Denotational Cost Semantics for Functional Languages with Inductive Types

Norman Danner [*]
Wesleyan University, USA
ndanner@wesleyan.edu

Daniel R. Licata [†]
Wesleyan University, USA
dlicata@wesleyan.edu

Ramyaa Ramyaa
Wesleyan University, USA
ramyaa@wesleyan.edu

Abstract

A central method for analyzing the asymptotic complexity of a functional program is to extract and then solve a recurrence that expresses evaluation cost in terms of input size. The relevant notion of input size is often specific to a datatype, with measures including the length of a list, the maximum element in a list, and the height of a tree. In this work, we give a formal account of the extraction of cost and size recurrences from higher-order functional programs over inductive datatypes. Our approach allows a wide range of programmer-specified notions of size, and ensures that the extracted recurrences correctly predict evaluation cost. To extract a recurrence from a program, we first make costs explicit by applying a monadic translation from the source language to a complexity language, and then abstract datatype values as sizes. Size abstraction can be done semantically, working in models of the complexity language, or syntactically, by adding rules to a preorder judgement. We give several different models of the complexity language, which support different notions of size. Additionally, we prove by a logical relations argument that recurrences extracted by this process are upper bounds for evaluation cost; the proof is entirely syntactic and therefore applies to all of the models we consider.

Categories and Subject Descriptors F.3.1 [*Logics and meanings of programs*]: Specifying and verifying and reasoning about programs; F.3.2 [*Logics and meanings of programs*]: Semantics of programming languages

General Terms Verification.

Keywords Semi-automatic complexity analysis.

[*] This material is based upon work supported by the National Science Foundation under grant no. 1318864.

[†] This material is based on research sponsored by The United States Air Force Research Laboratory under agreement number FA9550-15-1-0053. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the United States Air Force Research Laboratory, the U.S. Government or Carnegie Mellon University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada.
Copyright © 2015 ACM 978-1-4503-3669-7/15/08…$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

The typical method for analyzing the asymptotic complexity of a functional program is to extract a recurrence that relates the function's running time to the size of the function's input, and then solve the recurrence to obtain a closed form and big-O bound. Automated complexity analysis (see the related work in Section 7) provides helpful information to programmers, and could be particularly useful for giving feedback to students. In a setting with higher-order functions and programmer-defined datatypes, automating the extract-and-solve method requires a generalization of the standard theory of recurrences. This generalization must include a notion of recurrence for higher-order functions such as `map` and `fold`, as well as a general theory of what constitutes "the size of the input" for programmer-defined datatypes.

One notion of recurrence for higher-order functions was developed in previous work by Danner and Royer (2009) and Danner et al. (2013). Because the output of one function is the input to another, it is necessary to extract from a function not only a recurrence for the running time, but also a recurrence for the size of the output. These can be packaged together as a single recurrence that, given the size of the input, produces a pair consisting of the running time (called the *cost*) and the size of the output (called the *potential*). Whereas the former is the cost of executing the program to a value, the latter determines the cost of using that value. This generalizes naturally to higher-order functions: a recurrence for a higher-order function is itself a higher-order function, which expresses the cost and potential of the result in terms of a given recurrence for the cost and potential of the argument function. The process of extracting recurrences can thus be seen as a denotational semantics of the program, where a function is interpreted as a function from input potential to cost and output potential.

Building on this work, we give a formal account of the extraction of recurrences from higher-order functional programs over inductive datatypes, focusing how to soundly allow programmer-specified sizes of datatypes. We show that under some mild conditions on sizes, the cost predicted by an extracted recurrence is in fact an upper bound on the number of steps the program takes to evaluate. The size of a value can be taken to be (essentially) the value itself, in which case one gets exact bounds but must reason about all the details of program evaluation, or the size of a value can forget information (e.g. abstracting a list as its length), in which case one gets weaker bounds with more traditional reasoning.

We start from a call-by-value source language, defined in Section 2, with strictly positive inductive datatype definitions (which include lists and finitely and infinitely branching trees) Datatypes are used via case-analysis and structural recursion (so the language is terminating), but unlike in Danner et al. (2013), recursive calls are only evaluated if necessary—for example, recurring on one branch of a tree has different cost than recurring on both branches. The

cost of a program is defined by an operational cost semantics, an evaluation relation annotated with costs. For simplicity, the cost semantics measures only the number of function applications and recursive calls made during evaluation, but our approach to extracting recurrences generalizes to other cost models.

We extract a recurrence from such a program in two steps. First, in Section 3, we make the cost of evaluating a program explicit, by translating a source program e to a program $\|e\|$ in a complexity language. The complexity language has an additional type $\mathbf{C}$ for costs, and the translation to the complexity language is a call-by-value monadic translation into the writer monad $\mathbf{C}\times -$ (Moggi 1991; Wadler 1992). The translated program $\|e\|$ returns an additional result, which is the cost of running the original program e.

Second, we abstract values to sizes; we study both semantic and syntactic approaches. In Section 4, we give a size-based semantics of the complexity language, which relies on programmer-specified size functions mapping each datatype to the natural numbers (or some other preorder). Typical size functions include the length of a list and the size or depth of a tree. The semantics satisfies a *bounding theorem* (Theorem 7), which implies that the denotational cost given by composing the source-to-complexity translation with the size-based semantics is in fact an upper bound on the operational cost. We show some examples that the recurrence for cost extracted by this process is the expected one; later we also show that the results in Danner et al. (2013) carry over.

Alternatively, the abstraction of values to sizes can be done syntactically in the complexity language, by imposing a preorder structure on the values of the datatype themselves. For example, rather than mapping lists to numbers representing their lengths, we can order the list values by rules including $xs \leq (x{::}xs)$ and $(x{::}xs) \leq (y{::}xs)$. The second rule says that the elements of the list are irrelevant, quotienting the lists down to natural numbers, and the first generates the usual order on natural numbers. Formally, we equip the complexity language with a judgement $E \leq E'$ that can be used to make such abstractions. In Section 5, we identify properties of this judgement that are sufficient to prove a syntactic bounding theorem (Theorem 12), which states that the operational cost is bounded by the cost component of the complexity translation. The key technical notion is a logical relation between the source and complexity languages that extends the bounding relation of Danner et al. (2013) to inductive types. This proof gives a bounding theorem for any model of the complexity language that validates the rules for $\leq$. In Section 6, we show that these rules are valid in the size-based semantics of Section 4 (thereby proving Theorem 7), and we discuss several other models of the complexity language.

This gives a formal account of what it means to extract a recurrence from higher-order programs on inductive data. We leave an investigation of what it means to solve these higher-order recurrences to future work. Danner et al. (2015) is a full version of this paper.

2. Source Language with Inductive Data Types

The source language is a simply-typed λ-calculus with product types, function types, suspensions, and strictly positive inductive datatypes. Its syntax, typing, and operational semantics are given in Figure 1. We bundle sums and inductive types together as datatypes, rather than using separate $+$ and μ types, because below we do not want to consider sizes for the sum part separately. We assume a top-level signature ψ consisting of datatype declarations of the form

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

Each constructor's argument type is specified by a strictly positive functor ϕ. These include the identity functor (t), representing a recursive occurrence of the datatype; constant functors (τ), representing a non-recursive argument; product functors ($\phi_1 \times \phi_2$), representing a pair of arguments; and constant exponentials ($\tau \to \phi$), representing an argument of function type. For example for τ `list`, the argument type for `Nil` is `unit` (constant functor), and the argument type for `Cons` is $\tau \times t$ (product of constant and recursive arguments). We write $\phi[\tau] = \phi[\tau/t]$ for substitution for the single free type variable t in ϕ. We sometimes abbreviate further by dropping the type superscripts and writing $\texttt{datatype}\ \delta = \overline{C\ \texttt{of}\ \phi_C}$ and by writing C rather than C_i to refer to one of the constructors of the declaration. In the signature, each ϕ_C in each `datatype` declaration must refer only to datatypes that are declared earlier in the sequence, to avoid introducing general recursive datatypes (see the full paper for the formal definition). We write $C : (\phi \to \delta) \in \psi$ to mean that the signature ψ contains a datatype declaration of the form $\texttt{datatype}\ \delta = \ldots \mid C\ \texttt{of}\ \phi[\delta] \mid \ldots$. We generally elide the signature from typing, but sometimes write $\gamma \vdash_\psi e : \tau$ to include it. The elimination rule for a datatype δ is structural recursion, $\texttt{rec}^\delta(e, \overline{C \mapsto x.e_C})$. When $\phi_C = \texttt{unit}$, we assume $x \notin \mathrm{fv}(e_C)$ and write e_C instead of $x.e_C$.

Evaluation is call-by-value and products and datatypes are strict. However, unfolding datatype recursors requires substituting expressions (the recursor applied to the components of the value) for the variables standing for the recursive calls—running the recursive call first and substituting its value would require a function to make all possible recursive calls. We handle this using suspensions: when computing a τ by recursion, the result of a recursive call is given the type $\texttt{susp}\ \tau$. The values of type $\texttt{susp}\ \tau$ are $\texttt{delay}(e)$ where e is an expression of type τ; the elimination form `force` forces evaluation. When defining a recursive computation of result type τ, the branch for a constructor C has access to a variable of type $\phi_C[\delta \times \texttt{susp}\ \tau]$, which gives access both to the "predecessor" values of type δ and to the recursive results. This supports both case-analysis and structural recursion, and recursive calls are only computed if they are used.

For any strictly positive functor ϕ, the $\texttt{map}^\phi$ expression witnesses functoriality, essentially lifting a function $\tau_0 \to \tau_1$ to a function $\phi[\tau_0] \to \phi[\tau_1]$. It is used in the operational semantics for the recursor to insert recursive calls at the right places in ϕ (Harper (2013) provides an exposition). We will only need to lift maps $x : \tau_0.v : \tau_1$ whose bodies are syntactic values (or variables), and apply them to syntactic values (or variables), and we restrict `map` to this special case to simplify its cost semantics.

The cost semantics in Figure 1 defines the relation $e \downarrow^n v$, which means that the expression e evaluates to the value v in n steps. Our cost model charges only for the number of function applications and recursive calls made by datatype recursors. This prevents constant-time overheads from the encoding of datatypes using product and suspension types from showing up in the extracted recurrences. It is simple to adapt the denotational cost semantics below to other operational cost semantics, such as one that charges for these steps, or assigns different costs to different constructs.

Substitutions are defined as usual:

DEFINITION 1. We write θ for substitutions $v_1/x_1, \ldots, v_n/x_n$, and $\theta : \gamma$ to mean that $\mathrm{Dom}\ \theta \subseteq \mathrm{Dom}\ \gamma$ and $\emptyset \vdash \theta(x) : \gamma(x)$ for all $x \in \mathrm{Dom}\ \theta$. We define the application of a substitution θ to an expression e as usual and denote it $e[\theta]$.

LEMMA 1. *If $x \notin \mathrm{Dom}\ \theta$, then $e[\theta, x/x][e_1/x] = e[\theta, e_1/x]$.*

For source cost expressions n, we write $n \leq n'$ for the order given by interpreting these cost expressions as natural numbers (i.e. the free precongruence generated by the monoid equations for $(+, 0)$ and $0 \leq 1$). We have the following syntactic properties of evaluation:

LEMMA 2 (Value Evaluation).

