Wesleyan University The Honors College

Automated Cost Analysis of a Higher-Order Language in Coq

by

Jennifer Paykin
Class of 2012

A thesis submitted to the
faculty of Wesleyan University
in partial fulfillment of the requirements for the
Degree of Bachelor of Arts
with Departmental Honors in Mathematics and Computer Science

Middletown, Connecticut May, 2012

Acknowledgements

First and foremost I would like to thank my thesis advisor Norman Danner, for the opportunity to work on this project. You have always encouraged and expanded my academic pursuits, and your enthusiasm for your research has inspired mine. Thank you for all your hard work over the past two years.

I cannot thank my family enough for their unwavering support of my education and all my pursuits. You have always been there for me, cheering me on. Thank you to Jake for everything: for perpetual encouragement and support, and for keeping me optimistic when work overwhelms me. To all my friends at Wesleyan, especially my fellow thesis writers, thank you for being there to complain to and to celebrate with. You have made Wesleyan a great place to grow.

Abstract

The cost analysis of a program is a function from the size of its input to the number of steps the program takes to run. In this work we address two difficulties associated with constructing cost analyses. First, what is the cost of a higher-order program? If an expression takes a function as input, what is the "size" of that input? If an expression evaluates to a function, what is its "cost"? Second, how can we automate the cost analysis of arbitrary programs?

In this work we develop a static cost analysis system for a functional target language with structural list recursion. The cost of evaluating a target expression is defined to be the size of its evaluation derivation in the operational semantics. The complexity of a target expression is a pair consisting of a cost and a potential, which is a measure of size. A translation function $\|\cdot\|$ maps target expressions to complexities. Our main result is the following soundness theorem: If t is a term in the target language, then the cost component of $\|t\|$ is an upper bound on the cost of evaluating t.

We formalize the mathematical development of the system in the proof assistant Coq, a dependently typed functional programming language and interactive theorem prover. We implement target language typing and evaluation in Coq, as well as an independent complexity language to reason about complexities. By formalizing the proof of the soundness theorem in Coq, we obtain a system which automatically generates certified upper bounds on the complexity of target-language terms.

Contents

CHAPTER 1

Introduction

1.1. Cost Analysis

The analysis of algorithms underlies a vast amount of computer science. The cost of an algorithm can be thought of as a function from its input to the number of steps the algorithm takes to evaluate given that input. In many cases a cost analysis depends not on the value of the input, but on its size. For example, most functions which take a list as input must traverse the entire list, meaning that the algorithm's evaluation cost depends on the length of the input list.

Though cost analyses are extremely well-studied, they are traditionally performed by hand in a relatively ad-hoc manner. A number of tools, such as those by Wilhelm [2005] and Albert et al. [2007], have been developed to compute the cost of imperative programs. Theories regarding the cost analysis of functional and higher-order languages have also been developed, as by Sands [1990], Benzinger [2004] and Rosendahl [1989]. The ability to automatically generate cost bounds for arbitrary programs has important implications for all manner of real-world systems.

The analysis we develop in this thesis is inspired by the work of Danner and Royer [2007], which is discussed in Section 2.4. To obtain some intuition about how the analysis will play out, let us consider a simple example, which will be revisited numerous times in this paper. The `insert` function takes as input an integer x and a sorted list xs of integers and evaluates to the sorted list obtained from inserting x into xs. `insert` can be implemented as follows:

```
fun insert (x, []) = [x]
  | insert (x, y::ys) = if (x<=y) then x::y::ys else y::(insert(x,ys))
```

A simple recursive analysis of the code tells us that the cost of evaluating insert on a list of length 0 (while ignoring the size of the integer input x) is a small constant $c0$, and the cost of evaluation on a list of length $n + 1$ is at most a constant cN plus the cost of evaluating `insert`

on a list of length n. Since `insert` is structurally recursive, it is easy to prove that this upper bound holds by induction on the size of the list. The cost analysis of `insert` is then as follows:

```
fun insert_c 0 = c0
  | insert_c (S n) = cN + insert_c n
```

where `c0` and `cN` are constants. But this is not enough for our purposes. Since we are interested in higher-order languages, we need to consider algorithms which take functions as input. In that case, we need some way to represent the "size" of a function, or an equivalent attribute. In Chapter 3 we develop a notion of *potential* which acts as a measure of size. For type level-0 objects, we can think of potential as natural number size, but for arrow types the situation is different. In essence we use the potential of a function to encompass its cost analysis—the potential of f is a function from potentials p to the complexity of applying f to an input of size p. By *complexity* here we mean a pair of natural number costs and potentials. Hence the following would be the potential of `insert`:

```
fun insert_pot 0 = (c0,0)
  | insert_pot (S n) = (cN,1) + insert_pot n
```

because the potential of the empty list is 0 and the potential of the result list is 1 plus the potential of the recursive call. As a function evaluates to itself in constant time, the complexity of the entire `insert` function is `(1,insert_pot)`.

The goal of Chapter 3 is to construct a translation function which maps expressions in a target language to upper bounds on their complexity.

1.2. Development in Coq

To obtain a usable system from the translation scheme, we implement the system in Coq, a dependently typed programming language and proof assistant (see Coq Develelopment Team 2009). Our goal is to formally verify the proof of soundness for the translation function, so that the implementation produces certified upper bounds on the complexity of target language expressions.

The advantages of cost-analysis certification are numerous. Benzinger [2004] relates an anecdote he calls the *pigeonhole incident* which demonstrates the usefulness of certified complexity bounds. Working in a proof assistant called NUPRL, a research group at Cornell was developing a state-minimization algorithm but encountering exponential running times.[1] The problem with the state-minimization algorithm was eventually traced back to NUPRL's proof of the pigeonhole principle. Since the development of the pigeonhole proof had been so focused on program correctness and automation, its exponential running-time had been overlooked and remained unnoticed for a long stretch of time. The moral of the pigeonhole incident is that even in an environment of experienced developers, it is easy to make small implementation mistakes which blow up the run-time of a complex project.

In our implemented system with certified complexity bounds, it is possible to verify an upper bound on the complexity of a program by simply examining its translation. The problems induced by the pigeonhole incident might have been lessened significantly by the system's exposition of complexity bounds.

We chose to implement the translation scheme in Coq because of its flexibility as both a dependently-typed programming language and proof assistant. As a programming language, Coq uses dependent typing, which allows the type of a program to depend on not just other types, but on terms.

As an example, consider natural number division. The division algorithm tells us that for any $m, n \in \mathbb{N}$ such that $n \neq 0$, there exist $p, r \in \mathbb{N}$ such that $m = pn + r$ and $r < n$. The standard way to compute this p and r is by means of a function $\mathsf{div1}\ (m\ n\colon \mathbf{nat}) : \mathbf{nat} \times \mathbf{nat}$. Then we can prove the following theorem about $\mathsf{div1}$:

Theorem. *If $n \neq 0$ then $\mathsf{div1}(m, n) = (p, r)$ such that $m = pn + r$ and $r < n$.*

[1] NUPRL's constructive type theory allows users to prove theorems and then extract the theorem into a usable program through a process called program synthesis. For example, a theorem of $\forall m, n \in \mathbb{N}, \exists p \in \mathbb{N}, p = m + n$ would be synthesized into a function which, given m and n, produced the p such that $p = m + n$; in other words, the addition function.

Meanwhile, Coq allows us to define a function div2 which incorporates this theorem into the very type of the operator. In particular, we can construct

`Definition` div2 $(m : \textbf{nat})\ (n : \{n' : \textbf{nat} \mid n' > 0\}) : \{\, p : \textbf{nat}\ \&\ \{r : \textbf{nat} \mid m = pn + r \wedge r < n\}\}$.[2]

For one, div2 is only defined on m and n if n is accompanied by a proof that $n \neq 0$. Additionally, the result of applying div2 is a datatype consisting of natural numbers p and r along with a proof that $m = pn + r$ and $r < n$.

The expressiveness of Coq allows a good deal of freedom in implementing our translation system. In addition, Coq is a proof assistant, which means that it allows a user to state and prove theorems by applying a series of tactics. To prove a theorem in Coq, one starts with a goal and works backwards. Tactics reduce goals to smaller, more manageable goals, until they can be solved automatically or by a single tactic.[3]

More information about the theory behind Coq can be found in the Coq Reference Manual [2009].

1.3. Outline

Chapter 2 reviews a small portion of prior work done in the area of automatic complexity analysis. In Chapter 3 we develop a mathematical foundation for the complexity analysis. We describe our target language and its operational semantics, which is based on the typed λ-calculus with integer lists and structural recursion. We also develop a separate but parallel complexity language whose denotational semantics maps into the realm of complexities. The translation map $\|\cdot\|$ sends target expressions e to complexity expressions so that the denotation of $\|e\|$ gives an upper bound on the cost of evaluating the target expression.

In Chapter 4 we describe the implementation of our translation system in Coq. The process of formalizing the target and complexity languages in Coq requires a careful consideration of assumptions and precise definitions. The bounding relation and proof of soundness are also

[2] An element of type $\{a : A \mid P\ a\}$ is an element a of A along with a proof that the proposition P holds on a.

[3] Although Coq is not an automatic theorem prover, it does have some capacity for automation by means of the `auto` tactic and through `Hint` databases.

formalized in Coq, resulting in the systematic production of certified upper bounds for target language expressions.

Finally, Chapter 5 concludes our work with a summary of our main contributions, as well as areas for future work.

CHAPTER 2

Prior Work

2.1. Shultis's Tolls

In the technical report *On the Complexity of Higher-Order Programs*, Shultis [1985] presents one of the first theories for analyzing the complexity of higher-order functions. His goal is to construct a denotational semantics for a simple higher-order language that models both the value and some measure of the cost of an expression. As a cost model Shultis develops a system of "tolls" representing the potential cost of applying an expression to an argument.

The system Shultis develops distinguishes between the simple cost (0-toll) and (for $j \geq 1$) the j-toll of an expression. The cost of an expression e, written t^0_e, is a straightforward measure of the time required to evaluate the expression. For example, the cost of an expression $e_1 + e_2$ is equal to $t^0_{e_1} + t^0_{e_2} + 1$. On the other hand $t^0_{\lambda x.e} = 1$ because a lambda-abstraction cannot be evaluated further. If $j \geq 1$ then the j-toll of e, or t^j_e, measures the complexity of applying e to some value. In particular, the j-toll of $\lambda(x).e$ is a function from a value v to the $(j-1)$-toll of $e[v/x]$. A j-toll is said to be j levels of abstraction away from a simple cost.

The structure of tolls allows for reasoning about the complexity of a higher-order expression in an interesting way. Consider the length function on singly-linked lists:

$$\texttt{length} = \texttt{rec}\, f.\lambda(xs).\texttt{if}\ \mathit{null}(xs)\ \texttt{then}\ 0\ \texttt{else}\ 1 + f(\texttt{tl}(xs))$$

Intuition tells us that the cost of applying `length` to a list of length n should be less than or equal to $n\alpha$ for some constant α. In other words we can write $t^0_{\texttt{length}(xs)} \leq \alpha \times v_{\texttt{length}(xs)}$ where v_e is the value of the expression e. In actuality the toll induction rules can only show that $t^0_{\texttt{length}(xs)} \leq t^0_{xs} + \alpha \times v_{\texttt{length}(xs)}$. Therefore the cost of $\texttt{length}(xs)$ depends not only on

the length of xs but also on the cost of xs itself. Additionally, since evaluation uses a call-by-value semantics, in order to compute $v_{\texttt{length}(xs)}$ one must first evaluate xs entirely. In all, this computation has a lot of baggage which we would like to avoid.

Instead of calculating the 0-toll of $\texttt{length}(xs)$, consider the 1-toll of `length` itself. The 1-toll is a map from values to the cost of applying `length` to those values. Therefore we will be able to prove that

$$t^1_{\texttt{length}} = \lambda{<}vs{>}.2v_{\texttt{length}(vs)} + 2$$

where vs is a list value and comparison is done point-wise. Since vs is a value it is not necessary to evaluate the list when computing $v_{\texttt{length}(vs)}$. This trick of abstracting away the application cost embodies the meaning of the "cost" of `length` better than analyzing $\texttt{length}(xs)$ for an arbitrary list.

The proof of this statement illustrates how recursion is handled in the system. According to the given induction rules, it is sufficient to prove that, given some input ϕ, if $t^0_\phi = 1$ proves that $t^1_\phi = \lambda{<}vs{>}.2v_{\texttt{length } vs} + 2$, then

$$1 + t^1_{\lambda(xs).\texttt{if } null(xs) \texttt{ then } 0 \texttt{ else } 1+\phi(\texttt{tl } xs)} = \lambda{<}vs{>}.2v_{\texttt{length } vs} + 2$$

Assume $t^0_\phi = 1$ and $t^1_\phi = \lambda{<}vs{>}.2v_{\texttt{length } vs} + 2$. We can calculate that

$$\begin{aligned} t^1_{\lambda(xs).\texttt{if } null(xs) \texttt{ then } 0 \texttt{ else } 1+\phi(\texttt{tl } xs)} &= \lambda{<}vs{>}.t^0_{\texttt{if } null(vs) \texttt{ then } 0 \texttt{ else } 1+\phi(\texttt{tl } vs)} \\ &= \lambda{<}vs{>}.1 + \texttt{if } v_{null(vs)} \texttt{ then } t^0_0 \texttt{ else } t^1_0 + t^0_{\phi(\texttt{tl } vs)} \\ &= \lambda{<}vs{>}.1 + \texttt{if } v_{null(vs)} \texttt{ then } 1 \texttt{ else } 1 + t^1_\phi{<}v_{\texttt{tl } vs}{>} \end{aligned}$$

By the hypothesis regarding t^1_ϕ, we can show that this equal to

$$\begin{aligned} &\lambda{<}vs{>}.1 + \texttt{if } v_{null(vs)} \texttt{ then } 1 \texttt{ else } 1 + 2v_{\texttt{length (tl } vs)} + 2 \\ &= \lambda{<}vs{>}.\texttt{if } v_{null(vs)} \texttt{ then } 2 \texttt{ else } 2 + 2v_{\texttt{length } vs} - 2 + 2 \\ &= \lambda{<}vs{>}.2v_{\texttt{length } vs} + 2 \end{aligned}$$

In this example it is not necessary to evaluate the recursive subterm at all since assumptions about its toll are wrapped up in the inductive hypothesis.

Because this system incorporates general recursion, it is not always possible to calculate the toll of an expression, especially if the expression does not evaluate. The computation of

tolls of non-recursive expressions is a straightforward application of syntax-directed inference rules, and therefore could easily be automated. Tolls of recursive expressions are not calculated *per se*, but propositions can be proved about them. Therefore automation of the general system would need some kind of additional knowledge about the tolls of recursive expressions.

2.2. Sands's Translations

In *Calculi for the Time Analysis of Functional Programs*, Sands [1990] takes a different approach in his attempt to represent the cost of an expression. Sands puts forward a translation scheme which takes a program in the target language and returns another target program incorporating some kind of cost information. The motivation behind the translation is to keep the domain of the cost meta-language on firm syntactic and semantic ground. (We will see problems with this grounding in Benzinger [2004].) Another advantage is the flexibility to apply program transformations to either a target program or a translation. Sands develops explicit translations for a number of small languages including an initial first-order language, a curried higher-order language with call-by-value semantics, and a higher-order language using call-by-name semantics. We will focus here on the second language, which implements currying and a call-by-value parameter discipline.

Sands's language does not allow for anonymous function definitions, so a program (see Figure 2.1) consists of a set of recursive function definitions along with a closed expression. The semantics evaluates this closed expression in the context of the function definitions. A corresponding step-counting semantics records the number of non-primitive function calls in the evaluation derivation, and acts as a cost model for the target language. The correctness of the cost translation depends on its correspondence to the step-counting semantics.

Like Shultis, Sands requires some way to measure the cost of applying an arbitrary function to an argument. For each user-defined function f_i, he constructs two corresponding functions f_i' and cf_i such that $f_i'(v_1, \ldots, v_{n_i})$ is the value of f_i applied to the arguments $v_1, \ldots, v_{n_i}$ and $cf_i(v_1, \ldots, v_{n_i})$ is the cost of applying f_i to $v_1, \ldots, v_{n_i}$. We can liken Sands's notation to Shultis's by mapping f_i' to v_{f_i} and cf_i to $t_{f_i}^{n_i}$. But whereas Shultis allows lambda abstractions with associated tolls, in Sands's system the cost (image under $\mathcal{T}$) of any partially applied function is zero.

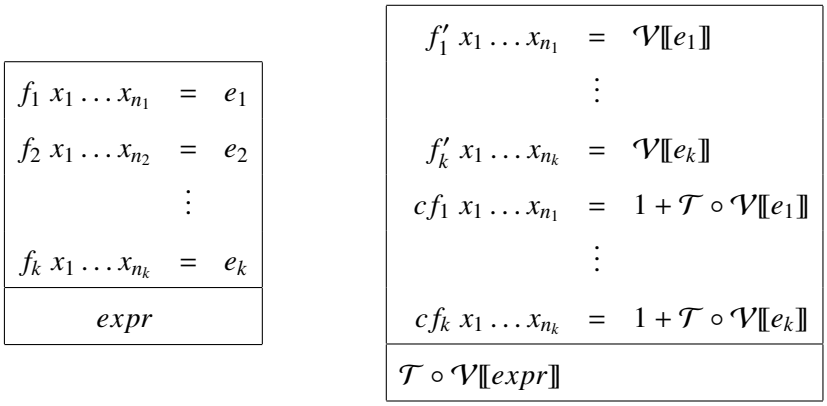

FIGURE 2.1. A program and corresponding cost program in Sands's curried higher-order language. The cost program is defined in terms of translations $\mathcal{T}[\![\cdot]\!]$ and $\mathcal{V}[\![\cdot]\!]$. For an expression e, $\mathcal{V}[\![e]\!]$ contains the value and cost information pertaining to partial function applications in e. $\mathcal{T}[\![e]\!]$ computes the cost of evaluating e, provided e is in the domain of $\mathcal{V}$.

In other words, Sands is measuring only the simple costs of expressions even though arbitrarily complex costs can be represented in the system.

Once the cost-functions (as the cf_i's are called) are generated, Sands attempts to factor them into context-free and context-sensitive parts. A context-free cost function does not depend on the cost of evaluating any of the input. A context-sensitive cost function may depend on this.

As an example, consider the function

$$\texttt{map}\ f\ xs = \texttt{if}\ null(xs)\ \texttt{then}\ [\,]\ \texttt{else}\ cons(f(\texttt{hd}\ xs))(\texttt{map}\ f\ (\texttt{tl}\ xs))$$

Using Sands's translation rules, we can produce a cost function

$$\texttt{cmap}\ f\ xs = 1 + \texttt{if}\ (xs)\ \texttt{then}\ 0\ \texttt{else}\ f\ \texttt{c@}\ (\texttt{hd}\ xs) + \texttt{cmap}\ f\ (\texttt{tl}\ xs)$$

where f c@ (hd xs) is the cost of applying f to the expression hd xs. Since f is an unknown parameter it is impossible to simplify this application cost. As a result, cmap is context-sensitive. For other functions f_i, it is possible to represent cf_i as the sum of cfF_i and cfS_i, where cfS_i is context-sensitive and cfF_i is context-free. This factorization is not always straightforward, but Sands presents a number of strategies to factor cost functions.

2.3. Benzinger's Annotations

Benzinger [2004] introduces the automated complexity analysis (ACA) system for the analysis of higher-order functional programs. ACA aims to fully automate the time complexity analysis of a program written in the language of the NUPRL proof development system. The automation of this analysis is completed using a combination of NUPRL's proof generation techniques and Mathematica's computer algebra system. Benzinger translates the target language of NUPRL into a meta-language in which he performs symbolic evaluation. The result of the symbolic evaluation is a set of parameterized recurrence equations describing the complexity of the original term. Though not the focus of the present work, Benzinger describes various methods for solving these recurrences to their closed form representations.

As a cost model for the target language of ACA, Benzinger adds a series of annotations to NUPRL's call-by-name operational semantics. If s and t are expressions and $s \downarrow t$ in NUPRL's semantics, then he says $s \downarrow t$ (in n) if n is some measure of the time complexity required to evaluate s to t. If t is a value (canonical) then the judgement can be written $s \downarrow\downarrow t$ (in n). The ACA system in particular models the exact time complexity of a term, though it could easily be adapted to represent space or another form of complexity.

Benzinger does not provide a syntactic structure for the complexity measure (in n). The lack of structure allows for greater flexibility in the possible ways to measure complexity, but it also makes reasoning about each method more complicated. Later we will see that this complication is abstracted away to the symbolic evaluation stage; in order to reason about the translation of the complexity in the meta-language, a user must provide reduction rules specific to the term being analyzed.

To illustrate this drawback, consider the time complexity evaluation of the `insert` expression in Benzinger's annotated operational semantics, where

$$\texttt{insert} \coloneqq \lambda x, xs.\texttt{listind}(xs; x :: [\,]; h, t, z.\texttt{lt}(x; h; x :: h :: t; h :: z))^1$$

Given a sorted integer list xs of length n, we expect $\texttt{insert}(x, xs)$ to evaluate to a sorted list consisting of every value in xs along with x. Intuitively we will want to prove that the time complexity of `insert` is linear in the size of xs. Due to NUPRL's call-by-name semantics, the

judgement we will actually prove is

$$\texttt{insert}(x, xs) \downarrow\downarrow w \text{ (in } m) \text{ for } m \leq \hat{xs} + (n+1)\hat{x} + 3n + 1$$

where

- xs is an sorted integer list of length n with a cost of $\hat{xs}$
- x is an integer with a cost of $\hat{x}$
- w is a sorted integer list consisting of the values of x along with elements of xs.

We will prove this judgement holds by induction on n, the size of xs.

If $n = 0$ then it must be the case that $xs \downarrow\downarrow [\,] \text{ (in } \hat{xs})$. Therefore, as a base case we see that

$$\dfrac{xs \downarrow\downarrow [\,] \text{ (in } \hat{xs}) \qquad \dfrac{x \downarrow\downarrow v \text{ (in } \hat{x}) \quad [\,] \downarrow\downarrow [\,] \text{ (in 0)}}{x :: [\,] \downarrow\downarrow v :: [\,] \text{ (in } \hat{x}+1)}}{\texttt{listind}(xs; x :: [\,]; h, t, z.\texttt{lt}(x; h; x :: h :: t; h :: z)) \downarrow\downarrow v :: [\,] \text{ (in } \hat{xs} + \hat{x} + 1)}$$

For an inductive step, suppose the claim holds on lists of size n and let xs be a list of size $n + 1$. Then $xs \downarrow\downarrow u :: us$ for us a list of length n. Let

$$\texttt{rec}(us) := \texttt{listind}(us; x :: [\,]; h, t, z.\texttt{lt}(x; h; x :: h :: t; h :: z))$$

be the recursive subterm. We will proceed by case analysis on the conditional in $\texttt{insert}(x, xs)$. If the value of x is less than the value of the head of xs, then we should have $\texttt{insert}(x, xs) \downarrow x :: xs$. In particular,

$$\dfrac{xs \downarrow\downarrow u :: us \text{ (in } \hat{xs}) \qquad \dfrac{x \downarrow\downarrow v \text{ (in } \hat{x}) \quad u \downarrow\downarrow u \text{ (in 0)} \quad x < u \qquad \dfrac{x{\downarrow\downarrow}v \text{ (in } \hat{x}) \quad u{\downarrow\downarrow}u \text{ (in 0)} \quad us{\downarrow\downarrow}us \text{ (in 0)}}{x{::}u{::}us{\downarrow\downarrow}v{::}u{::}us \text{ (in } \hat{x}{+}2)}}{\texttt{lt}(x; u; x :: u :: us; u :: \texttt{rec}(us)) \downarrow\downarrow v :: u :: us \text{ (in } 2\hat{x} + 3)}}{\texttt{listind}(x; x :: [\,]; h, t, z.\texttt{lt}(x; h; x :: h :: t; h :: z)) \downarrow\downarrow v :: u :: us \text{ (in } \hat{xs} + 2\hat{x} + 3)}$$

Notice that $\hat{xs} + 2\hat{x} + 3 \leq \hat{xs} + (n+2)\hat{x} + 3(n+1) + 1$ so the claim holds.

