upper bound on the evaluation cost of any typeable program in the source language. We now discuss related work on automating the construction of resource bounds from source code, including a few that have inspired this thesis. We also give several future extensions to this formalism that we hope to achieve.

1. Related Work

There is an extensive literature on automating the construction of resource bounds from source code. We focus on recent works that place emphasis on complexity analysis of higher-order programs. The Resource Aware ML Project (RAML) takes an automated complexity analysis approach based on type assignment. **?** presents a framework that automatically infers linear resource bounds for higher-order, polymorphic, recursive programs; exploiting linearity to allow inference. The system only works provided the input program does in fact have a linear resource cost. **?** and **?** extend this work to handle polynomial bounds for only first-order programs, and **?** extend the work to parallel programs. RAML uses a source language in which types are annotated with variables corresponding to resource usage. Then type inference in the annotated system boils down to solving a set of constraints with these variables. A key feature of this work is that it is able to establish tight bounds on programs which require amortized cost analysis.

? presents another framework in which complexity of higher-order functional programs can be automatically analyzed by using existing tools for complexity analysis of

first-order term rewrite systems. The system makes use of defunctionalisation, to transform higher-order programs into first-order rewrite systems such that the complexity of the latter reflects the complexity of the former. Since defunctionalised programs have a recursive structure too difficult for first-order provers, they apply appropriate program transformations; for which they show are at least complexity reflecting if not complexity preserving. This work is a complement rather than an alternative to existing methodologies, making use of preexisting tools by way of program transformations.

The most relevant work to this thesis is the framework for extraction of recurrences from higher-order functional programs with structural recursion presented in **?**. Instead of establishing fixed time bounds in terms of values, the work constructs expressions that bound the cost of the given program in terms of the input size. This thesis is in most part inspired by this work, which introduces the same notion of recurrence as the one used in this formalism. The source language is Gödel's System T augmented with lists of a natural numbers and supports only structural recursion on lists.

Another work based on **?** is the previously mentioned formalism presented in **?** which extends the framework higher-order functional programs including user-defined inductive datatypes. The work soundly allows programmer-specified sizes of datatypes and shows that, with some conditions on sizes, the cost predicted by the extracted recurrence is an upper bound on the number of steps the program takes to evaluate. A key technical notion of the work is a logical relation between the source and complexity languages that gives a bounding relation on inductive types similar to the one presented in this work. The work studies both semantic and syntactic approaches on abstracting values to size and gives a formal account on what it means to extract a recurrence from higher-order programs on inductive data.

2. Conclusions and Further Work

We have described a formalism for static complexity analysis of higher-order programs with general recursion by extracting a recurrence that yields an upper bound on the evaluation cost of said program written in the source language. It proceeds by

applying the extraction function to each source-language program e which returns a program $\|e\|$ which makes cost explicit. We prove a bounding theorem for the extraction function, which guarantees that the cost component of $\|e\|$ is an upper bound on the evaluation cost of e.

The extraction function described in Chapter 2 can be adapted to various cost models. For example, we could charge different costs for different steps rather than only counting the number of unrollings of the fix operator. Ideally, we also wish to allow users to abstract values to sizes and to add their own datatypes soundly.

While the source language now supports general recursion, it lacks user-defined datatypes. An obvious extension would be to support soundly allowing arbitrary inductive data types. The difficulty would lie in being able to extending the bounding relation which relies on compactness on forcing evaluation in the source language assuming evaluation in the complexity language. Doing so will allows us to support inductive data types such as streams, which are essentially coninductively defined values.

A desirable characteristic of the bounding relation would be to be able to reason about the cost of evaluating an expression at higher type. This would allow for us to revert the source language to a call-by-value setting rather than a call-by-name one. A possibility is to define the bounding relation on complexity language expressions applied to a function that forces consideration of all the evaluation cost at base type. While we saw that the extraction function itself does have the correct cost component, we can have bounding for higher order expressions that have the cost hidden within layers of abstractions. So if we are able to collect the costs scattered throughout abstractions and force them to collectively bound the cost of evaluating the source language expression and pose that as a condition, we could still reason about our bounding relation and have information about the cost of evaluating higher type expressions.

Bibliography

Martin Avanzini, Ugo Dal Lago, and Georg Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 152–164. ACM, 2015. doi: 10.1145/2784731.2784753.

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*, PLPV '13, pages 25–34. ACM, 2013. doi: 10.1145/2428116.2428123.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 140–151. ACM, 2015. doi: 10.1145/2784731.2784749.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2013.

Jan Hoffmann and Martin Hofmann. Amortized Resource Analysis with Polynomial Potential - A Static Inference of Polynomial Bounds for Functional Programs. In *In Proceedings of the 19th European Symposium on Programming (ESOP'10)*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer, 2010.

Jan Hoffmann and Zhong Shao. Automatic static cost analysis for parallel programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*, pages 132–157, New York, NY, USA, 2015. Springer-Verlag New York, Inc. doi: 10.1007/978-3-662-46669-8_6.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, November 2012. doi: 10.1145/2362389.2362393.

Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 223–236, New York, NY, USA, 2010. ACM. doi: 10.1145/1706299.1706327.