- *If $v \downarrow^n v'$ then $n \leq 0$ and $v = v'$.*

Types:

$$\begin{array}{rcl} \tau & ::= & \mathtt{unit} \mid \tau \times \tau \mid \tau \to \tau \mid \mathtt{susp}\,\tau \mid \delta \\ \phi & ::= & t \mid \tau \mid \phi \times \phi \mid \tau \to \phi \\ \mathtt{datatype}\,\delta & = & C_0^\delta\ \mathtt{of}\ \phi_{C_0}[\delta] \mid \ldots \mid C_{n-1}^\delta\ \mathtt{of}\ \phi_{C_{n-1}}[\delta] \end{array}$$

Expressions:

$$\begin{array}{rcl} v & ::= & x \mid \langle\,\rangle \mid \langle v, v\rangle \mid \lambda x.e \mid \mathtt{delay}(e) \mid C\,v \\ e & ::= & x \mid \langle\,\rangle \mid \langle e, e\rangle \mid \mathtt{split}(e, x.x.e) \mid \lambda x.e \mid e\,e \\ & & \mid \mathtt{delay}(e) \mid \mathtt{force}(e) \\ & & \mid C^\delta\,e \mid \mathtt{rec}^\delta(e, \overline{C \mapsto x.e_C}) \\ & & \mid \mathtt{map}^\phi(x.v, v) \mid \mathtt{let}(e, x.e) \\ n & ::= & 0 \mid 1 \mid n + n \end{array}$$

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

$$\frac{\gamma \vdash e : \tau}{\gamma \vdash \mathtt{delay}(e) : \mathtt{susp}\,\tau} \qquad \frac{\gamma \vdash e : \mathtt{susp}\,\tau}{\gamma \vdash \mathtt{force}(e) : \tau}$$

$$\frac{\gamma \vdash e : \phi_C[\delta]}{\gamma \vdash C^\delta\,e : \delta}$$

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

$$\frac{\gamma, x : \tau_0 \vdash v_1 : \tau_1 \quad \gamma \vdash v_0 : \phi[\tau_0]}{\gamma \vdash \mathtt{map}^\phi(x.v_1, v_0) : \phi[\tau_1]}$$

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

$$\frac{}{\mathtt{delay}(e) \downarrow^0 \mathtt{delay}(e)} \qquad \frac{e \downarrow^{n_0} \mathtt{delay}(e_0) \quad e_0 \downarrow^{n_1} v}{\mathtt{force}(e) \downarrow^{n_0+n_1} v}$$

$$\frac{}{\mathtt{map}^t(x.v, v_0) \downarrow^0 v[v_0/x]} \qquad \frac{}{\mathtt{map}^\tau(x.v, v_0) \downarrow^0 v_0}\ (t \text{ not free in } \tau)$$

$$\frac{e \downarrow^n v}{Ce \downarrow^n Cv}$$

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

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

$$\frac{}{\mathtt{map}^{\tau \to \phi}(x.v, \lambda y.e) \downarrow^0 \lambda y.\,\mathtt{let}(e, z.\mathtt{map}^\phi(x.v, z))}$$

Figure 1: Source language syntax and typing and operational semantics. Standard typing rules for variables, product types, function types, and `let` are omitted. In omitted operational rules, the costs are the sum of the costs of the subevaluations, except for $e_0\ e_1$, which adds 1.

- *For all* v, $v \downarrow^0 v$.

LEMMA 3 (Totality of map). *If* $\gamma \vdash \mathtt{map}^\phi(x.v_1, v_0) : \phi[\tau_1]$ *then* $\mathtt{map}^\phi(x.v_1, v_0) \downarrow^0 v$ *for some* v.

3. Making Costs Explicit

3.1 The Complexity Language

The complexity language will serve as a monadic metalanguage (Moggi 1991) in which we make evaluation cost explicit. The syntax and typing are given in Figure 2. The preorder judgement defined in Section 5 will play a role analogous to an operational or equational semantics for the complexity language.

Because we are not concerned with the evaluation steps of the complexity language itself, we remove features of the source language that were used to control evaluation costs. Product types are eliminated by projections, rather than `split`. We allow substitution of arbitrary expressions for variables, which is used in recursors for datatypes. Consequently, suspensions are not necessary. We treat $\mathsf{map}^\Phi(x.E, E_1)$ as an admissible rule (macro), defined by induction on Φ:

$$\frac{\Gamma, x : T_0 \vdash E_1 : T_1 \quad \Gamma \vdash E_0 : \Phi[T_0]}{\Gamma \vdash \mathsf{map}^\Phi(x.E_1, E_0) : \Phi[T_1]}$$

$$\begin{aligned} \mathsf{map}^t(x.E, E_0) &:= E[E_0/x] \\ \mathsf{map}^T(x.E, E_0) &:= E_0 \\ \mathsf{map}^{\Phi_0 \times \Phi_1}(x.E, E_0) &:= \langle \mathsf{map}^{\Phi_0}(x.E, \pi_0 E_0), \mathsf{map}^{\Phi_1}(x.E, \pi_1 E_0)\rangle \\ \mathsf{map}^{T \to \Phi}(x.E, E_1) &:= \lambda y.\,\mathsf{map}^\Phi(x.E, E_1\,y) \end{aligned}$$

The type $\mathbf{C}$ represents some domain of costs. The term constructors for $\mathbf{C}$ say only that it is a monoid $(+, 0)$ with a value 1 representing a single step. Costs can be interpreted in a variety of ways—e.g. as natural numbers and as natural numbers with infinity (Section 4).

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

LEMMA 4.

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

3.2 The Complexity Translation

Consider a higher-order function such as `map` on lists:

$$\begin{aligned} \mathtt{listmap} = \lambda(f, xs).\,\mathtt{rec}(xs,\ & \\ & \mathtt{Nil} \mapsto \mathtt{Nil} \\ & \mid \mathtt{Cons} \mapsto \langle y, \langle ys, r\rangle\rangle.\mathtt{Cons}(f\,y, \mathtt{force}(r))) \end{aligned}$$

The cost of $\mathtt{listmap}(f, xs)$ depends on the sizes of each element of xs, and the cost of evaluating f on elements of those sizes. However, since $\mathtt{listmap}(f, xs)$ might itself be an argument to another function (e.g. another `listmap`), we also need to predict the sizes of the elements of $\mathtt{listmap}(f, xs)$, which depends on the size of the output of f. Thus, to analyze `listmap`, we should be given a recurrence for the cost and size of $f(x)$ in terms of the size of x, and need to produce a recurrence that gives the cost and size of $\mathtt{listmap}(f, xs)$ in terms of the size of xs. We call the size of the value of an expression that expression's *potential*, because the size of the value determines what future uses of that value will cost.

This discussion motivates a translation $\|\cdot\|$ from source language terms to complexity language terms so that if $e{:}\tau$, then $\|e\|{:}\mathbf{C} \times \langle\!\langle \tau \rangle\!\rangle$. In the complexity language, we call an expression of type $\mathbf{C} \times \langle\!\langle \tau \rangle\!\rangle$ a *complexity*, an expression of type $\mathbf{C}$ a *cost*, and an expression

Types:

$$\begin{array}{rrcl} & T & ::= & \mathbf{C} \mid \mathsf{unit} \mid \Delta \mid T \times T \mid T \to T \\ & \Phi & ::= & t \mid T \mid \Phi \times \Phi \mid T \to \Phi \\ \mathtt{datatype}\ \Delta & & = & C_0^\Delta\ \mathtt{of}\ \Phi_{C_0}[\Delta] \mid \ldots \mid C_{n-1}^\Delta\ \mathtt{of}\ \Phi_{C_{n-1}}[\Delta] \end{array}$$

Expressions:

$$\begin{array}{rcl} E & ::= & x \mid 0 \mid 1 \mid E + E \mid \\ & & \langle\,\rangle \mid \langle E, E\rangle \mid \pi_0\, E \mid \pi_1\, E \mid \lambda x.E \mid E\, E \\ & & \mid C^\Delta\, E \mid \mathsf{rec}^\Delta(E, \overline{C \mapsto x.E_C}) \end{array}$$

Typing: $\Gamma \vdash E : T$.

$$\frac{\Gamma \vdash E : \Phi_C[\Delta]}{\Gamma \vdash C^\Delta\, E : \Delta}$$

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

Figure 2: Complexity language types, expressions, and typing.
The typing rules are standard for unit, product, and arrow types. $\mathbf{C}$ has a binary operation $+$ and elements 0 and 1.

of type $\langle\!\langle \tau \rangle\!\rangle$ a *potential*. We abbreviate $\mathbf{C} \times \langle\!\langle \tau \rangle\!\rangle$ by $\|\tau\|$. The first component of $\|e\|$ is the cost of evaluating e, and the second component of $\|e\|$ is the potential of e.

For example, `listmap` is a value, so its cost should be zero. On the other hand, its potential should describe what future uses of `listmap` will cost, in terms of the potentials of its arguments. For the type of `listmap` (uncurried), the above discussion suggests

$$\begin{array}{l} \langle\!\langle (\tau \to \sigma) \times (\tau\ \mathtt{list}) \to \sigma\ \mathtt{list} \rangle\!\rangle := \\ \quad (\langle\!\langle \tau \rangle\!\rangle \to \mathbf{C} \times \langle\!\langle \sigma \rangle\!\rangle) \times \langle\!\langle \tau\ \mathtt{list} \rangle\!\rangle \to \mathbf{C} \times \langle\!\langle \sigma\ \mathtt{list} \rangle\!\rangle \end{array}$$

For the argument function, we are provided a recurrence that maps τ-potentials to costs and σ-potentials. For the argument list, we are provided a τ `list`-potential. Using these, the potential of `listmap` must give the cost for doing the whole `map` and give a σ `list`-potential for the value. This illustrates how the potential of a higher-order function is itself a higher-order function.

As discussed above, we stage the extraction of a recurrence, and in the first phase, we do not abstract values as sizes (e.g. we do not replace a list by its length). Because of this, the complexity translation has a succinct description. For any monoid $(\mathbf{C}, +, 0)$, the writer monad (Wadler 1992) $\mathbf{C} \times -$ is a monad with

$$\begin{array}{l} \mathsf{return}(E) := (0, E) \\ E_1 \gg\!= E_2 := (\pi_0(E_1) + \pi_0(E_2(\pi_1(E_1))), \pi_1(E_2(\pi_1(E_1)))) \end{array}$$

The monad laws follow from the monoid laws for $\mathbf{C}$. Thinking of $\mathbf{C}$ as costs, these say that the cost of $\mathsf{return}(e)$ is zero, and that the cost of bind is the sum of the cost of E_1 and the cost of E_2 on the potential of E_1. The complexity translation is then a call-by-value monadic translation from the source language into the writer monad in the complexity language, where source expressions that cost a step have the "effect" of incrementing the cost component, using the monad operation

$$\mathsf{incr}(E : \mathbf{C}) : \mathbf{C} \times \mathsf{unit} := (E, \langle\,\rangle)$$

We write this translation out explicitly in Figure 3. When E is a complexity, we write E_c and E_p for $\pi_0\, E$ and $\pi_1\, E$ respectively (for "cost" and "potential"). We will often need to "add cost" to a complexity; when E_1 is a cost and E_2 a complexity, we write $E_1 +_c E_2$ for the complexity $(E_1 + (E_2)_c, (E_2)_p)$ (in monadic notation, $\mathsf{incr}(E_1) \gg E_2$). The type translation is extended pointwise to contexts, so $x : \tau \in \gamma$ iff $x : \langle\!\langle \tau \rangle\!\rangle \in \langle\!\langle \gamma \rangle\!\rangle$—the translation is call-by-value, so variables range over potentials, not complexities. For example, $\|x\| = (0, x)$, where the x on the left is a source variable and the x on the right is a potential variable. Likewise we assume that for every datatype δ in the source signature, we have a corresponding datatype δ declared in the complexity language.

We note some basic facts about the translation: the type translation commutes with the application of a strictly positive functor, which is used to show that the translation preserves types.

LEMMA 5 (Compositionality).

- $\|\phi[\tau]\| = \|\phi\|[\langle\!\langle \tau \rangle\!\rangle]$
- $\langle\!\langle \phi[\tau] \rangle\!\rangle = \langle\!\langle \phi \rangle\!\rangle[\langle\!\langle \tau \rangle\!\rangle]$

THEOREM 6. *If* $\gamma \vdash_\psi e : \tau$, *then* $\|\gamma\| \vdash_{\|\psi\|} \|e\| : \|\tau\|$.