Now suppose that the value of x is greater than or equal to the value of the head of xs. The inductive hypothesis states that $\texttt{rec}(us) \downarrow\downarrow ws \text{ (in } m)$ for some $m \leq (n+1)\hat{x} + 3n + 1$ (since

[1]The list induction term $\texttt{listind}(xs; b; h, t, z.s)$ performs structural induction on the list xs with base case b. In evaluation of the recursive term, the variable h is mapped to the head of xs, t is mapped to the tail and z is mapped to the inductive call on the tail of xs.

$\hat{us} = 0$). Therefore we can derive the following:

$$\dfrac{xs \downarrow\downarrow u :: us \ (\text{in } \hat{xs}) \qquad \dfrac{x \downarrow\downarrow v \ (\text{in } \hat{x}) \quad u \downarrow\downarrow u \ (\text{in } 0) \quad v > u \quad \dfrac{u \downarrow\downarrow u \ (\text{in } 0) \quad \texttt{rec}(vs) \downarrow\downarrow ws \ (\text{in } m)}{u :: \texttt{rec}(vs) \downarrow\downarrow u :: ws \ (\text{in } m+1)}}{\texttt{lt}(x; u; x :: u :: us; u :: \texttt{rec}(us)) \downarrow\downarrow u :: ws \ (\text{in } \hat{x} + m + 2)}}{\texttt{listind}(xs; x :: [\,]; h, t, z.\texttt{lt}(x; h; x :: h :: t; h :: z)) \downarrow\downarrow u :: vs \ (\text{in } \hat{xs} + \hat{x} + m + 3)}$$

Because $\hat{xs} + \hat{x} + m + 3 \leq \hat{xs} + (n+2)\hat{x} + 3(n+1) + 1$, the claim holds.

There are a few subtleties required in this analysis. First is that the statement of the analysis is in terms of an upper bound. Though the bound we proved is true, it is actually a tight upper bound provided there are no constraints on the input xs. Although the complexity of $\texttt{insert}(x, xs)$ is not equal to $\hat{xs} + (n+1)\hat{x} + 3n + 1$ for all xs, there is some xs which makes the equality hold. In that way one may think of this expression as a "least upper bound" on the complexity of $\texttt{insert}(x, xs)$.

In the case of `insert`, the bound holds exactly if one takes the complexity of the conditional to be equal to the maximum of the complexities of the branches. This approach has two advantages. First, as in this example, it greatly simplifies the representation of the complexity if one's goal is to identify a least upper bound measurement. Second, it eliminates the need to evaluate the test in order to identify the complexity of the conditional.

The second difficulty with this analysis is that there is no intuitive means of reasoning about the size of a data structure as simple as a singly-linked list. In the above proof we use the fact that if the size of xs is 0 then xs evaluates to the empty list, and if the size of xs is $n > 0$ then xs evaluates to a list $v :: vs$ where vs is a list of size $n - 1$. There is no built-in way to do size reasoning in Benzinger's meta-language, however. In order to do this reasoning, it is possible to construct reduction rules to apply in the symbolic analysis. This method is flexible but it comes at the cost of having a less straightforward analysis. Intuitively the size of the input to a function is an important feature in a cost analysis. The goal of many asymptotic analyses is to represent the complexity of a term as a function of the size of the input to the term. This goal motivates much of the work in this thesis.

2.4. Danner and Royer's Semantics

The style of complexity analysis in this thesis is inspired by the work of [Danner and Royer, 2007, 2009]. Their work describes a programming formalism called Affine Tiered Recursion, or ATR. ATR is a type-2 programming formalism in which every expression is polynomial-time bounded in both space and time. Their work is an example of implicit computational complexity in that they construct a language whose typeable expressions correspond exactly to the class of polynomial-time algorithms. The bounds are achieved by means of a restrictive type system based on the idea of data ramification.

In their paper *Adventures in time and space*, [Danner and Royer, 2007] define their formalism and attribute to it a cost model which describes the cost of evaluation for ATR expressions. Their system consists to two distinct semantics for the language—one which describes evaluation and a second, $\mathcal{T}[\![\cdot]\!]$, which provides upper bounds on the time complexity of expressions. Danner and Royer here introduce the idea of *time complexity*, upon which we base the complexity language described in Chapter 3. The time complexity of an expression e is made up of a cost—a natural number which acts as an upper bound on the cost of evaluating e—and a potential—which acts an as upper bound on e's length. Potentials of base types are natural numbers, while the potential of an expression e of arrow type is a map from potentials to the time complexity of applying e to something of that potential.

Next the authors define the semantic types under $\mathcal{T}[\![\cdot]\!]$, which reflect the structure of complexities. There are two translation functions on the ATR types. The first, $\|\cdot\|$, sends a type σ to $\mathbf{T}\times\ll\sigma\gg$, where $\mathbf{T}$ is the type of costs. The second function, $\ll\cdot\gg$, sends σ to its corresponding potential type; that is, if σ is a base type, $\ll\sigma\gg$ is a type representing sizes of that base type, and for an arrow type $\sigma\to\tau$, $\ll\sigma\to\tau\gg=\ll\sigma\gg\to\|\tau\|$.

The next step is to extend the maps $\|\cdot\|$ and $\ll\cdot\gg$ to send expressions in ATR of type σ to semantic values inside of $\mathcal{T}[\![\sigma]\!]$. Elements of the base types of ATR are string constants or oracles (type-1 functions over ω). The cost of a string constant a is the maximum of 1 and the length of a, which refers to the cost of reading the bits of a. The potential of a string is its length. For oracles, the cost is 1 because λ-expressions evaluate to themselves in a call-by-value

system. The potential of an oracle f is a function from potentials p to the maximum of the $\|f\ v\|$ such that $\ll v \gg \leq p$.

To process complexities in the image of $\mathcal{T}$, the authors define a $\star$ operator which implements a notion of application of one complexity to another. That is, if t_0 is the time complexity of e_0 and t_1 the time complexity of e_1, then $t_0 \star t_1$ is intended to be the time complexity of e_0 applied to e_1. Similarly, the $\Lambda_\star$ operator on semantic values demonstrates the abstraction of time complexity pairs. We will see these ideas surface in a slightly different form in Chapter 3.

Danner and Royer's work aims to construct a type-2 programming formalism whose typeable expressions are exactly the polynomial-time computable functions. The goals of the present work are different, in that we wish to reason explicitly about complexities. For our system there is no guarantee of polynomial-time bounds, and in addition we wish to reason about varying levels of complexities as complexities themselves. To achieve this we modify the $\mathcal{T}$ semantics defined by Danner and Royer and push their ideas into the creation of a complexity language, in which time complexities are represented by expressions.

In *Two algorithms in search of a type-system,* Danner and Royer [2009] develop a type system on ATR values, or time complexities. The type system reflects the structure of the time-complexity semantics. There is one cost type, T, and for each base type N_L in ATR there is a potential type T_L. Then the time-complexity types are the image of $\|\tau\|$ for τ a type in ATR, where $\|\tau\| = \mathsf{T}\times \ll \tau \gg$. The potential types are then the image of $\ll \tau \gg$, given by

$$\ll \mathsf{N}_l \gg = \mathsf{T}_L \qquad \ll \sigma \to \tau \gg = \ll \sigma \gg \to \|\tau\|$$

From this type system, the authors define a typing relation on time complexity polynomials.

The purpose of developing this type system is to have a greater ability to reason about time complexities. The authors develop a pair of bounding relations which compare closures and values of type τ in ATR with time complexities of type $\|\tau\|$ and $\ll \tau \gg$. The type system and bounding relation allow the authors of the paper to prove the soundness of new constructs of ATR aimed at giving the language breadth.

In Section 3.3 we show how we modify the time-complexity type system for our own complexity language, and in Section 3.4.3 we describe the bounding relation derived from this work.

CHAPTER 3

Mathematical Development of the Complexity Translation

3.1. Introduction

In this chapter we develop a higher-order functional language with structural recursion to serve as a target language for our complexity analysis. We define a parallel complexity language whose semantics represent the complexity of a term. To connect the two languages there is a translation function $\|\cdot\|$ from target expressions to complexity expressions such that the cost given by the complexity term is an upper bound on the size of the target term's evaluation derivation.

3.2. Target Language

We will take our target language to be the simply-typed lambda calculus along with integer lists and structural recursion. Because of the simplicity of this language, we will find it easy to reason about the complexity of target language terms. Structural recursion gives us a good starting point for the development of our complexity analysis without forcing us to deal with non-terminating computation upfront. In Chapter 5 we discuss possibilities for extending the target language to one with more general kinds of recursion.

3.2.1. Syntax. The target language expressions are given in Figure 3.1. Var is a set of variable identifiers, and in the `listcase` and `fold` expressions, x, xs and w range over elements in Var. The intended interpretation of `listcase` r `of` $(s, [x, xs]t)$ is a conditional on the size of the integer list r where s is the size zero case and t is an expression with the variables x and xs free. In t, x is meant to stand in for the head of the list r and xs for the tail. The interpretation of `fold` r `of` $(s, [x, xs, w]t)$ is similar except that t has x, xs and w free and w stands in for the value of the recursive step `fold tl`(xs) `of` $(s, [x, xs, w]t)$.

$$\begin{aligned}\mathbf{e} \in \text{Exp} ::= & x \text{ for } x \in \text{Var} \mid \texttt{true} \mid \texttt{false} \mid \texttt{n} \text{ for } n \in \mathbb{Z} \mid [\,] \mid \mathbf{e} :: \mathbf{e} \\ & \mid \texttt{if } \mathbf{e} \texttt{ then } \mathbf{e} \texttt{ else } \mathbf{e} \mid \mathbf{e} R \mathbf{e} \text{ for } R \in \{<, \leq, =, +, -, \times, \div\} \mid \lambda x.\mathbf{e} \\ & \mid \mathbf{e}\ \mathbf{e} \mid \texttt{listcase } \mathbf{e} \texttt{ of } (\mathbf{e}, [x, xs]\mathbf{e}) \mid \texttt{fold } \mathbf{e} \texttt{ of } (\mathbf{e}, [x, xs, w]\mathbf{e})\end{aligned}$$

FIGURE 3.1. Target Language Expressions

3.2.2. Typing. Next we present a grammar for types in the target language. The base types **b** consist of booleans, integers and integer lists. The target types are base types or arrow types of target types.

$$\begin{aligned}\tau \in \text{Type} ::= & \ \mathbf{b} \mid \tau \to \tau \\ \mathbf{b} ::= & \ \texttt{bool} \mid \texttt{int} \mid \texttt{int}* \\ \texttt{int}* ::= & \ E \mid \texttt{cons } \mathbf{of } (\texttt{int}, \texttt{int}*)\end{aligned}$$

We also define a denotational semantics for target types, as follows:

$$[\![\texttt{bool}]\!] = \{\texttt{tt}, \texttt{ff}\} \qquad [\![\texttt{int}]\!] = \mathbb{N}$$

$$[\![\texttt{int}*]\!] = \mathbb{N}^* \qquad [\![\sigma \to \tau]\!] = [\![\tau]\!]^{[\![\sigma]\!]}$$

A *type context* is a partial function $\Gamma : \text{Var} \rightharpoonup \text{Type}$ which maps variables to types. Typing of expressions derives judgements of the form $\Gamma \vdash e : \tau$ where e is a target language expression and τ is a type. The typing rules are given in Figure 3.2.

3.2.3. Semantics. To evaluate the target language we define a call-by-value operational semantics. The set Val of target values consists of constants, integer lists and abstractions.

$$\text{Val} ::= \texttt{tt} \mid \texttt{ff} \mid \texttt{n} \text{ for } n \in \mathbb{N} \mid (n_0, \ldots, n_{k-1}) \text{ for } k \geq 0, n_i \in \mathbb{N} \mid \lambda x.r \text{ for } r \in \text{Exp}$$

Define a *value environment* to be a partial function ξ from variables to value closures. A *value closure* is defined recursively to be a pair $v\,\xi$ where v is a value and ξ a value environment. Meanwhile, we simply call a pair $e\,\xi$ a *closure* when e is an expression.

In the operational semantics we derive statements of the form $e\,\xi \downarrow v\,\theta$. The evaluation inference rules are given in Figure 3.3.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ var \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ true \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ false$$

$$\frac{}{\Gamma \vdash \texttt{n} : \texttt{int}}\ num \qquad \frac{}{\Gamma \vdash [\,] : \texttt{int}*}\ empty \qquad \frac{\Gamma \vdash r : \texttt{int} \quad \Gamma \vdash s : \texttt{int}*}{\Gamma \vdash r :: s : \texttt{int}*}\ cons$$

$$\frac{\Gamma \vdash r : \texttt{bool} \quad \Gamma \vdash s : \tau \quad \Gamma \vdash t : \tau}{\Gamma \vdash \texttt{if}\ r\ \texttt{then}\ s\ \texttt{else}\ t : \tau}\ if$$

$$\frac{\Gamma \vdash s : \texttt{int} \quad \Gamma \vdash t : \texttt{int} \quad R \in \{<, \leq, =\}}{\Gamma \vdash sRt : \texttt{bool}}\ rel \qquad \frac{\Gamma \vdash s : \texttt{int} \quad \Gamma \vdash t : \texttt{int} \quad R \in \{+, -, \times, \div\}}{\Gamma \vdash sRt : \texttt{int}}\ op$$

$$\frac{\Gamma, x : \sigma \vdash r : \tau}{\Gamma \vdash \lambda x.r : \sigma \to \tau}\ lambda \qquad \frac{\Gamma \vdash r : \sigma \to \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash rs : \tau}\ app$$

$$\frac{\Gamma \vdash r : \texttt{int}* \quad \Gamma \vdash s : \tau \quad \Gamma, x : \texttt{int}, xs : \texttt{int}* \vdash t : \tau}{\Gamma \vdash \texttt{listcase}\ r\ \texttt{of}\ (s, [x, xs]t) : \tau}\ listcase$$

$$\frac{\Gamma \vdash r : \texttt{int}* \quad \Gamma \vdash s : \tau \quad \Gamma, x : \texttt{int}, xs : \texttt{int}*, w : \tau \vdash t : \tau}{\Gamma \vdash \texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t) : \tau}\ fold$$

FIGURE 3.2. Typing Rules for Target Expressions

Notice first of all that all integer constants, booleans, and lists evaluate to a value closure whose environment is empty. In the *op-rel* rule, $m\ R\ n$ is the semantic interpretation of R, which is either an operator in $\{+, -, \times, \div\}$ or one of the binary relations $\{<, \leq, =\}$.

In the *fold1* evaluation rule, we perform the recursive evaluation step by inserting a fresh variable y to hold the place for the value of the tail of r. This ensures that we do not evaluate r more than once. This is contrasted to the more naïve evaluation rule based on an operator `tl` with the following evaluation rules:

$$\frac{r\,\xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,]}{(\texttt{tl}\ r)\ \xi \downarrow (n_0, \ldots, n_{k-1})\,[\,]}\ tl$$

$$\frac{r\,\xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,] \quad (\texttt{fold}\ \texttt{tl}\ r\ \texttt{of}\ (s, [x, xs, w]t))\ \xi \downarrow v'\ \theta' \quad t\,\xi \left[\begin{smallmatrix} x \mapsto n[\,], w \mapsto v'\theta' \\ xs \mapsto (n_0, \ldots, n_{k-1})[\,] \end{smallmatrix}\right] \downarrow v\ \theta}{(\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t))\ \xi \downarrow v\ \theta}\ fold1'$$

The evaluation derivation of $\texttt{fold}\ (\texttt{tl}\ r)\ \texttt{of}\ (s, [x, xs, w]t)$ will have as a root derivation one of *fold0* or *fold1'*. In either case it consists of the derivation of $\texttt{tl}\ r$, and therefore the derivation of r, as a subderivation. So the derivation of r occurs at least twice in every evaluation whose root is the *fold1'* rule.

To avoid this wasteful behavior, we record the evaluation of r in the variable y. Since variables evaluate in one step by means of the *var* inference rule, we ensure that we need not evaluate r again in the subderivations.

$$\frac{}{x\,\xi[x \mapsto v\,\theta] \downarrow v\,\theta}\ \textit{var} \qquad \frac{}{\texttt{true}\,\xi \downarrow \texttt{tt}\,[\,]}\ \textit{bool-t} \qquad \frac{}{\texttt{false}\,\xi \downarrow \texttt{ff}\,[\,]}\ \textit{bool-f}$$

$$\frac{}{\texttt{n}\,\xi \downarrow \texttt{n}\,[\,]}\ \textit{num} \qquad \frac{}{[\,]\,\xi \downarrow ()\,[\,]}\ \textit{nil} \qquad \frac{r\,\xi \downarrow \texttt{n}\,[\,] \quad s\,\xi \downarrow (n_0, \ldots, n_{k-1})\,[\,]}{(r :: s)\ \xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,]}\ \textit{cons}$$

$$\frac{r\,\xi \downarrow \texttt{tt}\,[\,] \quad s\,\xi \downarrow v\,\theta}{(\texttt{if}\ r\ \texttt{then}\ s\ \texttt{else}\ t)\ \xi \downarrow v\,\theta}\ \textit{if-t} \qquad \frac{r\,\xi \downarrow \texttt{ff}\,[\,] \quad t\,\xi \downarrow v\,\theta}{(\texttt{if}\ r\ \texttt{then}\ s\ \texttt{else}\ t)\ \xi \downarrow v\,\theta}\ \textit{if-f}$$

$$\frac{r\,\xi \downarrow \texttt{m}\,[\,] \quad s\,\xi \downarrow \texttt{n}\,[\,] \quad p = m\ R\ n}{(rRs)\ \xi \downarrow \texttt{p}\,[\,]}\ \textit{op-rel}$$

$$\frac{}{(\lambda x.r)\ \xi \downarrow (\lambda x.r)\ \xi}\ \textit{lambda} \qquad \frac{r\,\xi \downarrow (\lambda x.t)\ \theta_1 \quad s\,\xi \downarrow w\,\theta_2 \quad t\,\theta_1[x \mapsto w\,\theta_2] \downarrow v\,\theta}{(rs)\ \xi \downarrow v\,\theta}\ \textit{app}$$

$$\frac{r\,\xi \downarrow ()\,[\,] \quad s\,\xi \downarrow v\,\theta}{(\texttt{listcase}\ r\ \texttt{of}\ (s, [x, xs]t))\ \xi \downarrow v\,\theta}\ \textit{listcase0}$$

$$\frac{r\,\xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,] \quad t\,\xi[x \mapsto n\,[\,], xs \mapsto (n_0, \ldots, n_{k-1})\,[\,]] \downarrow v\,\theta}{(\texttt{listcase}\ r\ \texttt{of}\ (s, [x, xs]t))\ \xi \downarrow v\,\theta}\ \textit{listcase1}$$

$$\frac{r\,\xi \downarrow ()\,[\,] \quad s\,\xi \downarrow v\,\theta}{(\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t))\ \xi \downarrow v\,\theta}\ \textit{fold0}$$

$$\frac{r\,\xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,] \quad (\texttt{fold}\ y\ \texttt{of}\ (s, [x, xs, w]t))\ \xi[y \mapsto (n_0, \ldots, n_{k-1})\,[\,]] \downarrow\ v'\ \theta' \quad t\,\xi \left[\begin{smallmatrix} x \mapsto \texttt{n}\,[\,], w \mapsto v'\,\theta' \\ xs \mapsto (n_0, \ldots, n_{k-1})\,[\,] \end{smallmatrix}\right] \downarrow v\,\theta}{(\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t))\ \xi \downarrow v\,\theta}\ \textit{fold1}$$

FIGURE 3.3. An Operational Semantics for the Target Language

3.2.4. Uniqueness of Evaluation Derivation. Evaluation judgements are proved by means of an evaluation derivation. An *evaluation derivation* for $t\,\xi \downarrow v\,\theta$ is a tree of inference rules from the operational semantics such that the root inference rule has $t\,\xi \downarrow v\,\theta$ as a conclusion. The size of an evaluation derivation $\mathcal{D}$ is the number of inference rules in the tree.

Theorem 1. *If $\mathcal{D}$ is an evaluation derivation of $t\,\xi \downarrow v\,\theta$ and $\mathcal{D}'$ is a derivation of $t\,\xi \downarrow v'\,\theta'$ then $\mathcal{D} = \mathcal{D}'$ and $v\,\theta = v'\,\theta'$*

Because the operational semantics of the target language is syntax-directed, there exist a number of inversion lemmas which specify how various inference rules apply. We use these inversion lemmas liberally in the following proof.

PROOF. We will proceed by induction on the height of $\mathcal{D}$.

If the height of $\mathcal{D}$ is one, then the root inference rule in $\mathcal{D}$ must be an axiom—one of *var*, *bool-t*, *bool-f*, *num*, *nil* or *lambda*. By the inversion lemmas for each of these, the root derivation of $\mathcal{D}'$ must be the same axiom.

Suppose $\mathcal{D}$ has height $h + 1$. We now perform a case analysis of the root inference rule $\mathcal{R}$ of $\mathcal{D}$. We know that $\mathcal{R}$ is not an axiom since $\mathbf{cost}(\mathcal{D}) > 1$.

If $\mathcal{R}$ is the *cons* inference rule, then we can write $\mathcal{D}$ as

$$\frac{\mathcal{D}_r \quad \mathcal{D}_s}{(r :: s)\ \xi \downarrow (n, n_0, \ldots, n_{k-1})\ [\,]}\ \mathit{cons}$$

where $\mathcal{D}_r$ is a derivation of $r\,\xi \downarrow n\ [\,]$ and $\mathcal{D}_s$ is a derivation of $s\,\xi \downarrow (n_0, \ldots, n_{k-1})\ [\,]$. By the inversion lemma for *cons* it must be the case that any other derivation of $(r :: s)\ \xi \downarrow (n', n'_0, \ldots, n'_{k'-1})\ [\,]$ has the same form. So we can write $\mathcal{D}'$ as

$$\frac{\mathcal{D}'_r \quad \mathcal{D}'_s}{(r :: s)\ \xi \downarrow (n', n'_0, \ldots, n'_{k-1})\ [\,]}\ \mathit{cons}$$

By the induction hypothesis, $\mathcal{D}_r = \mathcal{D}'_r$ and $\mathcal{D}_s = \mathcal{D}'_s$. But then $n = n'$ and $(n_0, \ldots, n_{k-1}) = (n'_0, \ldots, n'_{k'-1})$. Hence $\mathcal{D} = \mathcal{D}'$.

The same reasoning holds if $\mathcal{R}$ is one of *rel-op* or *app*. Since the inversion lemmas of these rules specify that the root inference rules of $\mathcal{D}$ and $\mathcal{D}'$ must be the same, we can apply the inductive hypothesis to prove the equality of the subderivations as well.

Suppose $\mathcal{R}$ is *if-t*. Then $\mathcal{D}$ is of the form

$$\frac{\mathcal{D}_b \quad \mathcal{D}_r}{(\texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ s)\ \xi \downarrow v\,\theta}\ \mathit{if\text{-}t}$$

where $\mathcal{D}_b$ is a derivation of $b\,\xi \downarrow \texttt{tt}\ [\,]$ and $\mathcal{D}_r$ is a derivation of $r\,\xi \downarrow v\,\theta$. Then, since $t = \texttt{if}\ b\ \texttt{then}\ r\ \texttt{else}\ s$, the root inference rule of $\mathcal{D}'$ must be one of *if-t* and *if-f*. In either case it has a subderivation $\mathcal{D}'_b$ of $r\,\xi \downarrow w\ [\,]$ where w is either $\texttt{tt}$ or $\texttt{ff}$. Since $\mathcal{D}_b$ has height h, we can apply the inductive hypothesis to prove that $\mathcal{D}_b = \mathcal{D}'_b$. But in that case $w = \texttt{tt}$ so the root inference rule of $\mathcal{D}'$ is *if-t*. After applying the inductive hypothesis to $\mathcal{D}_r$ we see that $\mathcal{D} = \mathcal{D}'$ and $v\,\theta = v'\,\theta'$.

The proofs of the remaining cases are very similar to that of *if-t* and will not be discussed here. □

Since every derivation of $t\,\xi$ is unique, we will use the notation $\mathbf{cost}(t\,\xi)$, or simply $\mathbf{cost}(t)$ if the choice of ξ is clear, to refer to the size of $t\,\xi$'s evaluation derivation. This notion of size will be our cost model for the target language.

3.3. Complexity Language

The overarching goal of this thesis is to produce a cost complexity analysis of terms in the target language. A persistent design problem has been how exactly to represent complexities. We start with a number of intuitions about what we want the analysis to look like. First, we want the complexity of a target term to measure the cost of its evaluation. This cost may be a measure of time or space used during evaluation, for example. Second, complexity should be compositional—the complexity of a term should depend on the complexity of its input. Third, we would like the cost analysis to be static, meaning that complexities should not refer to target language evaluation.

Shultis [1985], Sands [1992] and Benzinger [2004] address the first two of these requirements, but not the third. Each of their complexity analyses depend on the evaluation of the target term. The trade-off for a static analysis here is exactness—their complexity representations exactly correspond to the complexity of the target term. Take for example the if statement `if` r `then` s `else` t. In the operational semantics of the previous section, the cost of evaluating this expression is equal to one plus the cost of evaluating r plus *either* the cost of evaluating s *or* the cost of evaluating t, depending on whether r evaluates to `tt` or `ff`. Therefore the exact complexity of `if` r `then` s `else` t depends not only on the complexity of its subterms, but also on the *value* of r.

The key intuition behind the complexities developed here, which was inspired by Danner and Royer [2007], is that in many cases we are not interested in that level of exactness as part of a complexity analysis. Think of the if statement as a function: $\lambda x.$`if` x `then` s `else` t. There exist two inputs of the same complexity—`true` and `false`—which yield distinct evaluation costs for `if` x `then` s `else` t. In that case, it is better for the complexity of the entire expression to reflect the maximum complexity achievable for input of a given cost. With respect to the expression `if` r `then` s `else` t, this means its cost is one plus the cost of r plus the maximum of the costs of s and t.

In this section we develop a notion of complexity which meets the three criteria given above. In order to reason about complexities we develop a *complexity language* whose denotational semantics maps into the realm of complexities. The advantage of implementing complexities as a formal language is that we can perform semantic-preserving manipulations to make complexity expressions easier to read and easier to reason about. In the next section we construct a translation which sends target terms to complexity terms, giving a complexity meaning to target expressions.

3.3.1. Complexities. Let us now try to pin down the notion of complexities more formally. One component of a complexity must be cost, which should correspond to the natural numbers. However, as Shultis [1985] and others have pointed out, cost alone is not enough to represent the complexity of higher-order terms. The cost analysis of a function is a function on the size of its input. If we are to extend this intuition to refer to complexities in our target language we must specify what we mean by the "size" of a higher-order expression. We will refer to this augmented idea of size as the *potential* of a term.

In the target language integers and booleans are not defined inductively, so their sizes are some small constant, which we can generalize as the number 1. As `cons` expressions are lists of integers, we will define their potential to be a list of 1's. Our other option would be to define the potential of a `cons` expression to be the length of the list. We chose the more naïve interpretation with the hope that this approach can be generalized to other data structures. For example, suppose we modify the target language to contain binary trees labeled by integers. It is not immediately obvious whether the "size" of a tree refers to the number of nodes or the height of the tree. On the other hand, extending the current interpretation of potential, we describe the potential of a tree to be simply a tree of its potentials. We explore this generalization in more detail in Section 3.5.

For the case of integer lists we will use the notation $\overline{n}$ to refer to the unique length-n list of 1's.

$$\begin{aligned}\mathbf{e} \in \text{CExp} ::= & \mathbf{n} \textbf{ for } n \in \mathbf{C} \mid \mathbf{e} + \mathbf{e} \mid \mathbf{e} \vee \mathbf{e} \mid (\mathbf{e}, \mathbf{e}) \mid \mathbf{e}_c \mid \mathbf{e}_p \\ & \mid x \text{ for } x \in \text{Var} \mid \lambda_* x.\mathbf{e} \mid \mathbf{e} * \mathbf{e} \mid 1^{\texttt{bool}} \mid 1^{\texttt{int}} \mid \texttt{p}E \mid \texttt{pcons}(\mathbf{e}, \mathbf{e}) \\ & \mid \texttt{plistcase } \mathbf{e} \texttt{ of } (\mathbf{e}, [p, ps]\mathbf{e}) \mid \texttt{pfold } \mathbf{e} \texttt{ of } (\mathbf{e}, [p, ps, w]\mathbf{e})\end{aligned}$$

FIGURE 3.4. Complexity Language Expressions

Let $\mathbb{A}$ be the set of simple product types over $\{\mathbb{N}, \{1\}, \{1\}^*\}$. That is, $\mathbb{N}, \{1\}, \{1\}^* \in \mathbb{A}$, and $\mathbb{A}$ is closed under product and function types. This set $\mathbb{A}$ will be home to the semantic complexities and potentials.

3.3.2. Syntax. To reason about complexities we develop the *complexity language*, whose syntax is given in Figure 3.4. We distinguish the potential of an integer ($1^{\texttt{int}}$) from the potential of a boolean ($1^{\texttt{bool}}$). Since we no longer work with actual integers and booleans, we have no need for arithmetic operators and boolean relations. We do require addition of costs and the max operator $\vee$ to compare both costs and potentials. As discussed above, in order to avoid assuming a definition of size for user-defined datatypes we let the potential of an integer list be a list itself, with constructors $\texttt{p}E$ and $\texttt{pcons}(\mathbf{e}, \mathbf{e})$.

We allow the pairing of expressions, and we use the t_c and t_p notation to select the first and second element of a pair, respectively. Here the c stands for cost and the p for potential. The $*$ operator implements a notion of application of one pair to another, and λ_* is the corresponding abstraction operation.

The `plistcase` and `pfold` expressions are the complexity language counterparts to `listcase` and `fold`. Recall that `listcase` r `of` $(s, [x, xs]t)$ and `fold` r `of` $(s, [x, xs, w]t)$ branch on the value to which r evaluates. On the other hand, in `plistcase` r `of` $(s, [x, xs]t)$ and `fold` r `of` $(s, [x, xs, w]t)$, the denotation of r will be the potential of the list. Thus while `listcase` and `fold` branch on the value of a list, `plistcase` and `pfold` branch on its potential.

3.3.3. Types. Our typing grammar, given in Figure 3.5, defines complexity and potential types. Corresponding to the collection $\mathbb{A}$ are the *full complexity types* which consist of the datatype $\mathbf{C}$ and the potential base types, and which are closed under arrow and product types.

$$\tau \in \text{Full-CType} := \mathbf{C} \mid \mathtt{pb} \mid \tau \to \tau \mid \tau \times \tau$$
$$\mathtt{pb} := \mathtt{pbool} \mid \mathtt{pint} \mid \mathtt{pint}*$$
$$\sigma \in \text{CType} := \mathbf{C} \times \gamma$$
$$\gamma \in \text{PType} := \mathtt{pb} \mid \gamma \to \sigma$$

FIGURE 3.5. Complexity Language Types

Elements of type $\mathbf{C}$ correspond to costs in $\mathbb{N}$. The potential base types are $\mathtt{pbool} := 1^{\mathtt{bool}}$, $\mathtt{pint} := 1^{\mathtt{int}}$, and $\mathtt{pint}* := \mathtt{p}E \mid \mathtt{pcons}\ \mathbf{of}\ (\mathtt{pint}, \mathtt{pint}*)$.

Next we define the mutually recursive *potential* and *complexity types*. If γ is a potential type then $\mathbf{C} \times \gamma$ is a complexity type. The potential types consist of potential base types along with arrow types of the form $\gamma \to \sigma$ where γ is a potential type and σ a complexity type.

Since all complexity types are of the form $\mathbf{C} \times \gamma$ for γ a complexity type, we define $\gamma^{\square}$ to be shorthand for the type $\mathbf{C} \times \gamma$. Additionally we will use the notation $\gamma \to_{\square} \tau$ for $(\gamma \to \tau)^{\square}$ or equivalently, $\mathbf{C} \times (\gamma \to \tau)$.

We define the denotation function $[\![\cdot]\!] : \text{Full-CType} \to \mathbb{A}$ in the expected manner:

$$[\![\mathbf{C}]\!] = \mathbb{N} \qquad [\![\mathtt{pbool}]\!] = [\![\mathtt{pint}]\!] = \{1\} \qquad [\![\mathtt{pint}*]\!] = \{[\![\mathtt{pint}]\!]\}^* = \{1\}^*$$
$$[\![\sigma \to \tau]\!] = [\![\tau]\!]^{[\![\sigma]\!]} \qquad [\![\sigma \times \tau]\!] = [\![\sigma]\!] \times [\![\tau]\!]$$

3.3.4. Typing. A *complexity type context* is a partial function $\Gamma : \text{Var} \rightharpoonup \text{Full-CType}$. The typing rules for complexity language expressions are given in Figure 3.6. In it we assume η, γ are potential types and τ is any full complexity type.

Since costs are natural numbers of type $\mathbf{C}$, the semantics of the addition and maximum operators are the standard arithmetic operators. We want to be able to take maximums of two expressions of any (same) type however, so we must also define a max operation over each of our base potential types. For our current language it is easy to infer what these operators should be, but as we will see in Section 3.5, for more complicated datatypes we will need to adopt a different system.

$$\frac{}{\Gamma \vdash \mathtt{n} : \mathbf{C}}\ C \quad (\text{for } n \in \mathbf{C}) \qquad \frac{}{\Gamma \vdash 1^{\mathtt{bool}} : \mathtt{pbool}}\ bool \qquad \frac{}{\Gamma \vdash 1^{\mathtt{int}} : \mathtt{pint}}\ int$$

$$\frac{}{\Gamma \vdash \mathtt{p}E : \mathtt{pint}*}\ pempty \qquad \frac{\Gamma \vdash r : \mathtt{pint} \quad \Gamma \vdash s : \mathtt{pint}*}{\Gamma \vdash \mathtt{pcons}(r, s) : \mathtt{pint}*}\ pcons$$

$$\frac{\Gamma \vdash r : \mathbf{C} \quad \Gamma \vdash s : \mathbf{C}}{\Gamma \vdash r + s : \mathbf{C}}\ plus \qquad \frac{\Gamma \vdash r : \tau \quad \Gamma \vdash s : \tau}{\Gamma \vdash r \vee s : \tau}\ max$$

$$\frac{\Gamma \vdash r : \mathbf{C} \quad \Gamma \vdash s : \gamma}{\Gamma \vdash (r, s) : \gamma^{\Box}}\ pair \qquad \frac{\Gamma \vdash r : \gamma^{\Box}}{\Gamma \vdash r_c : \mathbf{C}}\ cost \qquad \frac{\Gamma \vdash r : \gamma^{\Box}}{\Gamma \vdash r_p : \gamma}\ pot$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ var \qquad \frac{\Gamma, x : \gamma^{\Box} \vdash r : \eta^{\Box}}{\Gamma \vdash \lambda_* x.r : \gamma \to_{\Box} \eta^{\Box}}\ lambda* \qquad \frac{\Gamma \vdash r : \gamma \to_{\Box} \eta^{\Box} \quad \Gamma \vdash s : \gamma^{\Box}}{\Gamma \vdash r * s : \eta^{\Box}}\ app*$$

$$\frac{\Gamma \vdash r : \mathtt{pint}* \quad \Gamma \vdash s : \gamma^{\Box} \quad \Gamma, p : \mathtt{pint}, ps : \mathtt{pint}* \vdash t : \gamma^{\Box}}{\Gamma \vdash \mathtt{plistcase}\ r\ \mathtt{of}\ (s, [p, ps]t) : \gamma^{\Box}}\ plistcase$$

$$\frac{\Gamma \vdash r : \mathtt{pint}* \quad \Gamma \vdash s : \gamma^{\Box} \quad \Gamma, p : \mathtt{pint}, ps : \mathtt{pint}*, w : \gamma^{\Box} \vdash t : \gamma^{\Box}}{\Gamma \vdash \mathtt{pfold}\ r\ \mathtt{of}\ (s, [p, ps, w]t) : \gamma^{\Box}}\ pfold$$

FIGURE 3.6. Complexity Language Typing Rules

We only allow pairing to form expressions of type $\gamma^{\Box}$ so that the cost r_c always has type $\mathbf{C}$ and the potential r_p always has a potential type. Meanwhile, if r has an arrow-box type $(\gamma \to_{\Box} \eta^{\Box})$ then it can be applied using the $*$ operator to operands of complexity type $\gamma^{\Box}$. Recall that $*$ is meant to act as application of complexity pairs. An element of type $\gamma \to_{\Box} \eta^{\Box}$ will evaluate to an element of $[\![\gamma \to_{\Box} \eta^{\Box}]\!]$; so it will be a pair (n, f) for a natural number n and a function $f \in [\![\eta^{\Box}]\!]^{[\![\gamma]\!]}$ which sends potentials in $[\![\gamma]\!]$ to complexities in $[\![\eta^{\Box}]\!]$. The $*$ operator, we will see in Section 3.3.5, adds n and the cost component of the operand to the result of applying f to the operand's potential.

It is possible for a variable to have any full complexity type, including potential types, through a complexity type context. In the `plistcase` and `pfold` rules, notice that the placeholder variables p and ps are of potential type, unlike w which has a complexity type. This is because `plistcase` and `pfold` expressions branch on r which has a potential type, namely `pint`$*$. In t, p stands in for the head of that list of potentials, and ps for its tail. On the other hand, w stands in for the value of the recursive call, which has complexity type.

3.3.5. Semantics. The denotational semantics we construct for the complexity language provides us with a semantic representation of costs and potentials. If Γ is a complexity type context, a Γ-environment is a function $\xi : \mathrm{dom}(\Gamma) \to \bigcup_{X \in \mathbb{A}} X$ such that $\xi(x) \in [\![\Gamma(x)]\!]$ for all x

$$
\begin{aligned}
[\![\Gamma \vdash \mathbf{n} : \mathbf{C}]\!]\xi &= n \\
[\![\Gamma \vdash r + s : \mathbf{C}]\!]\xi &= [\![r]\!]\xi + [\![s]\!]\xi \\
[\![\Gamma \vdash r \vee s : \tau]\!]\xi &= [\![r]\!]\xi \vee [\![s]\!]\xi \\
[\![\Gamma \vdash (r, s) : \gamma^{\square}]\!]\xi &= [\![\Gamma \vdash r : \mathbf{C}]\!] \times [\![\Gamma \vdash s : \gamma]\!] \\
[\![\Gamma \vdash r_c : \mathbf{C}]\!]\xi &= \mathbf{cost}([\![\Gamma \vdash r : \gamma^{\square}]\!]\xi) \\
[\![\Gamma \vdash r_p : \gamma]\!]\xi &= \mathbf{pot}([\![\Gamma \vdash r : \gamma^{\square}]\!]\xi) \\
[\![\Gamma \vdash x : \tau]\!]\xi &= \xi(x) \\
[\![\Gamma \vdash \lambda_* x.r : \gamma \to_{\square} \eta^{\square}]\!]\xi &= (1, \lambda v^{[\![\gamma]\!]}.[\![r]\!]\xi[x \mapsto (1, v)]) \\
[\![\Gamma \vdash r{*}s : \eta^{\square}]\!]\xi &= \mathtt{dally}\left(1 + ([\![r]\!]\xi)_c + ([\![s]\!]\xi)_c, ([\![r]\!]\xi)_p([\![s]\!]\xi)_p\right) \\
[\![\Gamma \vdash 1^{\mathtt{bool}} : \mathtt{pbool}]\!]\xi &= [\![\Gamma \vdash 1^{\mathtt{int}} : \mathtt{pint}]\!]\xi = 1 \\
[\![\Gamma \vdash \mathtt{p}E : \mathtt{pint}{*}]\!]\xi &= \overline{0} \\
[\![\Gamma \vdash \mathtt{pcons}(r, s) : \mathtt{pint}{*}]\!]\xi &= \overline{q+1} \qquad \text{if } [\![r]\!]\xi = 1 \text{ and } [\![s]\!]\xi = \overline{q} \\
[\![\Gamma \vdash \mathtt{plistcase}\ r\ \mathtt{of}\ (s, [p, ps]t) : \gamma^{\square}]\!]\xi &= \begin{cases} [\![s]\!]\xi & \text{if } [\![r]\!]\xi = \overline{0} \\ [\![s]\!]\xi \vee [\![t]\!]\xi[p \mapsto 1, ps \mapsto \overline{q}] & \text{if } [\![r]\!]\xi = \overline{q+1} \end{cases} \\
[\![\Gamma \vdash \mathtt{pfold}\ r\ \mathtt{of}\ (s, [p, ps, w]t) : \gamma^{\square}]\!]\xi &= \begin{cases} [\![s]\!]\xi & \text{if } [\![r]\!]\xi = \overline{0} \\ \left(2 + \mathtt{rec}_c + ([\![t]\!]\xi')_c,\right. & \\ \left.\ ([\![s]\!]\xi)_p \vee ([\![t]\!]\xi')_p\right) & \text{if } [\![r]\!]\xi = \overline{q+1} \end{cases} \\
&\text{where } \mathtt{rec} := [\![\mathtt{pfold}\ y\ \mathtt{of}\ (s, [p, ps, w]t)]\!]\xi[y \mapsto \overline{q}] \\
&\text{and } \xi' := \xi[p \mapsto 1, ps \mapsto \overline{q}, w \mapsto (1, \mathtt{rec}_p)]
\end{aligned}
$$

FIGURE 3.7. Denotational Semantics of the Complexity Language

in the domain. We attribute a denotation only to typeable terms, and so our denotation function is $[\![\Gamma \vdash \cdot : \tau]\!] : (\text{CExp} \times \Gamma\text{-context}) \rightharpoonup [\![\tau]\!]$. When Γ and τ can be inferred from the context, we may simply write $[\![\mathbf{e}]\!]\xi$ in place of $[\![\Gamma \vdash \mathbf{e} : \tau]\!]\xi$.

In Figure 3.7 we define the denotational semantics for the complexity language. We reuse the **cost** and **pot** (or $\cdot_c$ and $\cdot_p$ respectively) notation to denote projections on the first and second elements of a pair, respectively. We also have introduced a new notation, `dally`, which adds its first argument to the cost of the second argument. Hence $\texttt{dally}(r, s) = (r + \mathbf{cost}(s), \mathbf{pot}(s))$.

To evaluate the maximum of two expressions, we push the computation through to the semantics. For a, b of the same type we define

$$a \vee b = \begin{cases} \mathtt{m} \vee \mathtt{n} & a = m, b = n \in [\![\mathbf{C}]\!] \\ 1 & a, b \in [\![\texttt{pint}]\!], [\![\texttt{pbool}]\!] \\ \overline{m \vee n} & a = \overline{m}, b = \overline{n} \in [\![\texttt{pint}*]\!] \\ (a_c \vee b_c, a_p \vee b_p) & a, b \in [\![\sigma \times \tau]\!] \\ \lambda v.av \vee bv & a, b \in [\![\sigma \to \tau]\!] \end{cases}$$

For the case of `plistcase`'s denotation, suppose $[\![r]\!]\xi = \overline{q + 1}$ for some $\overline{q + 1} \in \{1\}^*$. In this case the denotation of `plistcase` r `of` $(s, [p, ps]t)$ is defined to be the maximum of the two branches $[\![s]\!]\xi$ and $[\![t]\!]\xi[p \mapsto 1, ps \mapsto \overline{q}]$. The reason behind this is guided by the following intuition: if $[\![r]\!]\xi$ is a potential larger than $[\![r']\!]\xi$, then we want the complexity $[\![\texttt{plistcase}\ r\ \texttt{of}\ (s, [p, ps]t)]\!]$ to be larger than that of $[\![\texttt{plistcase}\ r'\ \texttt{of}\ (s, [p, ps]t)]\!]$. This is the intuition, but the necessity of the choice is motivated by the translation function, defined in Section 3.4. The translation function $\|\cdot\|$ is a map from target to complexity expressions. Our goal is to ensure that the complexity of a target expression t is bounded by the complexity of $\|t\|$. If $r\,\xi \downarrow ()\,[\ ]$ for some r, and r is bounded by $\|r\|$, it is possible that $[\![\|r\|_p]\!]\xi^* = \overline{q + 1}$ for some q. Hence in order to ensure that `listcase` r `of` $(s, [x, xs]t)$ is bounded by $\|\texttt{listcase}\ r\ \texttt{of}\ (s, [x, xs]t)\|$, we need for the denotation of `plistcase` expressions in the $\overline{q + 1}$ case to be greater than the complexity of either branch by itself.

For the denotation of `pfold` expressions, we do a similar trick, but only in the potential component of the denotation. Since all recursive calls are structural, the cost component of the base case is included within the cost component of the recursive step. Additionally we add a

dallied cost of 2 to the denotation of the recursive case, to account for the extra cost of evaluating recursive `fold` expressions.

3.4. Translation

The translation we describe in this section yields upper bounds on the cost of target language expressions.

3.4.1. Expressions. In Figure 3.8 we provide a translation from target language expressions Exp to complexity language expressions CExp. No expressions are free—the cost of any expression is at least one. All boolean expressions have potential $1^{\texttt{bool}}$ and all integer expressions have potential $1^{\texttt{int}}$, as expected.