4. A Size-Based Complexity Semantics

In the above translation, the potential of a value has just as much information as that value itself. Next, we investigate how to abstract values to sizes, such as replacing a list by its length. In this section, we make this replacement by defining a size-based denotational semantics of the complexity language.

We need to be able to treat potentials of inductively-defined data in two different ways. On the one hand, potentials must reflect intuitions about sizes. To that end, we will insist that potentials be partial orders. On the other hand, to interpret rec expressions, we must be able to distinguish the datatype constructor that a potential represents. In other words, we need the potentials to also be (something like) inductive data types. We will have our cake and eat it too using an approach similar to the work on views (Wadler 1987). As hinted above, we interpret each datatype Δ in the complexity language as a partial order $[\![\Delta]\!]$. But we will also make use of the sum type $D^\Delta = [\![\Phi_{C_0}[\Delta]]\!] + \cdots + [\![\Phi_{C_{n-1}}[\Delta]]\!]$ (representing the unfolding of the datatype) and a function $size_\Delta : D^\Delta \to [\![\Delta]\!]$ (which represents the size of a constructor, in terms of the size of the argument to the constructor). When $\Phi_{C_i} = t$ (i.e. the argument to the constructor is a single recursive occurrence of the datatype), $size(inj_i\, x)$ is intended to represent an upper bound on the size of the values of the form $C\, v$, where v is a value of size at most x. To define the semantics of $\mathsf{rec}^\Delta(y, \overline{C \mapsto x.E_C})$, we consider all values $z \in D^\Delta$ such that $size_\Delta(z) \leq y$. We can distinguish between such values to (recursively) compute the possible values of the form $E_C[\ldots/x]$, and then take a maximum over all such values.

For example, for the inductive definitions of `nat` and `list` (where the list elements have type `nat`), suppose we want to construe the size of a `list` to be the number of all `nat` and `list` constructors. We implement this in the complexity semantics as

$$\begin{array}{rcl} [\![\mathsf{nat}]\!] & = & \mathbf{Z}^+ \\ D^{\mathsf{nat}} & = & \{*\} + [\![\mathsf{nat}]\!] \\ size_{\mathsf{nat}}(*) & = & 1 \\ size_{\mathsf{nat}}(m) & = & 1 + m \end{array}$$

$$\begin{array}{rcl} [\![\mathsf{list}]\!] & = & \mathbf{Z}^+ \\ D^{\mathsf{list}} & = & \{*\} + ([\![\mathsf{nat}]\!] \times [\![\mathsf{list}]\!]) \\ size_{\mathsf{list}}(*) & = & 1 \\ size_{\mathsf{list}}((m, n)) & = & 1 + m + n \end{array}$$

where $\mathbf{Z}^+$ is the non-negative integers.

$$
\begin{array}{rcl}
\|\tau\| & = & \mathbf{C} \times \langle\!\langle \tau \rangle\!\rangle \\
\langle\!\langle \mathtt{unit} \rangle\!\rangle & = & \mathsf{unit} \\
\langle\!\langle \sigma \times \tau \rangle\!\rangle & = & \langle\!\langle \sigma \rangle\!\rangle \times \langle\!\langle \tau \rangle\!\rangle \\
\langle\!\langle \sigma \to \tau \rangle\!\rangle & = & \langle\!\langle \sigma \rangle\!\rangle \to \|\tau\| \\
\langle\!\langle \mathtt{susp}\, \tau \rangle\!\rangle & = & \|\tau\| \\
\langle\!\langle \delta \rangle\!\rangle & = & \delta
\end{array}
$$

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

$\langle\!\langle \psi \rangle\!\rangle$ has, for each datatype δ in ψ
$\mathsf{datatype}\ \delta = C_0^\delta\ \mathsf{of}\ \langle\!\langle \phi_{C_0} \rangle\!\rangle[\delta], \ldots, C_{C_{n-1}}^\delta\ \mathsf{of}\ \langle\!\langle \phi_{n-1} \rangle\!\rangle[\delta]$

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

Figure 3: Translation from source types and expressions to complexity types and expressions. Recall that $\|e\|_c = \pi_0\|e\|$ and $\|e\|_p = \pi_1\|e\|$.

We define the size-based complexity semantics as follows. The base cases for an inductive definition of $(S^T, \leq_T)$ for every complexity type T consist of well-founded partial orders $(S^\Delta, \leq_\Delta)$ for every datatype Δ in the signature, such that $\leq_\Delta$ is closed under arbitrary maximums (see below for a discussion). We define $\mathbf{N}^\infty = \mathbf{N} \cup \{\infty\}$, where $\mathbf{N}$ is the natural numbers with the usual order and addition. We extend the order and addition to ∞ by $n \leq_{\mathbf{N}^\infty} \infty$ and $n + \infty = \infty + n = \infty + \infty = \infty$ for all $n \in \mathbf{N}$. For products and functions we define $S^{\mathsf{unit}} = \{*\}$ and $S^{T_0 \times T_1} = S^{T_0} \times S^{T_1}$ and $S^{T_0 \to T_1} = (S^{T_1})^{S^{T_0}}$, with the trivial, componentwise, and pointwise partial orders, respectively. Complexity types are interpreted into this type structure by setting $[\![\mathbf{C}]\!] = \mathbf{N}^\infty$ and $[\![T]\!] = S^T$ for each complexity type T.

Stating the conditions on programmer-defined size functions requires some auxiliary notions. For $\mathsf{datatype}\ \Delta = \overline{C\ \mathsf{of}\ \Phi_C}$, set $D^\Delta = [\![\Phi_{C_0}[\Delta]]\!] + \cdots + [\![\Phi_{C_{n-1}}[\Delta]]\!]$, writing $inj_i : [\![\Phi_{C_i}[\Delta]]\!] \to D^\Delta$ for the i^{th} injection. Next, we define a function sz^Φ with domain $[\![\Phi[\Delta]]\!]$ (the semantic analogue of the argument type of a datatype constructor). $sz^\Phi(a)$ is intended to be the maximum of the values of type $[\![\Delta]\!]$ from which a is built using pairing and function application. We want to define sz^Φ by induction on Φ, computing the maximum at each step. To ignore values not of type $[\![\Delta]\!]$ we assume an element $\bot \notin S^\Delta$ that serves as an identity for $\vee$; that is, we order $S^\Delta \cup \{\bot\}$ so that $\bot < a$ for all $a \in S^\Delta$. We define $sz^\Phi : [\![\Phi[\Delta]]\!] \to S^\Delta \cup \{\bot\}$ by induction on Φ as follows:

$$
\begin{array}{l}
sz^t(a) = a \\
sz^T(a) = \bot \\
sz^{\Phi_0 \times \Phi_1}(a) = sz^{\Phi_0}(a) \vee sz^{\Phi_1}(a) \\
sz^{T \to \Phi}(f) = \bigvee_{a \in [\![T]\!]} sz^\Phi(f(a))
\end{array}
$$

The key input to the size-based semantics is programmer-supplied size functions $size_\Delta : D^\Delta \to S^\Delta$ such that

$$sz^{\Phi_{C_i}}(a) <_{S^\Delta \cup \{\bot\}} (size_\Delta \circ inj_i)(a)$$

$size_\Delta$ represents the programmer's notion of size for inductively-defined values. The only condition, which is used to interpret the recursor, is that the size of a value is strictly greater than the size of any of its substructures of the same type. For example, this condition permits interpreting the size of a list as its length or its total number of constructors, and the size of a tree as its number of nodes or its height. Non-examples include defining the size of a list of natural numbers to be the number of successor constructors, and defining the size of all natural numbers to be a constant (though see Section 6.5 for a discussion of this latter possibility).

The interpretation of most terms is standard except for that of constructors and rec, which are given in Figure 4. We write map^{Φ, T_0, T_1} for semantic functions that mirror the definition of map, and we overload the notation C_i to stand for $inj_i : [\![\Phi_{C_i}[\delta]]\!] \to D^\delta$. The implementation of the recursors requires a bit of explanation, and is motivated by the goal to have $\|e\|$ bound the cost and potential of e. We expect that $[\![\|\mathtt{rec}^\delta(e, \overline{C \mapsto x.e_C})\|]\!]$, which depends on $[\![\mathsf{rec}^\delta(\|e\|_p, \overline{C \mapsto x.\|e_C\|})]\!]$, should branch on $[\![\|e\|_p]\!]$, evaluating to the appropriate $[\![\|e_C\|]\!]$. However, $[\![\|e\|_p]\!]$ will be a semantic value of type S^δ, whereas to branch, we need a semantic value of type D^δ. Furthermore, $[\![\|e\|_p]\!]$ is an *upper bound* on the size of e, so $[\![\|e\|_p]\!]$ does not tell us the precise form of e, and so we cannot use $[\![\|e\|_p]\!]$ to predict which branch the evaluation of the source rec expression will follow. We solve these problems by introducing a semantic *case* function, and define the denotation of rec expressions by taking a maximum over the branches for all semantic values that are bounded by the upper bound $[\![\|e\|_p]\!]$. This is the source of the requirement that base-type potentials be closed under arbitrary maximums. Although this requirement seems rather strong, in most examples it seems easy to satisfy. In particular, we think of most datatype potentials (sizes) as being natural numbers, and so we satisfy the condition by interpreting them by $\mathbf{N}^\infty$.

The restriction on $size_\Delta$ ensures that the recursion used to interpret rec expressions descends along a well-founded partial order, and hence is well-defined. The maximum may end up being a maximum over all possible values, but this simply indicates that our interpretation fails to give us precise information.

We illustrate this semantics on some examples. In order to ease the notation, we will occasionally write syntactic expressions for the corresponding semantic values (in effect, dropping $[\![\cdot]\!]$). We also write the *case* function as a branch on constructors; for example, we write $case(t, \mathsf{Emp} \mapsto x.\langle 1, 1\rangle \mid \mathsf{Node} \mapsto \langle y, t_0, t_1\rangle.e)$ for $case(t, \lambda x.\langle 1, 1\rangle, \lambda\langle y, t_0, t_1\rangle.e)$.

$$case^\delta : D^\delta \times \prod_C (S^{[\![\Phi_C[\delta]]\!]} \to S^\tau) \to S^\tau \qquad case(Cx, (\dots, f_C, \dots)) = f_C(x)$$

$$[\![Ce]\!]\xi = size(C([\![e]\!]\xi))$$
$$[\![\mathsf{rec}^\delta(E^\delta, \overline{C \mapsto x^{\phi_C[\delta\times\tau]}.E_C^\tau})]\!]\xi = \bigvee_{size\ z \le [\![E]\!]\xi} case(z, (\dots, f_C, \dots))$$
$$f_C(x) = [\![E_C]\!]\xi\{x \mapsto [\![\mathsf{map}^{\Phi_C}(w.\langle w, \mathsf{rec}(w, \overline{C \mapsto x.E_C})\rangle]\!]\xi, x)\}$$
$$= [\![E_C]\!]\xi\{x \mapsto map^{\Phi_C}(\lambda\!\!\lambda a.(a, [\![\mathsf{rec}(w, \overline{C \mapsto x.E_C})]\!]\xi\{w \mapsto a\}), x)\}$$

Figure 4: The interpretation of rec in the size-based semantics for the complexity language.

4.1 Booleans and Conditionals

In the source language we define booleans and their `case` construct:

$$\texttt{datatype bool} = \texttt{True of unit} \mid \texttt{False of unit}$$
$$\texttt{case}(e^{\texttt{bool}}, e_0^\tau, e_1^\tau) = \texttt{rec}(e, \texttt{True} \mapsto e_0 \mid \texttt{False} \mapsto e_1)$$

(recall our convention on writing e_C for $x.e_C$ when $\phi_C = \texttt{unit}$). In the semantics of the complexity language, we interpret bool as a one-element set $\{1\}$, so `True` and `False` are indistinguishable by "size." Our interpretation yields

$$\begin{aligned}
&[\![\|\texttt{case}(e, e_0, e_1)\|]\!] \\
&= \|e\|_c +_c \\
&\quad \bigvee_{size\ b \le \|e\|_p} case(b, \mathsf{True} \mapsto 1 +_c \|e_0\| \mid \mathsf{False} \mapsto 1 +_c \|e_1\|) \\
&= \|e\|_c +_c (case(\mathsf{True}, \mathsf{True} \mapsto 1 +_c \|e_0\| \mid \mathsf{False} \mapsto 1 +_c \|e_1\|) \\
&\quad \vee case(\mathsf{False}, \mathsf{True} \mapsto 1 +_c \|e_0\| \mid \mathsf{False} \mapsto 1 +_c \|e_1\|)) \\
&= (1 + \|e\|_c) +_c (\|e_0\| \vee \|e_1\|).
\end{aligned}$$

In other words, if we cannot distinguish between `True` and `False` by size, then the interpretation of a conditional is just the maximum of its branches (with the additional cost of evaluating the test). This is precisely the interpretation used by Danner et al. (2013).

4.2 Tree Membership

Next we consider an example that shows that the "big" maximum used to interpret the recursor can typically be simplified to the recurrence that one expects to see. We analyze the cost of checking membership in an `int`-labeled tree. We write e_0 `orelse` e_1 as an abbreviation for `case` $(e_0, \texttt{True} \mapsto \texttt{True} \mid \texttt{False} \mapsto e_1)$.

```
datatype tree = Emp of unit | Node of int × tree × tree
mem(t, x) = rec(t,
  Emp ↦ False
  Node ↦ ⟨y, ⟨t0, r0⟩, ⟨t1, r1⟩⟩.
    y = x orelse (force r0 orelse force r1))
```

For this example, we treat `int` (in the source and complexity languages) as a datatype with 2^{32} constructors where the equality test $x = y$ is implemented by a rather large case analysis. Let us define the size of a tree to be the number of nodes:

$$\begin{aligned}
[\![\mathsf{tree}]\!] &= \mathbf{N}^\infty \\
D^{\mathsf{tree}} &= \{*\} + \{1\} \times \mathbf{N}^\infty \times \mathbf{N}^\infty \\
size_{\mathsf{tree}}(\mathsf{Emp}) &= 0 \\
size_{\mathsf{tree}}(\mathsf{Node}(1, n_0, n_1)) &= 1 + n_0 + n_1
\end{aligned}$$

We would like to get the following recurrence for the cost of the `rec` expression when t has size n:

$$T(0) = 1 \qquad T(n) = \bigvee_{n_0+n_1+1=n} 6 + T(n_0) + T(n_1)$$

($x = y$ requires an application and two `case` evaluations; each `orelse` evaluation costs 1; and we charge for the `rec` reduction).

Working through the interpretation yields $[\![\|\texttt{mem}(t, x)\|]\!]_c = \|t\|_c + g(\|t\|_p) + 1$ where

$$\begin{aligned}
g(n) = [\![&\mathsf{rec}(z, \mathsf{Emp} \mapsto 1 \\
&\mathsf{Node} \mapsto \langle y, \langle t_0, r_0\rangle, \langle t_1, r_1\rangle\rangle.6 + (r_0)_c + (r_1)_c \\
&]\!]\{z \mapsto n\}.
\end{aligned}$$

We can calculate that $g(0) = 1$, and for $n > 0$:

$$\begin{aligned}
g(n) &= \bigvee_{\mathsf{size}\ t \le n} case(t, \\
&\qquad \mathsf{Emp} \mapsto 1 \\
&\qquad \mathsf{Node} \mapsto \langle y, n_0, n_1\rangle.6 + g(n_0) + g(n_1) \\
&= g(n-1) \vee \bigvee_{\mathsf{size}\ t = n} case(t, \dots) \\
&= g(n-1) \vee \bigvee_{1+n_0+n_1=n} case(\mathsf{Node}(1, n_0, n_1), \dots) \\
&= g(n-1) \vee \bigvee_{1+n_0+n_1=n} (6 + g(n_0) + g(n_1))
\end{aligned}$$

We now notice that when we take $n_0 = 0$ and $n_1 = n - 1$ we have

$$6 + g(n_0) + g(n_1) = 6 + g(0) + g(n-1) \ge g(n-1)$$

and hence

$$\begin{aligned}
g(n) &= g(n-1) \vee \bigvee_{1+n_0+n_1} (6 + g(n_0) + g(n_1)) \\
&= \bigvee_{1+n_0+n_1} (6 + g(n_0) + g(n_1))
\end{aligned}$$

which is precisely the recurrence we would expect.

4.3 Tree Map

Next, we consider an example that illustrates reasoning about higher-order functions and the benefits of choosing an appropriate notion of size. We analyze the cost of the map function for `nat`-labeled binary trees:

$$\begin{aligned}
\texttt{treemap}(f, t) = \texttt{rec}(t,& \\
&\mathsf{Emp} \mapsto \mathsf{Emp} \\
&\mathsf{Node} \mapsto \langle y, \langle t_0, r_0\rangle, \langle t_1, r_1\rangle\rangle. \\
&\qquad \mathsf{Node}(f(y), \texttt{force}\, r_0, \texttt{force}\, r_1).
\end{aligned}$$

Suppose the cost of evaluating f is monotone with respect to the size of its argument, where we define the size of a natural number n to be $1 + n$ (to count the zero constructor). The cost of evaluating $\texttt{treemap}(f, t)$ should be bounded by $1 + n \cdot (1 + f(s)_c)$, where n is the number of nodes in t, s is the maximum size of all labels in t,

and we write $f(s)_c$ for the cost of evaluating f on a natural number of size s (the map runs f on an input of size at most s for each of the n nodes, and takes an additional n steps to traverse the tree).

We take $[\![\mathsf{tree}]\!] = \mathbf{N}^\infty \times \mathbf{N}^\infty$, where we think of the pair (n, s) as (number of nodes, maximum size of label), and use the mutual ordering on pairs ($(n, s) < (n', s')$ iff $n \leq n'$ and $s < s'$ or $n < n'$ and $s \leq s'$). The size function is defined as follows:

$$size(\mathsf{Emp}) = (0, 0)$$
$$size(\mathsf{Node}(n, (n_0, s_0), (n_1, s_1))) = (1 + n_0 + n_1, \max\{n, s_0, s_1\}).$$

Let us write $g(m, s) = [\![\|\mathtt{rec}(\ldots)\|]\!]\{t \mapsto (m, s)\}$, so that $([\![\|\mathtt{treemap}\|]\!](f, (m, s)))_c = g(m, s) + 1$. We now show that $g(m, s) \leq m(1 + f(s)_c)$ by induction:

$$\begin{aligned}
&g(m,s)\\
&= \bigvee_{size\, z \leq (m,s)} case(z,\\
&\qquad \mathsf{Emp} \mapsto 1\\
&\qquad \mathsf{Node} \mapsto \langle n, (n_0, s_0), (n_1, s_0)\rangle.\\
&\qquad\qquad \big(1 + (f(n))_c + (g(n_0, s_0))_c + (g(n_1, s_1))_c\big)\\
&= 1 \vee\\
&\qquad \bigvee_{\substack{1+n_0+n_1 \leq m\\ \max\{n,s_0,s_1\} \leq s}} \big(1 + f(n)_c + (g(n_0, s_0))_c + (g(n_1, s_1))_c\big)\\
&\leq \bigvee_{\substack{1+n_0+n_1 \leq m\\ \max\{n,s_0,s_1\} \leq s}} (1 + f(n)_c +\\
&\qquad\qquad n_0 \cdot (1 + f(s_0)_c) + n_1 \cdot (1 + f(s_1)_c))\\
&\leq \bigvee_{\substack{1+n_0+n_1 \leq m\\ \max\{n,s_0,s_1\} \leq s}} (1 + n_0 + n_1)(1 + f(\max\{n, s_0, s_1\})_c)\\
&\leq m \cdot (1 + f(s)_c).
\end{aligned}$$

4.4 The Bounding Theorem for the Size-Based Semantics

The most basic correctness criterion for our technique is that a closed source program's operational cost is bounded by the cost component of the denotation of its complexity translation. However, to know that extracted *recurrences* are correct, it is not enough to consider closed programs; we also need to know that the potential of a function bounds that function's operational cost on all arguments, and so on at higher type. Thus, we use a logical relation. We first show a simplified case of the logical relation, where for this subsection only we do not allow datatype constructors to take functions as arguments (i.e., drop the $\tau \to \phi$ clause from constructor argument types ϕ). In Section 5, we consider the general case, which requires some non-trivial technical additions to the main definition.

DEFINITION 2.

1. Let e be a closed source language expression and a a semantic value. We write $e \sqsubseteq_\tau a$ to mean: if $e \downarrow^n v$, then
   (a) $n \leq a_c$; and
   (b) $v \sqsubseteq^{\mathrm{val}}_\tau a_p$.
2. Let v be a source language value and a a semantic value. We define $v \sqsubseteq^{\mathrm{val}}_\tau a$ by:
   (a) $() \sqsubseteq^{\mathrm{val}}_{\mathtt{unit}} 1$.
   (b) $\langle v_0, v_1\rangle \sqsubseteq^{\mathrm{val}}_{\tau_0 \times \tau_1} \langle a_0, a_1\rangle$ if $v_i \sqsubseteq^{\mathrm{val}}_{\tau_i} a_i$ for $i = 0, 1$.
   (c) $\mathtt{delay}(e) \sqsubseteq^{\mathrm{val}}_{\mathtt{susp}\ \tau} a$ if $e \sqsubseteq_\tau a$.
   (d) $C(v) \sqsubseteq^{\mathrm{val}}_\delta a$ if there is a' such that $v \sqsubseteq^{\mathrm{val}}_{\phi_C[\delta]} a'$ and $size(C(a')) \leq a$.[1]
   (e) $\lambda x.e \sqsubseteq^{\mathrm{val}}_{\sigma \to \tau} a$ if whenever $v \sqsubseteq^{\mathrm{val}}_\sigma a'$, $e[v/x] \sqsubseteq_\tau a(a')$.

THEOREM 7 (Bounding theorem). *If $e : \tau$ in the source language, then $e \sqsubseteq_\tau [\![\|e\|]\!]$.*

Rather than proving this bounding theorem directly, in Section 5 we identify syntactic constraints on the complexity language which allow the proof to be carried through (Theorem 12). Because the size-based semantics satisfies these syntactic constraints (see Section 6.1), we can prove that the logical relation defined in Section 5 implies the one defined above, giving Theorem 7 as a corollary.

5. The Syntactic Bounding Theorem

Rather than proving the bounding theorem for a particular model, such as the one from the previous section, we use a syntactic judgement $\Gamma \vdash E_0 \leq_T E_1$ to axiomatize the properties that are necessary to prove the theorem. The rules are in Figure 5; we omit typing premises from the figure, but formally each rule has sufficient premises to make the two terms have the indicated type. The first two rules state reflexivity and transitivity. The next rule (congruence) says that term contexts of a certain form (in the sequel, *congruence contexts*) are monotonic. The next three rules state the monoid laws for $\mathbf{C}$; we write $E_0 = E_1$ to abbreviate two rules $E_0 \leq E_1$ and $E_1 \leq E_0$. The final three rules (which we call "step rules") say that a β-redex is bigger than or equal to its reduct. The first five congruence contexts are the standard head elimination contexts used in logical relations arguments (principal arguments of elimination forms) and the next two say that $+$ is monotone.