In `if` statement translations, we perform the approximation discussed in Section 3.3. For every expression `if` r `then` s `else` t, $\|r\|_p$ has type `pbool` and so $[\![\|r\|_p]\!] = 1$. And yet r itself could evaluate to either `tt` or `ff`. To ensure that $\|$`if` r `then` s `else` $t\|$ provides an upper bound on the size of the target expression's evaluation derivation, we let the potential of $\|$`if` r `then` s `else` $t\|$ be the maximum of the potentials of $\|s\|$ and $\|t\|$.

In the `listcase` and `fold` translation, we reuse the `dally` construct, this time as syntactic shorthand within the complexity language. Since `plistcase` and `pfold` branch on the potential of $\|r\|$, so we use `dally` to account for the cost of $\|r\|$. Additionally, the variables p and ps in `plistcase` and `pfold` have potential type, but in $\|t\|$ the variables x and xs have complexity type. To reconcile these variables, we perform a syntactic substitution on $\|t\|$ during the translation, replacing x with $(\mathbf{1}, p)$ and xs with $(\mathbf{1}, ps)$.

3.4.2. Types. Since the complexity language consists of both potential types and complexity types, we need to reflect the distinction in the translation. The translation function $\|\cdot\| : \text{Type} \to \text{CType}$ sends τ to its corresponding complexity type, $C \times\!\!<\!\!<\tau>\!\!>$ or $<\!\!<\tau>\!\!>^{\square}$. The function $<\!\!<\cdot>\!\!>: \text{Type} \to \text{PType}$ sends τ to its potential type. For base types $\mathbf{b}$, the corresponding potential type is simply $\mathbf{pb}$. For arrow types on the other hand, $<\!\!<\sigma \to \tau>\!\!> = <\!\!<\sigma>\!\!> \to \|\tau\|$. Hence the potential of an arrow type is a function which takes potentials to complexity types. Note that $\|\sigma\| = <\!\!<\sigma>\!\!>^{\square}$ and $\|\sigma \to \tau\| = <\!\!<\sigma>\!\!> \to_{\square} \|\tau\|$.

$$
\begin{aligned}
\|x\| &= x \\
\|\texttt{true}\| = \|\texttt{false}\| &= (\mathbf{1}, 1^{\texttt{bool}}) \\
\|\mathbf{n}\| &= (\mathbf{1}, 1^{\texttt{int}}) \\
\|[\,]\| &= (\mathbf{1}, \mathtt{p}E) \\
\|r :: s\| &= (\mathbf{1} + \|r\|_c + \|s\|_c, \texttt{pcons}(\|r\|_p, \|s\|_p)) \\
\|\texttt{if } r \texttt{ then } s \texttt{ else } t\| &= \texttt{dally}(\mathbf{1} + \|r\|_c, \|s\| \vee \|t\|) \\
\|r\ R\ s\| &= (\mathbf{1} + \|r\|_c + \|s\|_c, 1^{\texttt{bool}}) \text{ for } R \in \{<, \leq, =\} \\
\|r \bullet s\| &= (\mathbf{1} + \|r\|_c + \|s\|_c, 1^{\texttt{int}}) \text{ for } \bullet \in \{+, -, \times, \div\} \\
\|\lambda x.r\| &= \lambda_* x.\|r\| \\
\|rs\| &= \|r\| * \|s\| \\
\|\texttt{listcase } r \texttt{ of } (s, [x, xs]t)\| &= \texttt{dally}\left(\mathbf{1} + \|r\|_c, \texttt{plistcase } \|r\|_p \texttt{ of } (\|s\|, [p, ps]\|t\|[\,{}^{x \mapsto (\mathbf{1},p)}_{xs \mapsto (\mathbf{1},ps)}\,])\right) \\
\|\texttt{fold } r \texttt{ of } (s, [x, xs, w]t)\| &= \texttt{dally}\left(\mathbf{1} + \|r\|_c, \texttt{pfold } \|r\|_p \texttt{ of } (\|s\|, [p, ps, w]\|t\|[\,{}^{x \mapsto (\mathbf{1},p)}_{xs \mapsto (\mathbf{1},ps)}\,])\right)
\end{aligned}
$$

FIGURE 3.8. Target Language Translation to Complexity Language

If Γ is a target type context, then $\|\Gamma\| = \{x : \|\sigma\| \mid (x, \sigma) \in \Gamma\}$ is a complexity type context. In the next theorem we prove that the translation scheme preserves typing judgements.

Theorem 2. *If* $\Gamma \vdash t : \tau$ *then* $\|\Gamma\| \vdash \|t\| : \|\tau\|$.

Before the proof of this theorem, we present a lemma that describes the type of a complexity expression which has undergone a syntactic substitution of the kind used in $\|\texttt{plistcase}\|$ and $\|\texttt{pfold}\|$.

Lemma 3. *If* $\Gamma, x : \gamma^{\square} \vdash r : \tau$ *then for any* $\mathbf{n}$ *of type* $\boldsymbol{C}$, $\Gamma, p : \gamma \vdash r[x \mapsto (\mathbf{n}, p)] : \tau$.

This lemma is proved by straightforward induction on r. With it we can go on to prove Theorem 2.

PROOF. We proceed by structural induction on the term t.

If $\Gamma, x : \tau \vdash x : \tau$ then $||\Gamma, x : \tau||(x) = ||\tau||$. Since $||x|| = x$, we have the desired result. The other axioms and the *cons*, *if*, *rel* and *op* inference rules are proved by straightforward application of the translation rules.

Now suppose $\Gamma \vdash \lambda x.r : \sigma \to \tau$. In order to prove $||\Gamma|| \vdash \lambda_* x.||r|| : \ll \sigma \gg \to_\Box ||\tau||$, we need to show that $||\Gamma||, x : \ll \sigma \gg^\Box \vdash ||r|| : ||\tau||$. By the *lambda* typing rule for the target language, we know that $\Gamma, x : \sigma \vdash r : \tau$. Hence the inductive hypothesis states that $||\Gamma, x : \sigma|| \vdash ||r|| : ||\tau||$ or in other words, $||\Gamma||, x : ||\sigma|| \vdash ||r|| : ||\tau||$. Since $||\sigma|| = \ll \sigma \gg^\Box$, the complexity language typing rule for *lambda*$*$ applies to give the desired result.

Now suppose $\Gamma \vdash rs : \tau$. By the target language typing inference rule, we know that there is some type σ with $\Gamma \vdash r : \sigma \to \tau$ and $\Gamma \vdash s : \sigma$. Applying the inductive hypothesis, we have that $||\Gamma|| \vdash ||r|| : \ll \sigma \gg \to_\Box ||\tau||$ and also $||\Gamma|| \vdash ||s|| : ||\sigma||$. But these are exactly the premises for the *app* inference rule for complexity language typing. Hence $||\Gamma|| \vdash ||r|| * ||s|| : ||\tau||$.

Suppose $\Gamma \vdash \texttt{listcase}\ l\ \texttt{of}\ (r, [x, xs]s) : \tau$. We have that $||\texttt{listcase}\ l\ \texttt{of}\ (r, [x, xs]s)|| = \texttt{dally}(\mathbf{1} + ||l||_c, \texttt{plistcase}\ ||l||_p\ \texttt{of}\ (||r||, [p, ps]s'))$ where $s' = ||s||[x \mapsto (\mathbf{1}, p), xs \mapsto (\mathbf{1}, ps)]$. First we will show that $||\Gamma|| \vdash ||l|| : \gamma^\Box$ for some potential type γ. From that it is sufficient to show that $||\Gamma|| \vdash \texttt{plistcase}\ ||l||_p\ \texttt{of}\ (||r||, [p, ps]s') : ||\tau||$.

In fact, $||\Gamma|| \vdash ||l|| : \texttt{pint}*^\Box$ by the inductive hypothesis. As a reminder, we have the following inference rule for the typing of `listcase` expressions:

$$\frac{\Gamma \vdash l : \texttt{int}* \quad \Gamma \vdash r : \tau \quad \Gamma, x : \texttt{int}, xs : \texttt{int}* \vdash s : \tau}{\Gamma \vdash \texttt{listcase}\ l\ \texttt{of}\ (r, [x, xs]s) : \tau}$$

As a result we know that $||\Gamma|| \vdash ||l||_p : \texttt{pint}*$ and also that $||\Gamma|| \vdash ||r|| : ||\tau||$. The remaining inductive hypothesis is that

$$||\Gamma||, x : \texttt{pint}^\Box, xs : \texttt{pint}*^\Box \vdash ||s|| : ||\tau||$$

By Lemma 3 this gives us that

$$||\Gamma||, p : \texttt{pint}, ps : \texttt{pint}* \vdash ||s||[x \mapsto (\mathbf{1}, p), xs \mapsto (\mathbf{1}, ps)] : ||\tau||$$

Hence $||\Gamma|| \vdash \texttt{plistcase}\ ||l||_p\ \texttt{of}\ (||r||, [p, ps]s') : ||\tau||$.

The same reasoning proves the `fold` case. □

3.4.3. Soundness. In this section we will show that the complexity translation of a target term yields an upper bound on its cost. Alone, the cost of a closure $t\,\xi$ is the number of inference rules in the derivation of $t\,\xi \downarrow v\,\theta$. However it is not enough just to show that $\mathbf{cost}(t) \leq \mathbf{cost}(\llbracket ||t|| \rrbracket \xi)$ because this comparison fails to consider the term's potential.

With this in mind, we define a bounding relation $\sqsubseteq$ so that $t\,\xi \sqsubseteq \chi$ if the entire complexity of t is bounded by the complexity χ. In particular, let $\Gamma \vdash t : \sigma$ and let ξ be a value environment consistent with Γ (that is, $\xi(x) : \Gamma(x)$ for all $x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}\xi$). Let χ be a complexity in the denotational semantics of the complexity language such that $\chi \in \llbracket ||\sigma|| \rrbracket$. Then $t\,\xi \sqsubseteq_\sigma \chi$ if two conditions hold:

(1) $\mathbf{cost}(t\,\xi) \leq \mathbf{cost}(\chi)$ and

(2) if $t\,\xi \downarrow v\,\theta$ then $v\,\theta \sqsubseteq^{\mathbf{pot}}_\sigma \mathbf{pot}(\chi)$

With the potential bounding relation $\sqsubseteq^{\mathbf{pot}}$ we can compare values in the target language with semantic potentials $q \in \mathbf{pot}(\llbracket ||\sigma|| \rrbracket)$. The following rules define the relation:

$$\mathtt{tt}\,\theta, \mathtt{ff}\,\theta \sqsubseteq^{\mathbf{pot}}_{\mathtt{bool}} 1$$

$$\mathtt{n}\,\theta \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}} 1$$

$$(n_0, \ldots, n_{k-1})\,\theta \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}*} \overline{q} \text{ if } k \leq q$$

$$(\lambda x.r)\ \theta \sqsubseteq^{\mathbf{pot}}_{\sigma\to\tau} q \text{ if } \left(z\,\eta \sqsubseteq^{\mathbf{pot}}_\sigma p \Rightarrow r\,\theta[x \mapsto z\,\eta] \sqsubseteq_\tau q(p)\right)$$

For ξ a Γ-environment and ξ^* a $||\Gamma||$-environment, we say that $\xi \sqsubseteq \xi^*$ if, for all $x \in \mathrm{dom}(\xi) \cap \mathrm{dom}(\xi^*)$, $x\,\xi \sqsubseteq \xi^*(x)$. Then, suppose $\Gamma \vdash t : \tau$ and $||\Gamma|| \vdash t^* : ||\tau||$. We say $t \sqsubseteq t^*$ if, for all Γ-environments ξ and $||\Gamma||$-environments ξ^* such that $\xi \sqsubseteq \xi^*$, we have $t\,\xi \sqsubseteq_\tau \llbracket t^* \rrbracket \xi^*$.

Again we introduce a substitution lemma to be used in the proof of soundness.

Lemma 4.

$$\llbracket t \rrbracket \xi[x \mapsto (a, b)] = \llbracket t[x \mapsto (a, y)] \rrbracket \xi[y \mapsto b]$$

The proof is straightforward by induction on t.

We are now in a position to prove that the maximum operator on complexities and potentials does not affect the bounding relation.

Lemma 5. *If $t\,\xi \sqsubseteq_\tau \chi$ then for all $\chi' \in \llbracket ||\tau|| \rrbracket$, $t\,\xi \sqsubseteq_\tau \chi \vee \chi'$.*

Proof. The cost component of the bounding relation refers exclusively to natural numbers. If $t\,\xi \sqsubseteq_\tau \chi$ then $\mathbf{cost}(t\xi) \leq \mathbf{cost}(\chi) \leq \mathbf{cost}(\chi) \vee \mathbf{cost}(\chi')$. So the theorem can be reduced to proving the potential bound. Suppose $t\,\xi \downarrow v\,\theta$. We want to show that $v\,\theta \sqsubseteq^{\mathbf{pot}}_\tau \mathbf{pot}(\chi) \vee \mathbf{pot}(\chi')$. If τ is a base type then the proof is trivial, so consider $\sigma \to \tau$. We know $v = \lambda x.r$ for some target expression r; we want to show that $(\lambda x.r)\ \theta \sqsubseteq^{\mathbf{pot}}_{\sigma\to\tau} (\chi \vee \chi')_p = \chi_p \vee \chi'_p$. Suppose $z\,\eta \sqsubseteq^{\mathbf{pot}}_\sigma \alpha$. We know that $r\,\theta[x \mapsto z\eta] \sqsubseteq_\tau \chi(\alpha)$ so by the inductive hypothesis, $r\,\theta[x \mapsto z\eta] \sqsubseteq_\tau \chi'(\alpha) \vee \mu$ for every $\mu \in [\![||\tau||]\!]$. But since $\chi'_p \in [\![||\tau||]\!]^{[\![<< \sigma >>]\!]}$, we get

$$r\,\theta[x \mapsto z\eta] \sqsubseteq_\tau \chi_p(\alpha) \vee \chi'_p(\alpha) = (\chi_p \vee \chi'_p)(\alpha)$$

Hence $(\lambda x.r)\ \theta \sqsubseteq^{\mathbf{pot}}_{\sigma\to\tau} \chi_p \vee \chi'_p$. □

Theorem 6 (Soundness)**.** *If* $\Gamma \vdash t : \tau$ *then* $t \sqsubseteq ||t||$.

Proof. We approach the proof by induction on the typing derivation $\mathcal{D}$ of $\Gamma \vdash t : \tau$. Fix $\xi \sqsubseteq \xi^*$. For simplicity in this proof, we will use the notation $||t||\xi^*$ to stand for $[\![||t||]\!]\xi^*$, the denotation of $||t||$ under the environment ξ^*.

($\mathcal{D} = \Gamma \vdash x : \tau$) Since $\xi \sqsubseteq \xi^*$, we know $x\,\xi \sqsubseteq_\sigma \xi^*(x) = ||x||\xi^*$.

($\mathcal{D} = \Gamma \vdash \texttt{true} : \texttt{bool}$) We know first of all that $\mathbf{cost}(\texttt{true}\,\xi) = 1$ and $\mathbf{cost}(||\texttt{true}||\xi) = \mathbf{cost}(1, 1) = 1$. Furthermore we must have $\texttt{true}\,\xi \downarrow \texttt{tt}\,[\,]$ and we know $\texttt{tt}\,[\,] \sqsubseteq^{\mathbf{pot}}_{\texttt{pbool}} 1$.

If $\mathcal{D}$ is the derivation for any of `false`, `n` or $[\,]$, the proof is similarly straightforward.

($\mathcal{D} = \Gamma \vdash r :: s : \texttt{int}*$) We are given the following two induction hypotheses:

$$\eta \sqsubseteq \eta^* \Rightarrow r\,\eta \sqsubseteq_{\texttt{int}} ||r||\eta^*$$

$$\eta \sqsubseteq \eta^* \Rightarrow s\,\eta \sqsubseteq_{\texttt{int}*} ||s||\eta^*$$

By unfolding the definition of the translation function, we see that

$$||r :: s|| = (1 + ||r||_c + ||s||_c, \texttt{pcons}(1^{\texttt{int}}, ||s||_p)) \qquad \text{and}$$

$$||r :: s||\xi^* = (1 + [\![||r||_c]\!]\xi^* + [\![||s||_c]\!]\xi^*, \overline{l + 1}) \qquad \text{where } [\![||s||_p]\!]\xi^* = \bar{l}$$

First, notice that the cost of $(r :: s)\ \xi$ is $1 + \mathbf{cost}(r) + \mathbf{cost}(s)$ which, by the induction hypothesis, is less than or equal to $1 + \mathbf{cost}(||r||\xi^*) + \mathbf{cost}(||s||\xi^*)$. Next we need to show that if

$t\,\xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,]$ then $(n, n_0, \ldots, n_{k-1})\,[\,] \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}*} \overline{l+1}$. Again by the induction hypothesis we know that $k \le l$, so the claim holds.

The cases for `if` statements, operators and relations are similar to the `cons` case, and will not be enumerated here.

($\mathcal{D} = \Gamma \vdash \lambda x.r : \sigma \to \tau$) As an inductive hypothesis we know that for all $\eta \sqsubseteq \eta^*$, $r\,\eta \sqsubseteq_\tau \|r\|\eta^*$. Notice that

$$\|\lambda x.r\|\xi^* = [\![\lambda_* x.\|r\|]\!]\xi^* = (1, \lambda v.\|r\|\xi^*[x \mapsto (1, v)])$$

so we want to show that $(\lambda x.r)\ \xi \sqsubseteq^{\mathbf{pot}}_{\sigma\to\tau} \lambda v.\|r\|\xi^*[x \mapsto (1, v)]$. Suppose $z\,\eta \sqsubseteq^{\mathbf{pot}}_\sigma p$. Then $\xi[x \mapsto z\eta] \sqsubseteq \xi^*[x \mapsto (1, p)]$. By the inductive hypothesis we know that $r\,\xi[x \mapsto z\eta] \sqsubseteq_\tau \|r\|\xi^*[x \mapsto (1, p)]$. So the abstraction rule for potential bounds gives us the desired judgement.

($\mathcal{D} = \Gamma \vdash rs : \tau$) By unraveling definitions,

$$\|rs\|\xi^* = \mathtt{dally}\left(1 + (\|r\|\xi^*)_c + (\|s\|\xi^*)_c, (\|r\|\xi^*)_p(\|s\|\xi^*)_p\right)$$

First we will prove the potential bound. Suppose $(rs)\ \xi \downarrow v\,\theta$ by the following evaluation rule:

$$\frac{r\,\xi \downarrow (\lambda x.r')\ \theta_1 \quad s\,\xi \downarrow w\,\theta_2 \quad r'\,\theta_1[x \mapsto w\theta_2] \downarrow v\,\theta}{(rs)\ \xi \downarrow v\,\theta}$$

We want to show that $v\,\theta \sqsubseteq^{\mathbf{pot}}_\tau \mathbf{pot}((\|r\|\xi^*)_p(\|s\|\xi^*)_p)$. Since $r\,\xi \downarrow (\lambda x.r')\ \theta_1$, we know that $(\lambda x.r')\ \theta_1 \sqsubseteq^{\mathbf{pot}}_{\sigma\to\tau} \mathbf{pot}\|r\|\xi^*$. Hence if $z\,\eta \sqsubseteq^{\mathbf{pot}}_\sigma p$ then $r'\,\theta_1[x \mapsto z\eta] \sqsubseteq_\tau (\|r\|\xi^*)_p(p)$. We know $w\,\theta_2 \sqsubseteq^{\mathbf{pot}}_\sigma (\|s\|\xi^*)_p$, so $r'\,\theta_1[x \mapsto w\theta_2] \sqsubseteq_\tau (\|r\|\xi^*)_p(\|s\|\xi^*)_p$. Therefore $v\,\theta \sqsubseteq^{\mathbf{pot}}_\tau \mathbf{pot}\left((\|r\|\xi^*)_p(\|s\|\xi^*)_p\right)$.

The previous observation also shows that $\mathbf{cost}(r'\theta_1[x \mapsto w\theta_2]) \le \mathbf{cost}\left((\|r\|\xi^*)_p(\|s\|\xi^*)_p)\right)$. Notice that

$$\begin{aligned}
\mathbf{cost}((rs)\ \xi) &= 1 + \mathbf{cost}(r\,\xi) + \mathbf{cost}(s\,\xi) + \mathbf{cost}(r'\theta_1[x \mapsto w\theta_2]) \\
&\le 1 + \mathbf{cost}(\|r\|\xi^*) + \mathbf{cost}(\|s\|\xi^*) + \mathbf{cost}\left((\|r\|\xi^*)_p(\|s\|\xi^*)_p)\right) \\
&= \mathbf{cost}(\|rs\|\xi^*)
\end{aligned}$$

by the induction hypotheses.

($\mathcal{D} = \Gamma \vdash \mathtt{listcase}\ r\ \mathtt{of}\ (s, [x, xs]t) : \tau$) Recall that

$$\|\mathtt{listcase}\ r\ \mathtt{of}\ (s, [x, xs]t)\| = \mathtt{dally}(1 + \|r\|_c, \mathtt{plistcase}\ \|r\|_p\ \mathtt{of}\ (\|s\|, [p, ps]\|t\|\left[\begin{smallmatrix} x\mapsto(1,p) \\ xs\mapsto(1,ps)\end{smallmatrix}\right]))$$

Let t' be $\|t\|[x \mapsto (1,p), xs \mapsto (1,ps)]$. Then

$$\|\texttt{listcase}\ r\ \texttt{of}\ (s,[x,xs]t)\|\xi^* = \texttt{dally}\left(1 + [\![\|r\|_c]\!]\xi^*, \begin{cases} \|s\|\xi^* & \text{if } [\![\|r\|_p]\!]\xi^* = \overline{0} \\ \|s\|\xi^* \vee [\![t']\!]\xi^*\left[\begin{smallmatrix} p\mapsto 1 \\ ps\mapsto\overline{q}\end{smallmatrix}\right] & \text{if } [\![\|t\|_p]\!]\xi^* = \overline{q+1} \end{cases}\right)$$

We have two cases to consider. First, suppose $r\,\xi \downarrow ()\,[\,]$. Then

$$\mathbf{cost}(\texttt{listcase}\ r\ \texttt{of}\ (s,[x,xs]t)) = \mathbf{cost}(r) + \mathbf{cost}(s) + 1 \leq \mathbf{cost}(\|r\|\xi^*) + \mathbf{cost}(\|s\|\xi^*) + 1$$

by the inductive hypotheses for r and s. The cost of $\|\texttt{listcase}\ r\ \texttt{of}\ (s,[x,xs]t)\|\xi^*$ is one of $1 + (\|r\|\xi^*)_c + (\|s\|\xi^*)_c$ or $1 + (\|r\|\xi^*)_c + (\|s\|\xi^*)_c \vee (\|t\|\xi^{**})_c$ for some ξ^{**}, depending on the value of $\|r_p\|\xi^*$. Both of these values bound the size of the evaluation derivation. Now, if $(\texttt{listcase}\ r\ \texttt{of}\ (s,[x,xs]t))\ \xi \downarrow v\,\theta$, we know that $s\,\xi \downarrow v\,\theta$ as well, so by the inductive hypothesis, $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} (\|s\|\xi^*)_p$. By Lemma 5 we also have

$$v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} (\|s\|\xi^*)_p \vee (\|t\|\xi^{**})_p$$

for any ξ^{**}, which gives us the desired result.

Next, suppose $r\,\xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,]$, and so $(\|r\|\xi^*)_p = \overline{q+1}$ for some $q \geq k$. By the inductive hypothesis for t, and by Lemma 4,

$$t\,\xi[x\mapsto n[\,],xs\mapsto(n_0,\ldots,n_{k-1})[\,]] \sqsubseteq_\tau \|t\|\xi^*[x\mapsto(\mathbf{1},1^{\mathtt{int}}),xs\mapsto(\mathbf{1},\overline{q})] = [\![t']\!]\xi^*[p\mapsto 1^{\mathtt{int}},ps\mapsto\overline{q}]$$

This gives us the desired upper bound on cost:

$$\begin{aligned}
\mathbf{cost}(\texttt{listcase}\ r\ \texttt{of}\ (s,[x,xs]t)\,\xi) &= 1 + \mathbf{cost}(r\,\xi) + \mathbf{cost}(t\,\xi[x\mapsto n[\,],xs\mapsto(n_0,\ldots,n_{k-1})[\,]]) \\
&\leq 1 + (\|r\|\xi^*)_c + ([\![t']\!]\xi^*[p\mapsto 1^{\mathtt{int}},ps\mapsto\overline{q}])_c \\
&\leq 1 + (\|r\|\xi^*)_c + (\|s\|\xi^*)_c \vee ([\![t']\!]\xi^*[p\mapsto 1,ps\mapsto\overline{q}])_c \\
&= (\|\texttt{listcase}\ r\ \texttt{of}\ (s,[x,xs]t)\|\xi^*)_c
\end{aligned}$$

Meanwhile, if $(\texttt{listcase}\ r\ \texttt{of}\ (s,[x,xs]t))\ \xi \downarrow v\,\theta$ then $t\,\xi[x\mapsto n[\,],xs\mapsto(n_0,\ldots,n_{k-1})] \downarrow v\,\theta$ and so $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} \mathbf{pot}([\![t']\!]\xi^*[p\mapsto 1^{\mathtt{int}},ps\mapsto\overline{q}])$. Thus by Lemma 5, $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} (\|s\|\xi^*)_p \vee ([\![t']\!]\xi^*[p\mapsto 1^{\mathtt{int}},ps\mapsto\overline{q}])_p$.

Before we prove the final case regarding `fold` expressions, we introduce the following two claims.

Lemma 7. *For all complexity expressions r,* $\boldsymbol{cost}([\![s]\!]\xi^*) \leq \boldsymbol{cost}([\![\texttt{pfold}\,r\,\texttt{of}\,(s,[p,ps,w]t)]\!]\xi^*)$.

Proof. We prove this claim by induction on the value of $[\![r]\!]\xi^*$. If $[\![r]\!]\xi^* = \overline{0}$, then the left hand side of the inequality is exactly the right hand side. If $[\![r]\!]\xi^* = \overline{q+1}$, then

$$([\![\texttt{pfold}\ r\ \texttt{of}\ (s,[p,ps,w]t)]\!]\xi^*)_c = 2 + \texttt{rec}_c + ([\![t]\!]\xi^*[p \mapsto 1, ps \mapsto \overline{q}, w \mapsto (1, \texttt{rec}_p)])_c$$

where $\texttt{rec} = [\![\texttt{pfold}\ y\ \texttt{of}\ (s,[p,ps,w]t)]\!]\xi^*[y \mapsto \overline{q}]$. By the inductive hypothesis , $([\![s]\!]\xi^*)_c \leq \texttt{rec}_c$ so it is less than the cost of $[\![\texttt{pfold}\ r\ \texttt{of}\ (s,[p,ps,w]t)]\!]\xi^*$. □

Lemma 8. *Suppose $s \sqsubseteq \|s\|$ and $t \sqsubseteq \|t\|$. Fix $\xi \sqsubseteq \xi^*$ and let $\xi_0 = \xi[y \mapsto (n_0, \ldots, n_{k-1})[\,]]$ and $\xi_0^* = \xi^*[y \mapsto \overline{q}]$, where $k \leq q$. Assume y is not free in either s or t. Then*

$$(\texttt{fold}\ y\ \texttt{of}\ (s,[x,xs,w]t))\ \xi_0 \sqsubseteq \texttt{dally}\Big(2, [\![\texttt{pfold}\ y\ \texttt{of}\ (\|s\|,[p,ps,w]t')]\!]\xi_0^*\Big)$$

where $t' = \|t\|[x \mapsto (1,p), xs \mapsto (1,ps)]$.

Proof. We prove the claim by induction on k. If $k = 0$ then the cost of the fold expression is $2 + \mathbf{cost}(s\,\xi_0)$, and the value of the fold expression under ξ_0 is the value to which $s\xi_0$ evaluates. We will prove the potential bound first. By the inductive hypothesis $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} (\|s\|\xi_0^*)_p$. Since the potential of $\texttt{dally}(2, [\![\texttt{pfold}\ y\ \texttt{of}\ (\|s\|,[p,ps,w]t')]\!]\xi_0^*)$ is either $(\|s\|\xi_0^*)_p$ or $(\|s\|\xi_0^*)_p \vee ([\![t']\!]\xi_0^{**})_p$ for some environment ξ_0^{**}, we know from Lemma 5 that the potential bound holds. The cost bound is proved by Lemma 7:

$$\begin{aligned}\mathbf{cost}(\texttt{fold}\ y\ \texttt{of}\ (s,[x,xs,w]t)\,\xi_0) &\leq 2 + (\|s\|\xi_0^*)_c \\ &\leq 2 + ([\![\texttt{pfold}\ y\ \texttt{of}\ (\|s\|,[p,ps,w]t')]\!]\xi_0^*)_c\end{aligned}$$

Suppose $\xi_0(y) = (n, n_0, \ldots, n_{k-1})[\,]$. Then $\xi_0^*(y) = \overline{q+1}$ for some $q \geq k$. By the induction hypothesis we know that

$$(\texttt{fold}\ y\ \texttt{of}\ (s,[x,xs,w]t))\ \xi[y \mapsto (n_0,\ldots,n_{k-1})[\,]] \sqsubseteq_\tau \texttt{dally}(2, \texttt{rec})$$

where $\texttt{rec} = [\![\texttt{pfold}\ y\ \texttt{of}\ (\|s\|,[p,ps,w]t')]\!]\xi^*[y \mapsto \overline{q}]$. So if

$$(\texttt{fold}\ y\ \texttt{of}\ (s,[x,xs,w]t))\ \xi[y \mapsto (n_0,\ldots,n_{k-1})[\,]] \downarrow v'\ \theta'$$

then $v'\ \theta' \sqsubseteq_\tau^{\mathbf{pot}} \texttt{rec}_p$, which means

$$\xi[x \mapsto n[\,], xs \mapsto (n_0,\ldots,n_{k-1})[\,], w \mapsto v'\theta'] \sqsubseteq \xi^*[x \mapsto (1,1), xs \mapsto (1,\overline{q}), w \mapsto (1, \texttt{rec}_p)]$$

Let $\xi_1 := \xi_0[x \mapsto n[\,], xs \mapsto (n_0, \ldots, n_{k-1})[\,], w \mapsto v'\theta']$ and let $\xi_1^* := \xi_0^*[p \mapsto 1, ps \mapsto \overline{q}, w \mapsto (1, \mathtt{rec}_p)]$. Since y does not occur free in t or t', and by the substitution result Lemma 4, $t\,\xi_1 \sqsubseteq_\tau [\![t']\!]\xi_1^*$. So

$$\begin{aligned}
&\mathbf{cost}(\mathtt{fold}\ y\ \mathtt{of}\ (s, [x, xs, w]t)\,\xi_0) \\
&= 2 + \mathbf{cost}(\mathtt{fold}\ y\ \mathtt{of}\ (s, [x, xs, w]t)\,\xi[y \mapsto (n_0, \ldots, n_{k-1})[\,]]) + \mathbf{cost}(t\,\xi_1) \\
&\leq 2 + (2 + \mathtt{rec}_c) + \left([\![t']\!]\xi_1^*\right)_c \\
&= 2 + \left([\![\mathtt{pfold}\ y\ \mathtt{of}\ (\|s\|, [p, ps, w]t')]\!]\xi_0^*\right)_c
\end{aligned}$$

For the potential bound, we know that if $(\mathtt{fold}\ y\ \mathtt{of}\ (s, [x, xs, w]t))\ \xi_0 \downarrow v\,\theta$ then $t\,\xi_1 \downarrow v\,\theta$, so $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} \mathbf{pot}([\![t']\!]\xi_1^*)$. By Lemma 5, we have

$$v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} (\|s\|\xi_0^*)_p \vee ([\![t']\!]\xi_1^*)_p = \mathbf{pot}\left([\![\mathtt{pfold}\ y\ \mathtt{of}\ (\|s\|, [p, ps, w]t')]\!]\xi_0^*\right)$$

□

Now we may finish up the proof of soundness.

$(\mathcal{D} = \Gamma \vdash \mathtt{fold}\ r\ \mathtt{of}\ (s, [x, xs, w]t) : \tau)$ If $r\,\xi \downarrow ()\,[\,]$ then

$$\begin{aligned}
\mathbf{cost}(\mathtt{fold}\ r\ \mathtt{of}\ (s, [x, xs, w]t)\,\xi) &= 1 + \mathbf{cost}(r\,\xi) + \mathbf{cost}(s\,\xi) \\
&\leq 1 + (\|r\|\xi^*)_c + (\|s\|\xi^*)_c \\
&\leq 1 + (\|r\|\xi^*)_c + ([\![\mathtt{pfold}\ \|r\|_p\ \mathtt{of}\ (\|s\|, [p, ps, w]t')]\!]\xi^*)_c \\
&= \mathbf{cost}(\|\mathtt{fold}\ r\ \mathtt{of}\ (s, [x, xs, w]t)\|\xi^*)
\end{aligned}$$

by Lemma 7. Additionally, if $r\,\xi \downarrow ()\,[\,]$ and $(\mathtt{fold}\ r\ \mathtt{of}\ (s, [x, xs, w]t))\ \xi \downarrow v\,\theta$ then we know $s\,\xi \downarrow v\,\theta$ as well. Hence $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} \mathbf{pot}(\|s\|\xi^*)$ and so $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} (\|s\|\xi^*)_p \vee ([\![t']\!]\xi^{**})_p$ for any environment ξ^{**}. Thus $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} \mathbf{pot}(\|\mathtt{fold}\ r\ \mathtt{of}\ (s, [x, xs, w]t)\|\xi^*)$.

Suppose $r\,\xi \downarrow (n, n_0, \ldots, n_{k-1})\,[\,]$. Before we prove the cost and potential bounds, we develop some of the implications of our induction hypotheses. By the hypothesis which states $r \sqsubseteq \|r\|$, $\|r\|_p\xi^* = \overline{q+1}$ for some $q \geq k$. Let $\mathtt{rec}^t := \mathtt{fold}\ y\ \mathtt{of}\ (s, [x, xs, w]t)$, let $t' := \|t\|[x \mapsto (1, p), xs \mapsto (1, ps)]$, and let

$$\mathtt{rec} := [\![\mathtt{pfold}\ y\ \mathtt{of}\ (\|s\|, [p, ps, w]t')]\!]\xi^*[y \mapsto \overline{q}]$$

By Lemma 8, $\texttt{rec}^t\,\xi[y \mapsto (n_0, \ldots, n_{k-1})[\,]] \sqsubseteq_\tau \texttt{dally}(2, \texttt{rec})$. So if $\texttt{rec}^t\,\xi[y\mapsto(n_0,\ldots,n_{k-1})[\,]] \downarrow v'\,\theta'$ then $v'\,\theta' \sqsubseteq_\tau^{\mathbf{pot}} \texttt{rec}_p$; hence

$$\xi[x \mapsto n[\,], xs \mapsto (n_0, \ldots, n_{k-1})[\,], w \mapsto v'\theta'] \sqsubseteq \xi^*[x \mapsto (1,1), xs \mapsto (1, \overline{q}), w \mapsto (1, \texttt{rec}_p)].$$

Now, let $\xi_1 = \xi[x\mapsto n[\,], xs\mapsto(n_0,\ldots,n_{k-1})[\,], w\mapsto v'\theta']$ and $\xi_1^* = \xi^*[p\mapsto 1, ps\mapsto\overline{q}, w\mapsto(1,\texttt{rec}_p)]$. Then by the substitution lemma, Lemma 4, we have $t\,\xi_1 \sqsubseteq_\tau [\![t']\!]\xi_1^*$.

For the cost bound, we have

$$\begin{aligned}
\mathbf{cost}(\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t)\,\xi) &= 1 + \mathbf{cost}(r\,\xi) + \mathbf{cost}(\texttt{rec}^t\xi[y\mapsto(n_0,\ldots,n_{k-1})[\,]]) + \mathbf{cost}(t\,\xi_1) \\
&\leq 1 + (\|r\|\xi^*)_c + (2 + \texttt{rec}_c) + ([\![t']\!]\xi_1^*)_c \\
&= 1 + (\|r\|\xi^*)_c + \mathbf{cost}([\![\texttt{pfold}\ \|r\|_p\ \texttt{of}\ (\|s\|, [p, ps, w]t')]\!]\xi^*) \\
&= \mathbf{cost}(\|\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t)\|\xi^*)
\end{aligned}$$

For the potential bound, suppose $(\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t))\ \xi \downarrow v\,\theta$. Then we must have $t\,\xi_1 \downarrow v\,\theta$ as well; hence $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} \mathbf{pot}([\![t']\!]\xi_1^*)$. So by Lemma 5,

$$v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} \mathbf{pot}(\|s\|\xi^*) \vee \mathbf{pot}([\![t']\!]\xi_1^*) = \mathbf{pot}(\|\texttt{fold}\ r\ \texttt{of}\ (s, [x, xs, w]t)\|\xi^*)$$

□

Theorem 6 gives us the goal we stated at the beginning of the section.

Corollary 9. *If* $\{\} \vdash t : \tau$ *then* $\boldsymbol{cost}(t\,[\,]) \leq \boldsymbol{cost}(\|t\|\,[\,])$.

3.4.4. A small example: `insert`. The bounding relation is our main tool to compare target language expressions and complexity expressions. In fact we can choose to express the bounding relation in one of two ways: either as $t\,\xi \sqsubseteq_\tau \chi$ for $\chi \in [\![\|\tau\|]\!]$ or as $t \sqsubseteq t^*$. The first case allows us to cleanly compare the complexities, but the second case allows us to work within the complexity language. For example, in this way we can compare the complexities of two target terms $r\xi$ and $s\xi'$ by testing if $r \sqsubseteq \|s\|$. Additionally, by working within the complexity language as opposed to simply working with denotations, we can more easily talk about the simplification of terms which bound a target expression.

In this section we will prove an upper bound on the complexity of the insert function, which inserts a number into a sorted list such that the resulted list is sorted. `insert` can be

implemented in the target language as follows:

$$\texttt{insert} := \lambda x.\lambda xs.\texttt{fold}\ xs\ \texttt{of}\ (x :: [\,], [y, ys, w]\texttt{if}\ x \leq y\ \texttt{then}\ x :: y :: ys\ \texttt{else}\ y :: w)$$

By soundness we know that the complexity of `insert` is bounded by its translation. However the direct translation of `insert` is extremely unwieldy. Consider the subterm of `insert`:

$$\begin{aligned}
&\|\texttt{if}\ x \leq y\ \texttt{then}\ x :: y :: ys\ \texttt{else}\ y :: w\| \\
&\quad = \texttt{dally}\,(\mathbf{1} + \|x \leq y\|_c, \|x :: y :: ys\| \vee \|y :: w\|) \\
&\quad = \texttt{dally}\,(\mathbf{1} + (\mathbf{1} + x_c + y_c, 1^{\texttt{int}})_c \\
&\qquad (\mathbf{1} + x_c + \|y :: ys\|_c, \texttt{pcons}(x_p, \|y :: ys\|_p)) \vee (\mathbf{1} + y_c + w_c, \texttt{pcons}(y_p, w_p))\big) \\
&\quad = \texttt{dally}\,(\mathbf{1} + (\mathbf{1} + x_c + y_c, 1^{\texttt{int}})_c, \\
&\qquad (\mathbf{1} + x_c + (\mathbf{1} + y_c + ys_c, \texttt{pcons}(y_p, ys_p))_c, \\
&\qquad \texttt{pcons}(x_p, (\mathbf{1} + y_c + ys_c, \texttt{pcons}(y_p, ys_p))_p) \vee (\mathbf{1} + y_c + w_c, \texttt{pcons}(y_p, w_p)))\big)
\end{aligned}$$

We have some intuition that adding constant costs and taking cost and potential projections of a pair does not alter the complexity of the target expression. After all, if $[\![s]\!]\xi^* = [\![t]\!]\xi^*$ for all ξ^*, then $r \sqsubseteq s \Leftrightarrow r \sqsubseteq t$. Performing these simple substitutions, the translation of $\texttt{if}\ x \leq y\ \texttt{then}\ x :: y :: ys\ \texttt{else}\ y :: w$ will have the same denotation as

$$\begin{aligned}
&\texttt{dally}\,(\mathbf{2} + x_c + y_c, \\
&(\mathbf{2} + x_c + y_c + ys_c, \texttt{pcons}(x_p, \texttt{pcons}(y_p, ys_p))) \vee (\mathbf{1} + y_c + w_c, \texttt{pcons}(y_p, w_p)))\big)
\end{aligned}$$

In fact, there is even more information we can deduce about this translation. Everywhere the variables y and ys occur, we know they will be mapped to $(1, p)$ and $(1, ps)$ respectively as part of the translation of `pfold`, outside the scope of our small example. Since $[\![y_c]\!] = [\![ys_c]\!] = 1$, we should be able to replace every instance of y_c and ys_c with $\mathbf{1}$. Similarly, since the variable x is bound by a lambda-abstraction, it will be mapped to $(1, v)$ in the denotation, so we will also replace x_c by $\mathbf{1}$. And because the terms x_p and p are of type $\texttt{pint}*$, we can replace them by $1^{\texttt{int}}$. Making these substitutions we obtain

$$\texttt{dally}\,\big(\mathbf{4}, (\mathbf{5}, \texttt{pcons}(1^{\texttt{int}}, \texttt{pcons}(1^{\texttt{int}}, ys_p))) \vee (\mathbf{2} + w_c, \texttt{pcons}(1^{\texttt{int}}, w_p))\big)$$

Ideally we would like to avoid the max operator altogether. We first notice that $\mathbf{9} + w_c \geq \mathbf{4} + (\mathbf{5} \vee (\mathbf{2} + w_c))$. Additionally, it is worth noting that if w gets mapped to the result of the recursive call on an input of length n, then the result will be a list of length $n + 1$, a fact which can be verified by induction. Therefore in our case $\texttt{pcons}(1^{\texttt{int}}, \texttt{pcons}(1^{\texttt{int}}, ys_p)) \vee \texttt{pcons}(1^{\texttt{int}}, w_p) = \texttt{pcons}(1^{\texttt{int}}, w_p)$.

Expanding this estimation to the entire expression `insert`, we define

$$\texttt{insert}_{\texttt{cmpxy}} := \lambda_* x.\lambda_* xs.\texttt{dally}\,(\mathbf{1} + xs_c,$$
$$\texttt{pfold}\ xs_p\ \texttt{of}\ ((\mathbf{3}, 1^{\texttt{int}} :: pE), [p, ps, w](\mathbf{9} + w_c, \texttt{pcons}(1^{\texttt{int}}, w_p))))\Big)$$

We first make the following observation about `insert`'s evaluation:

Lemma 10. *If* $xs\,\xi\downarrow(n_0, \ldots, n_{k-1})\,[\,]$ *and*

$$(\texttt{fold}\ xs\ \texttt{of}\ (x :: [\,], [y, ys, w]\texttt{if}\ x \leq y\ \texttt{then}\ x :: y :: ys\ \texttt{else}\ y :: w))\ \xi\downarrow(m_0, \ldots, m_{l-1})\,[\,]$$

then $l = k + 1$.

Proof. Let $\gamma := \texttt{if}\ x \leq y\ \texttt{then}\ x :: y :: ys\ \texttt{else}\ y :: w$. We prove this lemma by induction on k. If $k = 0$ then xs evaluates to the empty list, and so the whole expression evaluates to the value of $(x :: [\,])[\,]$, a list of length 1.

If $xs\,\xi\downarrow(n, n_0, \ldots, n_{k-1})\,[\,]$ then by the induction hypothesis the recursive expression evaluates to a list of length $k + 1$:

$$(\texttt{fold}\ xs\ \texttt{of}\ (x :: [\,], [y, ys, w]\gamma))\ \xi[xs \mapsto (n_0, \ldots, n_{k-1})]\downarrow(m, m_0, \ldots, m_{k-1})\,[\,]$$

The entire expression evaluates to either the value of $x :: y :: ys$ or $y :: w$ under the environment

$$\xi' := \xi[x \mapsto \ell[\,], xs \mapsto (n, n_0, \ldots, n_{k-1})[\,], y \mapsto n[\,], ys \mapsto (n_0, \ldots, n_{k-1})[\,], w \mapsto (m, m_0, \ldots, m_{k-1})[\,]]$$

Therefore $(x :: y :: ys)\ \xi' \downarrow (\ell, n, n_0, \ldots, n_{k-1})\,[\,]$ and $(y :: w)\ \xi' \downarrow (n, m, m_0, \ldots, m_{k-1})\,[\,]$ both evaluate to lists of length $k + 2$. □

Proposition 11. $\texttt{insert} \sqsubseteq \texttt{insert}_{\texttt{cmpxy}}$.

Proof. To prove the proposition, let

$$\gamma := \texttt{if}\ x \le y\ \texttt{then}\ x :: y :: ys\ \texttt{else}\ y :: w$$

$$\phi := \texttt{fold}\ xs\ \texttt{of}\ (x :: [\,], [y, ys, w]\gamma)$$

$$\text{and } \psi := \texttt{dally}(\mathbf{1} + xs_c, \texttt{pfold}\ xs_p\ \texttt{of}\ ((\mathbf{3}, 1^{\texttt{int}} :: pE), [p, ps, w](\mathbf{9} + w_c, 1^{\texttt{int}} :: w_p)))$$

so that ϕ is the body of `insert` and ψ is the body of $\texttt{insert}_{\texttt{cmpxy}}$. Unwinding the definition of $\sqsubseteq$, we want to prove that $(\lambda x.\lambda xs.\phi)\ [\,] \sqsubseteq_{\texttt{int}*} [\![\lambda_* x.\lambda_* xs.\psi]\!][\,]$. We can simplify the right hand side of this judgement to

$$[\![\lambda_* x.\lambda_* xs.\psi]\!][\,] = (1, \lambda v.(1, \lambda vs.[\![\psi]\!]\{x \mapsto (1, v), xs \mapsto (1, vs)\}))$$

Since the cost of evaluating $\lambda x.\lambda xs.\phi[\,]$ is 1, the cost requirement of the complexity bound is satisfied. To show the potential requirement, we must prove that if $z'\,\eta' \sqsubseteq^{\mathbf{pot}} r'$ then

$$(\lambda xs.\phi)\ \{x \mapsto z'\eta'\} \sqsubseteq (1, \lambda vs.[\![\psi]\!]\{x \mapsto (1, r'), xs \mapsto (1, vs)\})$$

Again the cost requirement clearly holds. So it is sufficient to show the following:

$$(*) \qquad \text{If } z'\,\eta' \sqsubseteq^{\mathbf{pot}}_{\texttt{int}} r' \text{ and } z\,\eta \sqsubseteq^{\mathbf{pot}}_{\texttt{int}*} r \text{ then } \phi\,\xi \sqsubseteq_{\texttt{int}*} [\![\psi]\!]\xi^*$$

provided $\xi = \{x \mapsto z'\eta', xs \mapsto z\eta\}$ and $\xi^* = \{x \mapsto (1, r'), xs \mapsto (1, r)\}$. Since $z\,\eta \sqsubseteq^{\mathbf{pot}}_{\texttt{int}*} r$, we know that z is an integer sequence $z = (n_0, \ldots, n_{k-1})$, η is the empty context and, since $r \in [\![\texttt{pint}*]\!]$, $r = \overline{q}$ for some $q \ge k$. We will proceed by induction on q.