These preorder rules are sufficient to prove the bounding theorem, and permit a variety of interpretations and extensions. If we impose no further rules, then $E_0 \leq E_1$ is basically weak head reduction from E_1 to E_0 (plus the monoid laws for $\mathbf{C}$). We can also add rules that identify elements of datatypes, in order to make those elements behave like sizes. For example, for lists of ints, we can say

$$\frac{}{E \leq \mathsf{Cons}(\_, E)} \qquad \frac{}{\mathsf{Cons}(E_1, E) \leq \mathsf{Cons}(E_2, E)}$$

and extend the congruence contexts with $\mathsf{Cons}(x, \mathcal{C})$. Then the second rule equates any two lists with the same number of elements, quotienting them to natural numbers, and the first rule orders these natural numbers by the usual less-than. Thus, considered up to $\leq$, lists are lengths.

Combining these rules with the ones used to prove the bounding theorem, the recursor for lists behaves like a monotonization of the original recursion (like the $\bigvee$ in the size-based complexity semantics). For example, for any specific list value $\mathsf{Cons}(x, xs)$, by the usual step rule, we have

$$E_1[(x, xs, \mathsf{rec}(xs, \mathsf{Nil} \mapsto E_0, \mathsf{Cons} \mapsto p.E_1))/p] \leq \mathsf{rec}(\mathsf{Cons}(x, xs), \mathsf{Nil} \mapsto E_0, \mathsf{Cons} \mapsto p.E_1)$$

But we can derive $\mathsf{Nil} \leq \mathsf{Cons}(x, xs)$, so we also have

$$\begin{array}{ll}
\mathsf{rec}(\mathsf{Nil}, \ldots) \leq \mathsf{rec}(\mathsf{Cons}(x, xs), \ldots) & \text{by congruence}\\
E_0 \leq \mathsf{rec}(\mathsf{Nil}, \mathsf{Nil} \mapsto E_0, \mathsf{Cons} \mapsto p.E_1) & \text{by the step rule}\\
E_0 \leq \mathsf{rec}(\mathsf{Cons}(x, xs), \mathsf{Nil} \mapsto E_0, \mathsf{Cons} \mapsto p.E_1) & \text{by transitivity}
\end{array}$$

and similarly for non-empty lists that are $\leq \mathsf{Cons}(x, xs)$. Thus, when we quotient lists to their lengths, the congruence and step

[1] Our restriction on the form of ϕ_C allows us to conclude that this definition is well-founded, even though the type gets bigger in clause (2d), because we can treat the definition of $\sqsubseteq^{\mathrm{val}}_\delta$ as an inner induction on the values. Allowing datatype constructors to take function arguments complicates the situation, and in Section 5 we must define a more general relation.

rules for rec (used to prove the bounding theorem) imply that the recursor is bigger than all of the branches for all smaller lists.

In Section 4, we used reasoning in the size-based semantics to massage the recurrence extracted from a program into a recognizable and solvable form. In future work, we plan to investigate how to do this massaging within the syntax of complexity language, using the rules we have just discussed and others. For example, while a recursion bounds what it steps to on all smaller values, we do not yet have a rule stating that it is a least upper bound. Here, we lay a foundation for this by proving the bounding theorem for the small set of rules in Figure 5.

5.1 The Bounding Relation

First, we extend Definition 2 to arbitrary datatypes. Fix a signature ψ. We will mutually define the following relations:

1. $e \sqsubseteq_\tau E$, where $\emptyset \vdash_\psi e : \tau$ and $\emptyset \vdash_{\|\psi\|} E : \|\tau\|$.
2. $v \sqsubseteq^{\mathrm{val}}_\tau E$, where $\emptyset \vdash_\psi v : \tau$ and $\emptyset \vdash_{\|\psi\|} E : \langle\!\langle \tau \rangle\!\rangle$.
3. $v \sqsubseteq^{\mathrm{val}}_{\phi,R} E$, where $\emptyset \vdash_\psi v : \phi[\delta]$ and $\emptyset \vdash_{\|\psi\|} E : \langle\!\langle \phi \rangle\!\rangle[\delta]$.
4. $e \sqsubseteq_{\phi,R} E$, where $\emptyset \vdash_\psi e : \phi[\delta]$ and $\emptyset \vdash_{\|\psi\|} E : \|\phi\|[\delta]$

In (3) and (4), $R(\emptyset \vdash_\psi v : \delta, \emptyset \vdash_{\|\psi\|} E : \delta)$, is any relation; these parts interpret strictly positive functors as relation transformers.

The definition is by induction on τ and ϕ. For datatypes, the signature well-formedness relation ψ sig ensures that datatypes are ordered, where later ones can refer to earlier ones, but not vice versa. Therefore, we could "inline" all datatype declarations: rather than naming datatypes, we could replace each datatype name δ by an inductive type $\mu[C \texttt{ of } \phi]$. The logical relation is defined using the subterm ordering for this "inlined" syntax. In addition to the usual subterm ordering for types τ and functors ϕ, we have that datatypes that occur earlier in ψ are smaller than later ones, and if $C : (\phi \to \delta) \in \psi$, then ϕ is smaller than δ.

DEFINITION 3.

1. We write $e \sqsubseteq_\tau E$ to mean: if $e \downarrow^n v$, then
   - $n \leq E_c$; and
   - $v \sqsubseteq^{\mathrm{val}}_\tau E_p$.
2. We write $v \sqsubseteq^{\mathrm{val}}_\tau E$ to mean:
   - $v \sqsubseteq^{\mathrm{val}}_{\mathtt{unit}} E$ is always true.
   - $\langle v_1, v_2 \rangle \sqsubseteq^{\mathrm{val}}_{\tau_1 \times \tau_2} E$ iff $v_1 \sqsubseteq^{\mathrm{val}}_{\tau_1} \pi_0\, E$ and $v_2 \sqsubseteq^{\mathrm{val}}_{\tau_2} \pi_1\, E$.
   - $\mathtt{delay}(e) \sqsubseteq^{\mathrm{val}}_{\mathtt{susp}\,\tau} E$ iff $e \sqsubseteq_\tau E$.
   - $v \sqsubseteq^{\mathrm{val}}_\delta E$ is inductively defined by
   $$\frac{C : (\phi \to \delta) \in \psi \quad v \sqsubseteq^{\mathrm{val}}_{\phi, -\sqsubseteq^{\mathrm{val}}_\delta -} E' \quad C\, E' \leq_\delta E}{C\, v \sqsubseteq^{\mathrm{val}}_\delta E}$$
   - $\lambda x.e \sqsubseteq^{\mathrm{val}}_{\tau_1 \to \tau_2} E$ iff (for all v_1 and E_1, if $v_1 \sqsubseteq^{\mathrm{val}}_{\tau_1} E_1$ then $e[v_1/x] \sqsubseteq_{\tau_2} E\, E_1$).
3. We write $v \sqsubseteq^{\mathrm{val}}_{\phi,R} E_p$ to mean:
   - $v \sqsubseteq^{\mathrm{val}}_{t,R} E$ if $R(v, E)$.
   - $v \sqsubseteq^{\mathrm{val}}_{\tau,R} E$ if $v \sqsubseteq^{\mathrm{val}}_\tau E$ (t not free in τ).
   - $\langle v, v' \rangle \sqsubseteq^{\mathrm{val}}_{\phi \times \phi',R} E$ if $v \sqsubseteq^{\mathrm{val}}_{\phi,R} \pi_0\, E$ and $v' \sqsubseteq^{\mathrm{val}}_{\phi',R} \pi_1\, E$.
   - $\lambda x.e_1 \sqsubseteq^{\mathrm{val}}_{\tau \to \phi,R} E_1$ if for all v and E, if $v \sqsubseteq^{\mathrm{val}}_\tau E$, then $e_1[v/x] \sqsubseteq_{\phi,R} (E_1\ E)$.
4. We write $e \sqsubseteq_{\phi,R} E$ to mean: if $e \downarrow^n v$, then
   - $n \leq E_c$; and
   - $v \sqsubseteq^{\mathrm{val}}_{\phi,R} E_p$.

The inner inductive definition of $v \sqsubseteq^{\mathrm{val}}_\delta E$ makes sense because R occurs strictly positively in $- \sqsubseteq^{\mathrm{val}}_{\phi,R} -$, and because (by signature formation) δ cannot occur in ϕ, so $- \sqsubseteq^{\mathrm{val}}_\delta -$ does not occur elsewhere in $- \sqsubseteq^{\mathrm{val}}_{\phi,R} -$. The relation on open terms considers all closed instances:

5. For a source substitution $\theta : \gamma$ and complexity substitution $\Theta : \Gamma$, we write $\theta \sqsubseteq^{sub}_\gamma \Theta$ to mean that for all $(x : \tau) \in \gamma$, $\theta(x) \sqsubseteq^{\mathrm{val}}_\tau \Theta(x)$.
6. For $\gamma \vdash e : \tau$ and $\Gamma \vdash E : \|\tau\|$, we write $e \sqsubseteq_\tau E$ to mean that for all $\theta : \gamma$ and $\Theta : \Gamma$, if $\theta \sqsubseteq^{sub}_\gamma \Theta$, then $e[\theta] \sqsubseteq_\tau E[\Theta]$.

We write $\mathcal{E} :: \mathcal{J}$ to mean that $\mathcal{E}$ is a derivation of any of the judgements just described. Because the relation for function types is a function between relations, derivations are infinitely-branching trees. A *subderivation* of such an $\mathcal{E}$ is any subtree of $\mathcal{E}$, which includes any application of an $\to$-type judgement. For example, if $\mathcal{E}_1 :: \lambda x.e_1 \sqsubseteq^{\mathrm{val}}_{\tau \to \phi,R} E_1$ and $\mathcal{E} :: v \sqsubseteq^{\mathrm{val}}_\tau E$, then the derivation of $e_1[v/x] \sqsubseteq^{\mathrm{val}}_{\phi,R} E_1\, E$ is a subderivation of $\mathcal{E}_1$.

Next, we establish some basic properties of the relation:

LEMMA 8 (Weakening).

1. *If* $e \sqsubseteq_\tau E$ *and* $E \leq_{\|\tau\|} E'$ *then* $e \sqsubseteq_\tau E'$.
2. *If* $v \sqsubseteq^{\mathrm{val}}_\tau E$ *and* $E \leq_{\langle\!\langle \tau \rangle\!\rangle} E'$ *then* $v \sqsubseteq^{\mathrm{val}}_\tau E'$.

Proof. Both clauses are proved simultaneously by induction on τ, using congruence for $\pi_0\,[\,]$, $\pi_1\,[\,]$ and $[\,]\ E$. See the full paper for details. □

LEMMA 9 (Compositionality).

1. $e \sqsubseteq_{\phi, -\sqsubseteq^{\mathrm{val}}_\tau -} E$ *iff* $e \sqsubseteq_{\phi[\tau]} E$.
2. $v \sqsubseteq^{\mathrm{val}}_{\phi, -\sqsubseteq^{\mathrm{val}}_\tau -} E$ *iff* $v \sqsubseteq^{\mathrm{val}}_{\phi[\tau]} E$.

Proof. (1) follows by post-composing with (2), and (2) follows by induction on ϕ. See the full paper for details. □

5.2 The Fundamental Theorem

First we state two lemmas which say that, when applied to related arguments, source-language map is bounded by complexity-language map, and that source-language rec is bounded by complexity-language rec.

LEMMA 10 (Map). *Suppose:*

1. $x : \tau_0 \vdash v_1 : \tau_1$ *and* $\emptyset \vdash v_0 : \phi[\tau_0]$;
2. $x : \langle\!\langle \tau_0 \rangle\!\rangle \vdash E_1 : \langle\!\langle \tau_1 \rangle\!\rangle$ *and* $\emptyset \vdash E_0 : \langle\!\langle \phi \rangle\!\rangle[\langle\!\langle \tau_0 \rangle\!\rangle]$;
3. $\mathcal{E} :: v_0 \sqsubseteq^{\mathrm{val}}_{\phi, -\sqsubseteq^{\mathrm{val}}_{\tau_0} -} E_0$;
4. *Whenever* $\mathcal{E}'$ *is a subderivation of* $\mathcal{E}$ *such that* $\mathcal{E}' :: v_0' \sqsubseteq^{\mathrm{val}}_{\tau_0} E_0'$, $v_1[v_0'/x] \sqsubseteq^{\mathrm{val}}_{\tau_0} E_1[E_0'/x]$; *and*
5. $\mathtt{map}^\phi(x.v_1, v_0) \downarrow^n v$.

Then $n = 0$ *and* $v \sqsubseteq^{\mathrm{val}}_{\phi[\tau_0]} \mathtt{map}^{\langle\!\langle \phi \rangle\!\rangle}(x.E_1, E_0)$. [2]

Proof. The proof is by induction on ϕ. Lemma 3 shows that $n = 0$. Omitted cases are in the full paper.

CASE: $\phi = \tau \to \phi_0$. Then $v_0 = \lambda y.e_0$ and $\mathcal{E}$ proves that for all $v' \sqsubseteq^{\mathrm{val}}_\tau E'$, $e_0[v'/y] \sqsubseteq_{\phi_0, -\sqsubseteq^{\mathrm{val}}_{\tau_0} -} E_0(E')$. Since $v_0 = \lambda y.e_0$, $v = \lambda y.\, \mathtt{let}(e_0, z.\mathtt{map}^\phi(x.v_1, z))$, so we must show that $\lambda y.\, \mathtt{let}(e_0, z.\mathtt{map}^\phi(x.v_1, z)) \sqsubseteq^{\mathrm{val}}_{\tau \to \phi_0[\tau_0]} \mathtt{map}^{\langle\!\langle \tau \to \phi_0 \rangle\!\rangle}(x.E_1, E_0)$. To do so, suppose $w \sqsubseteq^{\mathrm{val}}_\tau F$; we must show that

$$\mathtt{let}(e_0[w/y], z.\mathtt{map}^\phi(x.v_1, z)) \sqsubseteq_{\phi_0[\tau_0]} \mathtt{map}^{\|\phi_0\|}(x.E_1, E_0(F)). \quad (*)$$

[2] We could have said $\mathtt{map}^\phi(x.v_1, v_0) \sqsubseteq_{\phi[\tau_0]} \langle 0, \mathtt{map}^{\langle\!\langle \phi \rangle\!\rangle}(x.E_1, E_0) \rangle$ but this version of the lemma avoids needing the symmetric copy of the step rule for pairs.

$$\mathcal{C} ::= [\,] \mid \pi_0\,\mathcal{C} \mid \pi_1\,\mathcal{C} \mid \mathcal{C}\;E \mid \mathsf{rec}(\mathcal{C}, \overline{C \mapsto x.E_C}) \mid \mathcal{C} + E \mid E + \mathcal{C}$$

$$\frac{}{\Gamma \vdash E \leq_T E} \quad \frac{\Gamma \vdash E_0 \leq_T E_1 \quad \Gamma \vdash E_1 \leq_T E_2}{\Gamma \vdash E_0 \leq_T E_2} \quad \frac{\Gamma, x : T' \vdash \mathcal{C}[x] : T \quad \Gamma \vdash E_0 \leq_{T'} E_1}{\Gamma \vdash \mathcal{C}[E_0] \leq_T \mathcal{C}[E_1]}\ \text{(congruence)}$$

$$\frac{}{\Gamma \vdash 0 + E =_{\mathbf{C}} E} \quad \frac{}{\Gamma \vdash E + 0 =_{\mathbf{C}} E} \quad \frac{}{\Gamma \vdash (E_0 + E_1) + E_2 =_{\mathbf{C}} E_0 + (E_1 + E_2)}$$

$$\frac{\dfrac{}{\Gamma \vdash E_0[E_1/x] \leq_T (\lambda x.E_0)E_1} \quad \dfrac{}{\Gamma \vdash E_i \leq_{T_i} \pi_i\langle E_0, E_1\rangle} \qquad C : (\Phi \to \Delta) \in \Psi}{\Gamma \vdash E_C[\mathsf{map}^{\Phi}(y.\langle y, \mathsf{rec}(y, \overline{C \mapsto x.E_C})\rangle, E_0)/x] \leq_T \mathsf{rec}^{\Delta}(CE_0, \overline{C \mapsto x.E_C})}$$

Figure 5: Congruence contexts and the preorder judgement

Suppose

$$\frac{e_0[w/y] \downarrow^{n_0} w_0 \quad \mathtt{map}^{\phi_0}(x.v_1, w_0) \downarrow^{n_1} v'}{\mathtt{let}(e_0[w/y], z.\mathtt{map}^{\phi_0}(x.v_1, z)) \downarrow^{n_0+n_1} v'}$$

Since $w \sqsubseteq^{\mathrm{val}}_{\tau} F$, we have that $\mathcal{E}$ derives $e_0[w/y] \sqsubseteq_{\phi_0, -\sqsubseteq^{\mathrm{val}}_{\tau_0} -} E_0(F)$ and hence we have a subderivation $\mathcal{E}_0$ of $\mathcal{E}$ such that $\mathcal{E}_0 :: w_0 \sqsubseteq^{\mathrm{val}}_{\phi_0, -\sqsubseteq^{\mathrm{val}}_{\tau_0} -} (E_0(F))_p$. We now verify that (4) holds for $\mathcal{E}_0$ so that we can apply the induction hypothesis to to $\mathtt{map}^{\phi}(x.v_1, w_0)$. So suppose that $\mathcal{E}'_0$ is a subderivation of $\mathcal{E}_0$ such that $\mathcal{E}'_0 :: w'_0 \sqsubseteq^{\mathrm{val}}_{\tau_0} F'_0$. We need to show that $v_1[w'_0/x] \sqsubseteq^{\mathrm{val}}_{\tau_0} E_1[F'_0/x]$, and to do so it suffices to note that $\mathcal{E}'_0$ is a subderivation of $\mathcal{E}_0$, which in turn is a subderivation of $\mathcal{E}$.

We can now apply the induction hypothesis to conclude that $n_1 = 0$ and so:

$$n_0 + n_1 = n_0 \leq (E_0\,F)_c = (\mathsf{map}^{\|\phi\|}(x.E_1, E_0\,F))_c$$
$$v' \sqsubseteq^{\mathrm{val}}_{\phi[\tau_0]} \mathsf{map}^{\langle\langle\phi\rangle\rangle}(x.E_1, (E_0\,F)_p) = (\mathsf{map}^{\|\phi\|}(x.E_1, E_0\,F))_p.$$

Using β for pairs, these are the two conditions that must be verified to show (*), so this completes the proof. □

LEMMA 11 (Recursor). *Fix a datatype declaration* $\mathtt{datatype}\ \delta = \overline{C\ \mathtt{of}\ \phi}$. *If* $v \sqsubseteq^{\mathrm{val}}_{\delta} E$ *and for all* C, $e_C \sqsubseteq_{\phi_C[\delta \times \mathsf{susp}\,\tau]} E_C$, *then* $\mathtt{rec}(v, \overline{C \mapsto x.e_C}) \sqsubseteq \mathsf{rec}(E, C \mapsto x.1 +_{\mathsf{c}} E_C)$

Proof. By induction on $v \sqsubseteq^{\mathrm{val}}_{\delta} E$. The only case is

$$\frac{C : (\phi \to \delta) \in \psi \quad v' \sqsubseteq^{\mathrm{val}}_{\phi, -\sqsubseteq^{\mathrm{val}}_{\delta} -} E' \quad C\,E' \leq_{\delta} E}{C\,v' \sqsubseteq^{\mathrm{val}}_{\delta} E} \qquad (\dagger)$$

Assume $\mathtt{rec}(C\,v', \overline{C \mapsto x.e_C})$ evaluates. Then by inversion and Lemma 2 it was by

$$\frac{\begin{array}{l} C\,v' \downarrow^{0} C\,v' \\ \mathtt{map}^{\phi}(y.\langle y, \mathtt{delay}(\mathtt{rec}(y, \overline{C \mapsto x.e_C}))\rangle, v') \downarrow^{0} v'' \\ e_{\mathsf{C}}[v''/x] \downarrow^{n_2} v \end{array}}{\mathtt{rec}(C\,v', \overline{C \mapsto x.e_C}) \downarrow^{0+1+n_2} v} \qquad (*)$$

Using the premise that $CE' \leq_{\delta} E$ from (†), β for datatypes, and congruence, we note that

$$\begin{aligned} &\mathsf{rec}(E, C \mapsto x.1 +_{\mathsf{c}} E_C) \\ &\quad \geq \mathsf{rec}(C\,E', C \mapsto x.1 +_{\mathsf{c}} E_C) \\ &\quad \geq 1 +_{\mathsf{c}} E_C[\mathsf{map}^{\langle\langle\phi\rangle\rangle}(y.\langle y, \mathsf{rec}(y, C \mapsto x.1 +_{\mathsf{c}} E_C)\rangle, E')/x] \end{aligned}$$

Let us write E^* for $\mathsf{map}^{\langle\langle\phi\rangle\rangle}(y.\langle y, \mathsf{rec}(y, C \mapsto x.1 +_{\mathsf{c}} E_C)\rangle, E')$. Thus by congruence, transitivity, weakening, and β for pairs, it suffices to show

$$\begin{aligned} 1 + n_2 &\leq 1 + E_C[E^*/x]_c \\ v &\sqsubseteq^{\mathrm{val}} (E_C[E^*/x])_p \end{aligned}$$

By congruence for $+$, for the first goal it suffices to show $n_2 \leq E_C[E^*/x]_c$. Thus, if we can show $e_{\mathsf{C}}[v''/x] \sqsubseteq E_C[E^*/x]$, then applying it to the third evaluation premise of (*) gives the result. We can use our assumption that $e_C \sqsubseteq E_C$, as long as we show $v'' \sqsubseteq^{\mathrm{val}} E^*$. To do so, we use Lemma 10 applied to the second evaluation premise of (*) with

$$\begin{aligned} v_1 &= v' \qquad v = y.\langle y, \mathtt{delay}(\mathtt{rec}(y, \overline{C \mapsto x.e_C}))\rangle \\ E_1 &= E' \qquad E = y.\langle y, \mathsf{rec}(y, C \mapsto x.1 +_{\mathsf{c}} E_C)\rangle \end{aligned}$$

We have $\mathcal{E} :: v' \sqsubseteq^{\mathrm{val}}_{\phi, -\sqsubseteq^{\mathrm{val}}_{\delta} -} E'$ from the second premise of (†). Thus, to finish calling the theorem, we need to show that for all R-position subderivations of $\mathcal{E}$ deriving $v'_1 \sqsubseteq^{\mathrm{val}}_{\delta} E'_1$,

$$\langle v'_1, \mathtt{delay}(\mathtt{rec}(v'_1, \overline{C \mapsto x.e_C}))\rangle \sqsubseteq^{\mathrm{val}}_{\delta \times \mathsf{susp}\,\tau} \langle E'_1, \mathsf{rec}(E'_1, C \mapsto x.1 +_{\mathsf{c}} E_C)\rangle$$

By definition of value bounding at product types, weakening and β for pairs, it suffices to show

$$\begin{aligned} v'_1 &\sqsubseteq^{\mathrm{val}}_{\delta} E'_1 \\ \mathtt{delay}(\mathtt{rec}(v'_1, \overline{C \mapsto x.e_C})) &\sqsubseteq^{\mathrm{val}}_{\mathsf{susp}\,\tau} \mathsf{rec}(E'_1, C \mapsto x.1 +_{\mathsf{c}} E_C) \end{aligned}$$

The former we have, and for the latter by definition it suffices to show

$$\mathtt{rec}(v'_1, \overline{C \mapsto x.e_C}) \sqsubseteq_{\tau} \mathsf{rec}(E'_1, C \mapsto x.1 +_{\mathsf{c}} E_C)$$

Because $v'_1 \sqsubseteq^{\mathrm{val}}_{\delta} E'_1$ is an R-subderivation of $v' \sqsubseteq^{\mathrm{val}}_{\phi, -\sqsubseteq^{\mathrm{val}}_{\delta} -} E'$, and therefore a strict subderivation of $C\,v' \sqsubseteq^{\mathrm{val}}_{\delta} E$, we can use the inductive hypothesis on it, which gives exactly what we needed to show. □

THEOREM 12 (Bounding Theorem). *If* $\gamma \vdash e : \tau$, *then* $e \sqsubseteq_{\tau} \|e\|$.