If $q = 0$ then $z\eta = ()[\,]$. We can compute the evaluation derivation of ϕ under ξ as follows:

$$\dfrac{\dfrac{}{xs\,\phi\downarrow()\,[\,]} \quad \dfrac{\dfrac{}{x\,\phi\downarrow z'\,\eta'} \quad \dfrac{}{[\,]\,\phi\downarrow()\,[\,]}}{(x :: [\,])\ \phi\downarrow(z')\,[\,]}}{\phi\,\xi\downarrow(z')\,[\,]}$$

So $\phi\,\xi\downarrow(z')\,[\,]$ has size five. Additionally we compute the denotation of ψ under ξ^*:

$$\begin{aligned}
[\![\psi]\!]\xi^* &= [\![\texttt{dally}(\mathbf{1} + xs_c, \texttt{pfold}\ xs_p\ \texttt{of}\ ((\mathbf{3}, 1^{\texttt{int}} :: \texttt{p}E), [p, ps, w](\mathbf{9} + w_c, 1^{\texttt{int}} :: w_p)))]\!]\xi^* \\
&= \texttt{dally}(2, [\![\texttt{pfold}\ xs_p\ \texttt{of}\ ((\mathbf{3}, 1^{\texttt{int}} :: \texttt{p}E), [p, ps, w](\mathbf{9} + w_c, 1^{\texttt{int}} :: w_p))]\!]\xi^*) \\
&= \texttt{dally}(2, [\![(\mathbf{3}, 1^{\texttt{int}} :: \texttt{p}E)]\!]\xi^*) = (5, \overline{1})
\end{aligned}$$

Hence $\mathbf{cost}(\phi\xi) \le \mathbf{cost}([\![\psi]\!]\xi^*)$ and $(z')\,[\,] \sqsubseteq^{\mathbf{pot}}_{\texttt{int}*} \overline{1}$. Therefore $\phi\,\xi \sqsubseteq_{\texttt{int}*} [\![\psi]\!]\xi^*$ when $q = 0$.

Suppose $q = q' + 1$ for some $q' \in \mathbb{N}$. We know $z\eta = (n_0, \ldots, n_k)[\,]$ for some $k \leq q$. The inductive hypothesis states that if $a'\,\zeta' \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}} \omega'$ and $a\,\zeta \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}} \overline{q'}$ then

$$\phi\,\{x \mapsto (a', \zeta'), xs \mapsto (a, \zeta)\} \sqsubseteq_{\mathtt{int}*} [\![\psi]\!]\{x \mapsto (1, \omega'), xs \mapsto (1, \overline{q'})\}$$

First of all, let $\mathtt{rec} = [\![\psi]\!]\{x \mapsto (1, z), xs \mapsto \overline{q'}[\,]\}$. Since ψ has type $\mathtt{pint}*^{\square}$, it must be the case that $\mathtt{rec}_p = \overline{s_{\mathtt{rec}}}$ for some $s_{\mathtt{rec}} \in \mathbb{N}$. We can compute

$$\begin{aligned}
[\![\psi]\!]\xi^* &= \mathtt{dally}(2, [\![\mathtt{pfold}\ xs_p\ \mathtt{of}\ ((\mathbf{3}, 1^{\mathtt{int}} :: \mathtt{p}E), [p, ps, w](\mathbf{9} + w_c, 1^{\mathtt{int}} :: w_p))]\!]\xi^*) \\
&= \mathtt{dally}(4 + \mathtt{rec}_c, [\![\mathbf{9} + w_c, 1^{\mathtt{int}} :: w_p]\!]\xi^*[p \mapsto 1, ps \mapsto \overline{q'}, w \mapsto (1, \overline{s_{\mathtt{rec}}})]) \\
&= (13 + \mathtt{rec}_c, \overline{s_{\mathtt{rec}} + 1})
\end{aligned}$$

We have two cases to consider based on the value of k, the length of z. If $z\eta = ()[\,]$ then as we saw earlier in the proof, $\phi\,\xi\downarrow(z')\,[\,]$ has an evaluation derivation of size 5. Thus $\mathbf{cost}(\phi\xi) = 5 \leq 12 + \mathtt{rec}_c$ and $(z')\,[\,] \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}*} \overline{s_{\mathtt{rec}} + 1}$, meaning $\phi\,\xi \sqsubseteq_{\mathtt{int}*} [\![\psi]\!]\xi^*$. On the other hand, if $k = k' + 1$ then $k' \leq q'$ meaning that $(n_0, \ldots, n_{k'-1})\,[\,] \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}*} \overline{q'}$. So by the inductive hypothesis, $\phi\,\xi_r \sqsubseteq_{\mathtt{int}*} \mathtt{rec}$ where $\xi_r = \{x \mapsto z'\eta', xs \mapsto (n_0, \ldots, n_{k'-1})[\,]\}$. Let α be the evaluation derivation of

$$(\mathtt{fold}\ u\ \mathtt{of}\ (x :: [\,], [y, ys, w]\gamma))\ \xi[u \mapsto (n_1, \ldots, n_{k'})[\,]]\downarrow(m_0, \ldots, m_{\ell-1})\,[\,]$$

and β the derivation of

$$\gamma\,\xi[y \mapsto n_0[\,], ys \mapsto (n_1, \ldots, n_{k'})[\,], w \mapsto (m_0, \ldots, m_{\ell-1})[\,]]\downarrow v\,\theta$$

By the inductive hypothesis, $(\mathtt{fold}\ u\ \mathtt{of}\ (x :: [\,], [y, ys, w]\gamma))\ \xi[_{u \mapsto (n_1, \ldots, n_{k'})[\,]}] \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}*} \mathtt{rec}$ meaning that $\mathbf{cost}(\alpha) \leq \mathtt{rec}_c$ and $(m_0, \ldots, m_{\ell-1})\,[\,] \sqsubseteq^{\mathbf{pot}}_{\mathtt{int}*} \mathtt{rec}_p$. Meanwhile, the structure of β depends on the value to which $x \leq y$ evaluates under $\xi' = \xi[y \mapsto n_0[\,], ys \mapsto (n_1, \ldots, n_{k'})[\,], w \mapsto (m_0, \ldots, m_{\ell-1})[\,]]$. If $z' \leq n_0$ then β is

$$\dfrac{\dfrac{\overline{x\,\xi'\downarrow z'\,\eta'} \quad \overline{y\,\xi'\downarrow n_0\,[\,]} \quad z' \leq n_0}{(x \leq y)\ \xi'\downarrow \mathtt{tt}\,[\,]} \qquad \dfrac{\overline{x\,\xi'\downarrow z'\,\eta'} \quad \dfrac{\overline{y\,\xi'\downarrow n_0\,[\,]} \quad \overline{y\,s\downarrow\xi'\,(n_1, \ldots, n_{k'-1})[\,]}}{(y :: ys)\ \xi'\downarrow(n_0, \ldots, n_{k'-1})\,[\,]}}{(x :: y :: ys)\ \xi'\downarrow(n_0, \ldots, n_{k'-1})\,[\,]}}{(\mathtt{if}\ x \leq y\ \mathtt{then}\ x :: y :: ys\ \mathtt{else}\ y :: w)\ \xi'\downarrow(n_0, \ldots, n_{k'-1})\,[\,]}$$

If $z' > n_0$ then β is

$$\dfrac{\dfrac{\overline{x\,\xi' \downarrow z'\,\eta'} \quad \overline{y\,\xi' \downarrow n_0\,[\,]} \quad z' > n_0}{(x \le y)\ \xi' \downarrow \texttt{tt}\,[\,]} \quad \dfrac{\overline{y\,\xi' \downarrow n_0\,[\,]} \quad \overline{w\,\xi' \downarrow (m_0, \ldots, m_{\ell-1})\,[\,]}}{(y :: w)\ \xi' \downarrow (n_0, m_0, \ldots, m_{\ell-1})\,[\,]}}{(\texttt{if}\ x \le y\ \texttt{then}\ x :: y :: ys\ \texttt{else}\ y :: w)\ \xi' \downarrow (n_0, m_0, \ldots, m_{\ell-1})\,[\,]}$$

Therefore $\mathbf{cost}(\beta) \le 9$. By Lemma 10 we know that $\ell + 1 = k'$. So in both cases γ evaluates to a list of length k'.

Then the evaluation derivation for $\phi\xi$ is as follows:

$$\dfrac{\overline{xs\,\xi \downarrow (n_0, \ldots, n_{k'})\,[\,]} \quad \alpha \quad \beta}{\phi\,\xi \downarrow (l_0, \ldots, l_{k'-1})\,[\,]}$$

Thus we have

$$\begin{aligned}\mathbf{cost}(\phi\xi) &= 2 + \mathbf{cost}(\alpha) + \mathbf{cost}(\beta)\\ &\le 2 + \texttt{rec}_c + 9 = 11 + \texttt{rec}_c\\ &\le 12 + \texttt{rec}_c = \mathbf{cost}([\![\psi]\!]\xi^*)\end{aligned}$$

and also $(l_0, \ldots, l_{k'-1})\,[\,] \sqsubseteq^{\mathbf{pot}}_{\texttt{int}*} \texttt{rec}_p$. So we conclude that $\phi\,\xi \sqsubseteq_{[\![\psi]\!]\xi^*}$ in this case as well, completing the proof of $(*)$. □

3.5. Extensions to the Translation System

To increase the generality and usability of the target language, we would like to expand the language to allow for arbitrary user-defined datatypes. To do so we will need to automatically incorporate the datatypes into the translation scheme and complexity language. In this section we will step through the process for a simple datatype, the type of binary trees. We will then discuss some of the difficulties associated with generalizing this process to arbitrary datatypes.

Let `bintree` be the type of binary trees with integer-labeled nodes.

$$\texttt{bintree} := \texttt{Leaf} \mid \texttt{Node}\ \textbf{of}\ (\texttt{int}, \texttt{bintree}, \texttt{bintree}).$$

We add pattern matching and structural recursion for trees in the form of the `btcase` and `fold` operators. The typing and evaluation rules for these extensions to the target language are given in Figure 3.9.

$$\frac{}{\Gamma \vdash \texttt{Leaf} : \texttt{bintree}} \qquad \frac{\Gamma \vdash n : \texttt{int} \quad \Gamma \vdash l : \texttt{bintree} \quad \Gamma \vdash r : \texttt{bintree}}{\Gamma \vdash \texttt{Node}(n, l, r) : \texttt{bintree}}$$

$$\frac{\Gamma \vdash r : \texttt{bintree} \quad \Gamma \vdash s : \tau \quad \Gamma, x : \texttt{int}, y_l : \texttt{bintree}, y_r : \texttt{bintree} \vdash t : \tau}{\Gamma \vdash \texttt{btcase}\ r\ \texttt{of}\ (s, [x, y_l, y_r]t) : \tau}$$

$$\frac{\Gamma \vdash r : \texttt{bintree} \quad \Gamma \vdash s : \tau \quad \Gamma, x : \texttt{int}, y_l, y_r : \texttt{bintree}, w_l, w_r : \tau \vdash t : \tau}{\Gamma \vdash \texttt{btfold}\ r\ \texttt{of}\ (s, [x, y_l, y_r, w_l, w_r]t) : \tau}$$

$$\frac{}{\texttt{Leaf}\,\xi \downarrow \text{leaf}\,[\,]} \qquad \frac{n\,\xi \downarrow m\,[\,] \quad l\,\xi \downarrow u\,[\,] \quad r\,\xi \downarrow v\,[\,]}{(\texttt{Node}(n, l, r))\ \xi \downarrow \text{node}(m, u, v)\,[\,]}$$

$$\frac{r\,\xi \downarrow \text{leaf}\,[\,] \quad s\,\xi \downarrow v\,\theta}{(\texttt{btcase}\ r\ \texttt{of}\ (s, [x, y_l, y_r]t))\ \xi \downarrow v\,\theta} \qquad \frac{r\,\xi \downarrow \text{node}(m, u, v)\,[\,] \quad t\,\xi[x \mapsto m[\,], y_l \mapsto u[\,], y_r \mapsto v[\,]] \downarrow v\,\theta}{(\texttt{btcase}\ r\ \texttt{of}\ (s, [x, y_l, y_r]t))\ \xi \downarrow v\,\theta}$$

$$\frac{r\,\xi \downarrow \text{leaf}\,[\,] \quad s\,\xi \downarrow v\,\theta}{(\texttt{btfold}\ r\ \texttt{of}\ (s, [x, y_l, y_r, w_l, w_r]t))\ \xi \downarrow v\,\theta}$$

$$\frac{r\,\xi \downarrow \text{node}(m, u_l, u_r)\,[\,] \quad \mathcal{D}_l \quad \mathcal{D}_r \quad t\,\xi\left[x \mapsto m[\,], \begin{smallmatrix} y_l \mapsto u_l[\,], y_r \mapsto u_r[\,] \\ w_l \mapsto v_l\theta_l, w_r \mapsto v_r\theta_r \end{smallmatrix}\right] \downarrow v\,\theta}{(\texttt{btfold}\ r\ \texttt{of}\ (s, [x, y_l, y_r, w_l, w_r]t))\ \xi \downarrow v\,\theta}$$

where $\mathcal{D}_l := (\texttt{fold}\ y\ \texttt{of}\ (s, [x, y_l, y_r]w_l)w_r t)\ \xi[y \mapsto u_l[\,]] \downarrow v_l\,\theta_l$

and $\mathcal{D}_r := (\texttt{fold}\ y\ \texttt{of}\ (s, [x, y_l, y_r]w_l)w_r t)\ \xi[y \mapsto u_r[\,]] \downarrow v_r\,\theta_r$

FIGURE 3.9. Target typing and evaluation rules for `bintree`-related expressions.

In the complexity syntax we insert replicas of the constructors and of the case and fold operators. This mirrors the design of `pint`$*$, `plistcase` and `pfold`.

$$\texttt{pbintree} := \texttt{pLeaf} \mid \texttt{pNode}\ \textbf{of}\ (\texttt{pint}, \texttt{pbintree}, \texttt{pbintree})$$

Elements of type `pbintree` are potentials however, and the translation reflects this restriction.

$$\|\texttt{Leaf}\| = (\mathbf{1}, \texttt{pLeaf})$$

$$\|\texttt{Node}(m, l, r)\| = (\mathbf{1} + \|m\|_c + \|l\|_c + \|r\|_c, \texttt{pNode}(\|m\|_p, \|l\|_p, \|r\|_p))$$

$$\|\texttt{btcase}\ r\ \texttt{of}\ (s, [x, y_l, y_r]t)\| =$$

$$\texttt{dally}\left(1 + \|r\|_c, \texttt{pbtcase}\ \|r\|_p\ \texttt{of}\ (\|s\|, [p, p_l, p_r]\|t\|[x \mapsto (1,p), y_l \mapsto (1,p_l), y_r \mapsto (1,p_r)])\right)$$

$$\|\texttt{btfold}\ r\ \texttt{of}\ (s, [x, y_l, y_r, w_l, w_r]t)\| =$$

$$\texttt{dally}\left(1 + \|r\|_c, \texttt{pbtfold}\ \|r\|_p\ \texttt{of}\ (\|s\|, [p, p_l, p_r, w_l, w_r]\|t\|[x \mapsto (1,p), y_l \mapsto (1,p_l), y_r \mapsto (1,p_r)])\right)$$

$$[\![\Gamma \vdash \texttt{pLeaf} : \texttt{pbintree}]\!] = \text{pleaf}$$

$$[\![\Gamma \vdash \texttt{pNode}(m, l, r) : \texttt{pbintree}]\!] = \text{pnode}([\![m]\!], [\![l]\!], [\![r]\!])$$

$$[\![\texttt{pbtcase}\ r\ \texttt{of}\ (s, [p, p_l, p_r]t)]\!]\xi = \begin{cases} [\![s]\!]\xi & \text{if } [\![r]\!]\xi = \text{pleaf} \\ [\![s]\!]\xi \vee [\![t]\!]\xi[p \mapsto m, p_l \mapsto u, p_r \mapsto v] & \text{if } [\![r]\!]\xi = \text{pnode}(m, u, v) \end{cases}$$

$$[\![\texttt{pbtfold}\ r\ \texttt{of}\ (s, [p, p_l, p_r, w_l, w_r]t)]\!]\xi = \begin{cases} [\![s]\!]\xi & \text{if } [\![r]\!]\xi = \overline{0} \\ \Big(3 + \texttt{rec}_c^l + \texttt{rec}_c^r + ([\![t]\!]\xi')_c, & \\ \quad ([\![s]\!]\xi)_p \vee ([\![t]\!]\xi')_p\Big) & \text{if } [\![r]\!]\xi = \text{pnode}(q, u, v) \end{cases}$$

$$\text{where } \texttt{rec}^l := [\![\texttt{pbtfold}\ y\ \texttt{of}\ (s, [p, p_l, p_r, w_l, w_r]t)]\!]\xi[y \mapsto u]$$

$$\text{and } \texttt{rec}^r := [\![\texttt{pbtfold}\ y\ \texttt{of}\ (s, [p, p_l, p_r, w_l, w_r]t)]\!]\xi[y \mapsto v]$$

$$\text{and } \xi' := \xi \begin{bmatrix} p \mapsto q, p_l \mapsto u, p_l \mapsto v \\ w_l \mapsto (1, \texttt{rec}_p^l), w_r \mapsto (1, \texttt{rec}_p^r) \end{bmatrix}$$

FIGURE 3.10. Denotational semantics for `pbintree`-related expressions.

To the semantic values of the complexity language, we add a fourth copy of `bintree`'s constructors. This reflects the structure of $\overline{n}$, the list of 1's, which is the potential of a list.

$$[\![\texttt{pbintree}]\!] = \text{pleaf} \mid \text{pnode}\ \textbf{of}\ ([\![\texttt{pint}]\!], [\![\texttt{pbintree}]\!], [\![\texttt{pbintree}]\!])$$

The rules for the maximum operator on the semantics of the complexity language are as follows:

$$\text{pleaf} \vee \text{pleaf} = \text{pleaf} \qquad \text{pleaf} \vee \text{pnode}(n, u, v) = \text{pnode}(n, u, v)$$

$$\text{pnode}(n,u,v) \vee \text{pleaf} = \text{pnode}(n,u,v) \qquad \text{pnode}(n_1,u_1,v_1) \vee \text{pnode}(n_2,u_2,v_2) = \text{pnode}(n_1 \vee n_2, u_1 \vee u_2, v_1 \vee v_2)$$

The rest of the denotational semantics are given in Figure 3.10 Finally, we must add the following rules to the potential bounding relation:

$$\text{leaf}\ [\,] \sqsubseteq^{\textbf{pot}}_{\texttt{bintree}} \text{pleaf} \qquad \text{leaf}\ [\,] \sqsubseteq^{\textbf{pot}}_{\texttt{bintree}} \text{pnode}(q, q_l, q_r)$$

$$\text{node}(n, u, v)\ [\,] \sqsubseteq^{\textbf{pot}}_{\texttt{bintree}} \text{pnode}(q, q_l, q_r) \quad \text{if } n\ [\,] \sqsubseteq^{\textbf{pot}} q \text{ and } u\ [\,] \sqsubseteq^{\textbf{pot}} q_l \text{ and } v\ [\,] \sqsubseteq^{\textbf{pot}} q_r$$

The addition of `bintree`'s worked out very smoothly, but we encounter a number of problems when we move to more complicated datatypes. Consider the types of binary trees with both leaves and nodes labeled by integers; call this type `bintree'`. The process we used to

incorporate `bintree`'s into the translation system works perfectly well up to the definition of the denotational semantics of `pbtcase′` and `pbtfold′` expressions. We need only consider `pbtcase′` to understand where the problem lies. Consider $[\![\texttt{pbtcase}'\ r\ \texttt{of}\ ([p]s, [p, p_l, p_r]t)]\!]\xi$. If $[\![r]\!]\xi = \texttt{pleaf}(n)$ for some n then the denotation of the entire expression is $[\![s]\!]\xi[p \mapsto n]$. Meanwhile, if $[\![r]\!]\xi = \texttt{pnode}(n, u, v)$ then we need the denotation of the `pbtcase′` expression to be the maximum of the two branches. The right-hand-side is certainly $[\![t]\!]\xi[_{p\mapsto n, p_l\mapsto u, p_r\mapsto v}]$. But what of the left-hand-side? We need to evaluate $[\![s]\!]$ under the environment ξ updated with p mapped to some element of $[\![\texttt{pint}]\!]$.

In this example, we know that $[\![\texttt{pint}]\!] = \{1\}$, and so we use for the left-hand branch $[\![s]\!]\xi[p \mapsto 1]$. But it is easy to see that in any other situation, we cannot be assured of knowing how to fill in the variables in s bounded by the `pbtcase′` or `pbtfold′` expressions. Hence if the base case of a datatype takes any input of type other than `int` or `bool`, we cannot be assured of being able to evaluate both branches in the denotational semantics of the case and fold expressions.

Another telling example is the datatype for binary words:

$$\texttt{binword} := \varepsilon \mid 0\ \textbf{of}\ \texttt{binword} \mid 1\ \textbf{of}\ \texttt{binword}$$

The adaption of the translation system is straightforward all the way through the denotational semantics this time. If $[\![e]\!]\xi = 0w$ for example, then

$$[\![\texttt{pbwcase}\ e\ \texttt{of}\ (r, [p_0]s, [p_1]t)]\!]\xi = [\![r]\!]\xi \vee [\![s]\!]\xi[p_0 \mapsto w]$$

But the difficulty is when we attempt to define the maximum operator on the denotational semantics. We can easily compare ε to $0w$ or $1w$, but how do we define $0u \vee 1w$? The only reasonable option is to collapse the constructors 0 and 1 and consider only the length of the word when defining the max operator. But in this case we are left with the following result: $1\varepsilon \sqsubseteq 00\varepsilon$.

Consider the proof of soundness for the case expression $\texttt{bwcase}\ e\ \texttt{of}\ (r, [x_0]s, [x_1]t)$. It is possible that, for $\xi \sqsubseteq \xi^*$, we have $e\,\xi \downarrow 1\varepsilon[\ ]$ and $[\![\|e\|_p]\!]\xi^* = 01\varepsilon$. For the potential bound in

particular, we need to show that if $t\,\xi \downarrow v\,\theta$ then

$$v\,\theta \sqsubseteq^{\mathbf{pot}} \mathbf{pot}[\![||r||]\!]\xi^* \vee \mathbf{pot}[\![||s||[x_0 \mapsto (1, p_0)]\!]\xi^*[p_0 \mapsto 1\varepsilon]$$

But the inductive hypothesis gives us only that $v\,\theta \sqsubseteq^{\mathbf{pot}} \mathbf{pot}[\![||s||[x_1 \mapsto (1, p_1)]]\!]\xi^*[p_1 \mapsto u]$ for some u, which is completely unhelpful. To incorporate binary words and more complicated datatypes into the language one must take a different approach.

CHAPTER 4

Translation Implementation in Coq

One of the goals of this work is to produce a system which constructs certified upper bounds on the complexity of terms. We achieve this certification by implementing the translation scheme of the previous chapter in Coq and then proving the relevant theorems about the system.

To implement the target language in Section 4.1, we first construct the syntax and typing relations, and then move on to target language evaluation. For the complexity language in Section 4.2, we define its syntax and attribute to it a big-step operational semantics. In Section 4.3 we describe the translation function and bounding relation, and discuss approaches to proving the soundness theorem.

4.1. The Target Language and an Introduction to Coq

4.1.1. Target Syntax and Typing. To begin, we define the set of variables, which are indexed by the natural numbers.

```
Definition var := nat.
```

We build new datatypes in Coq by means of the `Inductive` declaration. This construct requires the user to provide names for each constructor along with the type of that constructor. Target language expressions are given in Figure 4.1. Since **exp** is a datatype, it is an element of `Type`. The constructors of **exp** are functions—for example, cons is a function which takes two arguments, both of type **exp**, and evaluates to an element of **exp**. In the definition, **rel** is an inductively defined datatype with three 0-ary constructors Lt, Le and Eq, and **op** is one with constructors Plus, Minus, Times and Div. **Z** are the integers in Coq, defined in the library ZArith.

```
Inductive exp : Type :=
  | Var : var → exp
  | Num : Z → exp
  | TT : exp
  | FF : exp
  | Nil : exp
  | Cons : exp → exp → exp
  | If : exp → exp → exp → exp
  | Rel : rel → exp → exp → exp
  | Op : op → exp → exp → exp
  | Lambda : var → exp → exp
  | Apply : exp → exp → exp
  | Listcase : exp → exp → var → var → exp → exp
  | Fold : exp → exp → var → var → var → exp → exp.
```

FIGURE 4.1. The target language expressions, defined in Coq.

Coq allows us to define patterns to act as notations for certain functions, and we take advantage of this feature for target language expressions. We define #*n* to be Num *n*, r::s to be Cons r s, Case *r* of (*s*,[*x*,*xs*]*t*) to be Listcase *r s x xs t*, and similarly Fold *r* of (*s*,[*x*,*xs*,*w*]*t*) to be Fold *r s x xs w t*.

Next we define the target language types, also by means of the `Inductive` declaration.

```
Inductive type : Type :=
  | Bool : type | Int : type | Ints : type | Arrow : type → type → type.
```

Before we can define the typing derivations, we must be able to represent type contexts. Recall that a type context is a partial function from variables to types. We define an environment to be a polymorphic partial function

`Definition` environment (*A*:`Type`) := var → **option** *A*.

Here **option** is an inductively defined datatype with two constructors: a zero-ary constructor None and a unary Some, which takes an element of type *A*. So if *G* has type environment *A* then *G x* = Some *a* if *G* is defined on *x*, and otherwise *G x* = None. We define get *G x* to be the result of applying *G* to *x*, the empty environment empty to be the map which sends every variable to None, and update and delete functions which modify environments as expected. The predicate equiv_empty holds on *G* if it is the empty environment; we cannot test point-wise equality directly. For the functions get, update and delete the value *A* can be inferred from the context, so for *G*:environment *A*, we can invoke delete *G x* for example, instead of delete *A G x*. For empty, *A* cannot be inferred, so for a particular *A0* we will define A0_empty:=empty *A0*.

Now we define a type context to be

`Definition` type_context := environment **type**.

Next we construct the target typing derivation. These derivation rules prove judgements of the form $\Gamma \vdash e : \tau$ for a type context Γ, an expression e and a type τ. In Coq, a judgement of this form has the type `Prop`, which stands for Proposition. Inhabitants of `Prop` are statements corresponding to classical logic, which can be combined using $\rightarrow$, $\forall$ and $\exists$. If *A* is a proposition of type `Prop` and *a* is an element of type *A*, then we say *a* is a *proof* of *A*. In Coq, proofs are constructed in two main ways. The first is interactively by means of the theorem-proving components of Coq, by applying a series of tactics to reduce goals to simpler goals. The second is by applying constructors of the `Inductive` construct. Just as we use `Inductive` to define datatypes of sort `Type`, we also use it to define predicates of sort `Prop`. The typing derivation is given in Figure 4.2. Each constructor provides a means of generating a proof of **typed** *G e t*.

The constructors of **typed** again are functions. For example, if pf_r is a proof of the proposition **typed** *G r* Int and pf_s is a proof of **typed** *G s* Ints, then tCons *G r s* pf_r pf_s is a proof of **typed** *G* (*r*::*s*) Ints.

4.1.2. The Proof of a Typing Judgement. To get an idea of how proof tactics work in Coq, let us prove a simple typing judgement. We will prove that $\{x : \texttt{int}\} \vdash 3 \leq x : \texttt{bool}$. First we fix *x* and define the type context G.

```
Inductive typed : type_context → exp → type → Prop :=
  | tVar : ∀(G:type_env)(v:var)(t:type), get G v = Some t → typed G (Var v) t
  | tNum : ∀(G:type_env)(n:Z), typed G (Num n) Int
  | tTT : ∀(G:type_env), typed G TT Bool
  | tFF : ∀(G:type_env), typed G FF Bool
  | tNil : ∀(G:type_env), typed G Nil Ints
  | tCons: ∀(G:type_env)(e1 e2:exp), typed G e1 Int → typed G e2 Ints →
      typed G (e1::e2) Ints
  | tIf : ∀(G:type_env)(e1 e2 e3:exp)(t:type),
      typed G e1 Bool → typed G e2 t → typed G e3 t →
      typed G (If e1 then e2 else e3) t
  | tRel : ∀(G:type_env)(e1 e2:exp)(r:rel), typed G e1 Int → typed G e2 Int →
      typed G (Rel r e1 e2) Bool
  | tOp : ∀(G:type_env)(e1 e2:exp)(o:op), typed G e1 Int → typed G e2 Int →
      typed G (Op o e1 e2) Int
  | tLambda : ∀(G:type_env)(v:var)(e:exp)(s t:type), typed (update G v s) e t →
      typed G (Lambda v e) (Arrow s t)
  | tApply : ∀(G:type_env)(e1 e2:exp)(s t:type), typed G e1 (Arrow s t) → typed G e2 s →
      typed G (Apply e1 e2) t
  | tListcase : ∀(G:type_env)(e1 e2 e3:exp)(x xs:var)(t:type),
      typed G e1 Ints → typed G e2 t →
      let G_x_xs := update (update G x Int) xs Ints in typed G_x_xs e3 t →
      typed G (Case e1 of (e2,[x,xs]e3)) t
  | tFold : ∀(G:type_env)(e1 e2 e3:exp)(x xs w:var)(t:type),
      typed G e1 Ints → typed G e2 t →
      let G_x_xs_w := update (update (update G x Int) xs Ints) w t in
      typed G_x_xs_w e3 t →
      typed G (Fold e1 of (e2,[x,xs,w]e3)) t.
```

FIGURE 4.2. An implementation of the target language typing derivation.

```
Definition x : var := 1.
Definition G : type_context := update type_empty x Int.
```

As mentioned above, type_empty is the empty type context empty type which maps every variable to None. We want to prove the following goal:

```
Goal typed G (Rel Lt (Num 3) (Var x)) Bool.
```

We begin the proof and start applying tactics. The strategy is to reduce the goal into smaller, more manageable goals until each can be solved by a single tactic. The first thing we notice is that our current goal is exactly the conclusion of the constructor tRel. This constructor gives us a means to prove the goal, if we had certain other proofs. That is, if pf_3 is a proof of $G \vdash 3 : \texttt{int}$ and pf_x is a proof of $G \vdash x : \texttt{int}$ then tRel G (Num 3) (Var x) Lt pf_3 pf_x is a proof of our current goal. But we do not have pf_3 and pf_x, unless we construct them first. Instead of working forward by constructing these two proofs and then applying the constructor, we can work backwards. We use the tactic `apply` tRel to apply the constructor to the current goal. Coq attempts to infer the remaining arguments to tRel, and it succeeds for Num 3, Var x and Lt. What it cannot infer are the proofs pf_3 and pf_x. However, since both of these proofs are of propositions, it is possible to construct them through more tactics. So the tactic `apply` tRel replaces the current goal with two brand new subgoals: **typed** G (Num 3) Int and **typed** G (Var x) Int.

The first subgoal is solved easily via `apply` tNum. Alternatively, the tactic `constructor` automatically searches the inductive predicate **typed** for a constructor whose conclusion matches the current goal, and applies that constructor.

The second subgoal is a little more tricky. We use `constructor` again to invoke the tVar constructor, but this time Coq generates a subgoal of get G x = Some Int. Our first instinct is to unfold definitions. We can do so with the `unfold` tactic:

```
unfold G, get, update, type_empty.
```

The resulting goal is an equality whose value depends on the function eq_nat_dec:

```
if eq_nat_dec x x then Some Int else None = Some Int
```

In general, equality is not decidable in Coq, and eq_nat_dec implements decidable equality for natural numbers. Clearly this goal must hold, because eq_nat_dec is reflexive. Coq can automatically infer reflexivity of eq_nat_dec for a particular number, but not for more general

patterns, like x, which is a folded definition. So to proceed we unfold the definition of x and simplify the expression by means of the `simpl` tactic:

`unfold` *x*. `simpl`.

This leaves us with a goal of Some Int = Some Int, which is solved easily by the `reflexivity` tactic.

In all, the following sequence of tactics proves our goal:

```
Goal typed G (Rel Var (3) (Var x)) Bool
Proof.
  apply tRel.
    (*LHS*) constructor.
    (*RHS*) constructor. unfold G, get, update, type_empty, x.
      simpl. reflexivity.
Qed.
```

4.1.3. Target Semantics. We define target values inductively as expected, with constructors num, tt and ff, nil and cons, and lambda. Next we need to define value environments, which are environments on value closures. Since value closures are defined recursively with value environments, the definition is not as straightforward as it was for type contexts. Instead we must define value environments recursively via the `Inductive` declaration:

`Inductive` **val_env** := vEnv : environment (**val** × **val_env**) → **val_env**.

With this construction, value environments no longer have type environment *A* for some *A*. This means that any theorems proved or definitions made about environments in general, like update and delete, do not transfer automatically to value environments. Coq has a coercion mechanism which converts one datatype to another under certain circumstances. In this case we can coerce elements of type **val_env** into those of type environment (**val** × **val_env**).

```
Coercion val_to_env (xi:val_env) :=
  match xi with vEnv xi ⇒ xi end : environment (val × val_env).
```

We then define a value closure to be the product of **val** and **val_env**, and a closure to be the product of **exp** and **val_env**.

```
Inductive eval : closure → val_closure → Prop :=
  | eVar : ∀ (x:var)(xi:val_env)(C:val_closure), Some(C) = get xi x → eval (Var x, xi) C
  | eTrue : ∀ (xi:val_env), eval (TT,xi) (tt,val_empty) | eFalse : ··· | eNum : ··· | eEmpty : ···
  | eCons : ∀ (xi:val_env) (r s:exp) (v1 v2:val),
      eval (r,xi) (v1,val_empty) → eval (s,xi) (v2,val_empty) → eval (r::s,xi) (cons v1 v2,val_empty)
  | eIf_t : ∀ (xi xi' : val_env)(r s t:exp)(v:val),
      eval (r,xi) (tt,val_empty) → eval (s,xi) (v,xi') → eval (If r then s else t,xi) (v,xi')
  | eIf_f : ··· | eRel : ···
  | eOp : ∀ (xi:val_env) (r s:exp) (o:op) (m n:Z) (v:val), eval (r,xi) (num m,val_empty) →
      eval (s,xi) (num n,val_empty) → op_eval o m n = v → eval (Op o r s,xi) (v,val_empty)
  | eLambda : ∀ (xi:val_env) (s:exp) (x:var), eval (Lambda x s,xi) (lambda x s,xi)
  | eApply : ∀ (xi theta theta_r theta_s:val_env)(r s t:exp)(x:var)(v v':val),
      eval (r,xi) (lambda x t,theta_r) → eval (s,xi) (v',theta_s) →
      eval (t,vEnv (update theta_r x (v,theta_s))) (v,theta) → eval (Apply r s,xi) (v,theta)
  | eListcase_n : ∀ (xi theta:val_env) (r s t:exp)(x xs:var)(v:val),
      eval (r,xi) (nil,val_empty) → eval (s,xi) (v,theta) → eval (Case r of (s,[x,xs]t),xi) (v,theta)
  | eListcase_c : ∀ (xi theta:val_env)(r s t:exp)(x xs:var)(u v vs:val),
      eval (r,xi) (cons v vs,val_empty)→
      let xi' := env (update (update xi x (v,val_empty)) xs (vs,val_empty)) in
      eval (t,xi') (u,theta) → eval (Case r of (s,[x,xs]t),xi) (u,theta)
  | eFold_n : ∀ (xi theta:val_env)(r s t:exp)(x xs w:var)(v:val),
      eval (r,xi) (nil,val_empty) → eval (s,xi) (v,theta) → eval (Fold r of (s,[x,xs,w]t),xi) (v,theta)
  | eFold_c : ∀ (xi xi_rec theta:val_env)(r s t:exp)(x xs w y:var)(u v vs v_rec:val),
      eval (r,xi) (cons v vs,val_empty) → let xi' := env(update xi y (vs,val_empty))
      in eval (Fold (Var y) of (s,[x,xs,w]t),xi') (v_rec,theta_rec) →
      let xi'' := env (update (update (update xi x (v,val_empty)) xs (vs,val_empty)) w (v_rec,xi_rec))
      in eval (t,xi'') (u,theta) → eval (Fold r of (s,[x,xs,w]t),xi) (u,theta).
```

FIGURE 4.3. The operational semantics of the target language as an inductive definition.

Next we move onto the evaluation derivation. It is straightforward to define evaluation as an inductive predicate, as we did for typing derivations. The predicate **eval** is given in Figure 4.3.

For the bounding relation we need to be able to compute the size of an evaluation derivation. Our first attempt at this computation is a recursive function:

`Fixpoint` deriv_cost (C : closure) (VC : val_closure) (pf : **eval** C VC) : **nat** := $\cdots$

The structure of this definition would be recursion on pf, the proof of **eval** C VC. But Coq rejects this definition, with the following error:

```
Elimination of an inductive object of sort Prop is not
allowed on a predicate in sort Set because proofs can be
eliminated only to build proofs.
```

The `Fixpoint` constructor defines a function into the type **nat**, so it is a predicate in sort `Set`. On the other hand, pf is an element of the predicate **eval** C VC of type `Prop`. The system does not allow us to destruct a proof, whose truth is undecidable, to construct a decidable function.

To get around this error, we would like to represent an evaluation derivation not as an object of type `Prop`, but as a concrete object of type `Type`, that is, a datatype. Luckily, in Coq it is remarkably easy to do so. We define an inductive datatype

`Inductive` **deriv** : closure $\rightarrow$ val_closure $\rightarrow$ `Type` := $\cdots$

with exactly the same constructors as **eval**. That is, for each constructor eC of type T, we attribute to **deriv** a constructor dC of type T.

Since **deriv** defines objects of sort `Type`, we can define deriv_cost in the natural way, recursing on the structure of the derivation.

With this setup, we can use **deriv** in many of the same ways we would use a proposition. For example, we can prove goals of the form **deriv** C VC, which is proved by constructing an element of that type. However we cannot use connectors on propositions, like $\exists$ and $\vee$ with **deriv**. Instead we retain both definitions, **eval** and **deriv**, and use them each when appropriate. We can easily prove that, given a derivation $\mathcal{D}$ of **deriv** C VC, the predicate **eval** C VC holds. However, we cannot prove the other direction, that **eval** C VC implies **deriv** C VC, because once again the proof of such a lemma would involve destructing an object of sort `Prop` to construct an object of sort `Type`.

```
Inductive VC_equiv : val_closure → val_closure → Prop :=
  | VCE : ∀ (v v':val)(theta theta':val_env), v=v' → vEnv_equiv theta theta' →
    VC_equiv (v,theta) (v',theta')
with vEnv_equiv : val_env → val_env → Prop :=
  | VEE : ∀ (xi xi':val_env), dom_eq xi xi' →
    (∀ (x:var)(VC VC':val_closure), get xi x = Some VC → get xi' x = Some VC' →
     VC_equiv VC VC') → vEnv_equiv xi xi'.
```

FIGURE 4.4. An equivalence relation on value closures and value environments.

Finally, we would like to prove the uniqueness of the evaluation derivation in Coq.

```
Theorem deriv_unique : ∀ (C:closure)(VC1 VC2:val_closure)
                       (D1:deriv C VC1)(D2:deriv C VC2), D1 = D2.
```

Since *D1* and *D2* have different types however, the expression *D1=D2* is ill-typed, so the theorem is rejected by Coq. Instead we define our own notion of equivalence for a derivation in the form of an inductively-defined predicate. We define deriv_equiv to hold on *D1*:**deriv** *C1 VC1* and *D2*:**deriv** *C2 VC2* if $C1 = C2$, $VC1 = VC2$ and $D1$ and $D2$ are formed from the same constructors.

This definition makes us consider carefully what we mean by equality of closures and value closures. Value closures consist of values, which we can compare explicitly, and value environments, which we cannot, because they are functions. When we say that two environments are equal, we mean that they are equal on every input. We define an equivalence relation on value closures and value environments by mutual recursion in Figure 4.4. The predicate dom_eq holds on *xi* and *xi'* if the domains of *xi* and *xi'* are exactly equal.

With these definitions of equality for value closures, value environments and derivations, we can try again to prove the uniqueness of the evaluation derivation.

```
Theorem deriv_unique : ∀ (C:closure)(VC1 VC2:val_closure)
                       (D1:deriv C VC1)(D2:deriv C VC2), deriv_equiv D1 D2.
```

We want to prove this theorem by induction on *D1*. Coq has a built-in `induction` tactic we can utilize. We introduce the variables *C VC1* and *D1* into the context and then use the tactic

`dependent induction`.[1] But when we try to perform this induction, the system complains that we cannot abstract over *D1* because the resulting goals would be ill-typed. In particular, induction on *D1* coerces *C* into its constructors, which affects the type of *D2*, **deriv** *C VC2*.

There are two ways to approach this problem. The first, more careful approach, would be to restate the lemma in the form "If *C1* and *C2* are equivalent closures then *D1*:**deriv** *C1 VC1* is equivalent to *D2*:**deriv** *D2 VC2*." Instead, we recognized that there are two important implications we obtain from the uniqueness result. These are (1) that whenever $t\,\xi \downarrow v\,\theta$ and $t\,\xi \downarrow v'\,\theta'$ we have $v\,\theta = v'\,\theta'$ and (2) that for derivations $\mathcal{D}_1$ of $t\,\xi \downarrow v\,\theta$ and $\mathcal{D}_2$ of $t\,\xi \downarrow v'\,\theta'$, $\mathbf{cost}(\mathcal{D}_1) = \mathbf{cost}(\mathcal{D}_2)$. The first result we can prove in terms of the predicate **eval**.

Lemma eval_equiv : ∀ (*C*:closure) (*VC1 VC2*:val_closure),

eval *C VC1* → **eval** *C VC2* → **VC_equiv** *VC1 VC2*.

The second, because it refers to the size of an evaluation derivation, must be in terms of **deriv**.

Lemma cost_equiv := ∀ (*C*:closure) (*VC1 VC2*:val_closure)

(*D1*:**deriv** *C VC1*) (*D2*:**deriv** *C VC2*), deriv_cost *D1* = deriv_cost *D2*.

From this lemma, we can define a predicate **deriv_bound** on a closure *C*, value closure *VC* and natural number *n*, which holds if the size of the derivation of **deriv** *C VC* is bounded by *n*.

For our purposes, these results together are sufficient to replace the theorem regarding uniqueness of the evaluation derivation.

4.2. The Complexity Language

4.2.1. Syntax. Complexity language expressions are defined straightforwardly with respect to the description in Section 3.3. Their syntax is given in Figure 4.5. One difference is that we implement both a **cLambda** as well as a **cLambda_s** constructor, with the first one representing regular λ-abstraction, and the second representing the λ_* operator. Similarly we have both operators with respect to application. We include **cLambda** and **cApply** so that we can convert values, for which we have the equivalent of a regular λ operator, into expressions. The function which does this conversion will be discussed later on.

[1] Dependent induction is an experimental tactic defined in the module `Coq.Program.Equality` which performs induction on a hypothesis with a dependent type.

```
Inductive cexp :=
  cVar : var → cexp | cNum : nat → cexp
  | int1 | bool1 | pNil | pCons : cexp → cexp → cexp
  | Pair : cexp → cexp → cexp | Cost : cexp → cexp | Pot : cexp → cexp
  | cPlus : cexp → cexp → cexp | cMax : cexp → cexp → cexp
  | cLambda : var → cexp → cexp | cApply : cexp → cexp → cexp
  | cLambda_s : var → cexp → cexp | cApply_s : cexp → cexp → cexp
  | pListcase : cexp → cexp → var → var → cexp → cexp
  | pFold : cexp → cexp → var → var → var → cexp → cexp.
```

FIGURE 4.5. Complexity language expressions implemented in Coq.

We implement some notation in Coq to refer to complexity expressions. We use ::p as infix notation for pCons, +++ for cPlus and _/ for cMax. Additionally, we use the notation [[*r*,*s*]] for Pair *r s*, pcase *r* of (*s*,[*x*,*xs*]*t*) for pListcase, and pfold *r* of (*s*,[*x*, *xs*,*w*]*t*) for pFold.

We define the full complexity types automatically from the mathematical definition in Chapter 3.3.3, with constructors pBool, pInt, pInts, C, cArrow and cPair. In the previous chapter, we defined potential and complexity types as subsets of the full complexity types. In our implementation we define them simply as predicates. Subset types in Coq are defined as **sig** types, of the form $\{x : A \mid Px\}$. However it is not the case that if y is an element of a **sig** type $\{x : A \mid Px\}$, then it is also an element of type A. Instead of converting back and forth between these two types, it is simpler for our purposes to deal with full complexity types separately from the predicates stating when *tau* is a complexity type (**ctype**) or a potential type (**ptype**).

We use Box and ArrowBox (or –>b) notation liberally in our proofs. The definition of the complexity type contexts **ctype_context** and typing relation are as expected.