Proof. By induction on the derivation of $\gamma \vdash e : \tau$. In each case we state the last line of the derivation, taking as given the premises of the typing rules in Figure 1. Omitted cases are in the full paper.

CASE: $\gamma \vdash \mathtt{rec}(e, \overline{C \mapsto x.e_C}) : \tau$. We need to show

$$\mathtt{rec}(e[\theta], \overline{C \mapsto x.e_C[\theta, x/x]}) \sqsubseteq \langle E_c + (E_r)_c, (E_r)_p\rangle$$

where $E = \|e\|[\Theta]$ and $E_r = \mathsf{rec}(E_p, C \mapsto x.(1 +_{\mathsf{c}} \|e_C\|[\Theta, x/x]))$. Suppose

$$\frac{\begin{array}{l} e[\theta] \downarrow^{n_0} Cv_0 \\ \mathtt{map}^{\phi_C}(y.\langle y, \mathtt{delay}(\mathtt{rec}(y, \overline{C \mapsto x.e_C[\theta, x/x]})\rangle), v_0) \downarrow^{0} v_1 \\ e_C[\theta, x/v_1] \downarrow^{n_2} v \end{array}}{\mathtt{rec}(e[\theta], \overline{C \mapsto x.e_C[\theta, x/x]}) \downarrow^{1+n_0+n_2} v}$$

By the induction hypothesis $e[\theta] \sqsubseteq E$, so $n_0 \leq E_c$ and $Cv_0 \sqsubseteq^{\mathrm{val}} E_p$. By Lemma 2 we can derive

$$\frac{\begin{array}{l} Cv_0 \downarrow^{0} Cv_0 \\ \mathtt{map}^{\phi_C}(y.\langle y, \mathtt{delay}(\mathtt{rec}(y, \overline{C \mapsto x.e_C[\theta, x/x]})\rangle), v_0) \downarrow^{0} v_1 \\ e_C[\theta, x/v_1] \downarrow^{n_2} v \end{array}}{\mathtt{rec}(Cv_0, \overline{C \mapsto x.e_C[\theta, x/x]}) \downarrow^{1+n_2} v}$$

So by Lemma 11 we have that $1 + n_2 \leq (E_r)_c$ and $v \sqsubseteq^{\mathrm{val}} (E_r)_p$. Putting these together, we have what we needed to show:

$$1 + n_0 + n_2 \leq E_c + (E_r)_c \qquad v \sqsubseteq^{\mathrm{val}} (E_r)_p$$

CASE: $\gamma \vdash \mathtt{map}^{\phi}(x.v_1, v_0) : \phi[\tau_1]$. Because v_1 is a sub-syntactic-class of e, we can upcast it and apply $\|v_1\|$ to it, producing a complexity expression. We must show that

$$\mathtt{map}^{\phi}(x.v_1[\theta, x/x], v_0[\theta]) \sqsubseteq \langle 0, \mathtt{map}^{\langle\langle\phi\rangle\rangle}(x.\|v_1\|[\Theta, x/x]_p, \|v_0\|[\Theta]_p)\rangle,$$

so suppose $\mathtt{map}^{\phi}(x.v_1[\theta, x/x], v_0[\theta]) \downarrow^{n} v$. By transitivity/weakening with β for pairs, it suffices to show:

$$n \leq 0 \qquad v \sqsubseteq^{\mathrm{val}} \mathsf{map}^{\langle\langle\phi\rangle\rangle}(\|v_1\|[\Theta, x/x]_p, \|v_0\|[\Theta]_p) \qquad (*)$$

We will apply Lemma 10 to $\mathtt{map}^{\phi}(x.v_1[\theta, x/x], v_0[\theta]) \downarrow^{n} v$ with

$$v_0 = v_0[\theta] \qquad v_1 = v_1[\theta, x/x]$$
$$E_0 = \|v_0\|[\Theta]_p \qquad E_1 = \|v_1\|[\Theta, x/x]_p$$

To establish condition (3) we apply the IH to v_0 to conclude that $v_0[\theta] \sqsubseteq_{\phi[\tau_0]} \|v_0\|[\Theta]$. Since $v_0[\theta]$ is a value, by Lemma 2, it evaluates to itself. Therefore $v_0[\theta] \sqsubseteq^{\mathrm{val}}_{\phi[\tau_0]} \|v_0\|[\Theta]_p$ and so by Lemma 9, $v_0[\theta] \sqsubseteq^{\mathrm{val}}_{\phi, - \sqsubseteq^{\mathrm{val}}_{\tau_0} -} \|v_0\|[\Theta]_p$.

To establish condition (4), assume $v_0' \sqsubseteq^{\mathrm{val}}_{\tau_0} E_0'$ (which is an R-subderivation of the above, but we won't use this fact). Using the substitution lemmas we need to show $v_1[\theta, v_0'/x] \sqsubseteq^{\mathrm{val}} \|v_1\|[\Theta, E_0'/x]_p$. Since $\theta, v_0'/x \sqsubseteq^{sub} \Theta, E_0'/x$, the IH on v_1 gives $v_1[\theta, v_0'/x] \sqsubseteq \|v_1\|[\Theta, E_0'/x]$ and since $v_1[\theta, v_0'/x]$ is a value, it evaluates to itself, so $v_1[\theta, v_0'/x] \sqsubseteq^{\mathrm{val}} \|v_1\|[\Theta, E_0'/x]_p$ as we needed to show.

Now we apply Lemma 10 to conclude (*). □

6. Models of the Complexity Language

A model of the complexity language consists of an interpretation of types as preorders, and of terms as maps between elements of those preorders, validating the rules of Figure 5. The congruence contexts $\mathcal{C}$, but not all terms, need to be monotone maps.

6.1 The Size-Based Complexity Semantics

We showed in Section 4 that the size-based semantics interpets the syntax of the complexity language; it is also a model of the preorder rules of Figure 5. Congruence is established by induction on $\mathcal{C}$; we do not need programmer-defined size functions to be monotonic, because there is no congruence context for datatype constructors.

$$\begin{aligned} &[\![\mathsf{rec}(CE_0, \overline{x \mapsto E_C})]\!]\xi \\ &\quad = \bigvee_{size\ z \leq size(C[\![E_0]\!]\xi)} case(z, (\ldots, f_C, \ldots)) \\ &\quad \geq case(C[\![E_0]\!]\xi, (\ldots, f_C, \ldots)) \\ &\quad = [\![E_C]\!]\xi\{x \mapsto [\![\mathsf{map}^{\Phi_C}(w.\langle w, \mathsf{rec}(w, \overline{x \mapsto E_C})\rangle, E_0)]\!]\xi\}. \end{aligned}$$

Verification of the pre-order axioms is straightforward. Therefore, Theorem 7 is a corollary of Theorem 12.

6.2 Infinite-Width Trees

Infinite-width trees can be defined by the declaration

$$\mathtt{datatype}\ \mathtt{tree} = E\ \mathtt{of}\ \mathtt{unit} \mid N\ \mathtt{of}\ \mathtt{int} \times (\mathtt{nat} \to \mathtt{tree})$$

Though every branch in such a tree is of finite length, the height of a tree is in general not a finite natural number. However, the size-based semantics adapts easily to interpret `tree` by a suitably large infinite successor ordinal, and then defining $size(N(x, f)) = \bigvee_{y \in [\![\mathtt{nat}]\!]} f(y) + 1$.

6.3 A Semantics Without Arbitrary Maximums

The language studied in Danner et al. (2013) can be viewed as a specific signature in the present language. Their language has a type of booleans, a type `int` of fixed-size integers, and a type `list` of integer lists. As in Example 4.2, we can treat `int` and `bool` as enumerated datatypes with unit-cost operations. The `list` type is defined as a datatype and its case and fold operators are easily defined using `rec`.

For this specific signature, we can give a semantics of the complexity language in which we interpret list by $\mathbf{N}$ instead of $\mathbf{N}^{\infty}$. Set $[\![\mathsf{Nil}]\!]\xi = 0$ and $[\![\mathsf{Cons}(E_0, E_1)]\!]\xi = [\![E_1]\!]\xi + 1$. Define a semantic primitive recursion operator by $rec(0, a, f) = a$ and $rec(n+1, a, f) = a \vee f(n, rec(n, a, f))$. Finally, set

$$[\![\mathsf{rec}(E)]\!]\xi = rec([\![E]\!]\xi, [\![E_{\mathsf{Nil}}]\!]\xi, \lambda\!\!\lambda n, w.[\![E_{\mathsf{Cons}}]\!]\xi\{x, xs, r \mapsto 1, n, w\}).$$

where $\mathsf{rec}(E) = \mathsf{rec}(E, \mathsf{Nil} \mapsto E_{\mathsf{Nil}}, \mathsf{Cons} \mapsto \langle x, \langle xs, r\rangle\rangle.E_{\mathsf{Cons}})$. Verifying the preorder rules from Figure 5 is straightforward in all cases except the last; we verify the Cons case as follows:

$$\begin{aligned} &[\![\mathsf{rec}(\mathsf{Cons}(E_0, E_1))]\!]\xi \\ &\quad = ([\![E_{\mathsf{Nil}}]\!]\xi\{x \mapsto 1\}) \vee \\ &\qquad ([\![E_{\mathsf{Cons}}]\!]\xi\{x, xs, r \mapsto 1, [\![E_1]\!]\xi, rec([\![E_1]\!]\xi, \ldots)\}) \\ &\quad \geq [\![E_{\mathsf{Cons}}]\!]\xi\{x, xs, r \mapsto 1, [\![E_1]\!]\xi, rec([\![E_1]\!]\xi, \ldots)\} \\ &\quad = [\![E_{\mathsf{Cons}}[E_0, \langle E_1, \mathsf{rec}(E_1)\rangle/x, \langle xs, r\rangle]]\!]\xi \\ &\quad = [\![E_{\mathsf{Cons}}[E_0, \mathsf{map}(y.\langle y, \mathsf{rec}(y)\rangle, \langle E_0, E_1\rangle)/x, xs, r]]\!]\xi. \end{aligned}$$

6.4 Exact Costs

If we wish to reason about exact costs, we can symmetrize the inequalities in Figure 5 into equalities, and add congruence for all contexts, which makes the $E_0 \leq E_1$ judgement into a standard notion of definitional equality. Then we can take the term model in the usual way, interpreting each type as a set of terms quotiented by this definitional equality. In this interpretation $\|e\|_c$ is a recurrence that gives the exact cost of evaluating e, but reasoning about such a recurrence involves reasoning about all of the details of the program.