4.2.2. Semantics. For the semantics of the complexity language, we define values inductively. We choose an inductive definition over a set-theoretic one for two reasons. Firstly, it is more difficult to reason about sets than about inductively defined datatypes in Coq. Secondly, and more importantly, we will end up implementing an operational, rather than denotational,

```
Inductive cval :=
  | vNum : nat → cval | vint1 | vbool1 | vpNil | vpCons : cval → cval → cval
  | vPair : cval → cval → cval
  | vLambda : ∀ (x:var) (e:cexp), closed_except x e → cval.
```

FIGURE 4.6. Complexity language values in Coq.

semantics on the complexity language. The operational semantics lends itself more naturally to the inductive definition, which is given in Figure 4.6. The predicate closed_except takes a variable x and a complexity expression e and evaluates to True if every variable not equal to x is not free in e. This allows us to easily show that for any vLambda x e, cLambda x e is a closed expression. This is important so that we do not require environments in order to type or destruct values.

To facilitate this, we define a function cval_to_cexp which maps complexity values to their corresponding complexity expressions. This function is used in the evaluation rules. We also define a cval_typed predicate which implements a typing relation on complexity values.

A complexity value environment of type cval_env is an environment on elements of type **cval**. For a complexity type context G, a G-environment is defined to be a complexity value environment xi along with a proof that xi is consistent with G. We construct this definition by means of a record type.

```
Record G_env (G:ctype_context) :=
  mkenv { cvenv : cval_env; consist_cond : consistent G cvenv }.
```

That is, mkenv is a constructor for elements of type **G_env** and cvenv and consist_cond are its projections. A G-environment xi is coerced automatically to cvenv xi of type cval_env which means we can use any of the standard definitions on environments with it.

As we mentioned above, we chose to implement an operational semantics on the complexity language rather than the denotational one spelled out in the previous chapter. The reason for this has to do with Coq's termination requirement—every function defined in Coq must be accompanied by a proof of its termination. This proof does not always have to be explicitly spelled out by the user, or Coq would be an extremely difficult language to work in. Therefore there are a

limited number of ways to define recursive functions in ways that Coq can automatically prove termination. One of the most common ways to define recursive functions in Coq is through the `Fixpoint` construct, which we have already seen. But we cannot implement our denotational semantics in this way, because the application rules, as well as the `fold` operator rule, are not structurally recursive. Paulin-Mohring [2009] and Benton et al. [2009] give examples of how to construct a denotational semantics in Coq. For the purposes of this thesis however, it was unnecessary to add such a layer of complexity.

We define our operational semantics as an inductive predicate **ceval** on complexity closures (a **cexp**-cval_env pair) and complexity values. Most of the evaluation rules are straightforward, but some are more complex than our definition of the target langauge semantics. We spell out a few of these rules in Figure 4.7. For the cLambda and cLambda_s evaluation rules, we produce values of the form vLambda x s, where no variable besides x is free in s. In particular, the closure (cLambda x r,xi) evaluates to vLambda x r', where r' is the result of filling in every free variable in s, except for x, with $xi(x)$. To achieve this we define a `Fixpoint` function fill on complexity expressions.

The pFold evaluation rule computes the recursive step by utilizing the cval_to_cexp function. That is, it first determines that **ceval** (r, xi) (vpCons u us) and then evaluates the recursive judgement, that **ceval** (cval_to_cexp us, cval_empty) v.

Defined mutually recursively with ceval is the predicate **ceval_max**, which implements the maximum operators on complexity values.

4.3. Translation and Soundness

4.3.1. Translation. The definition of the translation function is straightforward. We define a cexp_context to be an environment on complexity language expressions. We use these contexts to deal with syntactic substitution in the translation. That is, trans' is a fixpoint definition which takes an expression e and a complexity expression context G as input and evaluates to the expression e' given by translating e and filling in all variables defined in G with their corresponding complexity expressions under G. So when G is defined on x, trans' (cVar x) G evaluates to $G(x)$, and when G is undefined on x, trans' (cVar x) G = cVar x. We define the

```
Inductive ceval : cclosure → cval → Prop :=
  | ceVar : ··· | ceNum : ··· | ceint1 : ··· | cebool1 : ··· | ceNil : ··· | ceCons : ···
  | cePair : ··· | ceCost : ··· | cePot : ··· | cePlus_C : ··· | ceMax : ···
  | ceLambda : ··· | ceApply : ··· | ceListcase_n : ··· | ceFold_n : ···
  | ceLambda_s : ∀ (r s:cexp)(x:var)(xi:cval_env)(pf_closed: closed_except x s),
      s = fill r (delete xi x) → ceval (cLambda_s x r,xi) (vPair (vNum 1) (vLambda x s pf_closed))
  | ceApply_s : ∀ (r s r':cexp)(x:var)(xi:cval_env)(u v:cval)(m n p q:nat)(pf_r':closed_except x r'),
      ceval (r,xi) (vPair (vNum m) (vLambda x r' pf_r')) → ceval (s,xi) (vPair (vNum n) u) →
      ceval (r', update cval_empty x (vPair (vNum 1) u)) (vPair (vNum p) v) →
      q = 1 + m + n + p → ceval (cApply_s r s,xi) (vPair (vNum q) v)
  | ceListcase_c : ∀ (r s t:cexp) (x xs:var) (xi:cval_env)(u us v_s v_t v:cval),
      ceval (r, xi) (vpCons u us) → ceval (s, xi) v_s →
      ceval (t, update (update xi x u) xs us) v_t → ceval_max v_s v_t v →
      ceval (pcase r of (s,[x,xs]t),xi) v
  | ceFold_c: ∀ (r s t:cexp)(x xs w:var)(xi:cval_env)(m n p q:nat)(u us vs rec_p vt v:cval),
      ceval (r,xi) (vpCons u us) → let vs' := vPair (vNum m) vs
      in coqdockwlet rec' := vPair (vNum n) rec_p in let vt' := vPair (vNum p) vt
      in ceval (s,xi) vs' → ceval (pfold (cval_to_cexp us) of (s,[x,xs,w]t),xi) rec' →
      let xi_x_xs_w := update (update (update xi x u) xs us) w (vPair (vNum 1) rec_p)
      in ceval (t,xi_x_xs_w) vt' → q = 2 + n + p → ceval_max vs vt v →
      ceval (pfold r of (s,[x,xs,w]t),xi) (vPair (vNum q) v)

with ceval_max : cval → cval → cval → Prop :=
  | max_vint1 : ··· | max_vbool1 : ··· | max_vpNil_vpNil : ··· | max_vpNil_vpCons : ···
  | max_vpCons_vpNil : ··· | max_vpCons_vpCons : ··· | max_C : ··· | max_vPair : ···
  | max_vLambda : ∀ (x:var)(r s:cexp)(pf_r:closed_except x r)(pf_s:closed_except x s),
      ceval_max (vLambda x r pf_r) (vLambda x s pf_s)
                (vLambda x (r \_/ s) (max_closed_except x r s pf_r pf_s)).
```

FIGURE 4.7. Evaluation rule for the complexity language's operational semantics.

translation trans of a closed expression to be its image under trans' given the empty complexity expression context.

4.3.2. Bounding Relation. Next we move on to the bounding relation. The relation should be a predicate of type

$$\textbf{type} \to \mathsf{closure} \to \textbf{cval} \to \texttt{Prop}$$

whereas the potential bound should have type

$$\textbf{type} \to \mathsf{val\_closure} \to \textbf{cval} \to \texttt{Prop}$$

For the complexity bound $\sqsubseteq$, there are five conditions we must show. Let the input to $\sqsubseteq$ be the target type τ, the closure (t, ξ), and the complexity χ. The bounding relation requires

(1) There is some type environment Γ, consistent with ξ, such that $\Gamma \vdash t : \tau$;
(2) The complexity χ is an element of $[\![||\tau||]\!]$;
(3) χ is equal to a pair (n, χ') for some natural number n and potential χ';
(4) This same n is an upper bound on the size of the evaluation derivation of $t\,\xi \downarrow v\,\theta$ for some value closure $v\theta$; and
(5) $v\,\theta \sqsubseteq^{\mathbf{pot}}_{\tau} \chi'$.

The potential bound depends on τ and the values of v, θ and χ. Specifically, the potential bounding relation holds

(1) If v is any integer, θ is the empty environment and χ' is the complexity value vint1;
(2) If v is a boolean value, θ is the empty environment and χ' is the complexity value vbool1;
(3) If v is an integer list, θ is the empty environment and χ' is a list of vint1's whose length is greater than or equal to the length of v; or
(4) If $v = \lambda x.r$ for some expression r, $\chi' = \lambda x r'$ for some complexity expression r', and $\theta = \alpha \to \beta$ for types α and β, then the relation $\lambda x r\,\theta \sqsubseteq^{\mathbf{pot}}_{\alpha\to\beta} \chi'$ holds if, whenever $z\,\eta \sqsubseteq^{\mathbf{pot}}_{\alpha} p$ and $r'\,\{x \mapsto p\} \downarrow q$, $r\,\theta[x \mapsto z\eta] \sqsubseteq_{\beta} q$.

Our first attempt at this definition is as an inductive predicate, shown in Figure 4.8. We have previously defined the predicate **consistent_t** to hold on Γ and ξ if, for every x in the domain

```
Inductive bound : type → closure → cval → Prop :=
  | cBound : ∀ (tau:type)(t:exp)(xi:val_env)(X:cval),
    (∃ G:type_env, consistent_t G xi ∧ typed G t tau) →
    cval_typed X (type_trans tau) →
    (∀ (VC:val_closure)(n:nat)(Y:cval), eval_bound (t,xi) VC n →
     cval_equiv X (vPair (vNum n) Y) → pbound tau VC Y) →
    bound tau (t,xi) X
with pbound : type → val_closure → cval → Prop :=
  | pbInt : ∀ (n:Z), pbound Int (num n,val_empty) vint1
  | pbT : pbound Bool (tt,val_empty) vbool1
  | pbF : pbound Bool (ff,val_empty) vbool1
  | pbInts : ∀ (vs:val)(ws:cval), length_bound vs ws → pbound Ints (vs,val_empty) ws
  | pbLambda : ∀ (sigma tau:type)(x:var)(r:exp)(theta:val_env) (q:cexp)(Z:cval),
    (∀ (VC':val_closure)(p:cval), pbound sigma VC' p →
     ceval (q,update cval_empty x p) Z → bound tau (r,val_update theta x VC') Z) →
    pbound (sigma -> tau) (lambda x r,theta) (vLambda x q).
```

FIGURE 4.8. An incorrect definition of the bounding relation which violates the positivity requirement.

of both environments, $\xi(x)$ is a value of type $\Gamma(x)$. Additionally, **length_bound** is a predicate which takes a value *vs* and a complexity value *ws*, and holds if *vs* and *ws* are both lists and the length of *vs* is less than or equal to the length of *ws*. Furthermore, **cval_equiv** is an equivalence relation on complexity values.

When we try to pass this definition into coq, the system produces the following error:

```
Error:  Non strictly positive occurrence of "pbound" in
∀ (sigma tau:type)(x:var)(r:exp)(theta:val_env)(q:cexp)(s:cval),
    (∀ (VC':val_closure)(p:cval), pbound sigma VC' p →
     ceval (q,update cval_empty x p) s →
```

bound *tau* (*r*,val_update *theta x VC'*) *s*) →
pbound (*sigma* –> *tau*) (lambda *x r*,*theta*) (vLambda *x q*)

The positivity condition in Coq is a restriction on inductive definitions which is sufficient to prove termination. In its simplest incarnation, the positivity condition requires that all recursive occurrences of the inductive definition occur in the conclusion of a predicate. That is, X occurs positively in X, and X occurs positively in $\forall x : A, T$ if X does not occur in A and occurs positively in T. In actuality the condition has a bit more flexibility. In the **bound** example however, the recursive call is substantially embedded in the hypothesis of the constructor. The full definition of the positivity condition is described in the Coq Development Manual [2009].

Retaining the original specification of the potential bound, it is impossible to get around this constraint with an inductive definition. On the other hand, we know the definition of the bounding relation is well-formed, because it is indexed by the type τ, which decreases structurally. So far we have only defined predicates by means of inductive definitions. In fact, just as we can use the `Inductive` construct to define both datatypes and predicates, we can use the recursive function declaration `Fixpoint` to define predicates as well. Therefore our second attempt at defining the bounding relation is as a `Fixpoint` definition in Figure 4.9.

But Coq rejects this definition as well. The recursive call to pbound is invoked on *tau* instead of on a subterm of *tau*; for Coq this means the definition is not strictly structurally recursive. But we know that every call to pbound terminates, because every recursive call inside of pbound is invoked on a strict subterm.

Our solution is a small tweak to the fixpoint definition. Instead of defining cbound mutually with pbound, we can define pbound as an anonymous recursive function. Now the case which was problematic is implemented not as a recursive call but as the application of the anonymous fixpoint operator. This final definition is given in Figure 4.10.

So that we can refer to the potential bounding relation on its own, we subsequently define
`Fixpoint` pbound (*tau*:**type**) (*VC*:valclosureval_closure) (*Y*:**cval**) : `Prop` := ⋯
so that pbound is equivalent to the anonymous operator.

```
Fixpoint bound (tau:type) (t:exp) (xi:val_env) (X:) {struct tau} : Prop :=
  (∃ G:type_env, consistent_t G xi ∧ typed G t tau) ∧
  cval_typed X (type_trans tau) ∧
  ∃ VC:val_closure, ∃ n:nat, ∃ Y:cval,
  eval_bound (t,xi) VC n ∧ cval_equiv X (vPair (vNum n) Y) ∧ pbound tau VC Y

with pbound (tau:type) (VC:val_closure)(Y:cval) : Prop :=
  vtyped VC tau ∧ cval_typed Y (ptype_trans tau) ∧
  match tau,VC,Y with
    | Bool,(tt | ff,theta),vbool1 ⇒ equiv_empty theta
    | Int,(num z,theta),vint1 ⇒ equiv_empty theta
    | Ints,(vs,theta),_⇒ length_bound vs Y ∧ equiv_empty theta
    | sigma -> tau,(lambda x r,theta), vLambda y s pf ⇒
      ∀ (VC':val_closure)(Y' Z:cval), pbound sigma VC' Y' →
        ceval (s,update empty_cval y (vPair (vNum 1) Y')) Z →
        bound tau r (vEnv (update theta x VC')) Z
    | _,_,_⇒ False
  end.
```

FIGURE 4.9. An incorrect definition of the bounding relation which violates the structural recursion of the variable tau.

The bound on environments **env_bound** takes a type context G, a G-environment xi, a complexity type context G' which is point-wise equivalent to type_trans G, and a G'-environment xi'. Given these, the inductive predicate **env_bound** G xi G' xi' holds if, whenever get G x = Some t and get xi' x = Some X, then we have bound t (Var x,xi) X.

Additionally we define the bound on expressions, **exp_bound**. We define **exp_bound** r s to hold if, whenever **env_bound** G xi G' xi' such that **typed** G r tau for some type tau, **eval** (r,xi) VC for some value closure VC and **ceval** (s,xi') Y for some complexity value Y, we have bound tau (r,xi) Y.

```
Fixpoint bound (sigma:type)(t:exp)(xi:val_env)(X:cval) : Prop :=
  (∃ G:type_context, consistent_t G xi ∧ typed G t sigma) ∧
  cval_typed X (type_trans sigma) ∧
  ∃ n:nat, ∃ VC:val_closure, ∃ Y:cval,
  cval_equiv X (vPair (vNum n) Y) ∧ deriv_bound (t,xixi) VC n ∧

  (fix pbound (tau:type)(VC:val_closure)(Y:cval) : Prop :=
   vtyped VC tau ∧ cval_typed Y (ptype_trans tau) ∧
   match tau,VC,Y with
     | Bool,(tt | ff,theta),vbool1 ⇒ equiv_empty theta
     | Int,(num z,theta),vint1 ⇒ equiv_empty theta
     | Ints,(vs,theta),_⇒ length_bound vs Y ∧ equiv_empty theta
     | sigma -> tau,(lambda x r,theta), vLambda y s pf ⇒
       ∀ (VC':val_closure)(Y' Z:cval), pbound sigma VC' Y' →
         ceval (s,update empty_cval y (vPair (vNum 1) Y')) Z →
         bound tau r (vEnv (update theta x VC')) Z
     | _,_,_⇒ False
   end) sigma VC Y.
```

FIGURE 4.10. The final definition of the bounding relation, constructed as a definition by structural recursion.

4.3.3. Soundness. The general structure of the soundness proof in Coq is the same as the one laid out in Section 3.4.3. But because we are working in a much stricter system, we need to compile a small library of lemmas for the sole purpose of proving soundness.

A number of these lemmas involve ways to destruct the bounding relation into its various components. An example is the following, which states that if $t\,\xi \sqsubseteq_\tau \chi$ and $t\,\xi \downarrow v\,\theta$ then $v\,\theta \sqsubseteq_\tau^{\mathbf{pot}} \mathbf{pot}(\chi)$:

```
Lemma bound_pbound : ∀ (sigma:type) (t:exp) (xi:val_env) (VC:val_closure) (X Y:cval),
```

bound *sigma t xi* (vPair *X Y*) → **eval** (*t*,*xi*) *VC* → pbound *sigma VC Y*.

Since bound is defined by structural recursion on *tau*, we cannot simplify goals of the form bound *tau t xi X* into its separate components automatically when *tau* is an arbitrary variable. The projections like bound_pbound therefore provide us with a means of obtaining information about bounds even when we do not know the value of *tau*.

With a large collection of these results, we can prove the soundness result in Coq:

`Theorem` trans_sound : ∀ (*e*:**exp**), **exp_bound** *e* (trans *e*).

CHAPTER 5

Conclusion and Future Work

5.1. Summary and Main Contributions

The goal of this work is to produce a static complexity analysis of a higher-order language. We start with a simple target language in which to base our analysis. The target language we choose is the simply-typed lambda calculus with integer lists and structural list recursion. We measure the cost of a target language term by the size of its evaluation derivation in the big-step operational semantics. We develop a notion of the complexity of a target term, which reflects both the cost of evaluation as well as the size of the expression. Target terms and these semantic complexities are related via the bounding relation, defined in Section 3.4.3.

To better reason about complexities, we construct a complexity language that parallels the target one. Expressions in the complexity language are mapped into the realm of semantic complexities via a denotational semantics. The translation function maps target expressions into the complexity language. The main result of Chapter 3 is the soundness of the translation function under the bounding relation.

In Chapter 4 we describe the implementation of the translation system in Coq. The resulting system takes a target term and produces a certified upper bound on its complexity. During the implementation process we encountered a number of difficulties in translating the mathematical development of the system into Coq. The difficulties mainly stemmed from Coq's termination requirement, which states that every program written in Coq must be accompanied by a proof of its termination. Over the course of our work we were induced to construct our definitions and proofs in creative ways in order to abide by the requirement.

5.2. Future Work

5.2.1. Completing and Improving the Implementation. At this point in time, our implementation of the translation scheme in Coq is based on a small library of lemmas, some of which are unproven. Future work will fill in the gaps in the library to achieve a fully certified system. There are also areas for improvement in the implementation. First, as discussed in Section 4.1, it might be worthwile to change the definition of environments from partial functions to a finite list, in order to better compare equality of environments. Additionally, we have not fully explored the merits of a denotational semantics for the complexity language, and future work might implement such a semantics and compare the benefits and drawbacks of each approach.

5.2.2. Extensions to the Target Language. In Section 3.5 we briefly explored ways to expand the target language to include simple datatypes like trees. Future work would continue these efforts, with the goal of automatically generating translation schemes for arbitrary user-defined datatypes. One difficulty with this automation seems to be how to interpret the potential of an arbitrary datatype. Our original intention was to retain copies of the user-defined datatypes in the complexity language, thereby allowing us to perform an analysis without losing information about the structure of the datatype. However as we saw with the example of binary words in Section 3.5, full generalization cannot be achieved so straightforwardly.

An alternate approach might be for the user, when defining a new datatype, to also define a map from the datatype to the natural numbers. The base potential types of the complexity language would be the set of natural numbers. The challenge of this approach is then to define a clever translation of case and fold expressions on target datatypes into case and fold expressions on sizes.

Another area of future work is incorporating a more general recursion scheme into the target language. A next step after structural recursion is a divide-and-conquer approach, for example. For general recursion, we need to consider how the translation scheme responds to non-terminating computations. If we allow non-terminating functions in the target language, we want to ensure that the translation of an expression is still an upper bound on its complexity.

In particular, we need to ensure that if, for $\xi \sqsubseteq \xi^*$, the cost of $\|t\|$ under ξ^* is a natural number, then $t\xi$ has a (finite) evaluation derivation.

5.2.3. Program Transformations. One of the goals of the implementation of the translation system is to produce a usable system which automatically computes meaningful complexity bounds for target expressions. As the system currently stands, the translation produces messy, unreadable complexity language expressions, like we saw in Section 3.4.4. One extension to the work is to implement a number of bound-preserving program transformations on the complexity language like the ones we performed ad-hoc in the example. We would require that if $r \sqsubseteq s$ and s can be transformed into s' by a transformation rule then $r \sqsubseteq s'$. Among these transformations would be evaluation of closed expressions including arithmetic and application, as well as the solving the recurrences described by `plistcase` and `pfold` expressions. Then, given a target expression, the system would apply the translation function and program transformations and return a simple, readable and certifiably correct complexity bound.

Bibliography

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java bytecode. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 157–172, Berlin, Heidelberg, 2007. Springer-Verlag. doi: 10.1145/1363686.1363779.

Nick Benton, Andrew Kennedy, and Carsten Varming. Some domain theory and denotational semantics in Coq. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 115–130. Springer-Verlag, 2009. doi: 10.1007/978-3-642-03359-9_10.

Ralph Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318, June 2004. doi: 10.1016/j.tcs.2003.10.022.

Norman Danner and James S. Royer. Adventures in time and space. *Logical Methods in Computer Science*, 3(1), 2007. doi: 10.2168/LMCS-3(1:9)2007.

Norman Danner and James S. Royer. Two algorithms in search of a type-system. *Theory of Computing Systems*, 45, July 2009. URL `http://dl.acm.org/citation.cfm?id=1611588.1611604`.

Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin, editors, *From Semantics to Computer Science*, pages 383–413. Cambridge University Press, 2009. doi: 10.1017/CBO9780511770524.018. URL `http://hal.inria.fr/inria-00431806`.

Mads Rosendahl. Automatic complexity analysis. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 144–156, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: 10.1145/99370.99381.

David Sands. *Calculi for time analysis of functional programs*. PhD thesis, Department of Computing, Imperial College, Uniersity of London, September 1990.

David Sands. Operational theories of improvement in functional languages (extended abstract). In *Proceedings of the 1991 Glasgow Workshop on Functional Programming*, pages 298–311. Springer-Verlag, 1992.

Jon Shultis. On the complexity of higher-order programs. Technical Report CU-CS-288-85, University of Colorado, Boulder, 1985.

Coq Development Team. *The Coq proof assistant reference manual*, 2009. URL `http://coq.inria.fr/`.

Reinhard Wilhelm. Timing analysis and timing predictability. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in Computer Science*, pages 317–323. Springer Berlin / Heidelberg, 2005. doi: 10.1007/11561163_14.