6.5 Infinite Costs

Next, we consider a size-based model in which we drop the "increasing" requirement on the *size* functions from Section 4. Rather

than requiring a well-founded partial order for each datatype, we require an arbitrary partial order $(S^\tau, \leq_\tau)$ which we also interpret as a flat CPO (we do not require the interpretation of non-datatypes to be CPOs). The interpretation of rec expressions is then in terms of a general fixpoint operator. Define $\infty = \bigvee S^\Delta$ and identify ∞ with the bottom element of the CPO ordering. In this setting it may be that the interpretation of a rec expression does not terminate and hence, by our identification, evaluates to ∞. This turns out to be exactly the right behavior, as we can see in the following example.

Take the standard inductive definition of `nat` and interpret `nat` as some one-element set $\{1\}$ in the complexity languageNow compute the interpretation of the identity function:

$$\begin{aligned}
&[\![\|\texttt{rec}(y, \texttt{Zero} \mapsto \texttt{Zero}, \texttt{Succ} \mapsto x.\texttt{Succ}\, x)\|]\!] \\
&\quad = e(1) \\
&\quad = \bigvee_{size\, z \leq 1} case(z, \mathsf{Zero} \mapsto (0,1) \mid \mathsf{Succ} \mapsto \langle x, r\rangle.1 + e_c(x))
\end{aligned}$$

where

$$e(x) = \mathsf{rec}(x, \mathsf{Zero} \mapsto (0,1) \mid \mathsf{Succ} \mapsto \langle x, r\rangle.(1 + r_c, 1))$$

Since $size(\mathsf{Succ}(1)) = 1 \leq 1$, one of the *case* expressions in the maximum is $e_c(1)$. In other words, we have a nonterminating recursion in computing the complexity. We conclude $[\![\|\texttt{rec}(\ldots)\|_c]\!] = \infty$; in other words, we can draw no useful conclusion about the cost of this expression. What we have done in this example is to declare that we cannot distinguish values of type `nat` by size, and then we attempt to compute the cost of a recursive function on `nat`s in terms of the size of the recursion argument. The bound given by the bounding theorem is correct, just not useful; it does not even tell us that the computation terminates.

7. Related Work

There is a reasonably extensive literature over the last several decades on (semi-)automatically constructing resource bounds from source code. The first work concerns itself with first-order programs. Wegbreit (1975) describes a system for analyzing simple Lisp programs that produces closed forms that bound running time. An interesting aspect of this system is that it is possible to describe probability distributions on the input domain and the generated bounds incorporate this information. Rosendahl (1989) proposes a system based on step-counting functions and abstract interpretation for a first-order subset of Lisp. More recently the COSTA project (see, e.g., Albert et al. (2012)) has focused on automatically computing cost relations for imperative languages (actually, bytecode) and solving them (more on that in the next section). Debray and Lin (1993) develop a system for analyzing logic programs and Navas et al. (2007) extend it to handle user-defined resources.

The Resource Aware ML project (RAML) takes a different approach to the one we have described here, one based on type assignment. Jost et al. (2010) describe a formalism that automatically infers linear resource bounds for higher-order programs, provided that the input program does in fact have a linear resource cost. Hoffmann and Hofmann (2010) and Hoffmann et al. (2012) extend this work to handle polynomial bounds, though for first-order programs only, and Hoffmann and Shao (2015) extend it to parallel programs. RAML uses a source language that is similar to ours, but in which the types are annotated with variables corresponding to resource usage. Type inference in the annotated system comes down to solving a set of constraints among these variables. A very nice feature of this work is that it handles cases in which amortized analysis is typically employed to establish tight bounds, while our approach can only conclude (worst-case) bounds.

Danielsson (2003) uses an annotated monad (similar to $\mathbf{C} \times -$) to track running time in a dependently typed language, where size reasoning can be done via types. He emphasizes reasoning about amortized cost of lazy programs. He relies on explicit annotation of the program, which our complexity translation inserts automatically, and his correctness theorem is for closed programs, whereas we use a logical relation to validate extracted recurrences.

We now turn to work that is closest in spirit to ours, focusing on those aspects related to analysis of higher-order languages. Le Métayer's (1988) ACE system is a two-stage system that first converts FP programs (Backus 1978) to recursive FP programs describing the number of recursive calls of the source program, then attempts to transform the result using various program-transformation techniques to obtain a closed form. Shultis (1985) defines a denotational semantics for a simple higher-order language that models both the value and the cost of an expression. As a part of the cost model, he develops a system of "tolls," which play a role similar to the potentials we define in our work. The tolls and the semantics are not used directly in calculations, but rather as components in a logic for reasoning about them. Sands (1990) puts forward a translation scheme in which programs in a source language are translated into programs in the same language that incorporate cost information; several source languages are discussed, including a higher-order call-by-value language. Each identifier f in the source language is associated to a *cost closure* that incorporates information about the value f takes on its arguments; the cost of applying f to arguments; and arity. Cost closures are intended to address the same issue our higher-type potentials do: recording information about the future cost of a partially-applied function. Van Stone (2003) annotates the operational semantics for a higher-order language with cost information. She then defines a category-theoretic denotational semantics that uses "cost structures" to capture cost information and shows that the latter is sound with respect to the former. Benzinger (2004) annotates NuPRL's call-by-name operational semantics with complexity estimates. The language for the annotations is left somewhat open so as to allow greater flexibility. The analysis of the costs is then completed using a combination of NuPRL's proof generation and Mathematica. In all of these approaches the cost domain incorporates information about values in the source language so as to provide exact costs. Our approach provides a uniform framework that can be more or less precise about the source language values that are represented. While we can implement a version that handles exact costs, we can also implement a version in which we focus just on upper bounds, which we might hope leads to simpler recurrences.

8. Conclusions and Further Work

We have described a denotational complexity analysis for a higher-order language with a general form of inductive datatypes that yields an upper bound on the cost of any well-typed program in terms of the size of the input. The two steps are to translate each source-language program e into a program $\|e\|$ in a complexity language, which makes costs explicit, and then to abstract values to sizes. A consequence of the bounding theorem is that the cost component of $\|e\|$ is an upper bound on the evaluation cost of e. The bounding theorem is purely syntactic and therefore applies in all models of the complexity language. By varying the semantics of the complexity language (and in particular, the notion of size), we can perform analyses at different levels of granularity. We give several different choices for the notion of size, but ultimately this is too important a decision to take out of the hands of the user through automation.

The complexity translation of Section 3 can easily be adapted to other cost models. For example, we could charge different amounts for different steps. Or, we could analyze the work and span of parallel programs by taking $\mathbf{C}$ to be series-parallel cost graphs, something we plan to investigate in future work.

Another direction for future work is to handle different evaluation strategies. Compositionality is a thorny issue when considering call-

by-need evaluation and lazy datatypes, and as noted by Okasaki (1998), it may be that amortized cost is at least as interesting as worst-case cost. Sands (1990), Van Stone (2003), and Danielsson (2003) address laziness in their work, and as we already noted, RAML already performs amortized analyses.

We plan to extend the source language to handle general recursion. Part of the difficulty here is that the bounding relation presupposes termination of the source program (so that the derivation of $e \downarrow^n v$, and hence cost, is well-defined). One approach would be to require the user to supply a termination proof. Or, one could define the operational semantics of the source language co-inductively (as done by, e.g., (Leroy and Grall 2009)), thereby allowing explicitly for non-terminating computations. Another approach is to adapt the partial big-step operational semantics described by Hoffmann et al. (2012). Since our source language supports inductive datatype definitions of the form $\texttt{datatype strm} = \texttt{Cons of unit} \to \texttt{nat} \times \texttt{strm}$, adding general recursion will force us to understand how our complexity semantics plays out in the presence of what are essentially coinductively defined values. One could also hope to prove termination in the source language by first extracting complexity bounds and then proving that these bounds in fact define total functions. Another interesting idea along these lines would be to define a complexity semantics in which the cost domain is two-valued, with one value representing termination and the other non-termination (or maybe more accurately, known termination and not-known-termination); such an approach might be akin to an abstract interpretation based approach for termination analysis.

The programs $\|e\|$ are complex higher-order recurrences that call out for solution techniques. Benzinger (2004) addresses this idea, as do Albert et al. (2011, 2013) of the COSTA project. Another relevant aspect of the COSTA work is that their cost relations use non-determinism; it would be very interesting to see if we could employ a similar approach instead of the maximization operators that we used in our examples. Ultimately we should have a library of tactics for transforming the recurrences produced by the translation function to closed (possibly asymptotic) forms when possible.

References

E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46:161–203, 2011. doi: 10.1007/s10817-010-9174-1.

E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012. doi: 10.1016/j.tcs.2011.07.009.

E. Albert, S. Genaim, and A. N. Masud. On the inference of resource usage upper and lower bounds. *ACM Transactions on Computational Logic*, 14 (3):22:1–22:35, 2013. doi: 10.1145/2499937.2499943.

J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the Association for Computing Machinery*, 21(8):613–641, 1978. doi: 10.1145/359576.359579.

R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318(1-2):79 – 103, 2004. doi: 10.1016/j.tcs.2003.10.022.

N. A. Danielsson. Lightweight semiformal time complexity analysis for purely functional data structures. In A. Aiken and G. Morrisett, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–144. ACM Press, 2003. doi: 10.1145/1328438.1328457.

N. Danner and J. S. Royer. Two algorithms in search of a type system. *Theory of Computing Systems*, 45(4):787–821, 2009. doi: 10.1007/s00224-009-9181-y.

N. Danner, J. Paykin, and J. S. Royer. A static cost analysis for a higher-order language. In M. Might and D. V. Horn, editors, *Proceedings of the 7th workshop on Programming languages meets program verification*, pages 25–34. ACM Press, 2013. doi: 10.1145/2428116.2428123.

N. Danner, D. R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. arXiv:XXXX.XXXX, 2015.

S. K. Debray and N.-W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993. doi: 10.1145/161468.161472.

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

J. Hoffmann and M. Hofmann. Amortized resource analysis with polynomial potential: A static inference of polynomial bounds for functional programs. In A. D. Gordon, editor, *Programming Languages and Systems: 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, 2010. doi: 10.1007/978-3-642-11957-6_16.

J. Hoffmann and Z. Shao. Automatic static cost analysis for parallel programs. In J. Vitek, editor, *Programming Languages and Systems: 24th European Symposium on Programming, ESOP 2015*, volume 9032 of *Lecture Notes in Computer Science*, pages 132–157. Springer-Verlag, 2015. doi: 10.1007/978-3-662-46669-8_6.

J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *ACM Transactions on Programming Languages and Systems*, 34(3):14:1–14:62, 2012. doi: 10.1145/2362389.2362393.

S. Jost, K. Hammond, H.-W. Loidl, and M. Hofmann. Static determination of quantitative resource usage for higher-order programs. In M. Hermenegildo, editor, *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 223–236. ACM Press, 2010. doi: 10.1145/1706299.1706327.

D. Le Métayer. ACE: an automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988. doi: 10.1145/42190.42347.

X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009. doi: 10.1016/j.ic.2007.12.004.

E. Moggi. Notions of computation and monads. *Information And Computation*, 93(1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4.

J. Navas, E. Mera, P. López-Garcia, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In V. Dahl and I. Niemelä, editors, *Proceedings of Logic Programming: 23rd International Conference, ICLP 2007*, volume 4670 of *Lecture Notes in Computer Science*, pages 348–363, 2007. doi: 10.1007/978-3-540-74610-2_24.

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

M. Rosendahl. Automatic complexity analysis. In J. E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 144–156. ACM Press, 1989. doi: 10.1145/99370.99381.

D. Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, University of London, 1990.

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

K. Van Stone. *A Denotational Approach to Measuring Complexity in Functional Programs*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2003.

P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 307–313, 1987. doi: 10.1145/41625.41653.

P. Wadler. The essence of functional programming. In R. Sethi, editor, *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM Press, 1992. doi: 10.1145/143165.143169.

B. Wegbreit. Mechanical program analysis. *Communications of the Association for Computing Machinery*, 18(9):528–539, 1975. doi: 10.1145/361002.361016.